UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

An Energy-based Approach to Learning and Memory in Artificial Neural Networks

Permalink

https://escholarship.org/uc/item/1q39j23g

Author

Alonso, Nicholas

Publication Date

2023

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

An Energy-based Approach to Learning and Memory in Artificial Neural Networks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Cognitive Science

by

Nick Alonso

Dissertation Committee:
Professor Jeff Krichmar, Chair
Professor Mark Steyvers
Assistant Professor Aaron Bornstein
Professor Emre Neftci

2023

© 2023 Nick Alonso

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

VITA **ix**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**

2 Background **4**
2.1 Artificial Neural Networks 4
2.2 Bio-Constrained Machine Learning 5
2.3 Three Challenges for Bio-Constrained ANNs 6
2.3.1 Local Learning and Credit Assignment 7
2.3.2 Online Learning 8
2.3.3 Continual Learning 11
2.4 Two Approaches to Local Learning Algorithms 12
2.4.1 Backpropagation-based Algorithms 12
2.4.2 Energy-based Algorithms 15

3 Overview and Synthesis of Papers **20**
3.1 Toward A New Understanding of Energy-based Algorithms 20
3.2 Energy-based Models as Online Learners 24
3.3 Sparse, Quantized EBAs as Online-Continual Learner 26
3.4 Summary 29

4 A Theoretical Framework for Inference Learning **30**
4.1 Introduction 30
4.2 Related Works 32
4.3 Background 34
4.3.1 Notation 34
4.3.2 Generalized Backpropagation 34
4.3.3 Generalized Inference Learning 36

4.4 Theoretical Results . . . 38
4.4.1 Implicit Gradient Descent . . . 38
4.4.2 Main Results . . . 39
4.5 Experiments . . . 42
4.6 Limitations . . . 46
4.7 Discussion . . . 47

5 Understanding and Improving Optimization in Predictive Coding Networks 49
5.1 Introduction . . . 49
5.2 Background and Notation . . . 51
5.2.1 Notation . . . 51
5.2.2 Predictive Coding and Inference Learning . . . 52
5.2.3 Inference Learning Approximates Implicit Gradient Descent . . . 53
5.3 Theoretical Results . . . 55
5.3.1 IL with MQ and Implicit SGD . . . 55
5.3.2 IL is Sensitive to Second-Order Information . . . 57
5.3.3 Reinterpreting IL's Relation to BP and SGD . . . 58
5.4 The Matrix Update Equalization Optimizer . . . 59
5.5 Reducing Computation with Sequential Inference . . . 60
5.6 Experiments . . . 62
5.7 Related Work . . . 64
5.8 Limitations . . . 66
5.9 Discussion . . . 66

6 A Sparse Quantized Hopfield Network for Online-Continual Memory 68
6.1 Introduction . . . 68
6.2 Results . . . 70
6.2.1 Toward A Foundation for Local, Online-Continual Memory Models . 70
6.2.2 The Sparse Quantized Hopfield Network . . . 71
6.2.3 Experiments . . . 76
6.2.4 Further Comparison of SQHN Architectures . . . 84
6.3 Discussion . . . 87

7 Conclusions and Future Directions 91

Bibliography 95

Appendix A Chapter 4 Supplementals 105

Appendix B Chapter 5 Supplementals 151

Appendix C Chapter 6 Supplementals 169

LIST OF FIGURES

Page

2.1 Visualization of the difference between online and offline learning. 9

2.2 Circuitry of BP-based (left) and energy-based (right) algorithms. BP-based algorithms use external circuits to provide learning signals to each layer. These learning signals do not alter FF activities. EBAs directly update output layer activities to reduce loss, then feed-back and feed-forward signals alter activities at hidden layers through recurrent dynamics that keep the energy of the network small. 13

2.3 Visualization of the EBA. (Left) First, neuron activities are set to the free phase fixed point, which sits at the minima of the energy when the output layer is not clamped. Second, the clamped phase fixed point is computed by softly clamping the output layer to the target, which reduces the loss and reducing the energy w.r.t. to all neurons until an equilibrium is reached, at which point the gradients of the loss and energy will be equal. (Right) The weight update shifts the free phase activities toward clamped phase activities, reducing the loss of the free phase. 17

3.1 Summary of how the present EBA work contributes to improving ANNs. . . 29

4.1 A depiction of IL. Target activities $\hat{h}$ are initialized to FF activities h then updated to minimize energy F during the inference phase. Afterward, weights are updated to further minimize F. Weight updates 'align' with $\hat{h}$ activities so $\hat{h}$ become the new FF activities given the same data point. $\hat{h}$ are activities that will both improve the loss L and minimize $\|\Delta\theta\|^2$. This process is equivalent to minimizing the proximal objective and approximates implicit SGD. 40

4.2 We measure the proximal objective during the inference phase, where F is minimized w.r.t. activities $\hat{h}$, in a two hidden layer MLP with ReLU activations trained on MNIST. We test under four different initializations of $\hat{h}$. First, all $\hat{h}$ are initialized to h (FF Init.). Second, hidden layer $\hat{h}$ are initialized to h, while output layer $\hat{h}_N = y$ (FF + Full Clamp). For both of these initializations, the proximal loss is reduced to near zero for both the IL-prox and IL-SGD variants of IL (defined below). We next initialized $\hat{h}$ randomly (Rand Init. and Rand + Full Clamp). Both networks reduce the proximal loss, but nowhere near the global minima, which may help explain why the $\hat{h}$ needs to be initialized to FF activities when F is minimized with SGD. These results support theorem 4.1, which states that minimizing F w.r.t. $\hat{h}$ also minimizes the proximal objective. 43
4.3 Training runs for mini-batch size 1 (online) scenario, for classification and autoencoder tasks. MLP size 784-2x500-10 and autoencoder dimension 784-256-100-256-784 are trained on F-MNIST for 50000+ iterations. MLP size 3072-3x1024-10 and autoencoder 3072-1024-500-100-500-1024-3072 are trained on CIFAR-10 for 50000+ iterations. The first 4000 training iterations shown for F-MNIST on left. Full training run are shown for other simulations in right three plots. 46
4.4 Training runs for a subset of the algorithms on the data constrained scenarios. Each model trained on n data points from each category. 47

5.1 Measurement of the proximal objective ($\beta = 100$) during inference phase with fully clamped output layer of standard IL (IL) (20 iterations) and sequential IL (SeqIL) (5 iterations) with a fixed learning rate or MQ adaptive learning rates. Consistent with theorem 5.1, reducing energy F during the inference phase leads to a reduction of the proximal objective. 55
5.2 Average update magnitude of each weight matrix in MLP with three hidden layers. 59
5.3 Measurement of the mean of the squared local errors, e_l, at each hidden layer (layer 1-4) and the output layer (layer 5) in a fully connected network with four hidden layers. 61
5.4 MLPs trained on CIFAR-10 for 45 epochs, using either sequential inference with Adam (SeqIL-Adam) or simultaneous inference with Adam (SimIL-Adam). X-axis specifies the number of inference iterations used during training. . . . 62
5.5 Small convolutional networks trained on classification tasks. Training runs are averaged over 5 seeds. Shaded error bars show standard deviation across seeds. Each training run is shown up until convergence (the point of lowest test error +1 epoch). 63
5.6 Fully connected networks trained for classification tasks (MLP) and for autoencoder tasks (AE). Training runs are averaged over 5 seeds. Shaded error bars show standard deviation across seeds. The first 60 epochs of training shown for MLPs. For AEs, training runs are shown to convergence (lowest loss +1 epoch) or the max 100 epochs. 64

6.1 **A.** Tree structured, directed, acyclic graph. **B.** Diagram of recall process in terms of the node values and energy. Nodes take integer values. Correct recall finds the set of node values at the global maximum of the energy. Correct recall typically only occurs if the memory node has the correct value, since the memory node is fixed after the feedforward (FF) pass, after which it then adjusts the values of hidden nodes through feed-back/top-down signals. **C.** Neural network diagrams of the SQHN during associative recall and recognition. During recall the FF sweep propagates signals up the hierarchy, where the memory/root node retrieves the most probable (high energy) value, and propagates the signals down encouraging hidden nodes to take the values associated with that particular memory/value. 72

6.2 Online Continual Auto-Association. **Top** One hidden layer models with small (300), medium (1300), large (2300) node sizes. **Bottom** Three hidden layer models with small (200), medium (600), large (1000) node sizes. **A.** Recall accuracy and recall MSE during training on models with medium size hidden nodes. **B.** Cumulative recall MSE and recall accuracy for each model size. **C.** Order sensitivity for each model size. **D.** Online class incremental (OCI) versus online domain incremental (ODI) settings. 78

6.3 Noisy encoding task. **A.** Recall accuracy and recall MSE for one hidden layer models under white and noise and binary sample conditions. Example of the reconstruction during test time for SQHN+ model, in the case of 1, 6, and 12 samples. **B.** Recall accuracy and recall MSE for three hidden layer models under white and noise and binary sample conditions with reconstruction example for SQHN on the right. 82

6.4 Recognition task. **A.** The recognition accuracy for SQHN and MHN are shown in networks with 300 neurons at the hidden layer (300th iteration marked by vertical dotted line). **B.** The MSE for the SQHN model on the training MNIST data (Train Dist.), hold-out MNIST data (In Dist.), and the F-MNIST data (Out Dist.) **C.** The MSEs for the MHN model. 83

6.5 Comparison of SQHN architectures with different hidden layers. **A.** Depiction of various SQHN architectures. **B.** Recall accuracy across three data sets (CIFAR-100, Tiny Imagenet, Caltech 256) under the white noise and several occlusion scenarios. **C.** Visualization of the black, color, and noise occlusions. **D.** Recall accuracy during online training without noise (top row) and MSE on a test a test set (bottom row). The lower bound on recall accuracy posited by theorem C.2 (supp. C.2.7) marked by gray line. Models tested with different maximum number of neurons per node (200, 600, 1000). **E.** Recognition accuracy for three SQHN models, where 500 neurons are allocated to each node (vertical dotted line marks when all 500 neurons are grown). CIFAR-100 data used for train and in-distribution set, while a flipped pixel version of the Street View House Numbers (SVHN) dataset used for out of distribution. Best guessing strategy yields 66% accuracy (horizontal dotted line). 85

LIST OF TABLES

Page

4.1 Notation 34
4.2 Best test reconstruction loss after one epoch of training, mini-batch size 1. BCE was averaged across pixels and data-points. Top two scores highlighted in bold. All standard deviations $\leq .002$. 44
4.3 Best test accuracy on CIFAR-10 supervised task with mini-batch size 1 after 50,000 iterations (1 epoch) and with mini-batch size 64 with 77000 iterations. 44
4.4 Accuracies after 50,000 training iterations on CIFAR-10, mini-batch size 1. Fully connected networks with layer sizes 3072-3x1024-10. 45

5.1 Notation 52
5.2 Best test accuracies averaged across 5 seeds (mean $\pm$ standard dev.). Top scores and scores not significantly different from top score (according to two-sample t-test) are bolded. 64

6.1 **Top** Recall MSE of 1024 images from CIFAR-10 and Tiny ImageNet datasets. At test time, images are either corrupted with white noise (variance .2), 25% of the pixels dropped, or the right 25% of the pixels masked. **Bottom** Recall MSE of 128 images from CIFAR-10 and Tiny ImageNet datasets. At test time, images are either corrupted with white noise (variance .8), 75% of the pixels dropped, or the right 75% of the pixels masked. 77

ACKNOWLEDGMENTS

I would like to thank my family, Mom, Dad, Leanna, and Joey, for their continued support throughout my grad school career. I also need to thank my fiancé Marissa for the daily support she provided while I progressed through my PhD, and for never complaining when I talked endlessly about neural networks and AI. Couldn't have done it without y'all.

I also need to thank my co-advisors Emre Neftci and Jeff Krichmar for their excellent mentoring and for having the patience to let me explore the topics that really interested me. I feel lucky to have been part of both Emre and Jeff's lab and I am very happy with how our research collaborations turned out.

Thank you also to the other members of Emre's Neuromorphic Machine Intelligence Lab and Jeff's Cognitive Anteater Robotics Lab for their great work and enthusiasm in neuroscience and AI, which kept me inspired and impressed throughout my PhD.

My PhD work was partially funded by National Science Foundation (NSF) under grant 1652159 and by the National Science Foundation Grant IIS-1813785 and the Air Force Office of Scientific Research Grant FA9550-19-1-0306

Three papers are presented in this work. Chapter 4 is a version of a paper published in Neurips 2022 proceedings. Neurips does not have copyright privileges and only has a "non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper". Chapter 5 and 6 present preprints of papers not yet published and are under review.

VITA

Nick Alonso

EDUCATION

Doctor of Philosophy in Cognitive Science **2023**
University of California-Irvine *Irvine, CA*

Master of Arts in Philosophy **2019**
Georgia State University *Atlanta, GA*

Bachelor of Science in Cognitive Science and Philosophy **2017**
University of Michigan-Ann Arbor *Ann Arbor, MI*

PAPERS

Alonso, N., and Krichmar, J. (2023). A Sparse Quantized Hopfield Network for Online-Continual Memory. arXiv preprint arXiv:2307.15040.

Alonso, N., Krichmar, J., and Neftci, E. (2023). Understanding and Improving Optimization in Predictive Coding Networks. arXiv preprint arXiv:2305.13562.

Alonso, N., Millidge, B., Krichmar, J., and Neftci, E. O. (2022). A Theoretical Framework for Inference Learning. Advances in Neural Information Processing Systems, 35, 37335

Wang, F., **Alonso, N.**, and Teeter, C. (2022). Combining Spike Time Dependent Plasticity (STDP) and Backpropagation (BP) for Robust and Data Efficient Spiking Neural Networks (SNN) (No. SAND2022-16962). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Alonso, N., and Neftci, E. (2021). Tightening the Biological Constraints on Gradient-Based Predictive Coding. In International Conference on Neuromorphic Systems 2021 (pp. 1-9).

PROFESSIONAL EXPERIENCE

Graduate Research Assistant **2019–2023**
University of California, Irvine *Irvine, California*

Engineering Intern **Summer 2022**
Sandia National Labs *Albuquerque, New Mexico*

ABSTRACT OF THE DISSERTATION

An Energy-based Approach to Learning and Memory
in Artificial Neural Networks

By

Nick Alonso

Doctor of Philosophy in Cognitive Science

University of California, Irvine, 2023

Professor Jeff Krichmar, Chair

The standard method for training artificial neural networks (ANNs) is stochastic gradient descent (SGD) implemented by the backpropagation (BP) algorithm. Neuroscientists, however, readily agree it is unlikely that the brain uses BP, and BP/SGD has well-known performance limitations like slow convergence and issues with catastrophic forgetting. This raises the question of whether there are alternative learning algorithms that better fit with what is known about the brain and which may mitigate some of the performance drawbacks of BP/SGD. A class of more biologically plausible alternatives are *energy-based algorithms* (EBAs). Although EBAs have been used for decades, their theoretical foundations and performance on machine learning tasks are poorly understood. This thesis, through the presentation and synthesis of three papers, aims to develop the theoretical foundations and the empirical testing of EBAs on machine learning tasks. The first paper develops a novel theoretical description of a certain EBA, known as predictive coding (PC). The main claim is that predictive coding approximates a proximal algorithm, which is distinct from BP/SGD and was found to train faster than SGD, especially in online learning scenarios. The second paper describes proximal algorithms, which unlike SGD, are sensitive to second-order information and uses this to develop and test a novel optimizer for PC networks that allows for reductions in memory, compute, and improves convergence. The third paper develops

a novel EBA, called a sparse quantized Hopfield network, which performs various memory tasks and learns in online and continual settings. This model was shown to significantly outperform similar models trained with BP. The contributions of these papers are synthesized to support the more broad claims that 1) EBAs are proximal algorithms sensitive to second order information, 2) ANNs trained with EBAs are effective, principled online learners, and 3) sparse ANNs trained with EBAs can be effective, principled online-continual learners. Together, these claims suggest EBAs could be a promising alternative to BP/SGD for training ANNs and for developing theories for how brains learn.

Chapter 1

Introduction

The brain has an amazing ability to store and utilize information across massive networks of neurons. This fact suggests the brain implements a kind of *learning algorithm*, or a set of rules for altering model parameters to achieve some objective. A central question within cognitive neuroscience is what general learning algorithms are utilized by the brain. Although neuroscientists have made progress in understanding the biochemistry and dynamics of individual synapses in the brain, much less is known about the sorts of algorithms the brain uses to coordinate learning across large, deep neural networks to yield intelligent behavior.

Related questions are of central concern to the field of artificial intelligence, whose recent progress has been driven mainly by artificial neural networks (ANNs), which are trained using learning algorithms to perform various tasks. It is often thought that the field of deep learning could provide some ideas for understanding learning algorithms in the brain [102]. Deep learning has had amazing success training large ANNs with stochastic gradient descent (SGD) implemented by the backpropagation (BP) algorithm [108], and BP is now the standard for training deep networks. However, although it is still debated, neuroscientists increasingly agree BP-like algorithms are incompatible with much of what is known about

the brain (e.g., [25, 63, 133]). Further, unlike the brain, large ANNs trained with BP learn notoriously slow, suffer from catastrophic forgetting when learning multiple tasks continuously [29, 39], and do not allow for easy parallelization of the learning process [46], among other issues.

Considerations like these raise the question of whether there are principled and practically viable alternatives to BP-like algorithms for training deep ANNs, which better fit what we know about the brain and which avoid other shortcomings of BP. There are at least two motivations for researching bio-inspired alternatives to BP: 1) It could lead to novel theories of how large networks of neurons in the brain coordinate learning, and 2) It could lead to algorithms for ANNs that have performance advantages over SGD and BP (e.g., faster convergence with less catastrophic forgetting).

In this dissertation, I present a BP-alternative approach to training ANNs based on more biologically constrained *energy-based algorithms* (EBAs), which work in fundamentally different ways than BP. EBAs have been used to train ANNs for decades. However, their theoretical foundations and their ability to perform machine learning tasks are not fully understood. I present work in which I, along with my collaborators, lay the beginnings of a new theoretical foundation for EBAs and test and develop EBA models on various machine learning tasks. In particular, our work begins to show how and why EBAs are a mathematically justified method for training ANNs, have certain general advantages over first-order SGD, and are a particularly promising and principled basis for developing ANNs that learn in training scenarios similar to those faced by biological organisms, called online and continual learning scenarios (explained more below).

Three papers on EBAs published by myself and my collaborators during my PhD research are presented. Chapter 2 first provides a background of the relevant concepts and algorithms needed to understand the papers. Importantly, Chapter 2 provides a description of the three challenges facing the bio-constrained learning algorithms, which are a focus of the three

papers covered: local learning, online learning, and continual learning. Chapter 3 provides a brief overview and synthesis of the main results of the three papers. The aim of this synthesis is to provide a coherent story as to how and why EBAs are a promising basis to overcome the three challenges listed above, and how the results from my papers contribute to this narrative. More details are then provided by versions of the three papers presented in chronological order in Chapters 4,5, and 6. Chapter 7 discusses conclusions and future directions.

Chapter 2

Background

2.1 Artificial Neural Networks

ANNs were originally designed to mimic the way biological neural circuits operate [71]. Although standard ANN architectures ignore many biological details of the brain, they are often thought to capture important aspects of the way large networks of neurons interact. Many neuroscientists have argued that studying ANNs and using them as a modelling tool could provide a path toward better understanding the brain [67, 102]. The work presented in what follows is developed under this assumption.

Below we primarily concern ourselves with multi-layered ANN models. ANNs have layers of neurons, h_l, where l marks the layer. Layer h_0 is the input layer and h_L the output layer. Layers in between are hidden layers. Synapses link adjacent layers via synaptic weight matrices, e.g., layer l propagates signals to layer $l+1$ via matrix W_l. The input layer, h_0 is usually clamped to some input data, x. Feed forward values are computed via a matrix multiply between presynaptic activities and the weight matrix. For example, a linear ANN

with three hidden layers would compute a feed-forward (FF) output layer value as follows:

$$h_3 = W_2 W_1 W_0 h_0. \tag{2.1}$$

Non-linear functions, f, are typically added to network architectures such that FF signals from layer l to layer $l+1$ become $f(W_l h_l)$. To simplify notation, and without loss of generalization, we focus on linear networks in the next two sections. Note that although we discuss more complex networks that also have feedback signals, in most of these architectures, FF signals are computed using the variants of the equations above. Feedback signals are described in further detail when needed.

Deep ANNs are often described as storing (at least) two kind of memories within their synapses. First, are hetero-associative memories where an input vector x is associated with a distinct output target y, e.g. an image is associated with a target category label. Second, auto-associative memory tasks are those where an input x is associated with itself. The purpose of this task may be to develop, e.g., compression models like various kinds of auto-encoders or to remove corruption of input, such as in de-noising auto-encoders or Hopfield networks. Without loss of generality, our notation focuses on hetero-associative tasks in the next two chapters, though our comments extend to auto-associative networks as well.

2.2 Bio-Constrained Machine Learning

My research discussed below can be labeled as 'bio-constrained machine learning'. The research aims to develop ways of training deep ANNs for machine learning tasks that share similar constraints as the brain. For example, much of the work trains common deep network architectures on standard image data-sets for classification or self-supervised learning using learning algorithms that fit certain fundamental properties of circuitry and synaptic plasticity

in the brain.

Why pursue projects in bio-constrained machine learning? The motivation on the neuroscience side, is the hypothesis that getting bio-constrained models and algorithms to solve machine learning tasks at scale could yield new theories for how the brain works. Indeed, many of the most prominent and general theories in neuroscience have borrowed ideas from AI and machine learning, e.g., Bayesian brain hypothesis [50], free energy principle [33], and models of reward based learning [12]. Further, there is an argument that the algorithms and computations the brain uses performs well on large scale machine learning tasks. Therefore, any model or algorithm that does not perform well on the sorts of tasks humans do at relatively large scale is likely missing something important about the brain.

On the machine learning side, taking inspiration from the brain could yield new ideas useful for machine learning applications. Human brains are typically more energy efficient, faster learners, less forgetful, and can learn a wider variety of tasks than existing deep learning models. Thus, looking to neuroscience could provide new avenues for AI development.

2.3 Three Challenges for Bio-Constrained ANNs

In this section, I describe three challenges faced by brains that must learn in the real world: the challenges of local learning, online learning, and continual learning. One can also think of these challenges as determining the main biological constraints I considered when designing and testing learning algorithms, i.e., the constraint that learning algorithms must yield ANNs that are effective local, online, and continual learners. This list is not meant to be exhaustive, but I believe it covers some of the deepest challenges that the learning algorithms used by the brain must overcome.

2.3.1 Local Learning and Credit Assignment

Fundamental to deep learning is the notion of a *loss function*. A loss function, $\mathcal{L}(x, y, \theta)$ take as input data, x, desired output target, y, and model parameters θ and outputs a scalar that represents a measure of the network's performance. Typically, the loss computes some error, e.g., squared error, between the output of the network, h_{out} and the prediction target y. The goal of training is to update the model parameters to reduce the loss as much as possible over some data set.

In order to update the synapses of a deep neural network to reduce a loss, it is typically necessary to have some method for 'assigning credit', i.e., a description of the way each synapse contributed to the loss (e.g., a 'good' (+) or 'bad' (-) contribution) and how much the synapse contributed (e.g., a strong effect or a weak effect). In shallow perceptions, where there is only an input and output layer, this process is fairly straightforward, since each synapse connects directly to an output neuron, which can be directly assigned an error value. For example, a linear regression model of the form, $h_{out} = W_0 h_0$, would have output units, h_{out}, that directly take an error value, $e = y - h_{out}$, which can trivially be used to assign praise and blame to each parameter in W_0.

However, in a deep ANN with hidden layers, this task is not so straightforward, since there will be many synapses that do not directly connect to the output neurons used in the loss function. For example, in a linear MLP of the form $h_{out} = W_1 W_0 h_0$, it is much less obvious how we should assign praise and blame to the parameters in W_0 to improve e. The difficulty of assigning credit in deep networks with one or more hidden layers is known as the *credit assignment problem* [63].

Since biological organisms with nervous systems make associations and predictions, it follows that the brain is optimizing some loss function and therefore must have some method for solving the credit assignment problem. However, an important constraint on the brain's

learning algorithms, not necessarily shared by ANNs, is that of *locality*. That is, synaptic plasticity in the brain depends only on information spatially and temporally local to the synapse. In this sense, the brain uses a *local learning algorithm*. For example, classic Hebbian models of synaptic plasticity describe the change in a synapse as proportional to the product of pre- and post-synaptic firing rates present at the time of the update [38]. Other information used to update synapses may be timings of pre- and post-synaptic neuron spikes [119], statistics about local neuron activities, and/or neuromodulation of synapses.

2.3.2 Online Learning

By far, the most common training paradigm in deep learning is *offline learning*. Offline learning is a kind of pre-training method, where one first trains the model, then deploys the model to perform whatever task it was trained to perform. For example, AlphaFold [130], a deep learning model that predicts protein structure from amino acid sequences, is first trained offline on a database of amino-acid protein structure pairs, then deployed as a tool for scientists to use in research.

Data sets used in offline scenarios are stored locally such that they can be accessed and organized as desired by the engineers training the model. This offline procedure allows engineers to mini-batch data during training, i.e., provide a sample of multiple data points to the model each iteration and average updates over the data points. Mini-batching significantly reduces noise in the data compared to using individual data points each iteration. Offline scenarios also allow models to train over the same data set many times (i.e., for many 'epochs'), which allows slow learning models to keep improving performance until a minimum is reached, rather than to have their training cut short after a single pass through the data set. Because models are trained before being deployed, the relevant performance measure for offline learning is the *test loss*, $\mathcal{L}(X^{test}, Y^{test}, \theta^t)$, which is the average loss the

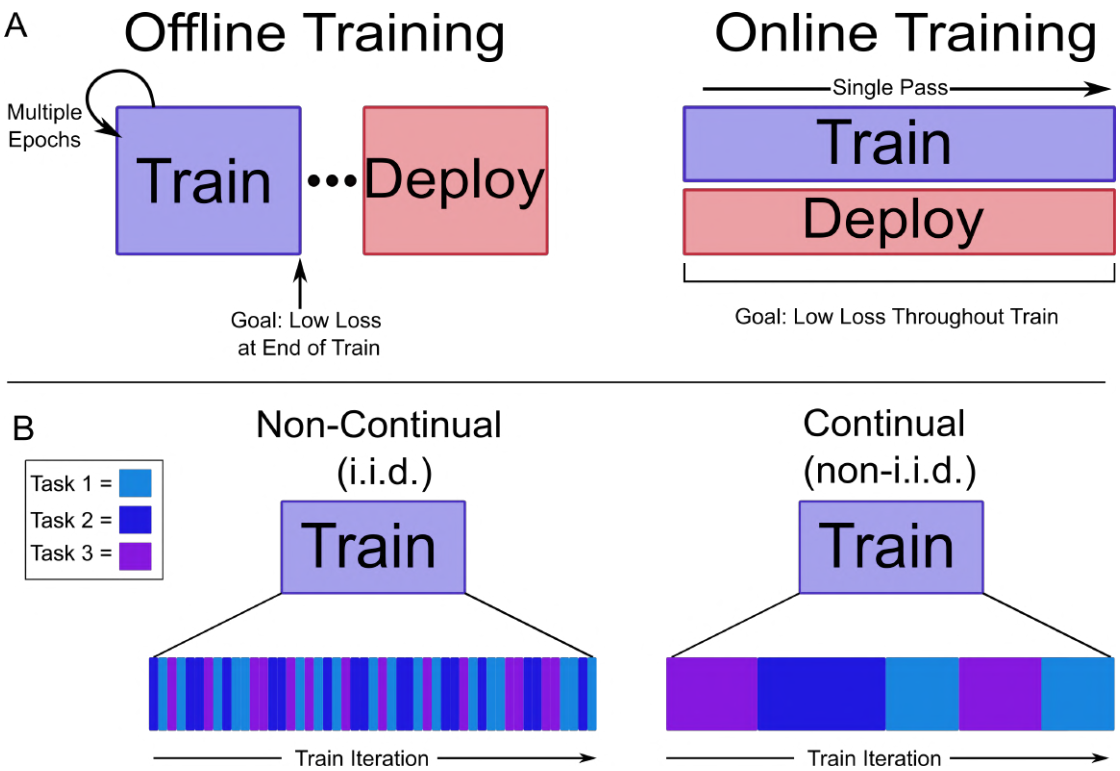

Figure 2.1: Visualization of the difference between online and offline learning.

model produces on hold-out data, (X^{test}, Y^{test}), which was unobserved during training. The test loss gives the engineer an estimate of how well the model will perform after training, when it is deployed and operating on new unobserved data (see 2.1 A, left for summary).

The brain, on the other hand, is never offline and must handle *online learning* scenarios. In online scenarios, it is assumed models both train and are deployed simultaneously. Animals, for example, both perform tasks important for their survival and learn simultaneously. Robots and sensor devices, (e.g. satellite or security cameras) that learn while performing tasks in the real world also must learn online. Typically, in online learning scenario it is assumed that: 1) the entire set of observations cannot be stored locally by the model, 2) the model performs only a single pass through the data set, and 3) the model sees only a single data point each training iteration. These properties reflect what it is like for animals and AI embedded in the real world, which receive a continuous stream of data from their sensors, and cannot store every observation in memory.

Importantly, online learners optimize a different kind of loss function than offline learning, known as a cumulative loss [115]. [1] The cumulative loss is the average of the loss across the entire training run:

$$\mathcal{L}^{T}_{Cumul.}(X, Y) = \sum_{t=0}^{T} \mathcal{L}(x^t, y^t, \theta^t), \tag{2.2}$$

where θ^t are the parameter values at iteration t and x^t, y^t are the data at t. Other cumulative performance measures may be used as well, e.g., a cumulative test loss $\sum_{t=0}^{T} \mathcal{L}(X^{test}, X^{test}, \theta^t)$. Cumulative losses are motivated by the fact in online scenarios models must perform well across its entire deployment, which implies it must perform well across its entire training run, not just at the end (see fig. 2.1 A, right).

Training a model online is more difficult than training offline for several reasons: 1) Because models only observe a single data point per iteration model updates are noisier than mini-batch updates, in the sense they are poorer estimates of the updates that reduce the loss over the entire training batch. 2) Only a single pass through the data set is performed in online scenarios, which means the model must somehow learn quickly (since it cannot learn from a data point more than once) without destroying previous knowledge (since it cannot revisit the previously observed data it has knowledge about). Balancing fast learning with knowledge retention in neural nets is equivalent to the *stability-plasticity* dilemma [1]. This dilemma is barely an issue in typical offline scenarios that allow for multiple epochs of training and mini-batching.

[1] Alternatively, a measure called 'regret' is also sometimes used, which is just the cumulative loss achieved by the model with the best performing parameter values minus the cumulative loss achieved by the trained model [115]. In either case, the model is trained to minimize cumulative loss.

2.3.3 Continual Learning

Offline scenarios, and much of the classic machine learning work on online learning (e.g., [23, 115]), assumes data is independently and identically distribution (i.i.d.). That is, it is assumed that the probability a data point is sampled during some training iteration, t, is independent of and equal to other data points in the train set. *Continual learning* scenarios are those where data is non-i.i.d. distributed. For example, a common continual learning scenario for classification tasks is called *class-incremental learning*, where a model, trained to performed classification, might be presented with data from class 1 for t training iterations, then class 2 for t iterations, and so on.

The main difficulty with continual learning is preventing catastrophic forgetting during learning, where the model forgets a previous task after learning a new task. It is well known that standard deep ANN architectures trained with BP suffer from catastrophic forgetting [39]. This difficulty is related to the stability-plasticity dilemma in the sense that in continual learning scenarios neural networks need to learn new tasks while preventing the forgetting of old ones, and therefore must allow for some plasticity while keeping synapses stable to some degree.

However, the pressure to learn quickly in continual learning is not present in recent continual learning benchmarks used by the machine learning community, since these tasks allow models to learn each task offline (for reviews see [131]). That is, in recent literature, models can train for multiple epochs on mini-batched data on task A before moving to task B where it again trains for multiple epochs on mini-batched data, and so on.

Much more difficult is the combined *Online-Continual* scenario where an agent learns from non-i.i.d. data presented one at a time, where each data-point is observed once. This task requires agents to both deal with the difficulties of online learning (noise and pressure for sample efficiency) with those of continual learning (catastrophic forgetting). The online-

continual scenario much better describes the sort of learning scenario embedded learners (e.g., animal and robots) face than just the online or continual scenario alone, since sensor inputs are non-i.i.d. and presented online. The combined online-continual learning task has only been developed into formal machine benchmarks by the deep learning community in the last few years [96, 35, 41, 64].

2.4 Two Approaches to Local Learning Algorithms

In this section, I present what I consider to be the two general approaches to local learning algorithms used in the literature. This list is not exhaustive. There are some algorithms that may not quite fit these two general categories, but nonetheless, as far as I can tell these two categories capture the most studied and best performing local learning algorithms in ANNs.

2.4.1 Backpropagation-based Algorithms

An SGD update performs a step down the loss gradient over ANN parameters: $\Delta\theta_{SGD} = -\alpha\frac{\partial\mathcal{L}}{\partial\theta}$, where α is the learning rate. The SGD update for an individual weight matrix in an ANN is therefore $\Delta W_{l,SGD} = -\alpha\frac{\partial\mathcal{L}}{\partial W_l}$. Gradients may be computed in many different ways. BP is one method for computing these gradients. BP treats ANNs as compositions of functions, which allows it to use the chain rule to differentiate through the function composition that describes the ANN. The gradient update computed in this way for an ANN is:

$$\Delta W_{l,BP} = -\alpha\frac{\partial\mathcal{L}}{\partial W_l} = -\alpha(\frac{\partial\mathcal{L}}{\partial h_{out}}\dots\frac{\partial h_{l+2}}{\partial h_{l+1}})\frac{\partial h_{l+1}}{\partial W_l}, \tag{2.3}$$

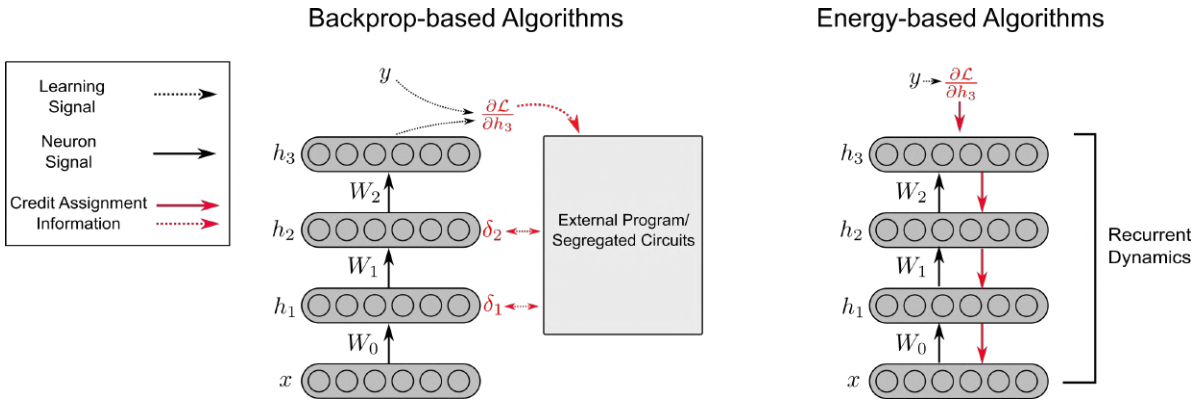

Figure 2.2: Circuitry of BP-based (left) and energy-based (right) algorithms. BP-based algorithms use external circuits to provide learning signals to each layer. These learning signals do not alter FF activities. EBAs directly update output layer activities to reduce loss, then feed-back and feed-forward signals alter activities at hidden layers through recurrent dynamics that keep the energy of the network small.

where h_l is the neuron activity at layer l. If we use a squared error loss, $\mathcal{L} = \frac{1}{2}\|y - h_{out}\|^2 = \frac{1}{2}\|e\|^2$, for a linear ANN this amounts to

$$\Delta W_{l,BP} = \alpha(W_{l+1}^\top W_{l+2}^\top \dots W_{L-1}^\top e)h_l^\top = \alpha\delta_{l+1}h_l^\top. \tag{2.4}$$

Notice that this update is *non-local* in the sense that one of the terms in the weight update, δ_{l+1}, explicitly involves the global loss gradients at the output layer, e, i.e., a weight update to W_l requires explicit knowledge of a quantity represented at a different spatial location in the network.

One class of local learning algorithms can be described as BP-derivatives. These algorithms begin with the BP algorithm and make alterations such that the resulting algorithm's weight updates are in some sense more local. Two general approaches are common for local spatial credit assignment.

First are, what I will call, direct error feedback (DF) methods. Instead of propagating the global error through a series of matrix multiplies, DF methods directly communicate the global loss gradient, e, to each layer, l, using some direct feedback matrix B_l. For a linear

ANN the resulting update is

$$\Delta W_{l,DF} = \alpha(B_l e)h_l^\top = \alpha\delta_{l+1}h_l^\top. \tag{2.5}$$

We can see that now the error signal is more local in the sense it is only one synaptic connection removed from the global loss. DF methods can be used to approximate SGD (e.g., [89, 86]), in which case $B_l \approx W_{l+1}^\top W_{l+2}^\top \dots W_{L-1}^\top$ or other optimization methods like Gauss-Newton optimization [74], which make different approximating assumptions about B_l.

Second, are what I will generally call target propagation methods. Target-prop methods work by setting up a local prediction 'target', $\hat{h}_l$, at each layer l using propagated errors signals. For example, the target at the the last hidden layer is $\hat{h}_{L-1} = h_{L-1} + B_{L-1}\delta_L$, where $\delta_L = -\frac{\partial \mathcal{L}}{\partial h_L}$ and B_{L-1} is some backward propagating matrix. Signals then propagate back to other hidden layers using the same process: $\hat{h}_l = h_l + B_l\delta_{l+1}$, where $\delta_{l+1} = \hat{h}_{l+1} - h_{l+1}$. The resulting weight update is

$$\Delta W_{l,TP} = \alpha(\hat{h}_{l+1} - h_{l+1})h_l^\top = \alpha\delta_{l+1}h_l^\top, \tag{2.6}$$

where δ_{l+1} is the error signal at layer $l+1$. This error signal is local in the sense that we can describe it in terms of an error at layer $l+1$, $\delta_{l+1} = \hat{h}_{l+1} - h_{l+1}$, rather than the output layer. Like DF methods, target-prop methods can be made equivalent or very close to SGD [28], Gauss-Newton [74], or other optimization methods [30].

Notice there are several properties all of these BP-variants share. First, they all use an update rule of the same form (equations 2.4, 2.5, 2.6), where pre-synaptic feed-forward activity, h_l is multiplied by an error signal, δ_{l+1}, at the next layer. Second, feedback signals do not interact with or alter FF activities h, i.e., feedback and feedforward signals are segregated, and feedback is in this sense external to the system being optimized (see figure 2.2). This second property is one important reason some believe the brain is likely not performing some

BP-derived algorithm, since the brain is highly interconnected, recurrent, and as such it is very difficult to find segregated circuitry in the abundance needed to implement a BP-like algorithm [63].

2.4.2 Energy-based Algorithms

Unlike BP-based algorithms, EBA algorithms are naturally local learning algorithms, recurrent, and do not require segregated circuits. EBAs work by updating neuron activities and parameters over time to reduce some energy function, E. Energy functions usually have the form of a summation over local energies:

$$E(h, \theta) = \sum_{l=0}^{L-1} E_l(W_l, h_l, h_{l+1}). \tag{2.7}$$

For example, two common energy measures are the free energy measure used in predictive coding (PC) models [100, 132] and a Hopfield energy (Hop) [44] used to train Hopfield network models:

$$\begin{aligned} E_{PC}(h, \theta) &= \sum_{l=0}^{L-1} \|h_{l+1} - W_l f(h_l)\|^2 \\ E_{Hop}(h, \theta) &= \sum_{l=0}^{L-1} f(h_{l+1}) W_l f(l) + f(h_l) W_l^\top f(h_{l+1}), \end{aligned} \tag{2.8}$$

There are multiple variants of these energies and many more derivatives in the literature.

Let's consider the hetero-associative task of associating a set of input vectors X with a set of output vectors Y. The goal of learning is to update parameters such that the loss produced in 'free phase' activities at the output layer h_L, reduces loss $\mathcal{L}(y^t, h_L)$, given x^t as input, across at x^t, y^t. Free phase activities, h, are those activities at the global minimum of the energy, $E(h, \theta, x)$, when only the input layer neuron values are clamped to input, x, but the

output layer values, h_L, are left free to change in a way that minimizes energy:

$$h = \arg\min_{h} E(h, \theta, x). \tag{2.9}$$

Often, these activities are found via gradient descent over activities, where activities are updated iteratively with the local energy gradients:

$$\Delta h_l \propto -(\frac{\partial E_l}{\partial h_l} + \frac{\partial E_{l+1}}{\partial h_l}) \tag{2.10}$$

The typical training procedure can be described as follows. Compute free phase activities, h. Then compute 'clamped phase' activities, $\hat{h}$, where output layer activities are clamped to the target in that they are updated to both minimize the loss $\mathcal{L}(y, \hat{h}_L)$ and the energy:

$$\hat{h} = \arg\min_{\hat{h}} \beta\mathcal{L}(y, \hat{h}_L) + E(\hat{h}, \theta, x), \tag{2.11}$$

where β is a scalar that controls the how soft the output layer clamp is, e.g., as $\beta \to \infty$ it is the case $h_L \to y$. This minimization of the energy in this phase can be interpreted as inferring hidden representation $\hat{h}$ that best describes input y, given x (see below for more detail).

Parameter updates that are most common in EBAs can then be described as further reducing the free and clamped phase energy difference E':

$$\begin{aligned} \theta^{t+1} &\approx \arg\min_{\theta} E(\hat{h}, \theta^t, x) - E(h, \theta^t, x) \\ &= \arg\min_{\theta} E'(\hat{h}, h, \theta^t, x), \end{aligned} \tag{2.12}$$

where the approximation in the first line can be interpreted as 'approximately minimizing' or 'reducing' the energy. The idea is that this update will make the free phase activities less

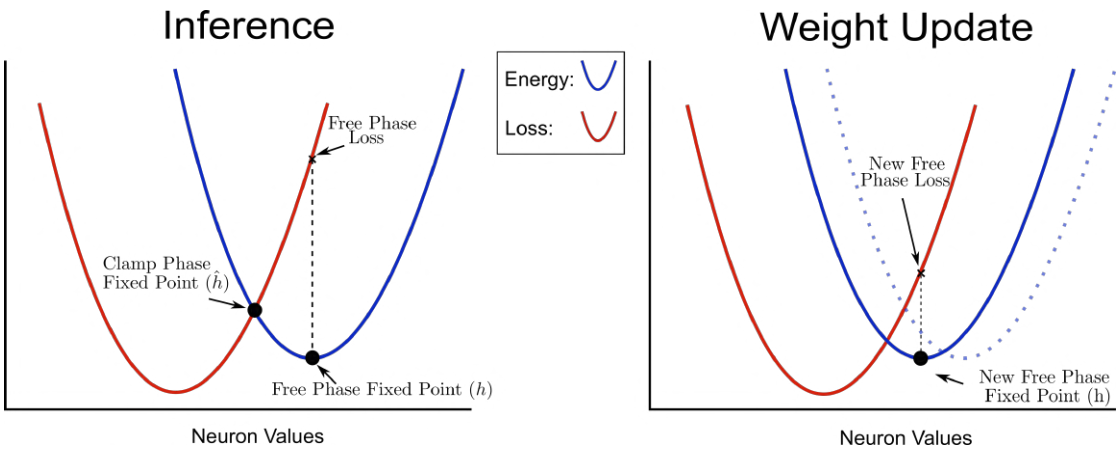

Figure 2.3: Visualization of the EBA. (Left) First, neuron activities are set to the free phase fixed point, which sits at the minima of the energy when the output layer is not clamped. Second, the clamped phase fixed point is computed by softly clamping the output layer to the target, which reduces the loss and reducing the energy w.r.t. to all neurons until an equilibrium is reached, at which point the gradients of the loss and energy will be equal. (Right) The weight update shifts the free phase activities toward clamped phase activities, reducing the loss of the free phase.

probable (higher energy), while making clamped phase more probable(lower energy) given the same input. The weight updates in other words should nudge free phase activities toward clamped phase activities, reducing loss (see figure 2.3).

The most common way to reduce energy w.r.t. weights is a gradient step:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E'}{\partial \theta} \approx \arg\min_{\theta} E'(\hat{h}^*, h, \theta^t, x, y). \tag{2.13}$$

Importantly, weight updates only require minimizing the local energies, and thus use local local learning rules, i.e., W_l is updated using only E_l. For example, gradient updates for PC and Hopfield networks are:

$$\begin{aligned} \frac{\partial E_{PC}}{\partial W_l} &= (h_{l+1} - W_l f(h_l)) f(h_l)^T \\ \frac{\partial E_{Hop}}{\partial W_l} &= f(h_{l+1}) f(h_l)^T, \end{aligned} \tag{2.14}$$

both of which are Hebbian like updates that are just products of pre- and post-synaptic

neuron activities. In the case of PC networks, this is true if we assume post synaptic neurons encode prediction errors; $e = h_{l+1} - W_l f(h_l)$.

Although the EBA described so far is for hetero-associative tasks, similar training procedures are used for auto-associative memory tasks, where the network learns to associate a vector input x with itself. For hetero-associative tasks, for example, one can either ignore the free phase computation and update, or run free phase update where no neurons are clamped at all similar to self-supervised training [43].

Although we have only described generic 'energy' functions so far, it is interesting to note probabilistic interpretations of EBAs. In particular, EBAs appear closely related to a variant of expectation maximization called maximum a posteriori learning. For example, Millidge et al. [78] show the PC energy is equivalent to the negative log of the joint distribution:

$$-\ln p(x, y, \hat{h}, \theta) = -\ln \exp(-E'_{PC}(\theta, h, \hat{h}, x)) = E'(\theta, h, \hat{h}, x). \tag{2.15}$$

Interestingly, Boltzman machines [43] use essentially the same idea, identifying the joint $p(x, \hat{h}, \theta)$ with $\frac{1}{Z}\exp(-E'_{Hop}(\theta, h, \hat{h}, x))$ using the Hopfield energy. This implies these EBAs are doing a kind of maximum a posteriori (MAP) learning, which can be described as:

1. $\hat{h}^* = \arg\min_{\hat{h}} -\ln p(x, y, \hat{h}, \theta)$
2. $\theta = \arg\min_{\theta} -\ln p(x, y, \hat{h}^*, \theta)$.

This implies the first step is performing a kind of MAP inference where $\hat{h}$ are the values of the hidden variable (the parameters of the hidden variables, e.g., Gaussian mean) that maximizes the posterior probability of these hidden variables given the inputs and parameters (which is equivalent to minimizing the negative log joint). It is unclear from the literature whether this equivalence to MAP learning is guaranteed for all EBAs that fit the generic form listed above. However, there does intuitively seem to be a strong resemblance: energy functions,

like joint distributions, are functions of ANN hidden states, parameters and data, and all EBAs seek to minimize these quantities during inference and learning, just as MAP learning seeks to minimize the negative log joint during inference and learning.

EBAs are more biologically constrained than BP in the sense they do not require segregated circuits or an external mechanism to provide information about the global loss. The standard learning rules are also local. However, until very recently, the literature on EBAs did not address certain important questions relating to the three challenges for bio-constrained algorithms noted above:

1. What sort of mathematical guarantees do EBAs have for reducing loss, and how are EBAs moving parameters across the loss landscape?
2. Can EBAs scale without requiring drastic increases in computation or memory?
3. Do EBAs provide any advantages in online and/or continual learning scenarios over BP or are they just as limited as BP/SGD in these learning scenarios?

In what follows, I outline how the research done during my PhD has helped make progress answering these questions and the three challenges for bio-constrained algorithms.

Chapter 3

Overview and Synthesis of Papers

In this section, I provide a brief summary of the results from the three papers presented in chapters 4 [8], 5 [7], and 6[6] and, importantly, I describe how the results from each paper link together and provide the beginnings of a solution to the local, online, and continual learning challenges to bio-constrained ANNs previously discussed.

3.1 Toward A New Understanding of Energy-based Algorithms

A complete local learning solution to the credit assignment problem should have at least the following properties: 1) a description of a local learning algorithm, 2) guarantees the algorithm will stably reduce the loss and/or converge, and 3) a description of how the algorithm moves the parameters across the loss landscape. EBAs provide a description of an algorithm for training neural networks. This alone, however, does not show under what conditions, if any, EBAs stably reduce the loss or explain how EBAs move parameters across the loss landscape. More mathematical analysis of the algorithm is therefore needed.

Until very recently, the main focus of the mathematical analyses of EBAs has been on their relationship to SGD. There is, in particular, a long history of analyzing similarities between Hopfield networks and SGD [83, 135, 16, 113], and, more recently, between PC and SGD [132, 120]. Interestingly, all energy-based models have essentially the same relationship to SGD, which is that in the limit where $\hat{h} \to h$ it is the case that $\frac{\partial E}{\partial W_l} \propto \frac{\partial \mathcal{L}}{\partial W_l}$. That is, as clamped phase activities, $\hat{h}$, approach free phase activities, h, the gradient of the energy w.r.t. each matrix becomes increasingly proportional to the loss gradient. This insight has provided a variety of ideas for altering the standard EBA algorithm to be more similar to SGD (e.g., [113, 120, 77]).

Although this analysis is highly useful for understanding how to alter the standard EBA algorithm to make it more similar to SGD, there is a concern that this unitary focus on analyzing the similarities between SGD and EBAs may be missing something important. For example, the limits in which EBAs approach SGD are not always well approximated in practice, and do not need to be for EBAs to perform well (e.g., see [8, 121]). Further, in these limits the link between hidden variables $\hat{h}$ and MAP inference becomes less clear because in these limits $\hat{h}$ becomes nearly insensitive to the input data. This raises the question of why EBAs still work when they do not closely approximate SGD and whether there is a better description of the way standard EBAs move parameters across the loss landscape.

These questions motivated my paper 'A Theoretical Framework for Inference Learning', covered in the next chapter, co-authored by Beren Millidge, Jeff Krichmar, and Emre Neftci [8]. This paper focused on the standard learning algorithms for the PC model, sometimes called inference learning (IL), and asked 'is there an existing optimization method that describes PC better than SGD?' The main thesis of the paper was that PC, under certain assumptions, is better described as performing *implicit gradient descent*, which is distinct

from standard SGD and can be described equivalently as a proximal algorithm:

$$\theta_{PC}^{t+1} \approx \theta_{prox}^{t+1} = \arg\min_{\theta} \mathcal{L}(x, y, \theta) + \frac{1}{2\alpha}\|\theta - \theta^2\|^2, \tag{3.1}$$

where θ_{PC}^{t+1} are the parameters after a PC update and θ^{t+1} are the parameters after a proximal update described on the right hand side of the equation. The proximal update sets the new parameters equal to those that minimize the loss and the distance between the new and old parameters. This update, therefore, finds new parameters that reduce the loss and remain in the proximity of old weights.

The proximal update has been proven to converge [94]. Further, I point out in a subsequent paper, which will be covered in Chapter 5, 'Understanding and Improving Optimization in Predictive Coding Networks' [7], along with co-authors Jeff Krichmar and Emre Neftci, a Taylor approximation of the proximal update shows that it approximates a regularized Newton's method of the form:

$$\theta_{prox}^{t+1} \approx \theta^t - \alpha(I + \alpha H)\frac{\partial \mathcal{L}}{\partial \theta^t}, \tag{3.2}$$

with error $\mathcal{O}(\alpha^2)$. Here H is the Hessian of the loss: $H = \frac{\partial^2 \mathcal{L}}{\partial \theta^{t2}}$.

What this implies is that when PC updates are closely approximating the proximal update they are sensitive to second-order (and likely even higher order) terms. This claim is consistent with a similar interpretation developed independently by another research group (see [45]) slightly after ours. A key prediction of this claim is that PC algorithms should reduce the loss more quickly than plain SGD, a purely first-order method, during training. Both of our papers found evidence of this. Alonso et al. found PC trained especially fast in the scenario with small mini-batches compared to SGD [8].

However, we also found PC sometimes converged to poor local minima, unless a memory

expensive optimizer was used. In our subsequent work, we found evidence this was because weight updates at early layers were very small, leading the network to converge to poor local minima near initialized parameters. (Note this finding is consistent with the proximal interpretation which keeps parameters near previous values.) We created a custom optimizer, the matrix update equalization optimizer, to mitigate this effect. The result was a PC-based learning algorithm that used no more memory than BP/SGD and only slightly more compute, but trained noticeably faster and converged to the same and similar losses.

There is, however, some limitations to this analysis. In particular, the approximation between PC and the proximal algorithm hold under several limiting assumptions: 1) regularization term is added to the energy of PC. This term is not uncommon in PC models but is non-standard. 2) Each term in the PC energy is weighted in a very particular way. These conditions turned out to be well-approximated by the standard PC algorithm in practice [8, 7], but still leave open the question of whether there is a description of PC that makes fewer assumptions.

A more general limitation is that we only analyze PC, and did not yet extend these results to other EBAs. Nonetheless, I believe there may be a general proximal algorithm description that applies to all EBAs. In particular, consider a distance function:

$$d_E(\theta, \theta^t) = E'(x, y, \hat{h}, \theta^t) - E'(x, y, \hat{h}, \theta), \tag{3.3}$$

where it is assumed $E'(x, y, \hat{h}, \theta^t) > E'(x, y, \hat{h}, \theta)$. Instead, of measuring the distance between parameters directly as with an L2 norm measure in the standard proximal algorithm, this measures the distance between parameters in *energy space*. This update finds the parameters that reduce the loss while also reducing the amount of change the parameters go through in energy space.

The proximal update that I believe can describe all EBAs is

$$\theta^t = \arg\min_{\theta} \mathcal{L}(\theta, x, y) + d_E(\theta, \theta^t), \tag{3.4}$$

which updates weights to both reduce the loss and the distance from old weight measured in energy-space. Proving that this update describes EBAs generally is a work in progress. However, I developed a similar proof in the appendix of my most recent paper discussed in chapter 6 [6]. The proof was for a particular EBA but the proof considered a general energy function and should therefore generalize. Thus, though more work is needed, the proximal interpretation of EBAs provides a promising basis for a general theoretical framework for EBAs.

3.2 Energy-based Models as Online Learners

The work outlined in the previous section suggests that EBAs can be interpreted as a kind of proximal algorithm. What use do proximal algorithms have? As it turns out the proximal algorithm is a principled method for online learning. Unfortunately, the links between online learning algorithms and proximal algorithms are not well described in the literature. The mathematical, optimization work on the proximal algorithm discussed in the previous section (e.g., see especially this canonical paper [94]) was developed independently of the algorithmic, machine learning work on online learning. Here I show the formal links between the two.

The online learning scenario has several properties: 1) the model sees one data point per iteration, 2) a single pass through the data set is performed, and 3) the model can only update each iteration T, using existing parameters, θ^T, and the data point (x^T, y^T). A common way of describing the desired update in the online scenario is as follows (e.g., see

[90]):

$$\theta^{T+1} = \arg\min_{\theta} \sum_{t=0}^{T} \gamma^t \mathcal{L}(\theta, x^t, y^t), \tag{3.5}$$

where γ is a weighting term. The idea is that each iteration we want to set our parameters equal to those that minimize the loss across the current data (x^T, y^T) and *all previously observed data*. The assumption is that new data or data in a test set will be drawn from the same or similar distribution as the old data. If we assume this distribution is stationary, we can set γ terms equal to one another. If it is non-stationary we may, e.g., have a decaying γ such that more recent data is given more weight.

The issue is that we cannot explicitly store previously observed data in the online scenario. Thus, it is common to assume all information about previously observed data is stored in the current parameters θ^T such that these parameters minimize the loss over all data from 0 to $T-1$ (e.g., see [90, 23]). To keep this loss low over previously observed data we therefore want to reduce the change in the current parameters θ^T. Let $d(\theta, \theta^T)$ be a distance measure between θ and θ^T such that $d(\theta, \theta^T) = 0$ when $\theta = \theta^T$. We thus have

$$\begin{aligned}
\theta^{T+1} &= \arg\min_{\theta} \sum_{t=0}^{T} \gamma^t \mathcal{L}(\theta, x^t, y^t) \\
&= \arg\min_{\theta} \gamma^T \mathcal{L}(\theta, x^T, y^T) + \sum_{t=0}^{T-1} \gamma^t \mathcal{L}(\theta, x^t, y^t) \\
&\approx \arg\min_{\theta} \mathcal{L}(\theta, x^T, y^T) + \frac{1}{\gamma^T} d(\theta, \theta^T),
\end{aligned} \tag{3.6}$$

which we can see has the exact form of a proximal algorithm. Interestingly, previous work has shown that if we use a KL-divergence distance measure under the assumption model parameters are interpreted as the mean of a Gaussian [24], we get back:

$$\theta^{T+1} = \arg\min_{\theta} \mathcal{L}(\theta, x^T, y^T) + \frac{1}{2\gamma^T} \|\theta - \theta^T\|^2, \tag{3.7}$$

which we can see is a standard proximal algorithm update. This update assumes γ terms decay and thus the data distribution is non-stationary.

Amazingly, the recursive least squares algorithm, which is the optimal algorithm for online learning of standard linear regression models, has been shown to be equivalent to a proximal algorithm similar to the one above [82], and similar algorithms have achieved state of the art on online binary classification [24]. This suggests the proximal algorithm approach to online learning is not just principled, but actually yields high performing algorithms in practice.

Importantly, these mathematical links suggest that we should see that EBAs, which are used to train ANNs, have advantages over BP/SGD in online scenarios, since these algorithms are implicitly performing a proximal update. This is precisely what we observe with PC. In our paper covered in the next chapter we observed that PC algorithms trained noticeably faster than BP/SGD in online learning scenarios on image classification and auto-encoder train tasks. Another research group, simultaneously and independently of us found similar improvements in small mini-batch scenarios (see [121]). At the time we did not have a formal explanation for why this was the case, but now we do. Further, another energy-based model that we test, called the sparsed quantized Hopfield network (SQHN), also performed better than similar architectures trained with SGD. Both PC and SQHN models trained quicker than SGD while converging to similar or better losses in online scenarios.

3.3 Sparse, Quantized EBAs as Online-Continual Learner

The last sections showed that EBA algorithms implicitly implement a kind of proximal algorithm. Each iteration this algorithm updates parameters to both reduce loss and the magnitude of weight updates. EBAs therefore implicitly provide some regularization, which reduces the amount of information the parameters lose during the update about previous

observed data: i.e., EBA updates implicitly reduce forgetting. This has been shown to be useful in online learning scenarios where data is i.i.d. distributed.

However, this regularization alone is unlikely to completely remove catastrophic forgetting in the more difficult non-i.i.d., continual learning scenarios. For example, although regularization methods combined with BP have proven to have some useful effects in easier, less biologically realistic continual learning settings in standard ANNs [49], they have not been effective alone to continual learning in more difficult scenarios (e.g., [64]). In these more difficult scenarios, some architectural or algorithmic modifications are generally needed.

Two general approaches have proven effective in the continual learning literature so far. First, there are replay methods (e.g., [20, 117, 5]). These methods use a memory unit that stores a buffer of previously observed data points or a generative model of previously observed data. Each iteration the model samples a mini-batch of data from the memory unit to combine with the new data. It then performs a standard mini-batch update. Replay, in other words, attempts to directly make continual learning more like offline i.i.d. learning by storing portions of the old data set directly.

Second, there are parameter isolation methods (e.g., [58, 140, 66, 65]). These methods allocate different parameters for different tasks and sub-tasks, such that only a subset of parameters are used and/or updated each training iteration. By only updating a small proportion of parameters, these algorithms ensure most information is retained in the network after the update.

One question for work on EBAs is whether there is any novel or unique method for replay or parameter isolation that can naturally be used by EBAs. Replay is architecture and algorithm-neutral in that it would benefit and could be used with just about any architecture and algorithm in the continual setting. Its downside, however, is that it requires storing a memory unit that eats up a large amount of memory, and typically grows significantly with

the size of the data set and task complexity.

In our paper 'A Sparse Quantized Hopfield Network for Online-Continual Memory', covered in chapter 6, we proposed that EBAs, alternatively, provide a principled, novel method for parameter isolation that naturally fits with the mechanics of the standard EBA. In particular, we begin with the description of EBAs as essentially implementing a kind of MAP learning algorithm training a kind of graphical model. We point out that when MAP learning is used in a graphical model with discrete, categorical random variables, each variable after MAP inference takes an integer value. These integer values can be represented as sparse, one-hot vectors. From this we derived the optimal weight update rule for the online setting, which ended up being identical to the PC update, which was $e_{l+1}\hat{h}_l$, where e is the post-synaptic prediction error, except now the pre-synaptic activity, $\hat{h}_l$, is a sparse, one-hot vector and the learning rate follows a certain decay schedule.

These sparse, one-hot activations ensure that weight updates are sparse. Whereas the standard PC update alters most parameters each iteration, this update only alters a single column on each matrix each iteration. Thus, we naturally have a form of parameter isolation. Further, the fact that the one-hot values are chosen via energy minimization itself has a further regularization effect: the one-hots chosen are those that require the smallest change to the parameters, where change is measured in energy space. Additionally, we proposed using methods for generating new parameters via neuron growth inspired by probabilistic models that used similar algorithms, which again is based on the idea that the network should grow a new neuron and allocate new parameters based on inference: allocate new parameters to those nodes that have high energy (low probability) given the current data. Thus, the method for parameter isolation is principled and easily derived from the EBA inference procedure.

We designed a model called the Sparse Quantized Hopfield Network (SQHN) based on these ideas, and tested it on various online-continual associative memory tasks. We found the

network performed essentially the same no matter what the ordering of the data was, learned very fast, and showed essentially no catastrophic forgetting. Details can be found in chapter 6.

3.4 Summary

Figure 3.1 is a summary of how each work makes progress on each problem: (left) the local credit assignment problem, (middle) the online learning problem, and (right) continual, (and online continual) learning problem. These will be discussed in further detail in subsequent chapters.

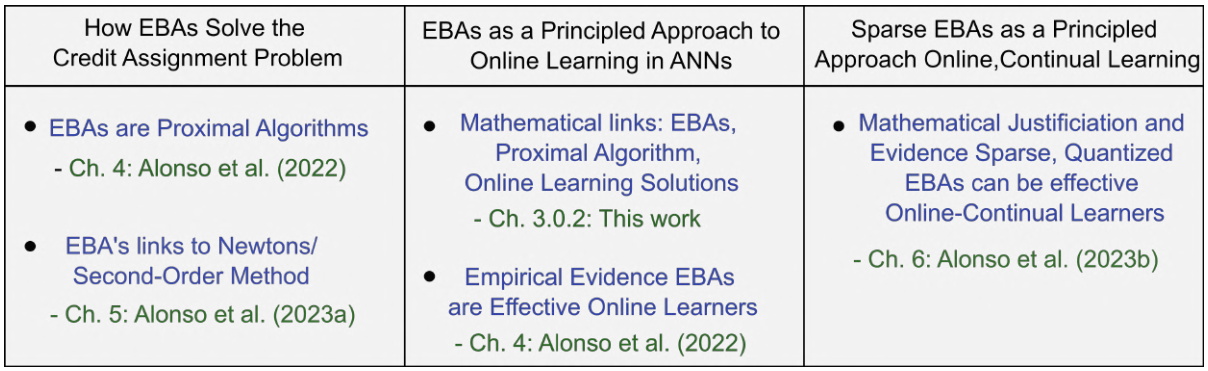

Figure 3.1: Summary of how the present EBA work contributes to improving ANNs.

Chapter 4

A Theoretical Framework for Inference Learning

4.1 Introduction

Backpropagation (BP) [107], the most successful and ubiquitously used learning algorithm in deep learning, has long been criticized as being incompatible with known neurobiology [25, 63]. For example, unlike the highly interconnected, recurrent circuits in the brain, BP performs credit assignment through a separate feedback stream that does not alter feed forward (FF) signals [25, 63]. Additionally, standard implementations of BP are largely incompatible with energy efficient, neuromorphic hardware, due, in part, to BP's non-local gradient computations. These concerns have motivated neuroscientists and engineers to search for alternatives to BP that use more biologically compatible, local learning rules.[1]

One alternative to BP is the inference learning algorithm (IL) [100, 132, 120]. IL is the

[1]This chapter is a slightly modified and condensed version of the paper titled "A theoretical framework for inference learning" published in Advances in Neural Information Processing Systems, 35, 37335-37348. The inclusion of this paper is in compliance with the permission policy of the original publisher.

standard learning algorithm used to train predictive coding (PC) models of biological neural circuits (e.g. [100, 31, 122, 11, 2, 18]). Within neuroscience, PC models have grown in popularity, where some even propose PC may be a canonical biological circuit [34, 33, 11]. IL works by first minimizing an energy function, known as free energy, w.r.t. neuron activities using the recurrent computations involved in predictive coding. At convergence, IL updates weights to further minimize energy. Unlike standard BP, IL uses local learning rules and recurrent circuits instead of a separate feedback stream to perform credit assignment. In addition to evidence the brain does predictive coding (and thus may do IL), recent empirical work has found evidence the brain may learn by first inferring target neural activities through recurrent processing then updating synapses to consolidate those neural activities, similar to IL [121]. IL has also been used to train deep networks for machine learning tasks. IL, for example, achieved comparable classification and recall accuracy to BP trained networks on supervised and auto-associative tasks (e.g, [132, 9, 111, 110], see also below).

Despite IL's wide use in biological models and success on machine learning tasks, a rigorous formal understanding of IL's optimization properties and their differences from BP is lacking. Providing such a characterization could yield several useful contributions: 1) A basis for a novel theory of how optimization and credit assignment may work in the brain, 2) a formal basis for improving, developing, and applying IL to machine learning tasks and neuromorphic hardware.

In this chapter, we develop a novel theoretical framework for IL that describes its optimization properties and their similarities and differences from BP. More specifically, 1) we expand beyond current literature by deriving a general form of the IL algorithm, which we call Generalized IL (G-IL). 2) We demonstrate our main result, which is that G-IL closely approximates an optimization method known as *implicit stochastic gradient descent* (implicit SGD), where the approximation is closest in the case of mini-batch size 1. Implicit SGD is distinct from explicit SGD (the standard form of SGD used in machine learning and the

sort implemented by BP). 3) We provide theoretical guarantees concerning IL's stability and a connection to Gauss-Newton optimization. 4) We identify a learning rule and variable settings needed in order for IL to equal implicit SGD and use this result to develop a novel implementation of IL called IL-prox. 5) Finally, we present extensive simulation results that support our theoretical findings, and for the first time show certain performance advantages of IL over BP, such as better stability across learning rates and improved convergence speed with biologically realistic online learning tasks. These results collectively suggest that, not only does IL better fit biological constraints, but it is also mathematically justified and has performance advantages over BP in more biologically realistic training scenarios.

4.2 Related Works

Inference Learning and Backpropagation Whittington and Bogacz [132] proved that parameter updates performed by the IL algorithm approach those of BP as the global loss approaches zero and optimized activities approach FF activities. This proof also implies, however, that non-zero IL updates essentially never equal those of BP, and suggests IL poorly approximates BP early in training when the loss is large and optimized activities deviate significantly from FF activities. This leaves open the question of whether there is a better description of IL's optimization strategy and the question of how IL is able to minimize the loss in a stable manner early in training. Other work has shown IL is formally related to variational expectation maximization [76, 68], which is a learning algorithm with convergence guarantees [81]. This analysis is a step forward though leaves open the question of how IL relates to BP and the broader framework of gradient-based methods that are the backbone of deep learning. The focus of the current work is to develop insights into these open questions. More recently, several works have altered IL to more closely approximate SGD and BP [79, 120]. These variants significantly change the original IL algorithm, and

some of the alterations seem hard to reconcile with neurobiology [79, 120]. Alternatively, instead of altering IL to better fit SGD and BP, we show the standard implementation of IL already closely approximates a working optimization method known as implicit SGD, and we identify the variable settings under which the standard implementation of IL better approximates implicit SGD.

Similar Algorithms Algorithms similar to IL have been proposed. For example, the alternating minimization algorithm [22], which was developed independently of the predictive coding and IL framework from neuroscience, updates weights to minimize local prediction errors similar to IL (see equation 5.1 below). However, [22] do not connect their algorithm to implicit SGD. A more recent variant of this algorithm by Qiao et al. [97] show some formal links to proximal operators, an optimization process related to implicit SGD [94] (see appendix A.2.1). However, they do not interpret the weight updates as performing the proximal update or implicit SGD as we do here with IL. Proximal operators have also recently been incorporated into learning algorithms for deep networks. Frerix et al. [30] developed a BP, proximal algorithm hybrid. Lau et al. [55] developed an algorithm that combined block coordinate descent with the proximal operator. Other works use proximal operators to reduce noise or perform other tasks, then BP is used to to update at convergence [73, 136]. All of these algorithms, however, are distinct from IL, as they use BP or other non-local gradient information to compute weight updates or local targets.

Target Propagation Finally, IL has some relation to target propagation algorithms, in which local target activities are computed and used to compute local error gradients to update weights. Target propagation [14] and difference target propagation [57] utilize approximate Gauss-Newton updates on activities to create local targets [74, 15], while other algorithms compute targets using gradient updates on activities [56, 93, 10, 54]. These algorithms have important differences from IL, e.g., they use different learning rules than IL, which we discuss further below. In sum, previous works have not developed the same mathe-

matical interpretation of IL that we do here and have not developed the same mathematical descriptions of the differences between IL and BP. Further, none of the works mentioned above produced the same empirical findings we do in our simulations below.

4.3 Background

4.3.1 Notation

Term	Description
W_n	Weight Matrix, pre-synaptic layer n
h_n	Feedforward Activity layer n, $h_n = W_{n-1} f(h_{n-1})$
$\hat{h}_n$	Optimized/Target Activity layer n
Δh_n	Optimized Activity Change layer n, $\Delta h_n = \hat{h}_n - h_n$
p_n	Local Prediction layer n, $p_n = W_{n-1} f(\hat{h}_{n-1})$
e_n	Local Error, layer n, either $e_n = \hat{h}_n - h_n$ or $e_n = \hat{h}_n - p_n$

Table 4.1: Notation

Notation describing a multi-layered feed-forward (FF) network (MLP) is summarized in table 5.1. To simplify notation, we assume that the bias is stored in an extra column on each weight matrix W_n and we assume a 1 is concatenated to the end of the pre-synaptic activity vector (h_n or $\hat{h}_n$ depending). All the following results should hold with and without biases.

4.3.2 Generalized Backpropagation

Consider a set of parameters $\theta^{(b)}$ at training iteration b with global loss function L. An explicit SGD iteration subtracts the gradient of the loss from the parameters: $\theta^{(b+1)} = \theta^{(b)} - \alpha \frac{\partial L}{\partial \theta^{(b)}}$, with step size α. BP [108] is the common algorithm for implementing SGD in deep networks. Here, to allow for easier comparison to IL, we consider an alternative implementation of SGD in deep networks that involves computing local target activities at

hidden and output layers, similar to [56], [10], [93]. Under this regime, each weight matrix W_n is updated using the gradient of local loss $l_{n+1} = \frac{1}{2}\|\hat{h}_{n+1} - h_{n+1}\|^2 = \frac{1}{2}\|e_{n+1}\|^2$. Here $\hat{h}_{n+1}$ is the local target, while h_{n+1} is the FF activity. The target at the output layer is computed $\hat{h}_N = h_N - \frac{\partial L}{\partial h_N}$. At hidden layer n, the local target is computed

$$\hat{h}_n = h_n - \frac{\partial l_{n+1}}{\partial h_n} = h_n - \frac{\partial L}{\partial h_n}. \tag{4.1}$$

It can be shown $h_n - \frac{\partial l_{n+1}}{\partial h_n} = h_n - \frac{\partial L}{\partial h_n}$ by first computing $\hat{h}_{N-1}$:

$$\begin{aligned} \hat{h}_{N-1} &= h_{N-1} - \frac{\partial l_N}{\partial h_{N-1}} = h_{N-1} - \frac{\partial l_N}{\partial h_N}\frac{\partial h_N}{\partial h_{N-1}} = h_{N-1} + e_N\frac{\partial h_N}{\partial h_{N-1}} \\ &= h_{N-1} + (h_N - \frac{\partial L}{\partial h_N} - h_N)\frac{\partial h_N}{\partial h_{N-1}} = h_{N-1} - \frac{\partial L}{\partial h_{N-1}} \end{aligned} \tag{4.2}$$

This same process can be applied recursively to show the remaining $\hat{h}_n$ are also global loss gradient steps over h_n. See appendix of [10] for similar proof.

Updates to weights W_n also use the gradient of the local loss such that $\Delta W_n = -\alpha\frac{\partial l_{n+1}}{\partial W_n}$. If we use the result from the above derivation that $-\frac{\partial l_n}{\partial h_n} = -\frac{\partial L}{\partial h_n}$, we can see this update is equivalent to a global loss gradient step over the weights:

$$\Delta W_n = -\alpha\frac{\partial l_{n+1}}{\partial h_{n+1}}\frac{\partial h_{n+1}}{\partial W_n} = -\alpha\frac{\partial L}{\partial h_{n+1}}\frac{\partial h_{n+1}}{\partial W_n} = -\alpha\frac{\partial L}{\partial W_n} = \alpha e_{n+1} f(h_n)^T. \tag{4.3}$$

This local target formulation of SGD is similar to other target propagation (TP) [14, 57] algorithms, which use essentially the same learning rule but compute targets slightly differently. For example, a well-known variant of TP is difference target propagation (DTP) [57], which computes local targets by approximating a Gauss-Newton update on activities rather than gradient updates [74, 15]: $\hat{h}_n \approx h_n - J^+_{N,n}e_N$, where $J^+_{N,n}$ is the pseudo-inverse of the Jacobian of the forward network from layer n to N and $e_N = y - h_N$. Recent attempts

to improve DTP generally do so by making DTP updates more similar to SGD updates (e.g. [74, 28]). Additionally, variants of BP algorithms, like random feedback alignment [62] and sign symmetric feedback alignment [134], approximate SGD. Since all of these methods approximate the (explicit) SGD update over weights and use similar learning rules, we consolidate them all under the heading of generalized BP (G-BP). The target-based formulation of G-BP is shown in algorithm 3.

4.3.3 Generalized Inference Learning

We now present a general description of the IL algorithm, which we call Generalized Inference Learning (G-IL). A summary of the algorithm is presented in algorithm 4. At each layer n of an MLP, G-IL stores two variables: $\hat{h}_n$ and p_n. At initialization these variables are set to feed-forward activity values: $\hat{h}_n = p_n = h_n$. Then $\hat{h}_n$ are altered over time to minimize the energy function F:

$$F = L(y, \hat{h}_N) + \sum_{n=1}^{N} \gamma_n \frac{1}{2} \|\hat{h}_n - p_n\|^2 + \sum_{n=1}^{N-1} \gamma_n^{decay} \frac{1}{2} \|\hat{h}_n\|^2, \tag{4.4}$$

where L is the global loss, $\frac{1}{2}\|\hat{h}_n - p_n\|^2$ are local losses, $\gamma_n^{decay}\frac{1}{2}\|\hat{h}_n\|^2$ is an optional regularization term, and γ are positive scalar weighting terms. The prediction p_n is computed as $p_n = W_{n-1} f(\hat{h}_{n-1})$. (Note p_n may also be computed as $p_n = f(W_{n-1}\hat{h}_{n-1})$. Our theoretical results use the first formulation, but we find little difference in performance between the two in practice. See simulations below.) G-IL first minimizes F w.r.t. to activities $\hat{h}$ (inference phase). While optimizing $\hat{h}$, predictions also change since $p_{n+1} = W_n f(\hat{h}_n)$. After $\arg\min_{\hat{h}} F$ is estimated, F is again minimized, now w.r.t. the weight, i.e. $\arg\min_W F$. Typical IL algorithms use SGD to perform the inference phase with either the LMS rule (below) or Adam optimizers to update weights (e.g. [100, 132]). G-IL, on the other hand, is general w.r.t. the optimization processes that approximate $\arg\min_W F$ and $\arg\min_{\hat{h}} F$. We consider two

weight updates in our analysis and simulations below. First, is a local error gradient update, which is equivalent to the least-mean squares (LMS) rule:

$$\arg\min_{W_n} F \approx W_n + \Delta W_n = W_n - \alpha \frac{\partial l_{n+1}}{\partial W_n} = W_n + \alpha e_{n+1} f(\hat{h}_n)^T, \tag{4.5}$$

where $e_{n+1} = \hat{h}_{n+1} - p_{n+1}$ and local loss $l_{n+1} = \frac{1}{2}\|e_{n+1}\|^2$. The LMS rule does not solve $\arg\min_W F$ but only approximates it. The next rule, which is equivalent to the normalized least mean squared rule (NLMS) [84] is

$$\arg\min_{W_n} F = W_n + \Delta W_n = W_n + \|f(\hat{h}_n)\|^{-2} e_{n+1} f(\hat{h}_n)^T. \tag{4.6}$$

The NLMS rule performs a gradient update but with an inverse squared ℓ_2 norm of the pre-synaptic activities as its step size instead of a hyper-parameter. The NLMS rule is the minimum-norm solution to $argmin_W F$ in the case where we use mini-batch size 1 (see proposition A.1). The LMS rule closely approximates the NLMS update in the sense the two are proportional. Comparing these rules to the update rule for G-BP (equation 4.3) we see the main difference is the pre-synaptic term used in the rule: BP uses FF activity h_n, while IL uses optimized/target activity $\hat{h}_n$. This difference leads to significantly different stability properties between the algorithms, as we note below.

Algorithm 1: Generalized BP

begin

`// Feedforward Pass`

$h_0 \leftarrow x^{(b)}$

for $n = 0$ **to** $N - 1$ **do**

$h_{n+1} \leftarrow W_n f(h_n)$

end

$\hat{h}_N \leftarrow y^{(b)}$

`// Compute Local Targets`

for $n = 1$ **to** N **do**

$\hat{h}_n \approx h_n - \frac{\partial L}{\partial h_n}$

end

`// Update Weight Matrices`

Eqn. 4.3

end

Algorithm 2: Generalized IL

begin

`// Feedforward Pass`

$\hat{h}_0 \leftarrow x^{(b)}$

for $n = 0$ **to** $N - 1$ **do**

$p_{n+1}, \hat{h}_{n+1} \leftarrow W_n f(h_n)$

end

`// Compute Local Targets`

$(\hat{h}_1, ...\hat{h}_N) \approx \arg\min_{\hat{h}} F$

`// Update Weight Matrices`

$(W_0, ...W_{N-1}) \approx \arg\min_W F$, Eqn. 4,5

end

4.4 Theoretical Results

4.4.1 Implicit Gradient Descent

Let the set of MLP weight parameters $\theta^{(b)} = [W_0^{(b)}, ...W_{N-1}^{(b)}]$, where the input data is $x^{(b)}$, the output target is $y^{(b)}$, and b is the current training iteration. The explicit SGD update uses the loss gradient produced by the current parameters:

$$\theta^{(b+1)} = \theta^{(b)} - \alpha \frac{\partial L(\theta^{(b)}, x^{(b)}, y^{(b)})}{\partial \theta^b}, \tag{4.7}$$

where α is step size and L is the loss measure. This update is explicit because the gradient can be readily computed given $\theta^{(b)}, x^{(b)}, y^{(b)}$. The *implicit* SGD update uses the loss gradient

of the parameters at the next training iteration:

$$\theta^{(b+1)} = \theta^{(b)} - \alpha \frac{\partial L(\theta^{(b+1)}, x^{(b)}, y^{(b)})}{\partial \theta^{(b+1)}}. \tag{4.8}$$

This is an implicit update because $\theta^{(b+1)}$ shows up on both sides of the equation. Unlike explicit SGD, $\theta^{(b+1)}$ cannot be readily computed using available quantities. However, the implicit gradient update is equivalent to the solution of the following optimization problem (see appendix equation A.5):

$$\theta^{(b+1)} = \arg\min_{\theta} \left(L(\theta, x^{(b)}, y^{(b)}) + \frac{1}{2\alpha} \|\theta - \theta^{(b)}\|^2 \right). \tag{4.9}$$

This update is equivalent to the proximal operator/update [94]. This proximal update changes parameters $\theta^{(b)}$ in a way that minimizes the loss L and the magnitude of the update, which helps keep $\theta^{(b+1)}$ in the proximity of $\theta^{(b)}$. (For more details on the proximal update and its relation to implicit SGD see appendix A.2.1).

4.4.2 Main Results

We now present our main result, which is to show G-IL is equivalent to implicit SGD in certain limits that are approximated well in practice. We focus on the case where single data-points (mini-batch size 1) are presented each training iteration. This scenario is more biologically realistic than mini-batching and is the case where IL best approximates implicit SGD. We discuss the case of mini-batching in appendix A.2.5. We first define a kind of IL algorithm that minimizes the proximal loss (equation 4.9) w.r.t. neuron activities and show G-IL approaches this algorithm in certain limits.

Definition 4.1. *Proximal Inference Learning (IL-prox). An algorithm identical to G-IL (algorithm 4), except activities are optimized according to* $\arg\min_{\hat{h}} prox = \arg\min_{\hat{h}}(L(\theta^*) +$

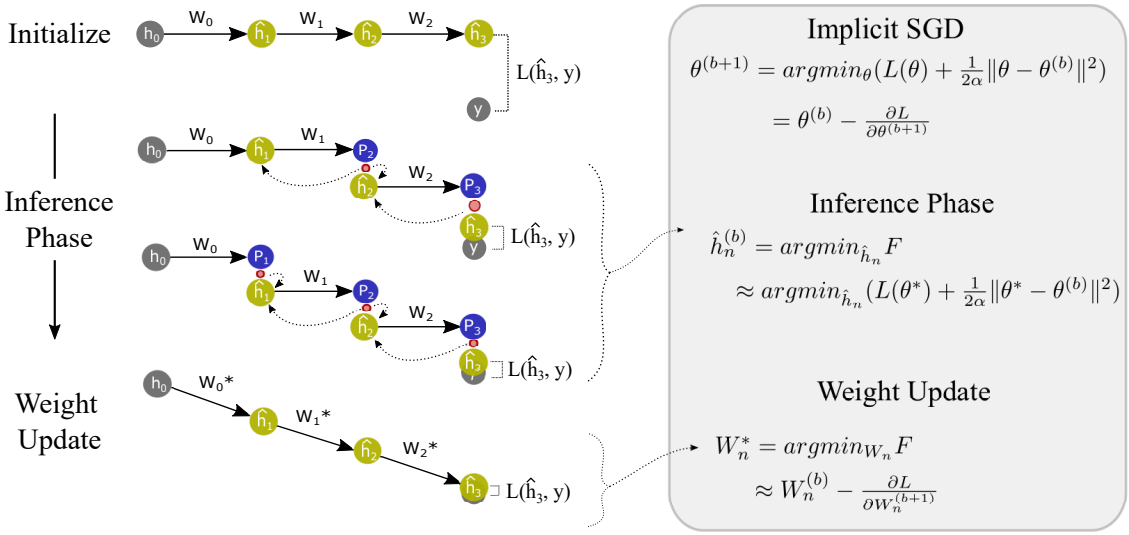

Figure 4.1: A depiction of IL. Target activities $\hat{h}$ are initialized to FF activities h then updated to minimize energy F during the inference phase. Afterward, weights are updated to further minimize F. Weight updates 'align' with $\hat{h}$ activities so $\hat{h}$ become the new FF activities given the same data point. $\hat{h}$ are activities that will both improve the loss L and minimize $\|\Delta\theta\|^2$. This process is equivalent to minimizing the proximal objective and approximates implicit SGD.

$\frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2)$ *and the NLMS rule is used to update weights.*

Here b is the current training iteration and θ^* are the parameters updated with the NLMS rule. IL-prox is the same as G-IL except during the inference phase, it minimizes the proximal loss w.r.t. activities $\hat{h}$ instead of the energy F. Intuitively, minimizing the proximal loss w.r.t. activities will result in $\hat{h}$ that yields a small loss after weights are updated and small weight update norms. Weight update norm $\frac{1}{2}\|\theta^* - \theta^{(b)}\|^2$ is known during the inference phase because the NLMS learning rule is used and known explicitly:

$$\frac{1}{2}\|\theta^* - \theta^{(b)}\|^2 = \frac{1}{2}\sum_n^N \|\Delta W_n\|^2 = \frac{1}{2}\sum_n^N \|\alpha_n e_{n+1} f(\hat{h}_n)^T\|^2,$$

where $\alpha_n = \|f(\hat{h}_n)\|^{-2}$. Thus, it can be minimized w.r.t. $\hat{h}_n$. As we explain in the appendix (see lemma A.1 and proposition A.1), the loss $L(\theta^*)$ is also explicitly known during the inference phase and can be optimized w.r.t. $\hat{h}_n$.

Now, let α be a 'global' learning rate hyper-parameter, and α_n be layer-wise learning rates used in the actual weight update ΔW_n.

Theorem 4.1. *Let* $\alpha_n = \|f(\hat{h}_n)\|^{-2}$ *and assume mini-batch size 1. In the limit where* $\gamma_n^{decay} \to \|e_{n+1}\|^2(1 - \frac{2}{\alpha_n^6})$, $\gamma_N \to \frac{\alpha_{N-1}}{\alpha}$, *and* $\gamma_n \to \alpha_{n-1}^{-1}$ *for all* $n < N$, *it is the case that* $\arg\min_{\hat{h}} F = \arg\min_{\hat{h}} L(\theta^*) + \frac{1}{2\alpha}\|\Delta\theta\|^2$. *Hence, in these limits G-IL is equivalent to IL-prox.*

The proof can be found in supplementary materials theorem A.1. This theorem states that when using the NLMS rule, G-IL is increasingly similar to IL-prox, as the γ weighting terms in the free energy (equation 5.1) approach the scalar values specified in the above theorem. In these limits, $\arg\min_{\hat{h}} prox = \arg\min_{\hat{h}} F$ and G-IL is equivalent to IL-prox. The intuitive explanation of this theorem goes as follows: minimizing the proximal loss (equation 4.9) requires minimizing L w.r.t. θ^* while also minimizing the update norm $\frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2$. The inference phase of IL minimizes L w.r.t. θ^* by initializing output layer activity $\hat{h}_N$ to FF output h_N then shifting it toward global target y. Upon a weight update this will yield a θ^* that produces a smaller loss L. See figure 4.4.1 for visualization. The magnitude of a weight update $\|e_{n+1}f(\hat{h}_n)^T\|^2$ intuitively depends on the magnitude of local prediction error e_{n+1}. F is a sum of the magnitudes of errors (see equation 5.1), so minimizing F minimizes the magnitudes of errors and consequently weight update norms $\frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2$. Importantly, the γ settings under which theorem 4.1 holds can be computed exactly in practice and are of reasonable magnitude that is easily approximated in practice (e.g., they do not approach ∞ or 0 and typically approach positive scalars). This is opposed to the limits under which IL approximates BP/explicit SGD, which is the limit where $\hat{h}_n \to h_n$ and thus $\Delta W_n \to 0$ [132], a condition that does not generally occur in practice.

Theorem 4.2. *Let* $\theta^{(b)}$ *be a set of MLP parameters at training iteration b. Let* $\theta_{prox}^{(b+1)} = argmin_\theta L(\theta) + \frac{1}{2\alpha}\|\theta - \theta^{(b)}\|^2$. *Let* $\theta_{IL-prox}^{(b+1)}$ *be the parameters updated by IL-prox (see def. 4.1) and* $\theta_{IL}^{(b+1)}$ *the parameters updated by G-IL under* γ *values in theorem 4.1. Assume mini-batch size 1. Under this assumption, it is the case* $\theta_{prox}^{(b+1)} = \theta_{IL-prox}^{(b+1)} = \theta_{IL}^{(b+1)}$.

This theorem states that the IL-prox update is equivalent to the proximal update (equation 4.9), and thus implicit SGD. It follows that, under the γ settings in theorem 4.1 the G-IL update is also equivalent to the proximal update/implicit gradient update.

Further theoretical results are developed in the appendix. A novel closed form description of IL targets is developed in A.3.5. The closed form description shows local targets approximate a regularized Gauss-Newton update on FF activities (see section A.3). We call this closed form approximation IL-GN. We study the stability of IL-GN when using the LMS learning rule, and compare to an analogous BP network, which uses Gauss-Newton updates (BP-GN). We show that IL-GN weight updates push output layer activities down their loss gradients *for any positive learning rate*, i.e., for any positive learning rate $\hat{h}_N^{(b)} - \hat{h}_N^{(b+1)} = j\frac{\partial L}{\partial \hat{h}_N^{(b)}}$ where j is a positive scalar (theorem A.3). We show BP-GN only has this property for a small range of learning rates (theorem A.4). We develop a partial explanation of why IL is more stable than BP that is based on differences in their learning rule (for details see section A.5). These results suggest that the differences in the way IL and BP compute and use local targets in their weight updates can explain much of their differences in stability across learning rate. The IL learning rule and target computations are advantageous over that of G-BP in this sense.

4.5 Experiments

In this section, we compare the performance of BP based algorithms to IL algorithms. BP algorithms compute and use explicit gradients to update weights, while IL algorithms compute and use approximate implicit gradients to perform updates. We test BP models that either perform simple gradient descent (**BP-SGD**) or use Adam optimizers (**BP-Adam**). **IL-SGD** is a simple implementation of IL that is similar to implementations of [100, 132]. It uses the LMS rule (local error gradient update) to update weights and optimizes $\hat{h}$ using

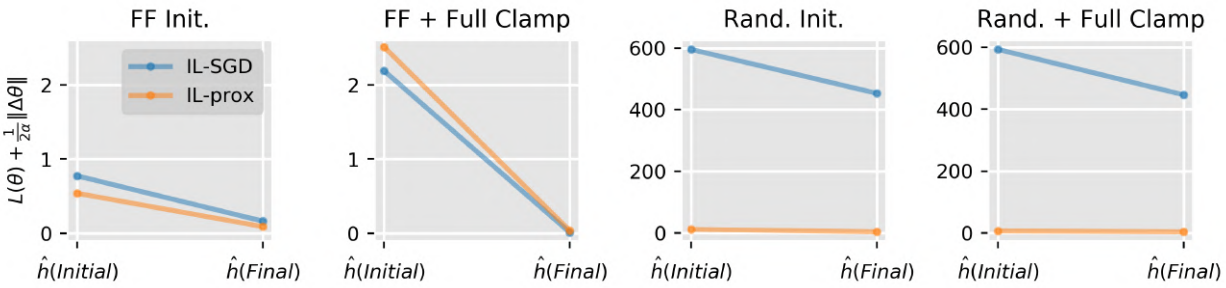

Figure 4.2: We measure the proximal objective during the inference phase, where F is minimized w.r.t. activities $\hat{h}$, in a two hidden layer MLP with ReLU activations trained on MNIST. We test under four different initializations of $\hat{h}$. First, all $\hat{h}$ are initialized to h (FF Init.). Second, hidden layer $\hat{h}$ are initialized to h, while output layer $\hat{h}_N = y$ (FF + Full Clamp). For both of these initializations, the proximal loss is reduced to near zero for both the IL-prox and IL-SGD variants of IL (defined below). We next initialized $\hat{h}$ randomly (Rand Init. and Rand + Full Clamp). Both networks reduce the proximal loss, but nowhere near the global minima, which may help explain why the $\hat{h}$ needs to be initialized to FF activities when F is minimized with SGD. These results support theorem 4.1, which states that minimizing F w.r.t. $\hat{h}$ also minimizes the proximal objective.

SGD to near convergence (25 update steps). **IL-Prox** is our novel algorithm that is based on definition 4.1. IL-prox uses the NLMS rule to update weights. Following theorem 4.1, the learning rate is only used to determine how much the output layer activities $\hat{h}_N$ are pushed toward the target y. As $\alpha \to \infty$, $\hat{h}_N \to y$ and as $\alpha \to 0$, $\hat{h}_N \to h_N$. IL-Prox optimizes $\hat{h}$ using SGD to near convergence (25 update steps). **IL-Prox Fast** is the same as the IL-Prox algorithm except it truncates the optimization of $\hat{h}$ to only 12 iterations. **IL-Adam** computes the weight gradients using IL-SGD, then uses Adam optimizers to update weights. **IL-prox Adam** computes the normalized weight gradient using IL-prox and uses Adam optimizers to update weights. All simulations average results over at least 5 seeds of each model type. All models use ReLU activations at hidden layers. Softmax is used at output layer for classification, while sigmoid is used on the autoencoder task.

Mini-batch Training Previous work has shown that IL algorithms can achieve the same accuracy as BP on small scale data sets (e.g., MNIST and Fashion-MNIST), in the case where Adam optimizers and medium sized mini-batches (e.g, 64) are used ([132, 9]. Here we train our BP and IL algorithms on CIFAR-10 with mini-batch size 64. Results are

Reconstruction BCE (mean±std.)					
Data	BP-SGD	BP-Adam	IL-SGD	IL-prox	IL-prox fast
F-MNIST	.303	**.282**	.306	**.286**	**.286**
CIFAR-10	.631	.622	.652	**.611**	**.611**

Table 4.2: Best test reconstruction loss after one epoch of training, mini-batch size 1. BCE was averaged across pixels and data-points. Top two scores highlighted in bold. All standard deviations $\leq .002$.

summarized in table 4.3. Consistent with previous results, IL algorithms, including our novel IL-prox, achieve near equal accuracy as BP-SGD and BP-Adam when using Adam optimizers. However, unlike previous studies, we also test IL-SGD and IL-prox algorithms without Adam, and find they do not perform as well as BP algorithms in this scenario. IL algorithms without Adam tend to converge more quickly but to shallower local minima. Adam, with its momentum and adaptive learning rate, may help prevent IL algorithms from getting caught in shallow local minima (see discussion).

Online Learning We find IL algorithms tend to converge significantly quicker and to more similar loss/accuracies as BP in a more biologically realistic, online learning scenario, where a single data-point is presented each training iteration and each data point is seen only once. This is especially apparent with CIFAR-10, shown in tables 4.3 and 4.2 and figure 4.3. IL-SGD and IL-prox algorithms perform similar to and even slightly better than BP-Adam, despite not using

CIFAR-10 Test Accuracy (mean±std.)		
Model	m-batch size=1	m-batch size=64
BP-SGD	36.834(±.478)	**57.044** (±.144)
IL-SGD	**43.894** (±.371)	46.924(±.248)
IL-prox	42.060(±.412)	49.844(±.359)
IL-prox Fast	**42.984** (±.530)	46.094(±.228)
BP-Adam	42.102(±.770)	**56.466** (±.228)
IL-Adam	40.724(±.160)	54.066(±.190)
IL-prox + Adam	42.802(±.390)	55.572(±.172)

Table 4.3: Best test accuracy on CIFAR-10 supervised task with mini-batch size 1 after 50,000 iterations (1 epoch) and with mini-batch size 64 with 77000 iterations.

momentum or stored, parameter-

wise adaptive learning rates like Adam. As far as we know, this is a novel result. Less significant speedups were also seen on MNIST and Fashion-MNIST classification tasks (see figure A.2). On autoencoder tasks, IL-prox algorithms, in particular, decreased loss significantly quicker than BP-SGD and even slightly quicker than BP-Adam (figure 4.3).

Learning Under Data Constraints We also find that IL algorithms can converge to better accuracy than BP in highly data constrained scenarios. We trained IL algorithms using on only 10, 100, or 500 data points from each category on F-MNIST and CIFAR-10, with mini-batch size 100. IL algorithms achieved better accuracies than BP-SGD in the 10 datapoint scenario on F-MNIST and CIFAR, and better accuracy on the 100 data point scenario on CIFAR, while performances evened out in the 500 data point scenario (see appendix table A.4 for details). IL-prox also tended to converge significantly quicker than other algorithms on Cifar-10 (figure 4.4).

Stability Test Explicit SGD is well-known to be highly sensitive to learning rate. Implicit SGD, on the other hand, is highly insensitive to learning rate and is unconditionally stable, i.e., it is able to discount the loss in a stable manner for nearly any positive learning rate [127]. We compare performance of BP-SGD, IL-SGD, and IL-prox on MNIST (table A.1) and CIFAR-10 (table 4.4) classification task across different learning rates. As a control, we train a hybrid of BP and IL-prox (BP-prox). BP-prox, like IL-prox, uses the learning rate only to

Stability Test: CIFAR-10 Test Accuracy						
Model	lr=.01	lr=.1	lr=1	lr=2.5	lr=10	lr=100
BP-SGD	35.48(±1.65)	–	–	–	–	–
IL-SGD	41.66(±1.65)	36.268(±.271)	–	–	–	–
BP-prox	38.39(±2.91)	23.978(±1.57)	12.89(±1.1)	12.10(±1.61)	12.21(±1.61)	–
IL-prox	34.97(±.67)	33.47(±2.56)	37.38(±1.46)	37.12(±1.81)	37.01(±1.01)	37.64(±.74)

Table 4.4: Accuracies after 50,000 training iterations on CIFAR-10, mini-batch size 1. Fully connected networks with layer sizes 3072-3x1024-10.

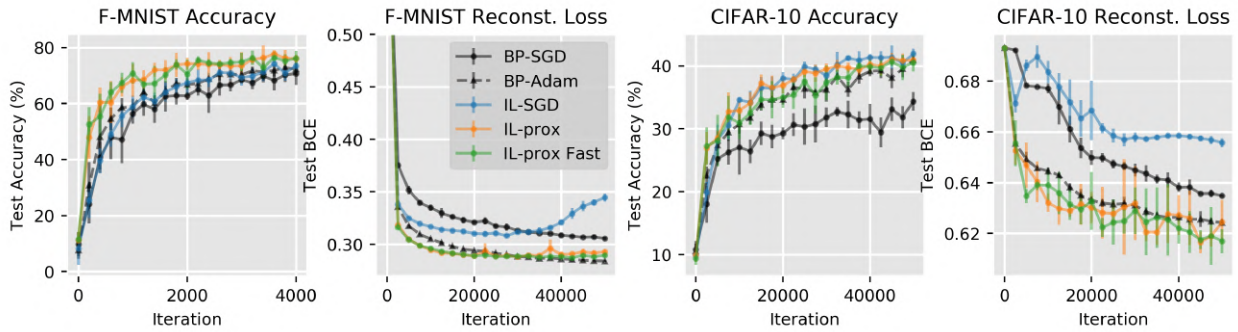

Figure 4.3: Training runs for mini-batch size 1 (online) scenario, for classification and autoencoder tasks. MLP size 784-2x500-10 and autoencoder dimension 784-256-100-256-784 are trained on F-MNIST for 50000+ iterations. MLP size 3072-3x1024-10 and autoencoder 3072-1024-500-100-500-1024-3072 are trained on CIFAR-10 for 50000+ iterations. The first 4000 training iterations shown for F-MNIST on left. Full training run are shown for other simulations in right three plots.

adjust the target at the output layer and uses the NLMS rule to update weights. Unlike IL-prox, BP-prox computes local targets using the G-BP algorithm rather than the IL algorithm. Results can be found in table A.1 and 4.4. Blank entries are those with accuracies below 12%. IL-SGD is more stable than BP-SGD, and IL-prox is more stable than BP-prox. This suggests that the way IL computes and uses targets contributes to stability. Additionally, IL-prox and BP-prox algorithms are more stable than SGD algorithms, showing that using the normalized gradient to update weights and α to adjust the output layer target (rather than scale weight updates) also contributes to stability.

4.6 Limitations

Our theoretical results show that IL algorithms closely approximate implicit SGD in the case where only a single data-point (i.e., mini-batch size 1) is presented each training iteration. This approximation is looser in the case where large mini-batches are used (see appendix A.2.5). This is no problem from a biological point of view, since the brain does not train with large mini-batches. Additionally, when Adam optimizers are used IL still performs similarly

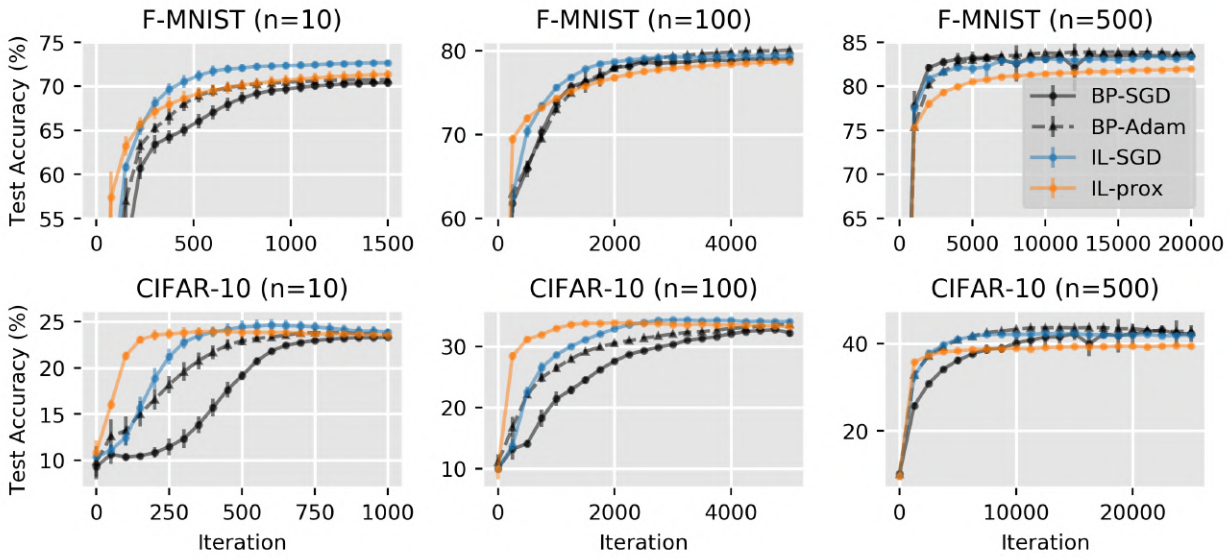

Figure 4.4: Training runs for a subset of the algorithms on the data constrained scenarios. Each model trained on n data points from each category.

to BP. However, it may still be desirable to further study the optimization properties of IL with mini-batches to improve performance on machine learning tasks where mini-batches are typically used.

4.7 Discussion

In this chapter, we showed that IL closely approximates implicit SGD, which is distinct from the explicit SGD performed by BP. This theoretical result, along with results that help to characterize how IL and BP behave differently, suggest a hypothesis, which as far was we know is novel: learning in the brain is more similar to implicit gradient descent than explicit gradient descent. This hypothesis would imply the learning characteristics of the brain behave differently from BP and explicit SGD, more generally. More empirical work will be needed to test this theory. However, recent evidence directly supports the idea the brain does something similar to IL [121]. Evidence that neural circuits perform predictive coding provides further, though more indirect, evidence for the idea.

We also found that IL often converged more quickly than BP and achieved similar test losses and accuracies in online and data constrained scenarios. However, in large mini-batch scenarios without data constraints and without the use of Adam optimizers, IL sometimes converged to worse accuracies than BP in classification tasks. With the use of Adam, IL's performance more closely matched that of BP. One possible explanation of this empirical finding may be related to the fact that IL, and implicit SGD more generally, tend to take smaller steps over parameters and a more direct path toward local minima than BP (e.g., see figures A.3 and table A.5). More direct paths to minima often lead to faster convergence. Smaller update magnitudes may also act as a kind of built-in regularization that aids in difficult, noisy learning scenarios, such as small mini-batch and data constrained scenarios. In cases where regularization is less advantageous (e.g., no data constraints and large mini-batches), IL's small update magnitudes may provide less of an advantage, and its more direct optimization paths may push parameters into shallow, nearby minima. Adam optimizers, with their momentum and adaptive learning rates, may help push parameters out of shallow minima. More work will be needed to fully characterize and explain these behavioral differences, but we believe the work done here provides a good formal basis for doing so.

These results further suggest IL may have advantages over BP in more biologically realistic machine learning tasks, such as lifelong, online learning scenarios, where streams of data are received by the neural network, a single datapoint at a time, and the model must adapt quickly to changes in the environment and minimize the magnitude of weight updates to prevent catastrophic forgetting. This possibility is especially interesting given that, unlike BP, IL's weight updates are both local in space and time and may thus be more compatible with energy efficient neuromorphic hardware, which is well suited for autonomous, embedded systems that run on batteries and receive data streams through sensors, similar to the brain.

Chapter 5

Understanding and Improving Optimization in Predictive Coding Networks

5.1 Introduction

Artificial neural networks (ANNs) were originally created to mimic biological neural circuits [71]. However, backpropagation (BP) [107], the now standard algorithm used to train ANNs, is difficult to reconcile with neurobiology [25, 63]. Researchers in computational neuroscience and neuromorphic computing have attempted to alter BP to better fit what is known about the brain [62, 61, 40, 101]. However, these altered versions of BP typically either fail to overcome the biological implausibilities of BP, such as BP's segregated feed-back signals that do not alter feed-forward activities, or perform significantly worse than standard BP. One promising alternative approach is based on the predictive coding (PC) model in neuroscience [100]. PC is a type of recurrent neural network typically trained with inference learning (IL).

IL is a variant of generalized expectation maximization [78], which works by first performing inference, where an objective known as free energy is reduced w.r.t. neuron activities via the recurrent PC circuits. Then weights are updated to further reduce free energy. Unlike BP, IL does not require a segregated feedback stream. Instead, its recurrent circuitry requires feed-forward and feedback signals to interact with each other. Further, like the brain and in contrast to BP, IL performs Hebbian-like updates that are local in space and time [132].

Previous work has found that networks trained with IL performed competitively with BP on classification and self-supervised tasks that use small and medium sized images [132, 9, 111, 110, 8, 121]. Recent work has even found some noticeable performance advantages over BP/SGD (e.g., [8, 121]), such as faster loss reduction and convergence with small mini-batches. These results suggest that IL may be useful in certain machine learning and neuromorphic computing applications, such as online learning scenarios which require small mini-batches. However, there are several challenges preventing IL from easily being applied to engineering problems: **1)** IL is much more computationally expensive than BP due primarily to IL's expensive inference phase. **2)** IL has only achieved comparable performance to BP when memory expensive optimizers like Adam are used. Without any optimizer, IL convergence is prone to fall in poor local minima [8], and the reason why this happens is not well understood. **3)** IL sometimes shows speedups in loss reduction and convergence over BP [8, 121], but the robustness of these speedups across models and tasks is unknown. In this chapter[1], we tackle these challenges through the following contributions:

1. We develop and test a non-standard implementation of IL's inference phase, called sequential inference, which propagates errors more quickly through the network allowing for a significant reduction of the computational cost of IL.

2. We show that IL weight updates are drastically smaller in magnitude in early layers,

[1]This chapter is a slightly modified and condensed version of the paper titled "Understanding and Improving Optimization in Predictive Coding Networks" which is under review for the proceedings of AAAI 2024.

which may lead the network to get caught in poor local minima near the initial parameters. We develop a custom optimizer called **M**atrix Update E**q**ualization (MQ) to address this problem. MQ requires no significant increase in memory or computation compared to IL and our simulations provide evidence that MQ prevents the convergence issues of IL at least as well as Adam.

3. We provide simulation results showing IL often reduces the loss more quickly than BP/SGD. We establish new theoretical results showing that IL is generally sensitive to second and higher-order information, which could explain this speed up.

In sum, in our simulations below, IL with sequential inference and the MQ optimizer requires no more memory than BP/SGD, requires only slightly more computation than BP/SGD, converges to similar losses as BP/SGD, and is sensitive to higher-order information, which often leads to faster convergence than BP/SGD. *As far as we know, this is the first time an energy-based learning algorithm (as defined in [113, 133]) has performed as well or better than BP/SGD on natural images in the ways described above without using memory expensive optimizers and without requiring significantly more computation than BP.* This combined with the fact that IL is more bio-plausible than BP suggests IL, with the modifications made here, may be a promising learning algorithm for neuromorphic and bio-inspired machine learning communities.

5.2 Background and Notation

5.2.1 Notation

Notation describing a multi-layered feed-forward (FF) network (MLP) is summarized in table 5.1. We assume a bias is stored in an extra column of each weight matrix W_l.

Term	Description
W_l	Weight Matrix, pre-synaptic layer l
h_l	Feedforward Activity layer l, $h_l = W_{l-1} f(h_{l-1})$
$\hat{h}_l$	Optimized/Target Activity layer l
p_l	Local Prediction layer l, $p_l = W_{l-1} f(\hat{h}_{l-1})$
e_l	Local Error, layer l, $e_l = \hat{h}_l - p_l$

Table 5.1: Notation

5.2.2 Predictive Coding and Inference Learning

Predictive coding (PC) networks are a kind of recurrent neural network typically trained with the inference learning algorithm (IL). IL can be interpreted as a variant of generalized Expectation Maximization algorithm (EM) [32, 78]. Like EM, IL proceeds in two steps: First, an energy function, known as free energy is minimized w.r.t. neuron activities. PC computations perform this minimization using gradient descent. Then weights are updated to further reduce free energy. Below, we present the energy function and equations for PC and IL weight updates.

Free Energy: The free energy, F, is defined here as follows:

$$F = \mathcal{L}(y, \hat{h}_L) + \sum_{l=1}^{L} \gamma_l \frac{1}{2} \|\hat{h}_l - W_{l-1} f(\hat{h}_{l-1})\|^2 + \sum_{l=1}^{L-1} \gamma_l^{decay} \frac{1}{2} \|f(\hat{h}_l)\|^2, \tag{5.1}$$

where $\hat{h}$ are layer activities, $\mathcal{L}$ is the global loss, $\frac{1}{2}\|f(\hat{h}_l)\|^2$ is an optional regularization term, f is a non-linearity, and γ are positive scalar weighting terms. The middle term can be interpreted as a summation over squared prediction errors, where prediction $p_l = W_{l-1} f(\hat{h}_{l-1})$ and prediction errors are $e_l = \hat{h}_l - p_l$. If we set the activities at the output layer equal to the prediction target, $y = \hat{h}_L$ and ignore the decay (set $\gamma^{decay} = 0$), as is common in practice, the energy is just the sum of prediction errors: $F = \sum_{l=1}^{L} \gamma_l \frac{1}{2} \|e_l\|^2$.

Inference: During the inference phase, which is similar to the E-step of EM [78], neuron activities $\hat{h}$ initialized to feed-forward activities h then are updated to reduce F. PC refers to

the the process that updates activities iteratively using partial gradients of local prediction errors. For example, PC circuits update $\hat{h}_l$ using gradients from e_l and e_{l+1}. Specifically, a single gradient update over a hidden layer $\hat{h}_l$ is

$$\Delta\hat{h}_l = \epsilon(-\frac{\partial F}{\partial \hat{h}_l}) = \epsilon(\gamma_{l+1} f'(\hat{h}_l) W_l^T e_{l+1} - \gamma_l e_l - \gamma_l^{decay} f'(\hat{h}_l)\hat{h}_l), \tag{5.2}$$

where ϵ is the step size. The output layer activities, $\hat{h}_L$, are either fully clamped to output targets ($\hat{h}_L = y$) or are softly clamped such that each iteration $\hat{h}_L$ is updated to reduce the loss $\mathcal{L}(\hat{h}_L, y)$ and $\gamma_L e_L$. Here we assume $\mathcal{L}(\hat{h}_L, y) = \frac{1}{2}\|y - p_L\|^2$, which yields a closed form solution that is computed at each iteration:

$$\hat{h}_L = \frac{1}{(1+\gamma_L)} y + \frac{\gamma_L}{(1+\gamma_L)} p_L. \tag{5.3}$$

The derivation can be found in appendix B.0.1. In practice, gradient updates are typically performed for 15+ iterations (e.g., [132, 9, 110]).

Learning: After the inference phase is completed, weights are updated to further reduce F. This update is typically performed by using the gradient of the local errors, e.g., W_l is updated with the gradient of $\frac{1}{2}\|e_{l+1}\|^2$:

$$\Delta W_l = -\alpha_l \frac{\partial F}{\partial W_l} = \alpha_l e_{l+1} f(\hat{h}_l)^T, \tag{5.4}$$

where $e_{l+1} = \hat{h}_{l+1} - W_l f(\hat{h}_l)$ and α_l is a layer-wise step size.

5.2.3 Inference Learning Approximates Implicit Gradient Descent

Previous works have analyzed the similarities between back-propagation (BP) and stochastic gradient descent (SGD) and IL (e.g., [132]). There, it was found that IL approaches BP/SGD

as activities $\hat{h}$ approach feed forward activity values h [132]. Although insightful, such analyses leave open the question of why IL is able to reduce the loss in a stable manner even when the $\hat{h}$ deviates significantly from h, which often occurs in practice [104]. Recently, an alternative theoretical framework for IL has been established that avoids this issue. In particular, it was shown that, in the case of a mini-batch size of one and under specific conditions on the γ terms in the energy equation and learning rates α_l, IL is equivalent to *implicit* stochastic gradient descent. These conditions on the γ and α terms are non-standard but are approximated well in standard implementations (see [8] and below). We emphasize that implicit SGD is not equivalent to standard SGD, which we call explicit SGD here. The difference between the two can stated as follows:

$$\begin{aligned} &\textbf{Explicit SGD: } \theta^{(b+1)} = \theta^{(b)} - \beta \frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}} \\ &\textbf{Implicit SGD: } \theta^{(b+1)} = \theta^{(b)} - \beta \frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}}, \end{aligned} \tag{5.5}$$

where θ are the model parameters, b is the current training iteration, and β is a 'global' learning rate acting over all parameters. While standard/explicit SGD updates parameters with the gradient of the loss w.r.t. the current parameters, implicit SGD updates parameters with the gradient of the loss w.r.t. the parameters at the next training iteration, $b+1$. This gradient cannot be explicitly computed given known values at iteration b. Hence, this update is computed implicitly. It can be shown that the implicit SGD update is equivalent to the *proximal update* [94, 127]:

$$\theta^{(b+1)} = \arg\min_{\theta} \mathcal{L}(\theta) + \frac{1}{2\beta}\|\theta - \theta^{(b)}\|^2 = \theta^{(b)} - \beta \frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}}. \tag{5.6}$$

The proximal algorithm sets the new parameters equal to the output of an optimization process that both minimizes the global loss $\mathcal{L}$ and the squared difference between the current and optimized parameters, which keeps the new parameters in the proximity of the old ones. The intuition for why IL approximates the proximal algorithm goes as follows: IL reduces F

w.r.t. to neuron activities before updating weights, which does two things: 1) $\hat{h}_L$ is updated to reduce the loss at the output layer (equation 5.3), which means that when weights are updated the loss is reduced given the same input. 2) Local errors e_l and activity $f(\hat{h}_l)$ magnitudes are minimized, which has the effect of minimizing the magnitude of the weight update since the weight updates are just outer products e_{l+1} and $f(\hat{h}_l)$. Thus, IL's inference phase yields weight updates that both minimize loss and the update norm, just like the proximal update.

5.3 Theoretical Results

5.3.1 IL with MQ and Implicit SGD

The IL algorithm was shown to be equivalent to the proximal algorithm/implicit SGD under a set of assumptions including that each weight update used a normalized step size: $\alpha_l = \|\hat{h}_l\|^{-2}$. This normalized step size solves the local prediction problem, such that local errors are minimized to zero given the same input [8] with mini-batch size of 1. The original PC/IL algorithm [100], however, does not use this normalized step size and instead treats α_l as a static hyper-parameter. Further, below we develop an optimizer that uses static step sizes for each layer. This raises the question of how this class of IL algorithms relate to the proximal algorithm. We show that even when a non-normalized step size is used to update weights, IL approximates implicit

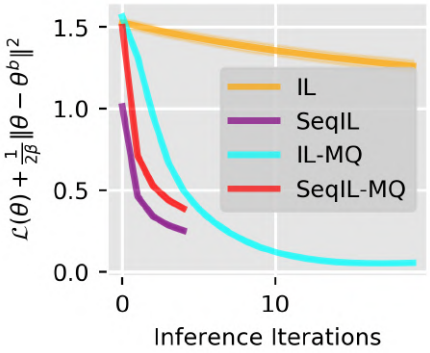

Figure 5.1: Measurement of the proximal objective ($\beta = 100$) during inference phase with fully clamped output layer of standard IL (IL) (20 iterations) and sequential IL (SeqIL) (5 iterations) with a fixed learning rate or MQ adaptive learning rates. Consistent with theorem 5.1, reducing energy F during the inference phase leads to a reduction of the proximal objective.

SGD under certain settings of the γ terms in the free
energy (equation 5.1) and in a certain limit concerning how much weight updates reduce local errors. In the next sections, we use the implicit SGD interpretation to further describe how IL is distinct from BP/SGD.

Let a 'static' scalar step size, α_l, refer to a scalar that does not change during the inference phase with changes to $\hat{h}_l$ (unlike the normalized step size). Let $\theta_{IL}^{(b+1)}$ be the parameters updated at iteration b by the IL algorithm with static learning rates. Let $\theta_{prox}^{(b+1)}$ be the parameters updated by the proximal algorithm (equation 5.6). The next theorem states that the IL algorithm with static learning rates is equivalent to the proximal algorithm under specific settings of the γ variables.

Theorem 5.1. *Consider the IL algorithm at training iteration b with static α_l and mini-batch size one. Assume that at each inference iteration we update the γ terms at hidden layers according to $\gamma_l = \alpha_{l-1}^2 \|f(\hat{h}_{l-1})\|^2$ and $\gamma_l^{decay} = \alpha_l^2 \|e_{l+1}\|^2$, and at the output layer $\gamma_L = \alpha_{L-1}(\frac{1}{\beta} + \|f(\hat{h}_{L-1})\|^2) - 1$. In the limit where $\hat{h}_{L-1}^{(b)} \to h_{L-1}^{(b+1)}$, it is the case that $\theta_{IL}^{(b+1)} = \theta_{prox}^{(b+1)}$.*

The theorem's proof is in appendix B.0.2 and its extension to larger mini-batches is in appendix B.0.3. Thus, under these conditions, IL is an implementation of the proximal algorithm. In particular, IL implements the proximal algorithm in an indirect way by defining the parameters, θ, optimized by the proximal update (eq. 5.6) as a function of hidden/auxiliary variables $\hat{h}$, i.e., $\theta = j(\hat{h})$, which is then optimized through updates to $\hat{h}$ (see B.0.2 for details). The limit in which this theorem is true is the limit where the last hidden layer activity $\hat{h}_{L-1}^{(b)}$ approaches the feed-forward activity after weights are updated, $h_{L-1}^{(b+1)}$, given the same input $x^{(b)}$. This limit holds in the case where local errors are fully minimized by the weight updates. The proof also depends on particular settings for the γ terms. In practice, these settings are usually not used, but we can see that the γ_l terms will generally be positive scalars and are therefore approximated in practice by weighting each term with

some positive scalar (see [8] and figure 5.1 and supplementary figure B.2). Further, we can see that $\gamma^{decay} \approx 0$ at most hidden layers, since the magnitude of errors are initially zero and remain small afterward, providing justification for the common practice of neglecting the decay term in F. Finally, the β term, which is the 'global' learning rate in the implicit SGD/proximal update, affects the clamping of the output layer: if $\beta \to 0$ then $\gamma_L \to \infty$ and $\hat{h}_L \to h_L$ (i.e., as β gets smaller the output layer is more softly clamped). If $\beta \to \infty$ then γ_L will get smaller and the output layer will become more strongly clamped. Indeed, in figure 5.1, we show empirically that several implementations of IL with a fully clamped output layer significantly reduce the proximal objective with a large β during the inference phase (see fig. B.2 and appendix B.0.5 for more detail and evidence).

5.3.2 IL is Sensitive to Second-Order Information

Recent work found that IL, even without optimizers, often reduces loss more quickly than vanilla BP/SGD, especially when small mini-batches are used [8]. Relatedly, it was found empirically by multiple works that IL often takes a shorter/more direct path toward local minima than BP/SGD [8, 121]. However, it has not been clearly explained mathematically why and how these shorter paths and quicker convergence are achieved. Here we provide some explanation based on the insights of Toulis et al. [127, 128], who showed that implicit SGD is sensitive to second-order information for small learning rates. Let $\Delta\theta_{IL}^{(b)}$ equal the IL update at iteration b.

Theorem 5.2. *Consider neural network parameters $\theta^{(b)}$ at iteration b with mini-batch size 1. Assume weight-wise step sizes α_l and the γ variables are set so that $\Delta\theta_{IL}^{(b)} = -\beta\frac{\partial\mathcal{L}(\theta^{(b+1)})}{\partial\theta^{(b+1)}}$. With these assumptions we have $\Delta\theta_{IL}^{(b+1)} = -\beta\frac{\partial\mathcal{L}(\theta^{(b+1)})}{\partial\theta^{(b+1)}} \approx -(I+\beta H)^{-1}\beta\frac{\partial\mathcal{L}(\theta^{(b)})}{\partial\theta^{(b)}}$ with error $\mathcal{O}(\beta^2)$, where $H = \frac{\partial^2\mathcal{L}(\theta^{(b)})}{\partial\theta^{(b)2}}$.*

For proof see appendix B.0.4. This theorem implies that implicit SGD, and IL when it

approximates implicit SGD closely, is sensitive to second-order information. Specifically, IL/implicit SGD approximates a shrinked version of the gradient, where the shrinkage is determined by the inverse Hessian. This use of second-order information, about the curvature of the loss landscape, can help account for why IL often converges more quickly and takes more direct paths toward minima than standard SGD. This theorem also implies that when β is small ($0 < \beta < 1$), IL closely approximates the Newton-Raphson algorithm with 'damping' (scaling by a small learning rate) and Levenburg-Marquardt regularization [88], which is: $\theta^{(b+1)} = \theta^{(b)} - \alpha(\mathbf{I}\lambda + H^{-1})\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}}$, where λ is a scalar weighting term and α the learning rate/damping term. Thus, for small but non-zero β and when IL approximates implicit SGD closely, IL approximates a regularized and damped version of the Newton-Raphson method with small error, $\mathcal{O}(\beta^2)$. Note that as $\beta \to \infty$ the approximation to Newton-Raphson becomes worse because IL becomes increasingly sensitive to even higher-order information in the Taylor-expansion. *As far as we know, this is the first time this relation between IL and the Newton Raphson-method and higher order information has been shown.* We find empirically that our implementations of IL reduces the proximal objective well for both large and some small values of β (fig. 5.1 and B.2), suggesting IL is generally sensitive to second and higher order information.

5.3.3 Reinterpreting IL's Relation to BP and SGD

Previous work found that in the limit where $\hat{h}_l \to h_l$, IL approaches BP/SGD [132, 133, 77, 104]. This limit is equivalent, in our notation, to the limit where the $\beta \to 0$ since in this case the output layer $\hat{h}_L$ approaches its initial value h_L, resulting in $\hat{h}_l \to h_l$. Notice that this is exactly what theorem 5.2 implies: as $\beta \to 0$ the Hessian term βH and the error, $\mathcal{O}(\beta^2)$, go to zero, and we are left with a standard SGD update. *Thus, consistent with these previous results, the implicit SGD interpretation predicts that IL should approach SGD as $\beta \to 0$ and $\hat{h} \to h$. However, theorems 5.1 and 5.2 imply that standard/explicit SGD is not*

the best interpretation of IL when $\beta > 0$, since the SGD approximation has a larger error, $\mathcal{O}(\beta)$, than approximations that take into account higher order terms, which have at most error of $\mathcal{O}(\beta^2)$.

5.4 The Matrix Update Equalization Optimizer

We present a novel optimizer, called Matrix Update Equalization (MQ), made specifically for the IL algorithm. Previous work (e.g. [8], also see below) observed that IL often converges to poor local minima when it does not use optimizers with parameter-wise adaptive learning rates, like Adam. However, Adam and similar optimizers are memory intensive. MQ seeks to provide similar benefits as Adam without any significant cost in memory or computation. It is motivated by two observations. First, Theorem 1 and 2 imply IL algorithms that only use matrix-wise scalar learning rates are sensitive to second and higher-order information. Second, IL updates, without the use of Adam, tend to be very small[2], especially at early layers (see figure 5.2), which likely explains why IL gets caught in shallow local minima nearer its initial values. Adam makes updates larger and more equal across matrices. These observations suggest that the main benefit of using optimizers like Adam with IL is that it prevents weight updates at early layers from becoming too small, pushing parameters further to deeper and flatter minima. This is contrary to the benefits Adam is supposed to provide BP/SGD, which is in large part to provide BP with an estimate of second order information [48] to speed up training.

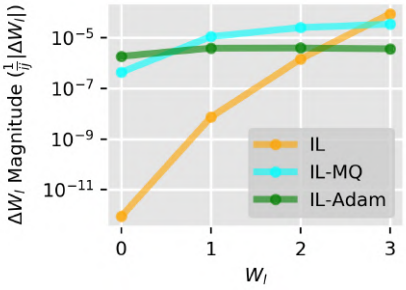

Figure 5.2: Average update magnitude of each weight matrix in MLP with three hidden layers.

Motivated by these observations, MQ uses only a single scalar learning rate per layer that

[2]It is worth noting that IL's tendency for small update magnitudes is consistent with the proximal interpretation.

adapts to prevent overly small update magnitudes and to better equalize the update magnitudes between matrices. MQ stores several scalars: a constant matrix-wise learning rate, α_l, for each matrix W_l, a constant global minimum learning rate, α_{min}, and matrix wise scalar v_l, which is updated continuously to adapt to the moving average of the update magnitude for matrix W_l. The MQ weight update for weight matrix W_l is

$$W_l^{(b+1)} = W_l^{(b)} - (\frac{\alpha_l}{v_l + r} + \alpha_{min})\frac{\partial F}{\partial W_l^{(b)}}, \tag{5.7}$$

where r is a constant that prevents division by zero and stabilizes learning. The minimum learning rate α_{min} is used to prevent learning from decreasing too much in any one matrix. The variable v_l is updated each training iteration b as follows:

$$v_l^{(b+1)} = (1 - \rho)v_l^{(b)} - \rho\frac{1}{ij}|\frac{\partial F}{\partial W_l^{(b)}}|, \tag{5.8}$$

where i and j refer to the row and column sizes, respectively, of the gradient $\frac{\partial F}{\partial W_l}$ and $|\frac{\partial F}{\partial W_l^{(b)}}|$ is the L-1 norm of the gradient. Finally, ρ is a global hyper-parameter (scalar). Note, although ρ is a hyper-parameter, we use a method for reducing bias early in training, which is explained in appendix B.0.5. Thus, v_l first averages the update over elements of the absolute value of the gradient (right term), then it takes a weighted average of this with the previous value of v_l. Figure 5.2 shows MQ reduces the differences in update magnitude between weights matrices nearly as well as Adam and prevents updates at early layers form becoming too small.

5.5 Reducing Computation with Sequential Inference

The inference phase accounts for most of the computational cost of the IL algorithm. Typically, activities are updated for $T = 15$ or more iterations (e.g., [132, 9, 110]), which requires

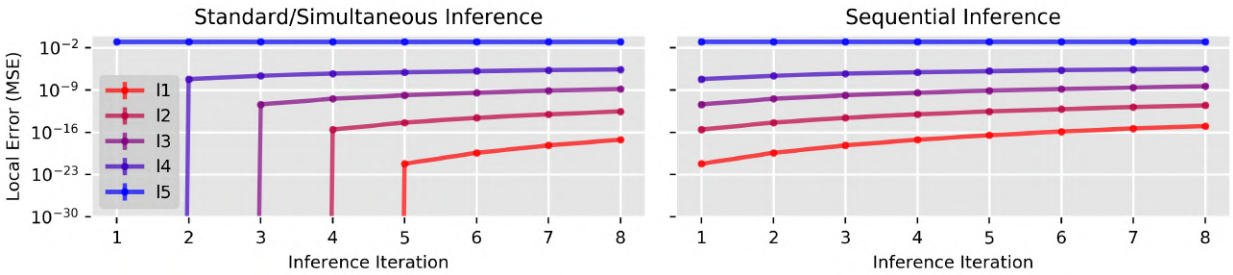

Figure 5.3: Measurement of the mean of the squared local errors, e_l, at each hidden layer (layer 1-4) and the output layer (layer 5) in a fully connected network with four hidden layers.

around $T(2L-1)$ matrix multiplications. The obvious way to reduce computation is to truncate the inference phase (reduce T). The difficulty with this approach is that the standard implementation of the inference phase requires at least $T \geq (L-2)$ in order for non-zero errors to form at each hidden layer (see figure 5.3), and typically more iterations to attain the best performance (see experiments section), e.g., a network with four hidden layers typically requires at least $T \geq 4$, and usually much more than 4 to perform well. In figure 5.3 and B.1, we show that the standard implementation requires large T because there is a delay in error propagating through the network. Each inference iteration, t, the standard method computes and stores errors at each layer, then these stored errors are used to update the activities according to equation 5.2 (see algorithm 3). This process simulates activities being updated simultaneously in time, since the update at one layer does not alter the updates or errors at other layers until the next iteration/time step. Instead of updating activities simultaneously, we propose updating them one at a time starting from the output layer and moving back in a sequence (see algorithm 4). We call this method *sequential inference*. Sequential inference removes the delay and progagates non-zero errors to every layer in a single iteration, allowing for smaller T (see figure 5.3 and supplementary figure B.1). [3]

[3]Though this is a simple alteration, we could only find one work where an implementation similar to sequential inference is used [104], and in that work this implementation was not discussed in the paper, analyzed, or used as a method for reducing computation. Instead, it seemed to be an arbitrary implementation decision.

5.6 Experiments

In this section, we test the performance of sequential inference and the MQ optimizer. The MQ optimizer works well in all models using following hyperparameters: $\alpha_{min} = .001$, $r = .000001$, and $\rho = .9999$ for fully connected and $\rho = .999$ for convolutional networks. Grid searches are used to find learning rates, α_l. Mini-batches size 64 are used in all simulations.

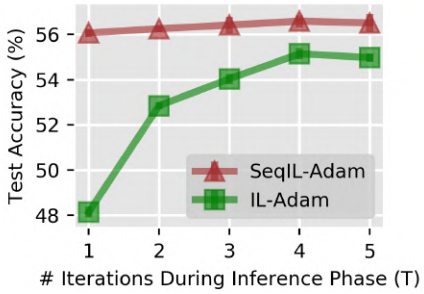

Figure 5.4: MLPs trained on CIFAR-10 for 45 epochs, using either sequential inference with Adam (SeqIL-Adam) or simultaneous inference with Adam (SimIL-Adam). X-axis specifies the number of inference iterations used during training.

Comparing Sequential and Simultaneous IL The standard/simultaneous inference method creates a delayed error propagation through the network, as illustrated in figure 5.3. We test how this delay effects performance at very small values of T on a classification of CIFAR-10 images. An MLP with four hidden layers, dimension 3072-4x1024-10, was trained on CIFAR-10 using both standard/simultaneous inference with Adam optimization (IL-Adam) and sequential inference with Adam optimization (SeqIL-Adam) for different values of T. Grid searches were used to find the learning rate and step size, ϵ, for activity updates. Models are trained over 45 epochs, about the amount of time needed for learning to near convergence. Results shown in figure 5.4. SeqIL-Adam performs approximately the same for all values of T (1-5), while IL-Adam's performance is highly sensitive to the value of T, with performance degrading significantly for smaller values of T.

Classification with Natural Images Next, we test the MQ optimizer combined with SeqIL on classifications tasks with natural image data sets: SVHN, CIFAR-10, and Tiny Imagenet. We train fully connected MLPs dimension 3072-3x1024-10 on SVHN and CIFAR-10, and small convolutional networks on SVHN, CIFAR-10, and Tiny Imagenet. All of IL

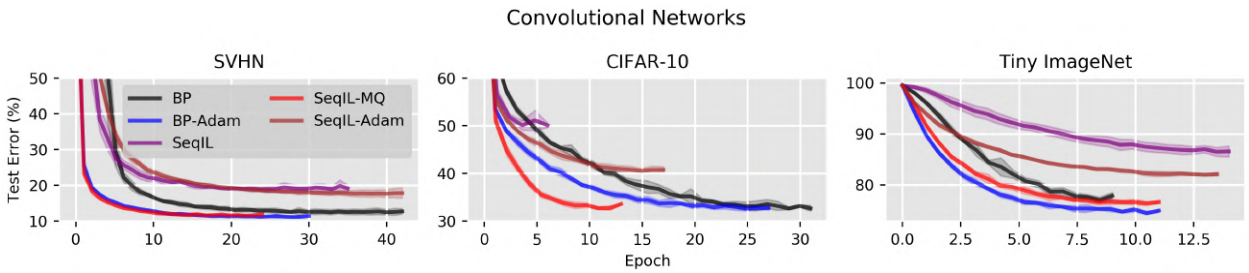

Figure 5.5: Small convolutional networks trained on classification tasks. Training runs are averaged over 5 seeds. Shaded error bars show standard deviation across seeds. Each training run is shown up until convergence (the point of lowest test error +1 epoch).

algorithms use a highly truncated inference phases of T=3. In addition to sequential IL with MQ (SeqIL-MQ), we test SeqIL with no optimizer (SeqIL) and SeqIL with Adam (SeqIL-Adam). We compare to the performance of BP-SGD and BP with Adam (BP-Adam). Grid searches over learning rates were performed to find the learning rate that yielded the best test performance at convergence. SeqIL-MQ test accuracies were never significantly worse than BP and slightly better than BP on Tiny ImageNet, while consistently performing comparably to BP-Adam (table 5.6). Further, SeqIL-MQ reduced loss more quickly early in training than BP-SGD in all simulations, and typically converged much more quickly (figure 5.5). SeqIL-MQ learning curves looked more similar to BP-Adam, despite the fact SeqIL-MQ uses about one third the memory of Adam per parameter. SeqIL-Adam matched the performance of SeqIL-MQ in fully connected networks but struggled in convolutional networks. We believe this is because in convolutional networks SeqIL-Adam had very small weight updates, which could not be increased by simply increasing learning rate due to instability (see app. fig B.3). Consistent with previous work on IL, we find that SeqIL without optimizers often reduces loss more quickly than BP early in training in MLPs, but struggled to converge to good minima.

Autoencoders We trained fully connected autoencoders (layer sizes 3072-1024-256-20-256-1024-3072) with ReLU at hidden layers and sigmoid at the output layer. Networks were trained on SVHN and CIFAR-10. A grid search was used to find a learning rate that

Classification Test Accuracies						
	SVHN (FC)	SVHN (Conv)	CIFAR (FC)	CIFAR (Conv)	TinyImNet Top 1	Top 5
BP	**84.40**$^{(\pm.12)}$	87.82$^{(\pm.67)}$	56.55$^{(\pm.08)}$	**67.76**$^{(\pm.43)}$	23.14$^{(\pm.42)}$	47.05$^{(\pm.46)}$
BP-Adam	84.08$^{(\pm.13)}$	**89.02**$^{(\pm.19)}$	56.15$^{(\pm.21)}$	**67.51**$^{(\pm.45)}$	**25.61**$^{(\pm.22)}$	**50.04**$^{(\pm.41)}$
SeqIL	82.19$^{(\pm.70)}$	81.55$^{(\pm.59)}$	54.24$^{(\pm.18)}$	50.67$^{(\pm.76)}$	13.88$^{(\pm.73)}$	33.42$^{(\pm1.16)}$
SeqIL-MQ	**84.51**$^{(\pm.17)}$	88.67$^{(\pm.18)}$	**57.70**$^{(\pm.21)}$	**67.50**$^{(\pm.19)}$	23.91$^{(\pm.31)}$	47.53$^{(\pm.48)}$
SeqIL-Adam	84.13$^{(\pm.15)}$	82.76$^{(\pm.87)}$	56.21$^{(\pm.20)}$	59.72$^{(\pm.48)}$	18.22$^{(\pm.29)}$	39.58$^{(\pm.46)}$

Table 5.2: Best test accuracies averaged across 5 seeds (mean ± standard dev.). Top scores and scores not significantly different from top score (according to two-sample t-test) are bolded.

produces the lowest test loss at convergence or after 100 epochs. Testing auto-encoders is important because, unlike classifiers, their output layers are large, which may make the task of propagating errors through the network easier for IL algorithms. IL models use T=6 inference iterations. We find that all IL models reduce the loss significantly faster than both BP and BP-Adam early in training (figure 5.6, right two plots). IL-MQ and IL-Adam converge to similar reconstruction losses as the BP algorithms.

5.7 Related Work

Theoretical work on IL Several recent works develop theoretical analyses of IL. Song et al. [121] argue IL and similar energy-based algorithms, like Constrastive Hebbian Learning

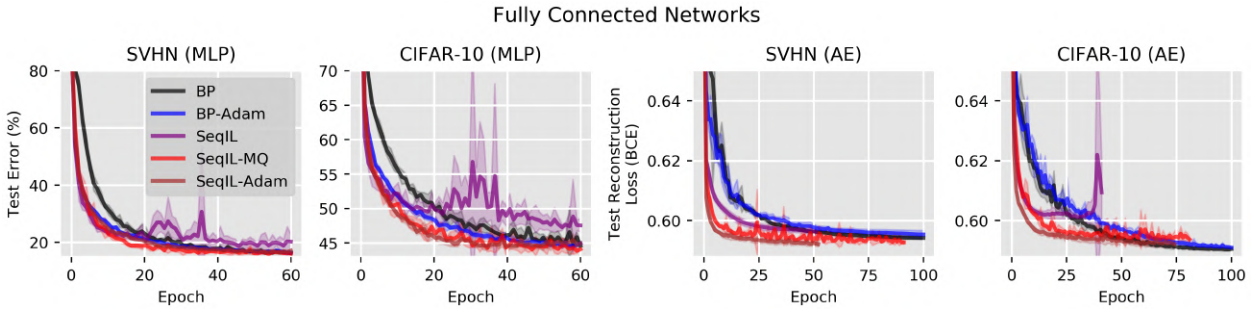

Figure 5.6: Fully connected networks trained for classification tasks (MLP) and for autoencoder tasks (AE). Training runs are averaged over 5 seeds. Shaded error bars show standard deviation across seeds. The first 60 epochs of training shown for MLPs. For AEs, training runs are shown to convergence (lowest loss +1 epoch) or the max 100 epochs.

[83], work differently than BP. Unlike BP, these algorithms first compute the desired neural activities, then update parameters to consolidate these neural activities (i.e., make them more probable). They call this approach 'prospective configuration'. They propose the brain likely learns in a way similar to prospective configuration and provide empirical support. Millidge et al. [78] provide a formal analysis showing the neural activities computed by prospective configuration algorithms are similar to regularized Gauss-Newton targets, and IL is a variant EM algorithm known as maximum a posteriori learning. Alonso et al. [8] developed an alternative, though compatible, analyses of IL that showed its approximation to the proximal algorithm/implicit SGD, and further examined the differences between IL and BP such as differences in stability across learning rates. The theoretical work here expands on these previous findings by showing 1) the results of Alonso et al. extend approximately to the more general case where a static learning rate is used at each layer (instead of a dynamic normalized learning rate) and 2) for the first time shows that, under the assumption IL is closely approximating implicit SGD, describes how IL is sensitive to higher order information. We believe our results provide a deeper understanding of how IL differs from BP/SGD than these previous works.

Algorithmic Work on IL The IL algorithm has recently been altered in several ways. One line of work altered IL in order to make its weight updates more similar to SGD (e.g., [120, 79]). Alonso et al. altered IL to make its weight updates better approximate the proximal algorithm/implicit SGD. Our alterations to IL differ from these previous works, as they do not seek to alter IL to make it more similar to SGD or implicit SGD. Salvatori et al. [112] developed a variant of IL based on incremental EM [85], called incremental PC, which works by updating weights each inference iteration instead of at the end of the inference phase. Incremental PC has some benefits (e.g., more bio-plausible and reduces free energy quickly during inference). However, incremental PC still suffers from the delayed error propagation problem and its implementation in [112] uses Adam optimizers. It is unclear whether incremental PC avoids convergence issues when Adam is not used. Incremental

PC also requires more matrix multiplications each training iteration than SeqIL, given the same number of inference iterations (T) (see appendix table B.3). Our goal was to reduce computation of IL while improving convergence, so we do not further develop incremental PC here.

5.8 Limitations

More testing is needed to establish how well our methods work in larger networks with more than 3-5 hidden layers and to establish how our algorithms' behavior compares to BP across different batch sizes (we only test with mini-batch size 64).

5.9 Discussion

Biologically motivated algorithms have historically faced at least two central challenges: 1) difficulty converging to test losses as good as BP/SGD when scaled to natural images [10] and/or 2) requiring far more computation than BP/SGD. A common method for dealing with one or both of these issues is to make one's bio-plausible algorithm of choice more similar to BP/SGD (e.g., [113, 120, 79, 54, 28]). In this chapter, we took a different strategy. We took a highly biologically constrained algorithm, IL, and showed its differences from BP/SGD are not undesirable properties that ought to be removed. Instead, IL differs from BP/SGD because it implements a different optimization method, which has useful sensitivity to higher-order information. On the basis of our analysis, we altered the way its recurrent processing worked to significantly reduce computation and we created a custom optimizer made specifically to address the convergence issue of the IL algorithm. This is not unlike the way many standard optimizers, e.g., Adam, were specifically created to address shortcomings of BP/SGD. The result is an algorithm that, in our simulations on natural images, matched

BP/SGD in terms of final test accuracy/loss, memory (table B.1), and training time (in seconds) (fig. B.4), while often reducing loss and converging more quickly than BP/SGD (in number of training iterations). These results suggest IL is a promising algorithm for further development and application in the bio-inspired and neuromorphic machine learning communities.

Chapter 6

A Sparse Quantized Hopfield Network for Online-Continual Memory

6.1 Introduction

A fundamental question in computational neuroscience and neuromorphic computing is the question of how to train deep neural networks using only local learning rules in the learning scenario faced by the brain, where data is noisy and presented in an online-continual fashion. Local learning rules use only information spatially and temporally adjacent to the synapse at the time of the update (e.g., pre-synaptic and post-synaptic neuron activity). Online-continual learning occurs when a stream of single data points are presented during training in non-independent and identically distributed (non-i.i.d.) fashion (e.g., several datasets are presented one dataset at a time). Brains must learn under these conditions, and neuromorphic hardware embedded in real world systems have similar constraints [26].[1]

[1]This chapter is a slightly modifed version of a pre-print of the same title co-authored by myself and Jeff Krichmar, which is currently under review. The full pre-print can be found at https://arxiv.org/abs/2307.15040.

Standard approaches to deep learning have not provided a solution to this problem. The standard approach trains neural networks with stochastic gradient descent (SGD) implemented by the backpropagation algorithm (BP) [107]. BP is a non-local learning algorithm generally considered biologically implausible [25, 123, 63] and is difficult to make compatible with neuromorphic hardware [87, 114]. Further, the standard training paradigm is offline learning, where data is mini-batched, i.i.d., non-noisy and can be passed over for multiple epochs during training. Learning in the noisy, online-continual scenario is much more difficult than the offline scenario. Unlike offline learners, online-continual learners must avoid problems like catastrophic forgetting and are pressured to deal with noise and to be more sample efficient (faster learners).

Furthermore, recent work on this problem has not approached all the aspects of the problem simultaneously. For example, although bio-plausible algorithms have been developed for deep networks, these algorithms are typically tested and developed for offline settings (e.g., [91, 132, 113, 109]). Although progress has been made on online-continual learning, essentially all of this work uses BP in some capacity (see [47, 131, 96, 35, 64, 41]). Some works do test local learning algorithms in online and continual settings separately, but do not address the online and continual setting simultaneously or focus on shallow recurrent networks (e.g., [13, 139, 138]). Therefore, novel local learning models that perform online-continual learning without resorting to BP are needed.

We attempt to remedy this situation by making the following contributions: 1) Unlike previous works on online-continual learning, which tend to focus on classification, we study the more general and basic task of *associative memory*, i.e., the basic process of storing and retrieving corrupted and partial patterns. We believe studying this basic task could yield ideas that apply across a wide range of tasks, rather than a single narrow task, like classification. 2) We propose a general approach to online-continual associative memory, based on the idea that a sparse, quantized neural code can both deal with noisy, partial input and prevent

catastrophic forgetting. Further, we propose implementing sparse, quantized neural codes using discrete graphical models that learn via algorithms similar to maximum a posteriori (MAP) learning, where MAP learning has the advantage of using local learning rules. 3) We implement this approach in a novel neural network called the sparse quantized Hopfield network (SQHN), an energy based model that optimizes a novel energy function and utilizes a novel learning algorithm that combines neuro-genesis (neuron growth) and local learning rules, both engineered specifically to yield high performance in the noisy and online-continual setting. 4) We develop two memory tasks, which are new to the recent machine learning literature on associative memory models, the noisy encoding task and an episodic memory task. 5) We run a variety of tests showing that SQHN significantly outperforms baselines on these new tasks and matches or exceeds state of the art (SoTA) on more standard associative memory tasks.

6.2 Results

6.2.1 Toward A Foundation for Local, Online-Continual Memory Models

Our goal is to design a model that can 1) deal with noisy, partial inputs in associative memory tasks, 2) learn in a sample-efficient way that avoids catastrophic forgetting in online-continual learning scenarios, and 3) uses only local learning rules. Our proposed approach has three parts.

First, we propose that *quantization* provides a principled approach to associative recall. Quantization is the process of mapping continuous valued inputs to a finite, discrete code, which is necessarily a process of pattern completion, i.e., a process where many distinct vectors are mapped to the same vector. The associative memory problem may also be cast

as one of pattern completion, where the goal is to reconstruct stored data points, x, given corrupted or partial versions, $\tilde{x}$, of it, i.e., $[\tilde{x}_0, \tilde{x}_1, ...] \rightarrow x$. Quantization can be used to perform this mapping via $[\tilde{x}_0, \tilde{x}_1, ...] \rightarrow h^* \rightarrow x$, where h^* is a single discrete latent code. Pattern completion is intuitively more difficult with continuous latent codes, since these codes may vary in an infinite number of ways, making it more difficult to map many corrupted versions of a data point to the same latent code and reconstruction (e.g., see experiments and discussion).

Second, we primarily use *parameter isolation* to avoid catastrophic forgetting, which is a strategy that has recent success in BP-based deep learning models (e.g.,[58, 140, 66, 65]). This strategy allocates subsets of new and old parameters to different tasks during training, as needed. By only using and updating small subsets of parameters each iteration, models are able to drastically avoid forgetting. However, there needs to be a principled method to decide which parameters to update or add at which times.

Third, we propose using an MAP learning algorithm as a local learning algorithm, in discrete-graphical models, which naturally implement quantization and parameter isolation. MAP learning works by first performing inference over hidden variables to find their specific values, h^*, that maximize the posterior $P(h^*|x, \theta) \propto P(h^*, x|\theta)$. In discrete graphical models these values are integers. Parameters are then updated to further increase the probability of the joint $P(h^*, x|\theta)$. Local learning rules are naturally used in this algorithm (supp. C.2.5). Further, if integer values are represented by sparse, one-hot vectors, updates are also sparse, i.e. perform parameter isolation.

6.2.2 The Sparse Quantized Hopfield Network

We develop a novel implementation of a discrete graphical model that uses primarily neural network operations. We call it the *sparse quantized Hopfield network* (SQHN). SQHNs

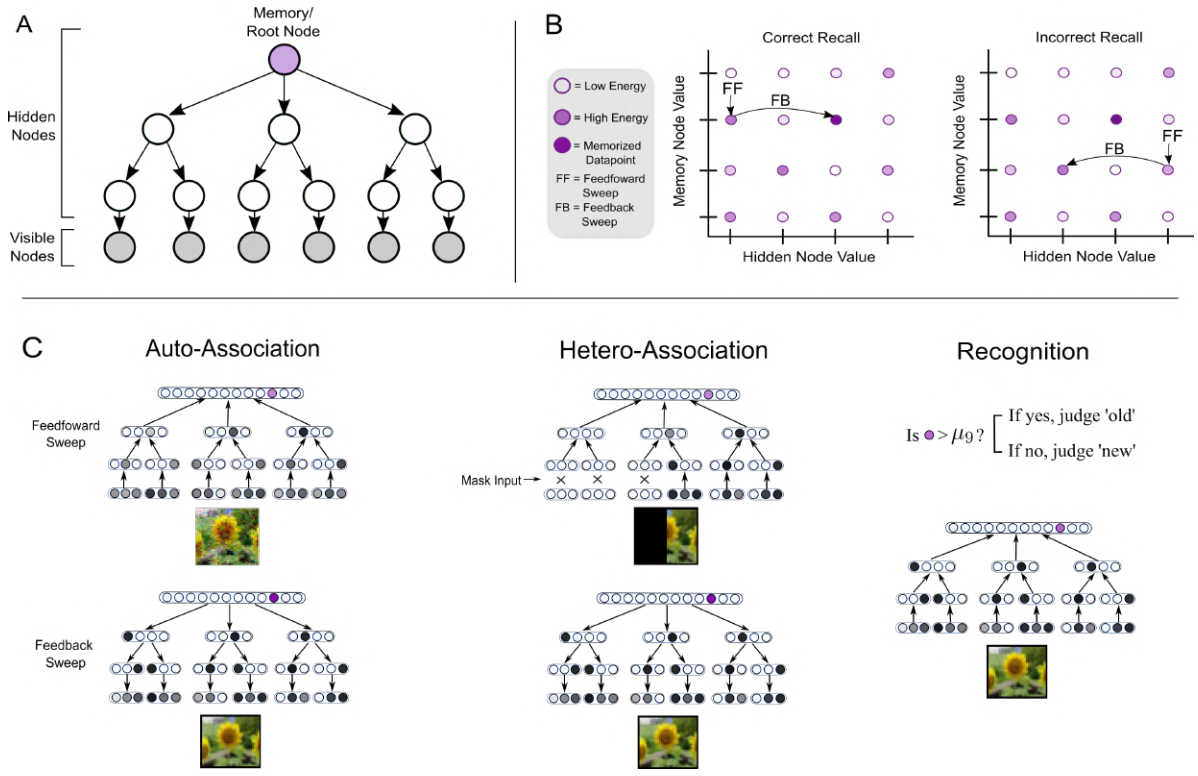

Figure 6.1: **A.** Tree structured, directed, acyclic graph. **B.** Diagram of recall process in terms of the node values and energy. Nodes take integer values. Correct recall finds the set of node values at the global maximum of the energy. Correct recall typically only occurs if the memory node has the correct value, since the memory node is fixed after the feedforward (FF) pass, after which it then adjusts the values of hidden nodes through feed-back/top-down signals. **C.** Neural network diagrams of the SQHN during associative recall and recognition. During recall the FF sweep propagates signals up the hierarchy, where the memory/root node retrieves the most probable (high energy) value, and propagates the signals down encouraging hidden nodes to take the values associated with that particular memory/value.

have architectural similarities to Hopfield networks and Bayesian networks. Unlike standard Bayesian networks, SQHNs are relatively easy to scale and more compatible with hardware that assumes vector matrix multiplication as the basic operation (e.g., GPUs and memristors). Unlike common Hopfield nets, SQHNs explicitly utilize quantization and implement a discrete, directed graphical model. Hidden nodes in SQHN models are assigned integer values during inference, represented by sparse one-hot vectors. Sparsity distinguishes SQHNs from prior quantized Hopfield networks (e.g., [69, 70]), which assign an integer value to each neuron. The sparse code we use ensures subsets of parameters are isolated during training.

Energy

SQHNs implement direct graphical models. In this chapter, we consider tree-architectures without loops (see Fig. 6.1), though many other architectures are possible. Visible nodes are nodes clamped to portions of the input (e.g., image patches), which are assumed, though not required, to be continuous. Each hidden node l represents categorical variables that take an integer value represented by a one-hot vector, h_l^*. We also notate clamped values at visible nodes as h_l^*. Conditional probability $p(h_l^*|pa_l)$ of node l given its parent value is parameterized by synaptic weight matrices. For example, in the simple case where l has one parent $p(h_l^*|pa_l)$ it is parameterized by matrix $M_{pa_l,l}$. The energy is a summation over conditional probabilities:

$$E(h^*, \theta) = \frac{1}{L} \sum_{l=0}^{L} p(h_l^*|pa_l). \tag{6.1}$$

Typically, in directed graphical models, like Bayesian Networks, the aim of MAP inference is to update the values of hidden nodes to maximize the joint probability of node states, which is the *product* of the conditional probabilities rather than the sum [17]. We show that the summation of the conditional probabilities approximates a kind of joint probability that takes into account uncertainty over parameters (supp. C.2.2). Taking into account this uncertainty is crucial for learning in online settings. Further, by approximating this joint distribution with the energy above, we can implement a model that performs inference using only standard artificial neural network operations that sum inputs to neurons rather than multiply, which would be needed with the standard joint distribution (supp. C.2.3).

Recall

SQHN models update neuron activities during recall in a way akin to MAP inference, where neurons are updated to maximize the energy:

$$h^* = \arg\max_{h^*} E(h^*, \theta, x), \tag{6.2}$$

where h^* is the set of one-hot vectors assigned to each node. We use a novel inference procedure, which we show empirically is a good approximate solution to this optimization problem (supp. fig. C.2.3). This approximate inference method uses standard neural network operations and is computationally cheap, only involving a single feed-forward/bottom-up (FF) and feedback/top-down (FB) sweep through the network. Mathematically, this recall process can be understood as follows: each data point x^t observed during training is stored at maxima of the energy. Therefore, given a corrupted input $\tilde{x}^t$, an SQHN can reconstruct the original by finding the latent code that maximizes the energy given $\tilde{x}^t$, which, during correct recall, will be the same latent code as the original data point (figure 6.1 B). Mechanistically, recall works by propagating signals up to the memory node which is assigned a memory value. Then a signal is propagated down, encouraging lower level hidden nodes to take values associated with that memory value (figure 6.1 C).

For a pseudo-code description, see supplementary algorithm 7.

Learning

SQHNs update their parameters using an algorithm akin to MAP learning, where each training iteration first perform inference to maximize energy w.r.t. activities and then update

weights to further increase energy.

$$\theta^T = \arg\max_{\theta} \sum_{t=0}^{T} E(\theta, h^{*,t}, x^t), \tag{6.3}$$

where matrices at hidden layers must meet certain normalization constraints, making this a constrained optimization problem. Importantly, weights are updated to maximize energy over *all previously observed data* points rather than just the one present at the current iteration. This helps prevent forgetting of previous data points in online scenarios. However, the update is performed using only the activities and data point from the current iteration (i.e., there is no buffer of previous data points or activities). The solution is a local Hebbian-like update rule:

$$\Delta M_{pa_l,l} = \frac{1}{c^*_{pa_l} + 1}(h^*_l - M_{pa_l,l} h^*_{pa_l}) h^{*\top}_{pa_l}, \tag{6.4}$$

where $c^*_{pa_l}$ is a count of the number of iterations the parent node value was activated during training. Because $h^*_{pa_l}$ is a sparse, one-hot vector, this is a sparse weight update that only alters the values in *a single column* of matrix $M_{pa_l,l}$.

Importantly, instead of randomly initializing weights, we initialize all weights equal to 0, then grow new neurons and synapses as needed. That is, before weights are updated, neurons are grown during inference. A new neuron is grown at node l if no neuron at the node has a value greater than some threshold. We use an exponentially decaying threshold based on the Dirichlet prior:

$$\epsilon = \frac{\alpha}{(t + \alpha)}, \tag{6.5}$$

where t is current training iteration and α is a hyper-parameter. Ablations show that if neuron growth, decaying growth threshold, or learning rate decay are removed, the model performs noticeably worse in online-continual settings (suppl. fig. C.2).

For pseudo-code description see supplementary, algorithm 8.

Episodic Memory Task

Below, we test SQHNs on a novel binary classification task, where the model must recognize whether some particular data point was observed previously during training or not. To perform recognition, we use the value of the neuron at the root/memory node with the maximum value. This maximum value tells us how similar or probable the features of the current data point is to the features of the most similar previously observed data point (supp. C.2.6). If the max activity at the memory node is above some threshold, the data point is judged to be old. If below, it is judged to be new. The threshold we use is the moving average, μ_l, of the activity values observed for each neuron:

$$\mu_{l,j}^t = \frac{1}{c_{l,j}^t} \sum_{n=1}^{t} h_{l,j}^{*,n} = \frac{c_{l,j}^t - 1}{c_{l,j}^t} \mu_{l,j}^{t-1} + \frac{1}{c_{l,j}^t} h_{l,j}^{*,t}, \tag{6.6}$$

where the equation on the right shows how to compute this average online. We show that this value is an estimate of the desired probability of $p(x^t = old|\theta^{(t-1)}, x^t) = .5$, which is the threshold at which it becomes more probable than not that the data point is old rather than new (supp. C.2.6).

For pseudo-code description of recognition, see supplementary algorithm 9.

6.2.3 Experiments

We tested SQHN on several tasks and compare its performance to SoTA and baselines. Specifically, we tested performance on: 1) Auto-Association. 2) Hetero-association. 3) Online Continual Auto-Association. 4) Noisy Encoding. 5) Episodic Memory.

Recall MSE - Moderate Corruption						
	White Noise		Pixel Dropout		Mask	
	CIFAR-10	TinyImgNet	CIFAR-10	TinyImgNet	CIFAR-10	TinyImgNet
GPCN(offline) [139]	$.0121^{(\pm.0001)}$	$.0067^{(\pm.0004)}$	$.0001^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0009^{(\pm.0000)}$	$.0001^{(\pm.0000)}$
BayesPCN [139]	$.0337^{(\pm.0007)}$	$.6606^{(\pm.0267)}$	$.0001^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0019^{(\pm.0000)}$	$.0000^{(\pm.0000)}$
BayesPCN(forget) [139]	$.0188^{(\pm.0002)}$	$.0176^{(\pm.0001)}$	$.0019^{(\pm.0000)}$	$.0008^{(\pm.0000)}$	$.0465^{(\pm.0001)}$	$.0235^{(\pm.0001)}$
MHN	$.1457^{(\pm.0097)}$	$.0955^{(\pm.0073)}$	$.4512^{(\pm.0558)}$	$.4545^{(\pm.0456)}$	$.5538^{(\pm.0149)}$	$.6101^{(\pm.0091)}$
MHN-Manhtn	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0020^{(\pm.0028)}$	$.0016^{(\pm.0012)}$
MHN-GradInf [139]	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0001^{(\pm.0000)}$
SQHN L1	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$
SQHN L2	$.0002^{(\pm.0001)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0002^{(\pm.0002)}$	$.0001^{(\pm.0000)}$	$.0008^{(\pm.0002)}$
SQHN L3	$.1076^{(\pm.0022)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$

Recall MSE - High Corruption						
	White Noise		Pixel Dropout		Mask	
	CIFAR-10	TinyImgNet	CIFAR-10	TinyImgNet	CIFAR-10	TinyImgNet
BayesPCN [139]	$.0755^{(\pm.0002)}$	$.0242^{(\pm.0002)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0006^{(\pm.0000)}$	$.0001^{(\pm.0000)}$
MHN-Manhtn	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.1840^{(\pm.0368)}$	$.1248^{(\pm.0884)}$	$.7644^{(\pm.2844)}$	$.7588^{(\pm.2188)}$
MHN-GradInf [139]	$.0052^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.3840^{(\pm.0010)}$	$.5630^{(\pm.0036)}$	$.3957^{(\pm.0000)}$	$.6378^{(\pm.0000)}$
SQHN L1	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$
SQHN L2	$.0904^{(\pm.0078)}$	$.0002^{(\pm.0004)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0008^{(\pm.0006)}$
SQHN L3	$.3324^{(\pm.0056)}$	$.0240^{(\pm.0032)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$	$.0000^{(\pm.0000)}$

Table 6.1: **Top** Recall MSE of 1024 images from CIFAR-10 and Tiny ImageNet datasets. At test time, images are either corrupted with white noise (variance .2), 25% of the pixels dropped, or the right 25% of the pixels masked. **Bottom** Recall MSE of 128 images from CIFAR-10 and Tiny ImageNet datasets. At test time, images are either corrupted with white noise (variance .8), 75% of the pixels dropped, or the right 75% of the pixels masked.

Auto-Association and Hetero-Association Comparison

We first compared SQHN models to SoTA associative memory models on auto-associative and hetero-associative recall tasks. For both tasks, unaltered data points from a set X_{train} are presented to the model during training. In auto-association, during testing the model is given corrupted versions, $\tilde{X}_{train}$, of training data, and the model is tasked with reconstructing the original data points. Here, corruption is added to the images with white noise. For hetero-association tasks, during testing portions of the input data are treated as missing. Following, the recent work of [139], we remove a certain number of pixels from the input data randomly (pixel dropout) or we remove a certain number of the right most pixels (mask).

Three SQHNs are tested: an SQHN with one hidden layer (SQHN L1), two hidden layers

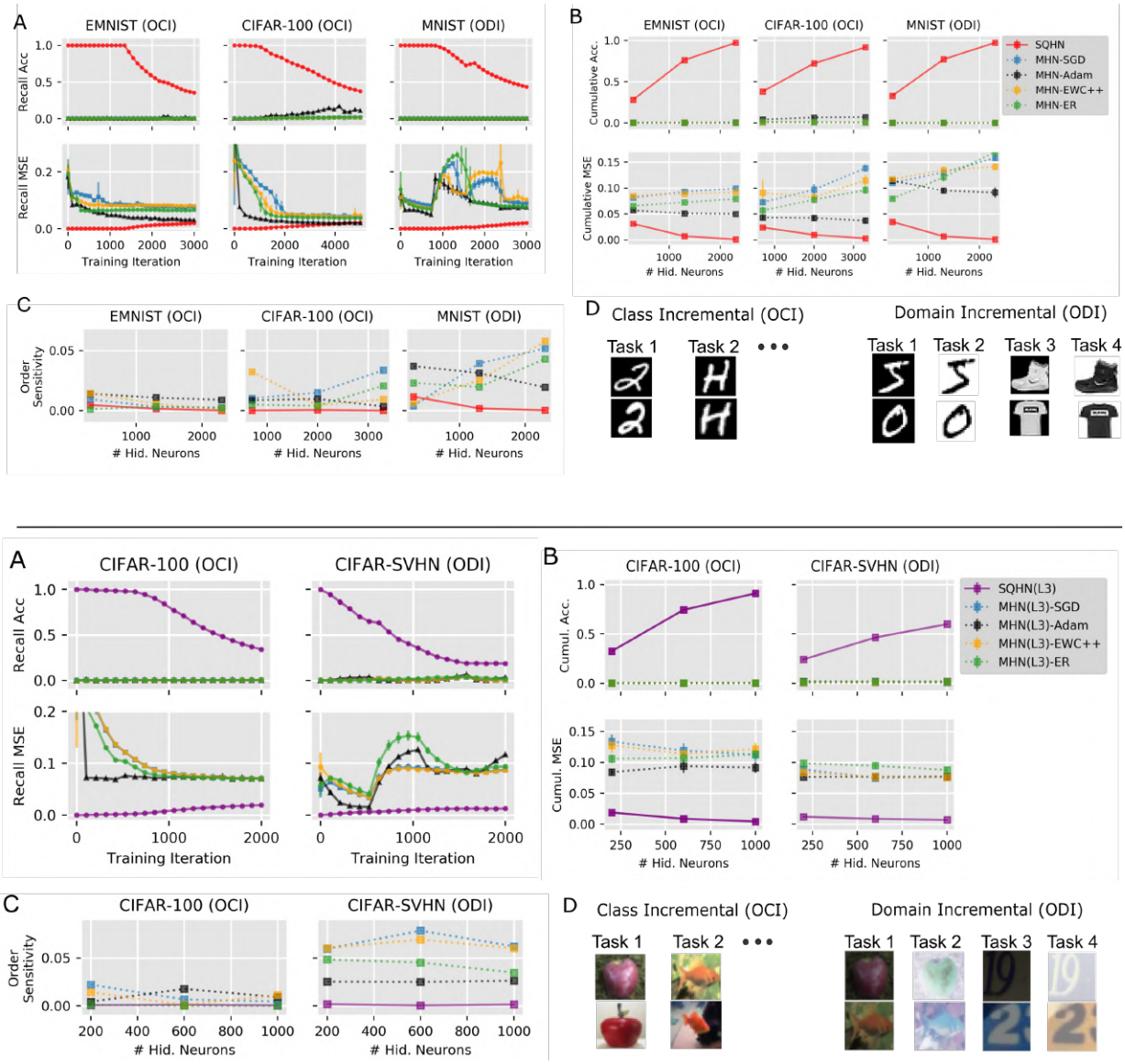

Figure 6.2: Online Continual Auto-Association. **Top** One hidden layer models with small (300), medium (1300), large (2300) node sizes. **Bottom** Three hidden layer models with small (200), medium (600), large (1000) node sizes. **A.** Recall accuracy and recall MSE during training on models with medium size hidden nodes. **B.** Cumulative recall MSE and recall accuracy for each model size. **C.** Order sensitivity for each model size. **D.** Online class incremental (OCI) versus online domain incremental (ODI) settings.

(SQHN L2), and three hidden layers (SQHN L3). We compare to two types of SoTA models: predictive coding networks (PCN) and modern Hopfield networks (MHNs). PCNs are neural network models [100], that implement a kind of probabilistic generative model with continuous latent variables [34]. Three types are compared: offline trained PCN (GPCN) [111]

and two online trained versions (BayesPCN, BayesPCN with forgetting) [139]. Continuous MHNs [99] have similarities to auto-encoders with a single hidden layer where a softmax activation is used. We compare to the original model of [99] (MHN), a version of this model by Millidge et al. [75], which showed better performance by using a Manhattan distance measurement in its recall operation (MHN-Manhtn) (see methods), and the MHN of [139] that used a novel gradient-based inference procedure (MHN-GradInf).

Results are in table 6.1. PCN models struggled on noisy, auto-association task. The MHN that uses Manhattan distance performed very well, and the MHN-GradInf and the one level SQHN model performed perfectly on all auto-association tests. The multi-level SQHNs were more sensitive to white noise on smaller CIFAR-10 images, but still performed well with moderate corruption, and performed very well when they have larger lower layer receptive field sizes, which they use on the larger Tiny Imagenet images.

The GPCN and BayesPCN achieved very low recall MSE on most masking tasks. MHN-grad and the MHN with Manhattan distance performed well on moderate masking, but failed completely on the high masking scenario. *The one and three level SQHN models performed perfectly on all masking tasks, while the two level performed nearly perfectly. SQHN models were the only models to match SoTA performance across both auto-associative and hetero-associative tasks.*

Online, Continual Auto-Association

Next, we test SQHNs on online, continual auto-association. The application for online learning algorithms are typically embedded learning systems (e.g., robots, sensing devices, etc.), where computationally and memory efficient algorithms are preferred. PCN networks are highly computationally expensive, requiring hundreds of neuron updates per training iteration (see [111, 139]), making them impractical as an online auto-associative memory

system. Thus, we compare SQHNs to the computationally efficient MHN model. However, we cannot perform the batch update that is typically used in auto-associative memory tests of MHNs. Instead, following previous work, (e.g., [52, 99]), we train MHNs with BP to reduce reconstruction error. As baseline comparisons, we train MHNs with BP/SGD and BP with an Adam optimizer. We also train with several compute-efficient algorithms common to continual learning: online elastic weight consolidation (EWC++) [19], which is a kind of regularized SGD, and episodic recall (ER) [20], which uses a small buffer to store a mini-batch of previously observed data points and SGD to update weights with the mini-batch.

We test on two kinds of online-continual auto-associative tasks: online class incremental (OCI) and online domain incremental (ODI). In both, data from each task is presented incrementally, one task at a time. In the OCI setting, each task consist images from the same class. In the ODI setting, there are four data sets, each are composed of visually distinct images (e.g., dataset 1 has bright images, dataset 2 has dim images, etc.). During training, models perform a single pass over each dataset, observing only a single data point each iteration, before switching to the next dataset. At testing, a noisy version of previously observed data are presented.

Performance is measured using recall MSE ($\mathcal{L}_{MSE}$) and following previous works [111, 139] recall accuracy:

$$\mathcal{A}^T = \frac{1}{T}\sum_{t=0}^{T} \mathbf{1}(\frac{1}{d}\|x^t - x^{t,new}\|^2 < \gamma), \tag{6.7}$$

where the indicator function $\mathbf{1}$ is one if the recall MSE is below threshold, γ, and zero otherwise. We also use a 'cumulative' (i.e., average) performance measure, which is common in online learning scenarios:

$$\mathcal{C}_{MSE} = \frac{1}{T}\sum_{t=0}^{T} \mathcal{L}^t_{MSE}, \qquad \mathcal{C}_{Acc} = \frac{1}{T}\sum_{t=0}^{T} \mathcal{A}^t, \tag{6.8}$$

where $\mathcal{L}^t_{MSE}$ and $\mathcal{A}^t$ is the recall MSE and recall accuracy, respectively, at iteration t given the model parameters and input data at iteration t. Cumulative measures are sensitive not just the final performance, but also to sample efficiency, i.e., how quickly the models improves performance. Finally, we use a novel measure of sensitivity to data ordering (S_{MSE}):

$$S_{MSE} = |\mathcal{C}^{OnCont}_{MSE} - \mathcal{C}^{On}_{MSE}| \tag{6.9}$$

This sensitivity measure is 0 when the model achieves the same cumulative MSE in the online (On) and online-continual (OnCont) settings, and increases as the performance differs. Models insensitive to ordering are highly useful in realistic scenarios where the way data is presented to the model is difficult to control and predict.

Results are shown in Figure 6.2. *SQHNs were highly insensitive to the ordering of the data in online-continual scenarios (Figure 6.2 C, top and bottom). They perform one-shot memorization until their capacity is reached, then their performance decays slowly (Figure 6.2 A, top and bottom), yielding very good cumulative performance (figure 6.2 B top and bottom). The BP-based MHNs learned too slowly to recall any data points and were much more sensitive to ordering, especially in the ODI setting. This lead to poor cumulative scores.* These results provide evidence the SQHN is a highly effective online-continual learner, especially in scenarios where fast learning is essential, and may have general benefits over SGD/BP based approaches.

Noisy Encoding

We tested SQHN models on a noisy encoding task, which is novel with respect to recent machine learning work on neural network based associative memory. In this task, each training iteration some number of noisy samples of an image are generated and presented to the network one at a time. Images are Gaussian (white noise added) or binary samples. The

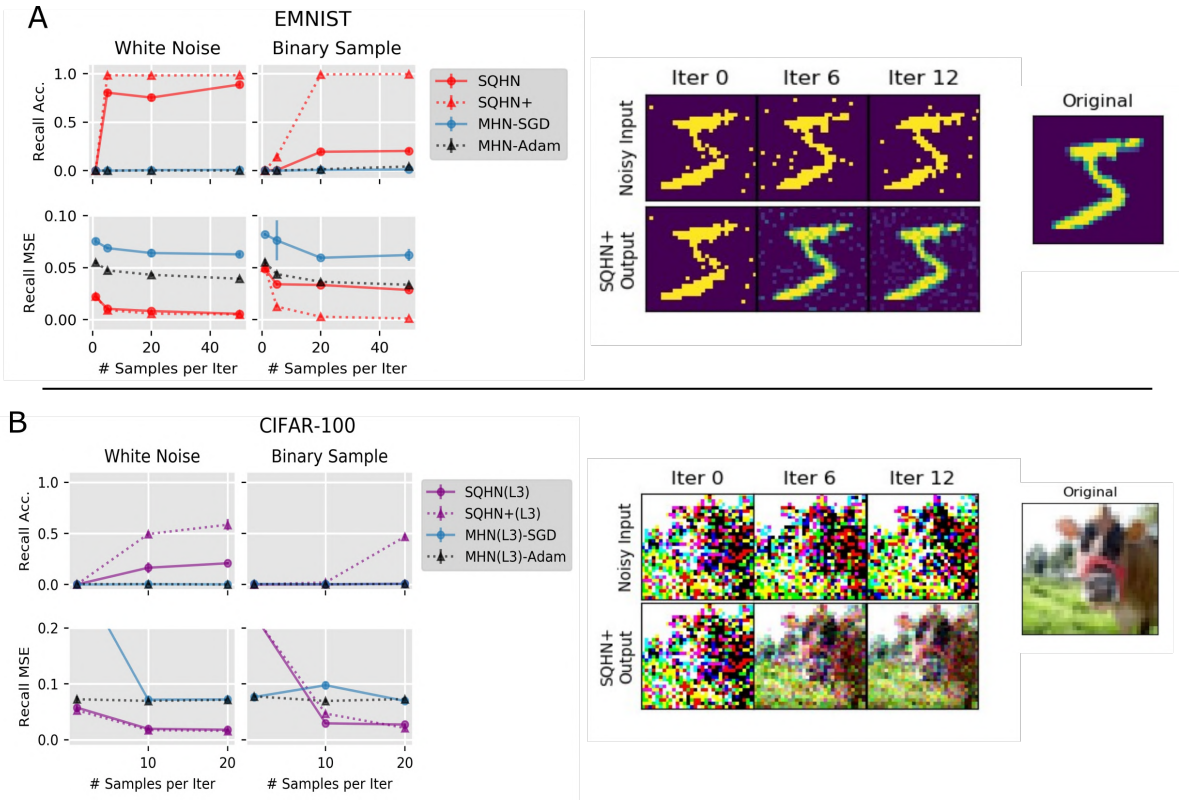

Figure 6.3: Noisy encoding task. **A.** Recall accuracy and recall MSE for one hidden layer models under white and noise and binary sample conditions. Example of the reconstruction during test time for SQHN+ model, in the case of 1, 6, and 12 samples. **B.** Recall accuracy and recall MSE for three hidden layer models under white and noise and binary sample conditions with reconstruction example for SQHN on the right.

models must encode the images, and reconstruct them at test time, when they are presented with the original non-corrupted versions. At no point during training is the model shown the non-corrupted version of the data. The purpose of this task is to specifically test the model's ability to demonstrate unsupervised learning in a noisy setting.

We compared the one and three hidden layer SQHN and MHN models on EMNIST and CIFAR-100, respectively. The one layer models had 300 hidden layer neurons, while the three hidden layer architectures had 150 neurons at each node. Networks were presented with 300 and 150 noisy images, respectively, so an inability to recall the images was more dependent on an inability to remove noise during learning rather than on capacity constraints.

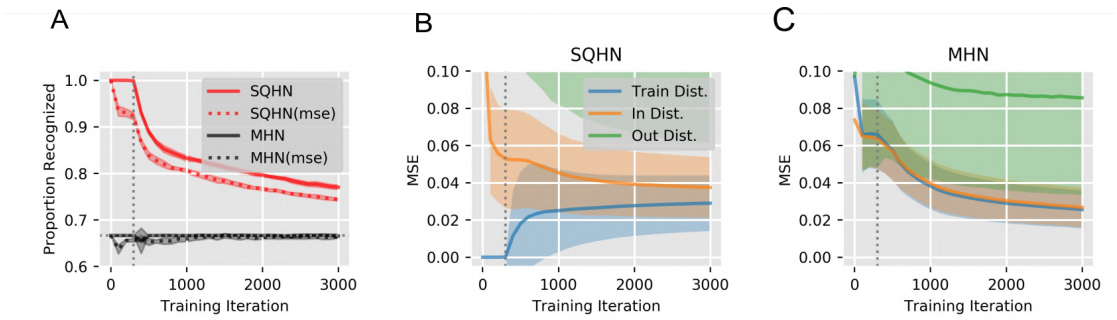

Figure 6.4: Recognition task. **A.** The recognition accuracy for SQHN and MHN are shown in networks with 300 neurons at the hidden layer (300th iteration marked by vertical dotted line). **B.** The MSE for the SQHN model on the training MNIST data (Train Dist.), hold-out MNIST data (In Dist.), and the F-MNIST data (Out Dist.) **C.** The MSEs for the MHN model.

In addition to testing the SQHN and MHN models, we tested an SQHN model with a slight alteration (SQHN+), where after the hidden states are computed for the first image sample, the hidden states are held fixed for the remaining duration of the training iteration. This ensures the latent code did not change during the encoding of the same image.

SQHN models, especially SQHN+, were highly effective at removing noise during learning (figure 6.3), *and both SQHN models significantly outperformed MHNs trained with BP and BP-Adam. This provides evidence the SQHN can be an effective learner under noise and outcompete similar BP-based models.*

Episodic Memory

Next, we tested models on a novel episodic memory task based on human and animal memory experiments, which we call episodic recognition. Recognition is hallmark ability of animal memory but has not, as far as we can find, been developed into a formalized machine learning test. In our task, models are presented with a sequence of training data points, X_{train}, in an online fashion. Afterward, the model is tested on a binary classification task, where the model must classify the data point presented as old (in the training set) or new (not from the

training set). The testing dataset is an equal portion of observed training images, X_{train}, new unobserved images from a related (in distribution) data set, X_{new-in}, and new images from an unrelated (out of distribution) data set, $X_{new-out}$. A high performing memory system will achieve significantly above chance accuracy, which will decay gracefully as the size of the data set increases.

Since there are no previous baselines to compare SQHN against for the episodic recognition task that we know of, we ran a simple comparison between an SQHN and MHN with single hidden layers (figure 6.4). Since MHN has not been used for an episodic recognition task, we created two methods for performing recognition in an MHN with one hidden layer. The first method uses the activities at the hidden layer as a measure of similarity to stored data points. The second method keeps a moving average of the recall MSE during training. If the hidden layer activity is above or if the MSE is below a threshold the model judges the data point is new. Grid search is used to set the threshold.

SQHN models performed perfectly until capacity is reached (vertical dotted line). Performance then decayed gradually, whereas MHN models were unable to do better than chance (figure 6.4 A). The performance differences seem due to the fact that SQHN models 'overfit' the training data (figure 6.4 B), early in training allowing it to recognize X_{new-in} as less probable than X_{old-in}. The MHN model on the other hand performed identically with old and new in-distribution data. This allowed MHN to generalize better earlier, but it prevented the MHN from being able to distinguish previously observed data points from similar data points that were not present in the training distribution.

6.2.4 Further Comparison of SQHN Architectures

Finally, we did a thorough comparison of SQHN models that have one, two, or three hidden layers (figure 6.5 A), to better establish what the advantages and disadvantages are of adding

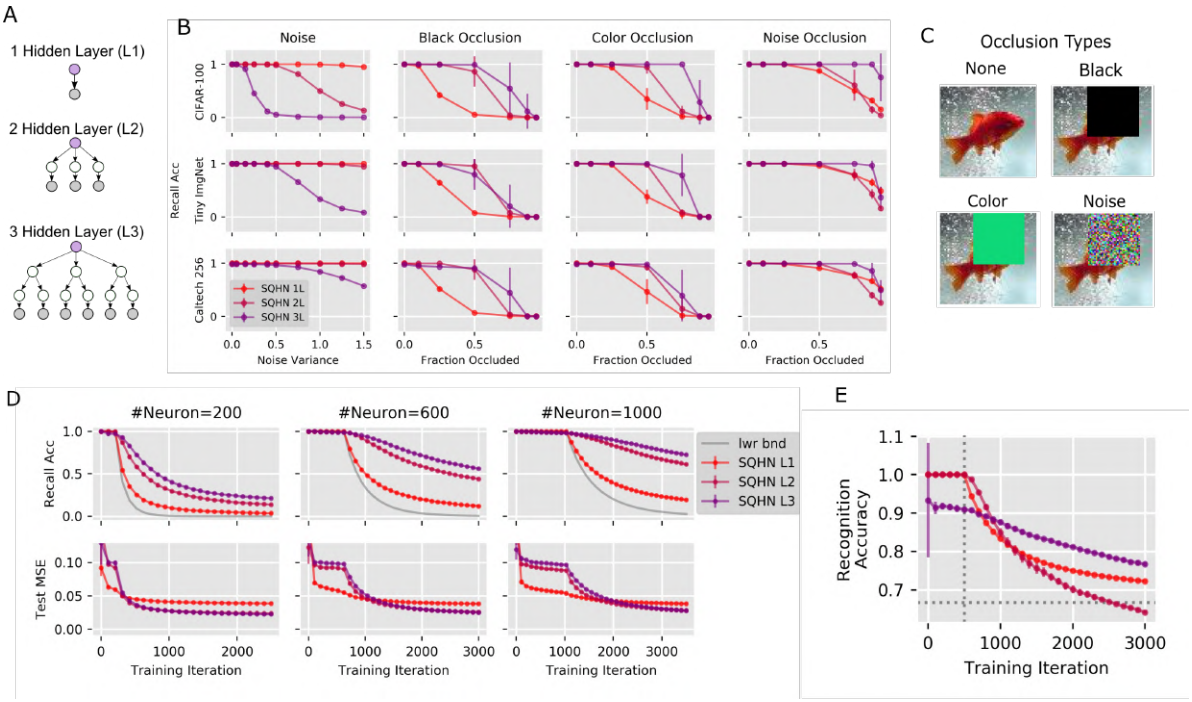

Figure 6.5: Comparison of SQHN architectures with different hidden layers. **A.** Depiction of various SQHN architectures. **B.** Recall accuracy across three data sets (CIFAR-100, Tiny Imagenet, Caltech 256) under the white noise and several occlusion scenarios. **C.** Visualization of the black, color, and noise occlusions. **D.** Recall accuracy during online training without noise (top row) and MSE on a test a test set (bottom row). The lower bound on recall accuracy posited by theorem C.2 (supp. C.2.7) marked by gray line. Models tested with different maximum number of neurons per node (200, 600, 1000). **E.** Recognition accuracy for three SQHN models, where 500 neurons are allocated to each node (vertical dotted line marks when all 500 neurons are grown). CIFAR-100 data used for train and in-distribution set, while a flipped pixel version of the Street View House Numbers (SVHN) dataset used for out of distribution. Best guessing strategy yields 66% accuracy (horizontal dotted line).

more hidden layers to the SQHN.

First, we tested how sensitive SQHN models are to corruption in several auto-association tasks (figure 6.5 B). During training each model memorizes 1000 images from the CIFAR-100, TinyImageNet, or Caltech256 datasets. During testing images are either corrupted with white noise or, what we call, an occlusion. In the occlusion scenario, pixels in a rectangular region of random shape and position are set equal to either 0 (black occlusion), a random color (color occlusion), or white noise (noise occlusion). (Note this is distinct from masking,

since models treat pixels as corrupted rather than missing.) Results are in figure 6.5 B. All SQHN models performed recall perfectly when corruption was small. Adding more hidden layers tended to improve performance on occlusion tasks, likely because trees represent data as a part-whole hierarchies and therefore can better ignore corrupted parts of the input during inference. Architectures whose bottom hidden layer nodes had larger receptive fields performed better on the noise task. The one layer SQHN had the largest receptive field so it performed the best. For the largest images (CalTech256), however, receptive field sizes at the bottom layers were large for all models (minimum 8x8), and all models performed similarly on noisy recall.

Next, we observed auto-associative recall performance during online (i.i.d.) learning for SQHN models with different numbers of layers and node sizes (figure 6.5 D). All SQHN models performed one-shot memorization until minimum capacity is reached (which is the number of neurons at hidden nodes, see theorem 1 supp. C.2.7). Deeper SQHN's recall accuracy decayed at a much slower rate. We suspected this was due to their ability to reuse primitive feature representations at lower nodes to generalize better across multiple data points. We tested these models on test/hold-out data and indeed found the tree architectures generalized better to new data than the one hidden layer model.

Finally, we compared SQHN models on the episodic recognition task. (figure 6.5 E). Models had 500 neurons at each node. The X_{train} data were CIFAR-10 train images, X_{new-in} were CIFAR-10 images from a hold/out test set, and $X_{new-out}$ were images from the the SVHN dataset with flipped pixels. Figure 6.5 (E) shows all SQHN models were able to perform well above chance accuracy (66%, horizontal dotted line) even when far more training data was presented than the number of neurons at the memory node.

In sum, *adding more levels to tree structured SQHNs significantly improves auto-associative recall with occlusion, slows decay of recall accuracy after the network hits capacity, and improves generalization without significant loss in recognition ability. While adding levels*

can make the SQHN more sensitive to noise, this seems limited to small images.

6.3 Discussion

Artificial neural networks were originally designed to mimic the way biological neural circuits process information [71]. The way biological circuits *learn* to process information, however, is an open question. In particular, it is unknown how the brain uses local learning rules effectively in noisy, online-continual settings. In this chapter, we proposed a general local learning approach to the basic task of storing and retrieving patterns in noisy, online-continual scenarios. We proposed using a sparse, quantized neural code to deal with noisy and partial inputs and to prevent catastrophic forgetting, and implementing this strategy via a discrete graphical model that performed MAP learning, an algorithm that uses local learning rules. We implemented this approach in the novel SQHN model.

Our results support the effectiveness of our approach and the SQHN. First, we found the quantized neural code of the SQHN was advantageous in auto-associative recall over similar models that use a continuous latent codes. In particular, PC models, like the SQHN, implement directed graphical models and learn via MAP learning [78]. However, because PC models use a continuous latent code, they were much more sensitive to noise than SQHN models. This lends credence to the idea that using a quantized neural code helps significantly with auto-association. MHN models performed similarly to SQHNs on the noise auto-association task. This is likely because under the hyper-parameter settings that yielded the best performance, these MHNs essentially implemented a discrete latent code (supplementary C.2.8). However, the operation MHNs use to map inputs to latent codes, is not as effective as that of SQHNs in the high masking settings.

Second, we found SQHNs significantly outperformed similar MHN architectures trained

with BP-based algorithms on online-continual learning tasks. The SQHN trained faster, demonstrated one-shot memorization, showed only a small, stable forgetting rate, and was largely insensitive to ordering. It achieved all of this while using little extra memory, no episodic memory buffer, and little compute. Part of these performance advantages may be attributable to the parameter isolation approach generally. However, the SQHN also uses an effective learning rate schedule to prevent forgetting (fig.C.2), and we also find mathematically that using MAP inference to set the one-hot values at hidden nodes is a principled way of deciding which parameters to update. In particular, it yields a set of activities and updates that require only a small change to existing parameters (supp. C.2.7). This suggests the exciting prospect that MAP learning in discrete models, like SQHNs, provides a justified and bio-plausible way to perform parameter isolation, which unlike previous parameter isolation methods (e.g., [58, 140, 66, 65]), does not require the computation of non-local global loss gradients to decide how to isolate parameters.

Third, the sparse quantized code reduced the negative effects of noise during the noisy encoding process by yielding a hidden latent code that was largely stable across noisy samples. If we fixed the latent code to be perfectly stable, as we did with SQHN+, then performance improves even more. The sparse updates also helped prevent noise from interfering with previously recorded memories.

Finally, the SQHN proved to be highly effective in the episodic recognition task. The memory node of the SQHN stored an explicit, itemized record of previously observed feature representations of input data (see supp. C.2.6). During recall the memory node performs a nearest neighbor operation by finding the item with the highest energy. This operation turned out to yield a straightforward method for detecting new versus old data points, is similar to classic cognitive models of episodic recognition (e.g., [116]), and it performed well even when the memory node was pushed past its capacity. Models like the MHN trained with BP, on the other hand, did not naturally learn an explicit record of previously observed

feature prototypes and struggled to distinguish new from similar, old data points.

Importantly, SQHN's energy function yields a straightforward way to implement a directed graphical model with largely standard artificial neural network operations. In particular, since the energy was a sum of probabilities, rather than a product, neurons in the SQHN architecture summed inputs from children and parent nodes rather than multiplied (as belief networks do [17]), yielding standard neural network operations which are easily scaled and more compatible with hardware built specifically to handle vector matrix multiplies. SQHN does so without the need to move probabilities to the log domain, which can sometimes yield unusual properties like the need to represent very large negative numbers (i.e., overflow issues). The energy can also be justified as an approximation to the joint distribution when uncertainty (i.e., a prior distribution) is placed over parameters (supp. C.2.2). Taken together these results suggest that our general approach, and the SQHN implementation of it, could provide a highly promising basis for building neuromorphic online-continual learners.

It is also interesting to note the striking similarities between SQHNs and models of memory from neuroscience. In particular, in addition to utilizing sparse codes and local learning rules like the brain, the memory node of the SQHNs has similarities to hippocampus (HP). HP is a region of the brain closely tied to episodic memory. We showed in our experiments the memory node, like the HP, is highly effective for episodic recognition. Further, the HP, like the memory node in SQHNs, is often proposed to be at the top of the cortical hierarchy [72], and some theories propose a central function of the HP is to retrieve and return previously stored patterns given partial, noisy inputs from cortex [124, 125]. This is precisely what the memory node of SQHNs does (see figure 6.1). Maybe most interesting, we find that on average neurons grow more rapidly and for longer periods in the memory node than the rest of the network and synapses in the memory node tend to have larger step sizes (more flexibility) on average (fig. C.3). This is highly consistent with observations that HP grows neurons into adulthood while the rest of the cortex stops in early adulthood

[80], and the observation that synapses are highly flexible in HP compared to the rest of the cortex [53]. Importantly, the SQHN was not engineered to have these properties. Rather these properties emerged from the SQHN learning algorithm as it solved the online-continual learning problem.

Future work will need to assess whether and how the SQHN provides possible new ideas or insights into these topics in neuroscience. Further, on the machine learning side, future work will need to assess the SQHN on tasks that require generalization. Although we found SQHNs with simple tree architectures to be highly efficient at storing and retrieving training data, and preventing forgetting, we believe somewhat more complicated SQHN architectures will be needed for high performance on tasks like classification or self-supervised learning where generalizing to new data points is needed. Nonetheless, the results here suggest SQHNs could provide a promising approach for learning these tasks in the online-continual setting.

Chapter 7

Conclusions and Future Directions

This dissertation presented and synthesized recent work by myself and my collaborators on EBAs, a class of learning algorithms for ANNs that is more bio-constrained than the standard BP/SGD approach. There were two main motivations for taking a more bio-constrained approach to training ANNS: such an approach could 1) lead to new learning algorithms with performance benefits over BP/SGD, and 2) lead to new theories of how learning works in the brain. Let's consider each of these hypotheses in turn.

First, do EBAs seem to have potential performance benefits of BP/SGD? The recent AI boom was driven by the paradigm of offline training with BP/SGD on GPUs. Although this paradigm has produced highly impressive AI models, the limits of this approach are now being recognized. It costs, for example, tens of millions of dollar in energy costs to train the largest AI models in this paradigm. Further, without a good solution to the online-continual learning scenario, autonomous systems, like self-driving cars, are unable to dynamically react and learn to new environments not seen during pre-training, leading to severe failures when deployed [51]. In this dissertation, we presented results that suggested that, unlike BP, EBAs naturally use local learning rules, are sensitive to second-order information, are well-

suited for online learning, and can be used to develop principled, highly effective continual learners. These performance benefits could be helpful in several applications areas where BP/SGD struggles. For example, the locality of EBA learning rules could make them better suited for energy-efficient neuromorphic hardware [87].[1] Their natural ability for online and continual learning may also make EBAs well-suited for edge computing devices, like robots, sensing devices, and wearables. EBAs, therefore, provide a promising research direction that could lead to embedded AI systems that are more animal-like in their energy-efficiency and cognitive adaptability to real-world environments than standard BP/SGD-based approaches.

Second, do EBAs provide a promising hypothesis for how the brain learns? Although my dissertation did not directly contribute to answering this question directly, recent work has directly investigated it. In particular, a notable work by Song et al. [121] provided a mechanistic description of how EBAs work different than BP/SGD: BP first updates weights, then recomputes new more desirable neuron values using the new weights, whereas EBAs first compute the desired neuron values then update weights to consolidate these values. This description was then used to make different empirical predictions about how plasticity in the brain should work and found that the mechanics of EBAs better fit with a variety of empirical evidence than the mechanics BP.

Despite these promising results, there are areas where EBAs need more work in the near future. Although more empirical work needs to be down to test the hypothesis that the brain uses an EBA, my future work will focus on areas where EBAs can be improved in machine learning:

- **Theory**: Although the proximal algorithm interpretation of PC seems more accurate than the BP/SGD interpretation, more work needs to be done to see how well this interpretation extends to all EBAs and whether there is a kind of proximal al-

[1]Neuromorphic hardware is a kind hardware inspired by the brain's architecture which can be orders of magnitude more energy efficient than standard Von Neumann architectures. Typically, in order for ANNs to learn while implemented on a neuromorphic chip local learning rules must be used (e.g., see [98])

gorithm description that makes fewer approximating assumptions than those made by [8]. I suggested one possible direction that seems promising, which a novel proximal/regularization term that measures the distance between parameters in energy space, but this idea needs to be more rigorously fleshed out.

- **Scaling**: EBAs have yet to be scaled to architectures as large as those trained with BP. It remains to be seen, for example, what the scaling laws are for EBAs and what sorts of architectures are (and are not) well suited for EBAs at larger scales. It may be that those architectures that work well with BP do not do so with EBAs at scale and vice versa. Further, developments to optimizers, like our MQ optimizer could also further aid this direction.

- **Online-Continual Learning** Our SQHN model was one of the first, as far as I know, that used a novel EBA algorithm designed specifically for online continual learning. This suggests there are potential improvements that could be made in the future to the model, and new sparse, dynamic architectures, like the SQHN, coupled with new EBA learning rules could be a very promising future for developing ANN that learn effectively in online-continual scenarios.

Longer term, I believe work on EBAs and the field of deep learning more generally could benefit from thinking more deeply about *self-organization*. Self-organizing systems, which include all biological organisms, are those that display global patterns of behavior through local interactions among their parts [118]. In contrast to self-organizing systems are those that display some global pattern by way of an external controller providing explicit instructions for how to organize, e.g., a 3D printer producing a particular object via an inputted blueprint. EBAs fit the description of self-organization, as they use local learning rules to optimize an global loss function, and their local learning signals are provided by the internal dynamics of the system. BP-based algorithms, on the other hand, although they may use rules that are local in some sense, always requires some external program or segregated

circuits to provide explicit instruction for how to change parameters.

However, there is much more to self-organizing neural systems than altering existing synapses. Neurons and synapses are also grown and organize into specific structures during development, and do so without the help of external controllers instructing them how to grow. The SQHN network we designed did this to an extent and it proved to be important for the results we obtained: if the network weights were randomly initialized rather than grown the network did not perform nearly as well C.2. However, much of the architecture, e.g., the number of nodes and the graph structure, were pre-designed. Principled methods that steer growth and development of an entire ANN architectures, including the number of nodes and structures, as well as its learning, using a principled EBA method could yield networks even better suited for learning in the real-world. In this case, network architectures would not be pre-designed. Rather, *local development programs* would be pre-designed and used to grow and develop the architectures in response to their inputs. In any case, there is clearly many more promsing avenues for designing bio-constrained EBAs to help solve some of the deepest problems in AI, and these results could, in turn, give us a deeper understanding of the brain.

Bibliography

[1] W. C. Abraham and A. Robins. Memory retention–the synaptic stability versus plasticity dilemma. *Trends in neurosciences*, 28(2):73–78, 2005.

[2] R. A. Adams, S. Shipp, and K. J. Friston. Predictions not commands: active inference in the motor system. *Brain Structure and Function*, 218(3):611–643, 2013.

[3] S. Ahmad and J. Hawkins. Properties of sparse distributed representations and their application to hierarchical temporal memory. *arXiv preprint arXiv:1503.07469*, 2015.

[4] R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, and T. Tuytelaars. Memory aware synapses: Learning what (not) to forget. In *Proceedings of the European conference on computer vision (ECCV)*, pages 139–154, 2018.

[5] R. Aljundi, M. Lin, B. Goujaud, and Y. Bengio. Gradient based sample selection for online continual learning. *Advances in neural information processing systems*, 32, 2019.

[6] N. Alonso and J. Krichmar. A sparse quantized hopfield network for online-continual memory. *arXiv preprint arXiv:2307.15040*, 2023.

[7] N. Alonso, J. Krichmar, and E. Neftci. Understanding and improving optimization in predictive coding networks. *arXiv preprint arXiv:2305.13562*, 2023.

[8] N. Alonso, B. Millidge, J. Krichmar, and E. O. Neftci. A theoretical framework for inference learning. *Advances in Neural Information Processing Systems*, 35:37335–37348, 2022.

[9] N. Alonso and E. Neftci. Tightening the biological constraints on gradient-based predictive coding. In *International Conference on Neuromorphic Systems 2021*, pages 1–9, 2021.

[10] S. Bartunov, A. Santoro, B. A. Richards, L. Marris, G. E. Hinton, and T. Lillicrap. Assessing the scalability of biologically-motivated deep learning algorithms and architectures. *arXiv preprint arXiv:1807.04587*, 2018.

[11] A. M. Bastos, W. M. Usrey, R. A. Adams, G. R. Mangun, P. Fries, and K. J. Friston. Canonical microcircuits for predictive coding. *Neuron*, 76(4):695–711, 2012.

[12] H. M. Bayer and P. W. Glimcher. Midbrain dopamine neurons encode a quantitative reward prediction error signal. *Neuron*, 47(1):129–141, 2005.

[13] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein, and W. Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature communications*, 11(1):3625, 2020.

[14] Y. Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

[15] Y. Bengio. Deriving differential target propagation from iterating approximate inverses. *arXiv preprint arXiv:2007.15139*, 2020.

[16] Y. Bengio and A. Fischer. Early inference in energy-based models approximates back-propagation. *arXiv preprint arXiv:1510.02777*, 2015.

[17] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[18] R. Bogacz. A tutorial on the free-energy framework for modelling perception and learning. *Journal of mathematical psychology*, 76:198–211, 2017.

[19] A. Chaudhry, P. K. Dokania, T. Ajanthan, and P. H. Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *Proceedings of the European conference on computer vision (ECCV)*, pages 532–547, 2018.

[20] A. Chaudhry, M. Rohrbach, M. Elhoseiny, T. Ajanthan, P. K. Dokania, P. H. Torr, and M. Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

[21] H. Cho, C. W. Lee, and S. W. Kim. Derivation of a new normalized least mean squares algorithm with modified minimization criterion. *Signal Processing*, 89(4):692–695, 2009.

[22] A. Choromanska, B. Cowen, S. Kumaravel, R. Luss, M. Rigotti, I. Rish, P. Diachille, V. Gurev, B. Kingsbury, R. Tejwani, et al. Beyond backprop: Online alternating minimization with auxiliary variables. In *International Conference on Machine Learning*, pages 1193–1202. PMLR, 2019.

[23] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive aggressive algorithms. 2006.

[24] K. Crammer, A. Kulesza, and M. Dredze. Adaptive regularization of weight vectors. *Advances in neural information processing systems*, 22, 2009.

[25] F. Crick. The recent excitement about neural networks. *Nature*, 337(6203):129–132, 1989.

[26] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

[27] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.

[28] M. Ernoult, F. Normandin, A. Moudgil, S. Spinney, E. Belilovsky, I. Rish, B. Richards, and Y. Bengio. Towards scaling difference target propagation by learning backprop targets. *arXiv preprint arXiv:2201.13415*, 2022.

[29] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[30] T. Frerix, T. Möllenhoff, M. Moeller, and D. Cremers. Proximal backpropagation. *arXiv preprint arXiv:1706.04638*, 2017.

[31] K. Friston. A theory of cortical responses. *Philosophical transactions of the Royal Society B: Biological sciences*, 360(1456):815–836, 2005.

[32] K. Friston. Hierarchical models in the brain. *PLoS computational biology*, 4(11):e1000211, 2008.

[33] K. Friston. The free-energy principle: a unified brain theory? *Nature reviews neuroscience*, 11(2):127–138, 2010.

[34] K. Friston and S. Kiebel. Predictive coding under the free-energy principle. *Philosophical transactions of the Royal Society B: Biological sciences*, 364(1521):1211–1221, 2009.

[35] J. Gallardo, T. L. Hayes, and C. Kanan. Self-supervised training enhances online continual learning. *arXiv preprint arXiv:2103.14010*, 2021.

[36] D. George, W. Lehrach, K. Kansky, M. Lázaro-Gredilla, C. Laan, B. Marthi, X. Lou, Z. Meng, Y. Liu, H. Wang, et al. A generative vision model that trains with high data efficiency and breaks text-based captchas. *Science*, 358(6368):eaag2612, 2017.

[37] S. J. Gershman and D. M. Blei. A tutorial on bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[38] W. Gerstner and W. M. Kistler. Mathematical formulations of hebbian learning. *Biological cybernetics*, 87(5):404–415, 2002.

[39] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[40] J. Guerguiev, T. P. Lillicrap, and B. A. Richards. Towards deep learning with segregated dendrites. *Elife*, 6:e22901, 2017.

[41] T. L. Hayes and C. Kanan. Online continual learning for embedded devices. *arXiv preprint arXiv:2203.10681*, 2022.

[42] D. Heckerman. *A tutorial on learning with Bayesian networks*. Springer, 1998.

[43] G. E. Hinton. Boltzmann machine. *Scholarpedia*, 2(5):1668, 2007.

[44] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[45] F. Innocenti, R. Singh, and C. L. Buckley. Understanding predictive coding as an adaptive trust-region method. *arXiv preprint arXiv:2305.18188*, 2023.

[46] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *International conference on machine learning*, pages 1627–1635. PMLR, 2017.

[47] K. Khetarpal, M. Riemer, I. Rish, and D. Precup. Towards continual reinforcement learning: A review and perspectives. *Journal of Artificial Intelligence Research*, 75:1401–1476, 2022.

[48] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[50] D. C. Knill and A. Pouget. The bayesian brain: the role of uncertainty in neural coding and computation. *TRENDS in Neurosciences*, 27(12):712–719, 2004.

[51] J. L. Krichmar, W. Severa, M. S. Khan, and J. L. Olds. Making bread: biomimetic strategies for artificial intelligence now and in the future. *Frontiers in neuroscience*, 13:666, 2019.

[52] D. Krotov and J. J. Hopfield. Dense associative memory for pattern recognition. *Advances in neural information processing systems*, 29, 2016.

[53] D. Kumaran, D. Hassabis, and J. L. McClelland. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7):512–534, 2016.

[54] A. Laborieux, M. Ernoult, B. Scellier, Y. Bengio, J. Grollier, and D. Querlioz. Scaling equilibrium propagation to deep convnets by drastically reducing its gradient estimator bias. *Frontiers in neuroscience*, 15:129, 2021.

[55] T. T.-K. Lau, J. Zeng, B. Wu, and Y. Yao. A proximal block coordinate descent algorithm for deep neural network training. *arXiv preprint arXiv:1803.09082*, 2018.

[56] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28, 1988.

[57] D.-H. Lee, S. Zhang, A. Fischer, and Y. Bengio. Difference target propagation. In *Joint european conference on machine learning and knowledge discovery in databases*, pages 498–515. Springer, 2015.

[58] S. Lee, J. Ha, D. Zhang, and G. Kim. A neural dirichlet process mixture model for task-free continual learning. *arXiv preprint arXiv:2001.00689*, 2020.

[59] Y. Li, E. Schofield, and M. Gönen. A tutorial on dirichlet process mixture modeling. *Journal of mathematical psychology*, 91:128–144, 2019.

[60] Z. Li and D. Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

[61] Q. Liao, J. Leibo, and T. Poggio. How important is weight symmetry in backpropagation? In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[62] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7(1):1–10, 2016.

[63] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, 2020.

[64] Z. Mai, R. Li, J. Jeong, D. Quispe, H. Kim, and S. Sanner. Online continual learning in image classification: An empirical survey. *Neurocomputing*, 469:28–51, 2022.

[65] A. Mallya, D. Davis, and S. Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82, 2018.

[66] A. Mallya and S. Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[67] A. H. Marblestone, G. Wayne, and K. P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in computational neuroscience*, 10:94, 2016.

[68] J. Marino. Predictive coding, variational autoencoders, and biological connections. *Neural Computation*, 34(1):1–44, 2022.

[69] S. Matsuda. Quantized hopfield networks for integer programming. *Systems and computers in Japan*, 30(6):1–12, 1999.

[70] S. Matsuda. Theoretical analysis of quantized hopfield network for integer programming. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*, volume 1, pages 568–571. IEEE, 1999.

[71] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[72] B. L. McNaughton. Cortical hierarchies, sleep, and the extraction of knowledge from memory. *Artificial Intelligence*, 174(2):205–214, 2010.

[73] T. Meinhardt, M. Moller, C. Hazirbas, and D. Cremers. Learning proximal operators: Using denoising networks for regularizing inverse imaging problems. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1781–1790, 2017.

[74] A. Meulemans, F. S. Carzaniga, J. A. Suykens, J. Sacramento, and B. F. Grewe. A theoretical framework for target propagation. *arXiv preprint arXiv:2006.14331*, 2020.

[75] B. Millidge, T. Salvatori, Y. Song, T. Lukasiewicz, and R. Bogacz. Universal hopfield networks: A general framework for single-shot associative memory models. *arXiv preprint arXiv:2202.04557*, 2022.

[76] B. Millidge, A. Seth, and C. L. Buckley. Predictive coding: a theoretical and experimental review. *arXiv preprint arXiv:2107.12979*, 2021.

[77] B. Millidge, Y. Song, T. Salvatori, T. Lukasiewicz, and R. Bogacz. Backpropagation at the infinitesimal inference limit of energy-based models: Unifying predictive coding, equilibrium propagation, and contrastive hebbian learning. *arXiv preprint arXiv:2206.02629*, 2022.

[78] B. Millidge, Y. Song, T. Salvatori, T. Lukasiewicz, and R. Bogacz. A theoretical framework for inference and learning in predictive coding networks. *arXiv preprint arXiv:2207.12316*, 2022.

[79] B. Millidge, A. Tschantz, and C. L. Buckley. Predictive coding approximates backprop along arbitrary computation graphs. *arXiv preprint arXiv:2006.04182*, 2020.

[80] G.-l. Ming and H. Song. Adult neurogenesis in the mammalian brain: significant answers and significant questions. *Neuron*, 70(4):687–702, 2011.

[81] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[82] E. Moroshko, N. Vaits, and K. Crammer. Second-order non-stationary online learning for regression. *J. Mach. Learn. Res.*, 16:1481–1517, 2015.

[83] J. R. Movellan. Contrastive hebbian learning in the continuous hopfield model. In *Connectionist models*, pages 10–17. Elsevier, 1991.

[84] J.-I. Nagumo and A. Noda. A learning method for system identification. *IEEE Transactions on Automatic Control*, 12(3):282–287, 1967.

[85] R. M. Neal and G. E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. *Learning in graphical models*, pages 355–368, 1998.

[86] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis. Event-driven random back-propagation: Enabling neuromorphic deep learning machines. *Frontiers in neuroscience*, 11:324, 2017.

[87] E. O. Neftci, H. Mostafa, and F. Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

[88] Y. Nesterov and B. T. Polyak. Cubic regularization of newton method and its global performance. *Mathematical Programming*, 108(1):177–205, 2006.

[89] A. Nøkland. Direct feedback alignment provides learning in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[90] M. Opper and O. Winther. A bayesian approach to on-line learning. 1999.

[91] R. C. O'Reilly. Biologically plausible error-driven learning using local activation differences: The generalized recirculation algorithm. *Neural computation*, 8(5):895–938, 1996.

[92] R. C. O'Reilly, D. R. Wyatte, and J. Rohrlich. Deep predictive learning: a comprehensive model of three visual streams. *arXiv preprint arXiv:1709.04654*, 2017.

[93] A. G. Ororbia and A. Mali. Biologically motivated algorithms for propagating local target representations. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4651–4658, 2019.

[94] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[95] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

[96] G. I. Parisi and V. Lomonaco. Online continual learning on sequences. In *Recent Trends in Learning From Data: Tutorials from the INNS Big Data and Deep Learning Conference (INNSBDDL2019)*, pages 197–221. Springer, 2020.

[97] L. Qiao, T. Sun, H. Pan, and D. Li. Inertial proximal deep learning alternating minimization for efficient neutral network training. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3895–3899. IEEE, 2021.

[98] B. Rajendran, A. Sebastian, M. Schmuker, N. Srinivasa, and E. Eleftheriou. Low-power neuromorphic hardware for signal processing applications: A review of architectural and system-level design approaches. *IEEE Signal Processing Magazine*, 36(6):97–110, 2019.

[99] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

[100] R. P. Rao and D. H. Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87, 1999.

[101] B. A. Richards and T. P. Lillicrap. Dendritic solutions to the credit assignment problem. *Current opinion in neurobiology*, 54:28–36, 2019.

[102] B. A. Richards, T. P. Lillicrap, P. Beaudoin, Y. Bengio, R. Bogacz, A. Christensen, C. Clopath, R. P. Costa, A. de Berker, S. Ganguli, et al. A deep learning framework for neuroscience. *Nature neuroscience*, 22(11):1761–1770, 2019.

[103] H. Ritter, A. Botev, and D. Barber. Online structured laplace approximations for overcoming catastrophic forgetting. *Advances in Neural Information Processing Systems*, 31, 2018.

[104] R. Rosenbaum. On the relationship between predictive coding and backpropagation. *Plos one*, 17(3):e0266102, 2022.

[105] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen. Sparse coding via thresholding and local competition in neural circuits. *Neural computation*, 20(10):2526–2563, 2008.

[106] M. D. Rugg and A. P. Yonelinas. Human recognition memory: a cognitive neuroscience perspective. *Trends in cognitive sciences*, 7(7):313–319, 2003.

[107] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[108] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[109] J. Sacramento, R. Ponte Costa, Y. Bengio, and W. Senn. Dendritic cortical microcircuits approximate the backpropagation algorithm. *Advances in neural information processing systems*, 31, 2018.

[110] T. Salvatori, L. Pinchetti, B. Millidge, Y. Song, R. Bogacz, and T. Lukasiewicz. Learning on arbitrary graph topologies via predictive coding. *arXiv preprint arXiv:2201.13180*, 2022.

[111] T. Salvatori, Y. Song, Y. Hong, L. Sha, S. Frieder, Z. Xu, R. Bogacz, and T. Lukasiewicz. Associative memories via predictive coding. *Advances in Neural Information Processing Systems*, 34, 2021.

[112] T. Salvatori, Y. Song, B. Millidge, Z. Xu, L. Sha, C. Emde, R. Bogacz, and T. Lukasiewicz. Incremental predictive coding: A parallel and fully automatic learning algorithm. *arXiv preprint arXiv:2212.00720*, 2022.

[113] B. Scellier and Y. Bengio. Equilibrium propagation: Bridging the gap between energy-based models and backpropagation. *Frontiers in computational neuroscience*, 11:24, 2017.

[114] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19, 2022.

[115] S. Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[116] R. M. Shiffrin and M. Steyvers. A model for recognition memory: Rem—retrieving effectively from memory. *Psychonomic bulletin & review*, 4:145–166, 1997.

[117] H. Shin, J. K. Lee, J. Kim, and J. Kim. Continual learning with deep generative replay. *Advances in neural information processing systems*, 30, 2017.

[118] J. Sneyd, G. Theraula, E. Bonabeau, J.-L. Deneubourg, and N. R. Franks. *Self-organization in biological systems*. Princeton university press, 2001.

[119] S. Song, K. D. Miller, and L. F. Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926, 2000.

[120] Y. Song, T. Lukasiewicz, Z. Xu, and R. Bogacz. Can the brain do backpropagation? exact implementation of backpropagation in predictive coding networks. *Advances in neural information processing systems*, 33:22566, 2020.

[121] Y. Song, B. G. Millidge, T. Salvatori, T. Lukasiewicz, Z. Xu, and R. Bogacz. Inferring neural activity before plasticity: A foundation for learning beyond backpropagation. *bioRxiv*, 2022.

[122] M. W. Spratling. Predictive coding as a model of biased competition in visual attention. *Vision research*, 48(12):1391–1408, 2008.

[123] Stork. Is backpropagation biologically plausible? In *International 1989 Joint Conference on Neural Networks*, pages 241–246. IEEE, 1989.

[124] T. J. Teyler and P. DiScenna. The hippocampal memory indexing theory. *Behav Neurosci*, 100(2):147–54, 1986.

[125] T. J. Teyler and J. W. Rudy. The hippocampal indexing theory and episodic memory: updating the index. *Hippocampus*, 17(12):1158–69, 2007.

[126] P. Toulis, E. Airoldi, and J. Rennie. Statistical analysis of stochastic gradient methods for generalized linear models. In *International Conference on Machine Learning*, pages 667–675. PMLR, 2014.

[127] P. Toulis and E. M. Airoldi. Implicit stochastic gradient descent for principled estimation with large datasets. *ArXiv e-prints*, 2014.

[128] P. Toulis and E. M. Airoldi. Stochastic gradient methods for principled estimation with large datasets., 2016.

[129] P. Toulis and E. M. Airoldi. Asymptotic and finite-sample properties of estimators based on stochastic gradients. *The Annals of Statistics*, 45(4):1694–1727, 2017.

[130] M. Varadi, S. Anyango, M. Deshpande, S. Nair, C. Natassia, G. Yordanova, D. Yuan, O. Stroe, G. Wood, A. Laydon, et al. Alphafold protein structure database: massively expanding the structural coverage of protein-sequence space with high-accuracy models. *Nucleic acids research*, 50(D1):D439–D444, 2022.

[131] L. Wang, X. Zhang, H. Su, and J. Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023.

[132] J. C. Whittington and R. Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation*, 29(5):1229–1262, 2017.

[133] J. C. Whittington and R. Bogacz. Theories of error back-propagation in the brain. *Trends in cognitive sciences*, 23(3):235–250, 2019.

[134] W. Xiao, H. Chen, Q. Liao, and T. Poggio. Biologically-plausible learning algorithms can scale to large datasets. *arXiv preprint arXiv:1811.03567*, 2018.

[135] X. Xie and H. S. Seung. Equivalence of backpropagation and contrastive hebbian learning in a layered network. *Neural computation*, 15(2):441–454, 2003.

[136] D. Yang and J. Sun. Proximal dehaze-net: A prior learning-based deep network for single image dehazing. In *Proceedings of the european conference on computer vision (ECCV)*, pages 702–717, 2018.

[137] J. Yang, K. Zhou, Y. Li, and Z. Liu. Generalized out-of-distribution detection: A survey. *arXiv preprint arXiv:2110.11334*, 2021.

[138] B. Yin, F. Corradi, and S. M. Bohté. Accurate online training of dynamical spiking neural networks through forward propagation through time. *Nature Machine Intelligence*, pages 1–10, 2023.

[139] J. Yoo and F. Wood. Bayespcn: A continually learnable predictive coding associative memory. *Advances in Neural Information Processing Systems*, 35:29903–29914, 2022.

[140] J. Yoon, E. Yang, J. Lee, and S. J. Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

[141] F. Zenke, B. Poole, and S. Ganguli. Continual learning through synaptic intelligence. In *International conference on machine learning*, pages 3987–3995. PMLR, 2017.

Appendix A

Chapter 4 Supplementals

A.1 Preliminaries

The theoretical analyses developed in this appendix focuses on the learning scenario where single data-points are presented to an MLP each training iteration, similar to online learning scenarios. We focus on this scenario in large part because it is more biologically realistic than training with large mini-batches, and a central goal of the paper is to analyze IL in order to better understand how optimization and credit assignment may work in the brain. Other analyses of learning algorithms make similar simplifying assumptions (e.g., see [74]). We discuss the case of mini-batches briefly in section A.2.5, and our empirical simulations in this paper and previous works in the literature show that IL algorithms are able to perform competitively with BP when trained on mini-batches (e.g., see [132, 9, 110] as well as in the scenario where single data-points are presented. This suggests using IL with large mini-batches is approximated by the case of training with single data points. We leave it to future work to analyze differences in the behavior of IL in the case of small versus large mini-batches in more detail.

Our analyses considers the version of IL that uses local predictions computed $p_{n+1} = W_n f(\hat{h}_n)$ at hidden layers and $p_1 = W_0 \hat{h}_0$ at the input layer. This kind of prediction can be interpreted as a prediction of neuron pre-activations. This is slightly different from predictions of neuron activations: $p_{n+1} = f(W_n \hat{h}_n)$. Although we only analyse the pre-activation version of the prediction, we find the two behave very similarly in practice (e.g., our IL-SGD model uses prediction of activations, while our IL-prox models use predictions of pre-activations and both achieve similar performance on a variety of tasks). Thus, our analyses here should apply approximately to IL models that use the activation prediction equation.

Finally, we will sometimes use two slightly different expressions to describe the loss produced by MLP parameter θ. First, $L(\theta^{(b)}, x^{(b)}, y^{(b)}) = L(h_N^{(b)}, y^{(b)})$ refers to the loss produced by MLP parameters $\theta^{(b)}$ at training iteration b, given data point $x^{(b)}$ and prediction target $y^{(b)}$ during the feedforward pass. This is distinct from $L(\hat{h}_N, y^{(b)})$, which refers to the loss between the output layer optimized activities $\hat{h}_N$ and target $y^{(b)}$. Note that both h_N and $\hat{h}_N$ are initialized to $W_{N-1} f(h_{N-1})$. Importantly, assume in MLPs with non-linearities some non-linearity is applied to the activities at the output layer before being inputted into loss. For example, in the case where cross entropy loss is used, one would compute $L(\sigma(h_N^{(b)}), y^{(b)})$, where σ may be sigmoid in the case binary cross entropy or softmax in the case a categorical cross entropy loss. To simplify notation, we do not write this σ below, but assume it is 'built' into the loss as needed. None of the proofs below are affected by this assumption.

A.2 G-IL and Implicit SGD

A.2.1 A Brief Introduction to Implicit SGD

Let $\theta^{(b)}$ be a set of parameters, $x^{(b)}$ input data, and global output target is $y^{(b)}$, where b is the current training iteration. The explicit SGD update uses the loss gradient produced by

the current parameters:

$$\theta^{(b+1)} = \theta^{(b)} - \alpha \frac{\partial L(\theta^{(b)}, x^{(b)}, y^{(b)})}{\partial \theta^b}, \tag{A.1}$$

where α is step size and L is the function being minimized. This update is explicit because the gradient can be readily computed given $\theta^{(b)}, x^{(b)}, y^{(b)}$. The *implicit* SGD update uses the loss gradient of the parameters at the next training iteration:

$$\theta^{(b+1)} = \theta^{(b)} - \alpha \frac{\partial L(\theta^{(b+1)}, x^{(b)}, y^{(b)})}{\partial \theta^{(b+1)}}. \tag{A.2}$$

This is an implicit update because $\theta^{(b+1)}$ shows up on both sides of the equation. Unlike explicit SGD, $\theta^{(b+1)}$ cannot be readily computed using available quantities. One can still perform an implicit gradient update, however, by computing the solution of the following optimization problem:

$$\theta^{(b+1)} = \arg\min_{\theta}(L(\theta, x^{(b)}, y^{(b)}) + \frac{1}{2\alpha}\|\theta - \theta^{(b)}\|^2). \tag{A.3}$$

This update is known as the proximal algorithm or proximal point method [94] and is equivalent to performing an implicit SGD update in the stochastic setting (see below). This algorithm is a specific application of the more general proximal operator:

$$prox_{\alpha f}(z) = \arg\min_{\theta}(f(\theta) + \frac{1}{2\alpha}\|\theta - z\|^2), \tag{A.4}$$

where f is a function with scalar output. We can see the proximal algorithm changes parameters $\theta^{(b)}$ in a way that minimizes the loss L and the magnitude of the update, which helps keep $\theta^{(b+1)}$ in the proximity of $\theta^{(b)}$. We can also see the proximal algorithm only requires using the known quantities $\theta^{(b)}, x^{(b)}, y^{(b)}$.

The equivalence of the proximal algorithm in the stochastic setting to the implicit SGD

update (equation A.2) can be shown easily using the fact that at the minimum of the proximal objective $\theta = \theta^{(b+1)}$ and the gradient of A.3 is zero:

$$\begin{aligned} 0 &= \frac{\partial L}{\partial \theta^{(b+1)}} + \frac{1}{\alpha}(\theta^{(b+1)} - \theta^{(b)}) \\ \theta^{(b+1)} &= \theta^{(b)} - \alpha \frac{\partial L}{\partial \theta^{(b+1)}}, \end{aligned} \tag{A.5}$$

where $L = L(\theta^{(b+1)}, x^{(b)}, y^{(b)})$.

Explicit and implicit SGD have similar convergence guarantees. Analysis of the convergence properties of implicit SGD in linear regression models was done by [127, 129], in generalized linear models by [126]. However, explicit and implicit SGD have importantly distinct stability properties. It is well-known that explicit SGD is highly sensitive to learning rate. Implicit SGD, on the other hand, is *unconditionally stable* in the sense that equation A.3 monotonically decreases the loss function L for $0 < \alpha$, as it is implied by equation A.3 that $L(\theta^{(b+1)}) + \frac{1}{\alpha 2}\|\theta^{(b+1)} - \theta^{(b)}\|^2 \leq L(\theta^{(b)})$. One can also gain an intuition for the unconditional stability of implicit SGD by noting the differences in the way the learning rate α is used in the explicit SGD update (equation A.1) versus the proximal algorithm/implicit update (equation A.3). In the explicit SGD update α is used to scale the gradient, which clearly implies that as $\alpha \to \infty$ so does the magnitude of the update. However, with the proximal algorithm α only controls the relative weighting of the two terms in equation A.3. As $\alpha \to \infty$ we see the regularization term $\frac{1}{2\alpha}\|\theta - \theta^{(b)}\|^2$ is down-weighted to 0. In this case, the updated parameters are simply those that best minimize the loss function L, which lead to an update that reduces the loss given $x^{(b)}$. For further details on the stability of explicit and implicit SGD in linear models see [127, 126, 129].

A.2.2 G-IL Local Targets as Target FF Activities

In this section, we show that when a single datapoint is presented each training iteration, there are two interesting properties concerning the local targets computed by IL and the NLMS rule used by IL. We express these properties in lemma A.1 and proposition A.1. This lemma and proposition are used in proofs below.

First, we show that IL local targets $\hat{h}_n^{(b)}$, at training iteration b, become the FF activities $h_n^{(b+1)}$ at the next training iteration given the same data point when weights are updated to fully minimize local prediction errors. In this sense, IL local targets are target FF activities.

Let $\theta^{(b)}$ be the set of MLP weight matrices $\theta^{(b)} = [W_0^{(b)}, ..., W_{N-1}^{(b)}]$. Let the solution parameters at iteration b be $\theta^* = \arg\min_\theta F$, where F is the energy function defined in equation 5.1. In the IL algorithm, minimizing F w.r.t. weight W_n equates to updating weights in a way that minimizes the local prediction error $e_{n+1} = \frac{1}{2}\|\hat{h}_{n+1} - p_{n+1}\|^2$ (see section 4.3.3). In the case where the MLP is only presented with a single data-point each iteration, this local learning problem can be solved such that there is zero error. More specifically, let W_n^* be the weight matrix at layer n of θ^*. When training with single data-points, this solution weight matrix has the property $\hat{h}_{n+1}^{(b)} = W_n^* f(\hat{h}_n^{(b)})$ (for an example of one solution with this property see proposition A.1). It follows trivially that $\hat{h}_n$ are equivalent to the FF activities of θ^* when given the same data-point $x^{(b)}$ as input:

Lemma A.1. *Consider a non-linear MLP trained with G-IL at iteration b with mini-batch size 1. Let h_n^* be the FF activities of solution parameters θ^* at layer n given data-point $x^{(b)}$. It is the case that $h_n^* = \hat{h}_n^{(b)}$.*

Proof. At initialization, $\hat{h}_0^{(b)} = x^{(b)}$ and $h_0^* = x^{(b)}$. Thus, $\hat{h}_0^{(b)} = h_0^*$. Next, by definition at the input layer $\hat{h}_1^{(b)} = W_0^* \hat{h}_0^{(b)}$. This implies that $\hat{h}_1^{(b)} = W_0^* \hat{h}_0^{(b)} = W_0^* \hat{h}_0^* = h_1^*$. This property holds for remaining layers because by definition $\hat{h}_{n+1}^{(b)} = W_n^* f(\hat{h}_n^{(b)})$, which implies

$\hat{h}_2^{(b)} = W_1^* f(\hat{h}_1^{(b)}) = W_1^* f(h_1^*) = h_2^*$. This same procedure can then be repeatedly applied to show the same is true of all remaining layers n. $\square$

The solution matrices W_n^* are non-unique in the case where single-data points are used. However, there are unique minimum-norm solution matrices, which can be computed using the NLMS rule (equation 4.6) as we now show.

Proposition A.1. *Consider an non-linear MLP trained with G-IL under the NLMS rule (equation A.7) at iteration b mini-batch size 1. For all n, the updated matrix* $W_n^{(b+1)}$ *is the minimum norm solution matrix, i.e.* $W_n^{(b+1)} = \arg\min_{W_n^{(b+1)}} \frac{1}{2}\|W_n^{(b+1)} - W_n^{(b)}\|^2$, *subject to the constraint that* $\hat{h}_n^{(b)} = W_n^{b+1} f(\hat{h}_n^{(b)})$.

Proof. Here we follow the proof of [21], which proved a similar result concerning linear regression models trained with a variant of NLMS.

Using the method of Lagrangian multipliers we can rewrite $\arg\min_{W_n} \frac{1}{2}\|W_n^{(b+1)} - W_n^{(b)}\|^2$, subject to $\hat{h}_n^{(b)} = W_n^{b+1} f(\hat{h}_n^{(b)})$ as as follows

$$P = \frac{1}{2}\|W_n^{(b+1)} - W_n^{(b)}\|^2 + \lambda e_{n+1}^{(b+1)}, \tag{A.6}$$

where λ is the Lagrangian multiplier and $e_{n+1}^{(b+1)} = \hat{h}_{n+1}^{(b)} - W_n^{(b+1)} f(\hat{h}_n^{(b)})$. First we compute the gradient of P w.r.t. the weights and set to 0,

$$\frac{\partial P}{\partial W_n^{(b+1)}} = W_n^{(b+1)} - W_n^{(b)} + \lambda f(\hat{h}_n^{(b)})^T = 0. \tag{A.7}$$

The partial w.r.t. the Lagrangian is then

$$\frac{\partial P}{\partial \lambda} = \hat{h}_{n+1}^{(b)} - W_n^{(b+1)} f(\hat{h}_n^{(b)}) = 0. \tag{A.8}$$

Rearranging A.7 we get

$$W_n^{(b+1)} = W_n^{(b)} - \lambda f(\hat{h}_n^{(b)})^T. \tag{A.9}$$

Substituting A.9 into A.8 we get

$$\begin{aligned} 0 &= \hat{h}_{n+1}^{(b)} - (W_n^{(b)} - \lambda f(\hat{h}_n^{(b)})^T) f(\hat{h}_n^{(b)}) \\ \lambda &= -\|f(\hat{h}_n^{(b)})\|^{-2} e_{n+1}^{(b)} \end{aligned} \tag{A.10}$$

.

Finally, substituting the value for λ into A.9 we get

$$W_n^{(b+1)} = W_n^{(b)} + \|f(\hat{h}_n^{(b)})\|^{-2} e_{n+1}^{(b)} f(\hat{h}_n^{(b)})^T, \tag{A.11}$$

which is exactly equal to the NLMS rule used in the IL algorithm (equation 4.6). □

In sum, when training with single data-points, activities $\hat{h}$ optimized by IL are target FF activities in the sense that they become FF activities after F is minimized completely w.r.t. weights and local errors go to zero. The NLMS rule yields such a solution. One important implication of this is that the LMS update (equation 5.4), which is commonly used in practice, is a good approximation of the NLMS solution since the two are proportional.

This lemma and proposition allow us to do something important, which lays the basis for the connection between implicit SGD and G-IL. In particular, this lemma and proposition show us that the network solution θ^*, its FF activities h_n^*, and $\Delta\theta$ can be expressed implicitly in terms of $\hat{h}$. For example, lemma A.1 shows that $h_n^* = \hat{h}_n$. This implies that we know what

the loss produced by θ^*:

$$L(\theta^*, x^{(b)}, y^{(b)}) = L(\hat{h}_N^{(b)}, y^{(b)}), \tag{A.12}$$

where $L(\hat{h}_N, y^{(b)})$ is the loss measure, which in the case of MLPs is some measure of the difference between prediction target $y^{(b)}$ and FF output layer value, which for θ^* is equivalent to $\hat{h}_N$ (as implied by lemma A.1).

Additionally, proposition A.1 shows that each ΔW_n can be expressed as $\|f(\hat{h}_n^{(b)})\|^{-2} e_{n+1}^{(b)} f(\hat{h}_n^{(b)})^T$, where e_{n+1} again is can be expressed in terms of optimized activities: $e_{n+1} = \hat{h}_{n+1}^{(b)} - W_n^{(b)} f(\hat{h}_n^{(b)})$. This means

$$\frac{1}{2}\|\theta^* - \theta^{(b)}\|^2 = \frac{1}{2}\sum_n^N \|\Delta W_n\|^2 = \frac{1}{2}\sum_n^N \|\|f(\hat{h}_n^{(b)})\|^{-2}(\hat{h}_{n+1}^{(b)} - W_n^{(b)} f(\hat{h}_n^{(b)}))f(\hat{h}_n)^T\|^2. \tag{A.13}$$

This allows for the possibility to minimize the proximal quantity w.r.t. $\hat{h}_n$. That is, it allows for an IL algorithm, where during the inference phase the algorithm approximates $\arg\min_{\hat{h}}(L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2)$, where $L(\theta^*)$ and $\|\theta^* - \theta^{(b)}\|^2$ are defined implicitly in terms of $\hat{h}$ using equations A.12 and A.13. At each step in the inference phase $\hat{h}$ are updated to minimize the proximal loss, which results in a change in θ^*, since θ^* is defined in terms of the $\hat{h}$. After a minimum is reached weights are updated with the NLMS rule such that $\hat{h}$ values become the FF values given the same data point. The inference phase, in other words, finds the FF values of a θ^* that best minimizes the proximal loss, then uses those FF values to update the actual parameters $\theta^{(b)}$ such that $\theta^{(b+1)} = \theta^*$ (see figure 4.4.1 for visualization).

Let's define such an algorithm as proximal inference learning (IL-prox), since it, like G-IL, computes $\hat{h}_n$ by minimizing an objective w.r.t. activities, then updates weights afterward:

Definition A.1. *Proximal Inference Learning (IL-prox). An algorithm identical to G-IL (algorithm 4), except activities are optimized to approximate* $\arg\min_{\hat{h}_n}(L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2)$

and weights are updated using the NLMS rule.

Again, the θ^* here is defined implicitly in terms of the $\hat{h}$ values using equations A.12 and A.13. In the next, section we show that under certain variable settings it is the case that $\arg\min_{\hat{h}} F = \arg\min_{\hat{h}}(L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2)$.

A.2.3 Generalized Inference Learning Approximates Proximal Inference Learning

Let α be a 'global' learning rate hyper-parameter for θ, and α_n be the layer-wise learning rate used in the weight update ΔW_n. Finally, let θ^* be defined as the parameters after the NLMS update is applied to each weight matrix, as in the last section.

Theorem A.1. *Let* $\alpha_n = \|f(\hat{h}_n)\|^{-2}$. *In the limit where* $\gamma_n^{decay} \to \|e_{n+1}\|^2(1 - \frac{2}{\alpha_n^6})$, $\gamma_N \to \frac{\alpha_{N-1}}{\alpha}$, *and* $\gamma_n \to \alpha_{n-1}^{-1}$ *for all* $n < N$, *it is the case that* $\arg\min_{\hat{h}} F = \arg\min_{\hat{h}} L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2$. *Hence, in these limits, G-IL is equivalent to IL-prox.*

Proof. To prove this statement, we show that $\frac{\partial F}{\partial \hat{h}_n} = 0 \iff \frac{\partial Prox}{\partial \hat{h}_n} = 0$, which implies $\arg\min_{\hat{h}} F = \arg\min_{\hat{h}} L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2$. We first compute the gradients of the IL energy function F w.r.t. activities $\hat{h}$, then the gradient of the proximal loss, which is defined in terms of equation A.12 and A.13.

Gradient of F In IL networks, activities are updated with local gradients of F, as follows:

At the output layer,

$$\frac{\partial F(\hat{h}_N)}{\partial \hat{h}_N} = \frac{\partial L(y, \hat{h}_N)}{\partial \hat{h}_N} + \gamma_N e_N. \tag{A.14}$$

and at hidden layers:

$$\frac{\partial F(\hat{h}_n)}{\partial \hat{h}_n} = -\gamma_{n+1} f'(\hat{h}_n) W_n^T e_{n+1} + \gamma_n e_n + \gamma_n^{decay} f'(\hat{h}_n)\hat{h}_n, \tag{A.15}$$

where $f'(\hat{h}_n) = \frac{\partial f}{\partial \hat{h}_n}$. We see the gradient of the loss L term is only taken w.r.t. the local output layer activity $\hat{h}_N$, and the gradient of prediction error at layers n and $n+1$ are taken w.r.t. to local layer n. We assume whatever process is used to minimize F similarly only uses local information such that its minima are described by the above gradients when they equal zero.

Gradient of Prox w.r.t. Output Layer We first compute gradients for the output layer. As with G-IL, we assume only local gradients of the proximal loss are used to update the activities. The gradients of the proximal loss w.r.t. output layer activity $\hat{h}_N$ is

$$Prox(\hat{h}_N) = \underbrace{L(\hat{h}_N, y)}_{prox_1} + \underbrace{\frac{1}{2\alpha}\|\Delta W_{N-1}\|^2}_{prox_2}, \tag{A.16}$$

since $\hat{h}_N$ is local to L and since W_{N-1} are the only weights local to $\hat{h}_N$.

The gradient $\frac{\partial Prox(\hat{h}_N)}{\partial \hat{h}_N}$, can be expressed in terms of $prox_1$ and $prox_2$ as follows: $\frac{\partial Prox(\hat{h}_N)}{\partial \hat{h}_N} = \frac{\partial prox_1(\hat{h}_N)}{\partial \hat{h}_N} + \frac{\partial prox_2(\hat{h}_N)}{\partial \hat{h}_N}$. Clearly, $\frac{\partial prox_1(\hat{h}_N)}{\partial \hat{h}_N} = \frac{\partial L(\hat{h}_N, y)}{\partial \hat{h}_N}$. The second term $\frac{\partial prox_2(\hat{h}_N)}{\partial \hat{h}_N}$ can be computed using the chain rule. First,

$$prox_2(\hat{h}_N) = \frac{1}{2\alpha}\|\Delta W_{N-1}\|^2 = \frac{1}{2\alpha}\|\alpha_{N-1} e_N f(\hat{h}_{N-1})^T\|^2. \tag{A.17}$$

where $p_N = W_{N-1} f(\hat{h}_{N-1})$, f is an element-wise non-linearity, and $e_N = \hat{h}_N - p_N$. Then using the chain rule we have

$$\frac{\partial prox_2}{\partial \hat{h}_N} = \frac{\partial prox_2}{\partial \Delta W_{N-1}} \frac{\partial \Delta W_{N-1}}{\partial e_N} \frac{\partial e_N}{\partial \hat{h}_N}, \tag{A.18}$$

where $\frac{\partial prox_2}{\partial \Delta W_{N-1}} = \frac{\alpha_{N-1}}{\alpha} e_n f(\hat{h}_{N-1})^T$ and $\frac{\partial \Delta W_{N-1}}{\partial e_N} \frac{\partial e_N}{\partial \hat{h}_N} = \alpha_{N-1} f(\hat{h}_{N-1})$. Together we get

$$\begin{aligned} \frac{\partial Prox(\hat{h}_N)}{\partial \hat{h}_N} &= \frac{\partial L(\hat{h}_N, y)}{\partial \hat{h}_N} + \frac{\alpha_{N-1}^2}{\alpha} \|f(\hat{h}_{N-1})\|^2 e_N \\ &= \frac{\partial L(\hat{h}_N, y)}{\partial \hat{h}_N} + \frac{\alpha_{N-1}}{\alpha} e_N. \end{aligned} \tag{A.19}$$

Note that here $\alpha_{N-1} = \|f(\hat{h}_{N-1})\|^{-2}$ as noted in the theorem. In the limit where $\gamma_N \to \frac{\alpha_{N-1}}{\alpha}$, this gradient comes to $\frac{\partial L(\hat{h}_N, y)}{\partial \hat{h}_N} + \gamma_N e_N$, which is equivalent to the gradient $\frac{\partial F}{\partial \hat{h}_N}$ (see equation A.14).

Gradient w.r.t. Hidden Layers Next, we compute $\frac{\partial Prox(\hat{h}_n)}{\partial \hat{h}_n}$ for hidden layers. The components of $Prox$ local to hidden layer $\hat{h}_n$ are

$$Prox(\hat{h}_n) = \underbrace{\frac{1}{2\alpha} \|\Delta W_n\|^2}_{prox_1} + \underbrace{\frac{1}{2\alpha} \|\Delta W_{n-1}\|^2}_{prox_2}. \tag{A.20}$$

whose gradient can be expressed in terms of $prox_1$ and $prox_2$: $\frac{\partial Prox(\hat{h}_n)}{\partial \hat{h}_n} = \frac{\partial prox_1(\hat{h}_n)}{\partial \hat{h}_n} + \frac{\partial prox_2(\hat{h}_n)}{\partial \hat{h}_n}$. As above, the gradient $\frac{\partial prox_2(\hat{h}_n)}{\partial \hat{h}_n}$ is computed using the chain rule.

$$prox_2 = \frac{1}{2\alpha} \|\Delta W_{n-1}\|^2 = \frac{1}{2\alpha} \|\alpha_{n-1} e_n f(\hat{h}_{n-1})^T\|^2, \tag{A.21}$$

where $p_n = W_{n-1} f(\hat{h}_{n-1})$ and $e_n = \hat{h}_n - p_n$. Using the chain rule

$$\frac{\partial prox_2(\hat{h}_n)}{\partial \hat{h}_n} = \frac{\partial prox_2}{\partial \Delta W_{n-1}} \frac{\partial \Delta W_{n-1}}{\partial e_n} \frac{\partial e_n}{\partial \hat{h}_n}, \tag{A.22}$$

where $\frac{\partial prox_2}{\partial \Delta W_{n-1}} = \frac{\alpha_{n-1}}{\alpha} e_n f(\hat{h}_{n-1})^T$ and $\frac{\partial \Delta W_{n-1}}{\partial e_n} \frac{\partial e_n}{\partial \hat{h}_n} = \alpha_{n-1} f(\hat{h}_{n-1})$. Multiplying together we

get

$$\frac{\partial prox_2(\hat{h}_n)}{\partial \hat{h}_n} = \frac{\alpha_{n-1}^2}{\alpha}\|f(\hat{h}_{n-1})\|^2 e_n. \tag{A.23}$$

Now we derive $\frac{prox_1(\hat{h}_n)}{\hat{h}_n}$:

$$prox_1 = \frac{1}{2\alpha}\|\Delta W_n\|^2 = \frac{1}{2\alpha}\|\alpha_{n-1} e_{n+1} f(\hat{h}_n)^T\|^2, \tag{A.24}$$

where $p_{n+1} = W_n f(\hat{h}_n)$ and $e_{n+1} = \hat{h}_{n+1} - p_{n+1}$. Using the chain rule

$$\begin{aligned}\frac{\partial prox_1}{\partial \hat{h}_n} &= \frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial e_{n+1}}\frac{\partial e_{n+1}}{\partial p_{n+1}}\frac{\partial p_n}{\partial \hat{h}_n} \\ &+ \frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial f(\hat{h}_n)^T}\frac{\partial f(\hat{h}_n)^T}{\partial \hat{h}_n} \\ &+ \frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial \|f(\hat{h}_n)\|^{-2}}\frac{\partial \|f(\hat{h}_n)\|^{-2}}{\partial \hat{h}_n}.\end{aligned} \tag{A.25}$$

Unlike the previous gradients, we now need propagate the gradient through the learning rate, which is what is done by the term $\frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial \|f(\hat{h}_n)\|^{-2}}\frac{\partial \|f(\hat{h}_n)\|^{-2}}{\partial \hat{h}_n}$ above.

First, $\frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial e_{n+1}}\frac{\partial e_{n+1}}{\partial p_{n+1}}\frac{\partial p_n}{\partial \hat{h}_n} = -\frac{\alpha_n^2}{\alpha}\|f(\hat{h}_n)\|^2 f'(\hat{h}_n) W_n^T e_{n+1}$. Next, $\frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial f(\hat{h}_n)^T}\frac{\partial f(\hat{h}_n)^T}{\partial \hat{h}_n} = \frac{\alpha_n^2}{\alpha}\|e_{n+1}\|^2 f'(\hat{h}_n)\hat{h}_n$. Finally, $\frac{\partial prox_1}{\partial \Delta W_n}\frac{\partial \Delta W_n}{\partial \|f(\hat{h}_n)\|^{-2}}\frac{\partial \|f(\hat{h}_n)\|^{-2}}{\partial \hat{h}_n} = \frac{\alpha_n^2}{\alpha}(\frac{-2\|e_n\|^2}{\alpha_n^6}) f'(\hat{h}_n)\hat{h}_n$, which results in

$$\begin{aligned}\frac{\partial prox_1}{\partial \hat{h}_n} &= -\frac{\alpha_n^2}{\alpha}\|f(\hat{h}_n)\|^2 f'(\hat{h}_n) W_n^T e_{n+1} + \frac{\alpha_n^2}{\alpha}\|e_{n+1}\|^2 f'(\hat{h}_n)\hat{h}_n + \frac{\alpha_n^2}{\alpha}(\frac{-2\|e_{n+1}\|^2}{\alpha_n^6}) f'(\hat{h}_n)\hat{h}_n \\ &= \frac{\alpha_n^2}{\alpha}(-\|f(\hat{h}_n)\|^2 f'(\hat{h}_n) W_n^T e_{n+1} + \|e_{n+1}\|^2 f'(\hat{h}_n)\hat{h}_n + \frac{-2\|e_{n+1}\|^2}{\alpha_n^6} f'(\hat{h}_n)\hat{h}_n) \\ &= \frac{\alpha_n^2}{\alpha}(-\alpha_n^{-1} f'(\hat{h}_n) W_n^T e_{n+1} + \|e_{n+1}\|^2(1 + \frac{-2}{\alpha_n^6}) f'(\hat{h}_n)\hat{h}_n)\end{aligned} \tag{A.26}$$

Now we substitute our $\frac{\partial prox_1}{\partial \hat{h}_n}$ and $\frac{\partial prox_2}{\partial \hat{h}_n}$ terms back into the gradient of $Prox(\hat{h}_n)$:

$$\frac{\partial Prox(\hat{h}_n)}{\partial \hat{h}_n} = \frac{\alpha_n^2}{\alpha}(-\alpha_n^{-1} f'(\hat{h}_n) W_n^T e_{n+1} + \alpha_{n-1}^{-1} e_n + \|e_{n+1}\|^2 (1 + \frac{-2}{\alpha_n^6}) f'(\hat{h}_n)\hat{h}_n) \quad \text{(A.27)}$$

Finally, we set this gradient equal to 0 (i.e. at a minimum of Prox) and in the limit where $\gamma_n^{decay} \to \|e_{n+1}\|^2(1 + \frac{-2}{\alpha_n^6})$ and $\gamma_n \to \alpha_{n-1}^{-1}$ for all $n < N$. To simplify notation, we just label this limit as lim

$$\begin{aligned} 0 &= \lim \frac{\alpha_n^2}{\alpha}(-\alpha_n^{-1} f'(\hat{h}_n) W_n^T e_{n+1} + \alpha_{n-1}^{-1} e_n + \|e_{n+1}\|^2 (1 + \frac{-2}{\alpha_n^6}) f'(\hat{h}_n)\hat{h}_n) \\ &= -\gamma_{n+1} f'(\hat{h}_n) W_n^T e_{n+1} + \gamma_n e_n + \gamma_n^{decay} f'(\hat{h}_n)\hat{h}_n. \end{aligned} \quad \text{(A.28)}$$

When set to zero, $\frac{\alpha_n^2}{\alpha}$ cancels. The result is exactly equal to $\frac{\partial F}{\partial \hat{h}_n}$ (equation A.15). It follows that, in these limits, for any hidden layer n it is the case that $\frac{\partial F}{\partial \hat{h}_n} = 0 \iff \frac{\partial Prox}{\partial \hat{h}_n} = 0$. Since the same result holds at the output layer, it follows that $\arg\min_{\hat{h}} F = \arg\min_{\hat{h}} L(\theta^*) + \frac{1}{2\alpha}\|\Delta\theta\|^2$. Hence, under these γ settings G-IL is equivalent to IL-prox. □

A.2.4 G-IL Approximates Implicit SGD

Here we present the theorem showing the IL-prox, and G-IL with the γ setting noted in the above theorem, are equivalent to the proximal algorithm and thus implicit SGD. An intuitive way to think about the relation between the typical proximal algorithm and IL-prox is that IL-prox does what the typical proximal algorithm would do, but in reverse order. The standard proximal algorithm $argmin_\theta L(\theta) + \frac{1}{2\alpha}\|\theta - \theta^{(b)}\|^2$ minimizes the proximal loss w.r.t. parameters, then the new parameters can be used to compute new FF values given the data-point $x^{(b)}$. IL prox, on the other hand, first computes the new FF values of the

parameters that best minimize the proximal loss given $x^{(b)}$ during the inference phase. Then it updates weights so they become the parameters that best minimize the proximal loss. Since both optimization problems are unconstrained (and thus optimize over the same space of possible parameter values), they yield the same parameter updates in the end.

Theorem A.2. *Let $\theta^{(b)}$ be a set of MLP parameters at training iteration b. Let $\theta_{prox}^{(b+1)} = argmin_\theta\, L(\theta) + \frac{1}{2\alpha}\|\theta - \theta^{(b)}\|^2$. Let $\theta_{IL-prox}^{(b+1)}$ be the parameters updated by IL-prox (see 4.1) and $\theta_{IL}^{(b+1)}$ the parameters updated by G-IL under parameter setting in theorem 4.1. Assume mini-batch size 1. Under these assumptions, it is the case $\theta_{prox}^{(b+1)} = \theta_{IL-prox}^{(b+1)} = \theta_{IL}^{(b+1)}$.*

Proof. The weight update procedure performed by IL-prox can be described as follows:

$$\theta_{IL-prox}^{(b+1)} = argmin_\theta\, F(argmin_{\hat{h}} L(\theta^*) + \|\theta^* - \theta^{(b)}\|^2), \tag{A.29}$$

where $argmin_\theta\, F$ is computed using the NLMS rule and θ^* are the parameters updated with the NLMS rule (equation A.7). Lemma A.1 and proposition A.1 imply that the $\hat{h}$ produced by $argmin_{\hat{h}} L(\theta^*) + \frac{1}{2\alpha}\|\theta^* - \theta^{(b)}\|^2)$ are the FF activities of θ^*. The fact that $argmin_\theta F$ is computed using the NLMS rule implies that the resulting parameters are the θ^* of the optimized activities. $\hat{h}$ are unbounded, real-valued vectors, which implies that $argmin_{\hat{h}} L(\theta^*) + \|\theta^* - \theta^{(b)}\|^2$ is an unbounded optimization problem and also implies that the possible θ^* values are also unbounded (because they can vary over a set of FF values that have unbounded possible values). This implies that $argmin_{\hat{h}} L(\theta^*) + \|\theta^* - \theta^{(b)}\|^2$ outputs the FF values of the parameters θ^* that best minimize the proximal loss. Then $argmin_\theta F$ uses those FF activities to update $\theta^{(b)}$ such that it become equal to the θ^* that best minimize the proximal loss. This implies $\theta_{IL-prox}^{(b+1)} = argmin_\theta L(\theta) + \|\theta - \theta^{(b)}\|^2$ and thus $\theta_{prox}^{(b+1)} = \theta_{IL-prox}^{(b+1)}$.

Further, this conclusion implies that under the γ parameter settings and learning rates noted in theorem 4.1, the parameter update produced by G-IL, $\theta_{IL}^{(b+1)}$ also equals $\theta_{prox}^{(b+1)}$, since $\theta_{prox}^{(b+1)} = \theta_{IL-prox}^{(b+1)} = \theta_{IL}^{(b+1)}$, given the proof above and theorem 4.1. □

A.2.5 A Note about Mini-batches

Our analysis above focuses on the more biologically realistic scenario where a single data-point is presented each training iteration. In this case, the NLMS rule provides the minimum-norm solution to $argmin_\theta F$ and the LMS update approximates this solution in the sense it is proportional to the NLMS update. However, in the case where mini-batches of size > 1 are used, the NLMS rule is not a solution to $argmin_\theta F$. The gradient update of F w.r.t. W_n is

$$W_n^{b+1} = W_n^{(b)} - \alpha_n \frac{\partial F}{\partial W_n} = W_n^{(b)} + \alpha_n (H_{n+1}^{T(b)} - W_n^{(b)} f(H_n^{T(b)})) f(H_n^{(b)}), \tag{A.30}$$

where f is an element-wise non-linearity and H_n is the mini-batch matrix of neurons activities. Each row of H_n is a $\hat{h}_n$ for one data-point in the mini-batch.

The solution to the local error minimization problem is found by setting the gradient equal to zero and solving for W_n:

$$\begin{aligned} 0 &= (H_{n+1}^{T(b)} - W_n f(H_n^{T(b)})) f(H_n^{(b)}) \\ W_n &= H_{n+1}^{T(b)} f(H_n^{(b)}) (f(H_n^{T(b)}) f(H_n^{(b)}))^{-1}, \end{aligned} \tag{A.31}$$

This update is generally not equal or proportional to the gradient update. Gradient updates are thus poorer approximations of the solution(s) to local least mean squared problem in the case of mini-batches size > 1 than they are in the case with mini-batches size equal to one. This may provide some insight into why the IL models we tested do not show performance advantages over BP-SGD when mini-batches are used as opposed to when single datapoints are used to update weights. Future research could explore alternative learning rules that better approximate the solution above in the mini-batch case.

A.3 A Closed Form Description of G-IL Local Targets

In this section, we derive a closed form description of $argmin_{\hat{h}} F$ for linear MLPs. In addition to simplifying notation, we focus on linear networks because we aim to use this closed form description to analyze the stability properties of linear networks trained with IL in the next section.

A.3.1 A Brief Intro to Gauss-Newton Optimization and Ridge Regression

Let θ be a vector of parameters, y a vector of prediction targets, X a data matrix. The linear least squares problem is defined as

$$\arg\min_{\theta} \|y - X\theta\|^2. \tag{A.32}$$

Its closed form solution is $(XX^T)^{-1}X^Ty$ which equals X^+y when X has linearly independent columns. X^+ is the left pseudo-inverse of X. For matrix M the left pseudo-inverse has the property $I = M^+M$. When the regularization term $\lambda\|\theta\|^2$ is added to A.32 we get ridge regression whose closed form solution is $(XX^T + \lambda I)^{-1}X^Ty$, where I is the identity matrix and λ is a scalar.

The Gauss-Newton method is an iterative method used to solve non-linear least squares problems, which has some similarities to the solutions to linear least squares problem just mentioned. Let $e^{(b)}$ be the residuals of the model at b, i.e. $e^{(b)} = y^{(b)} - f(x^{(b)}; \theta^{(b)})$, where f is the non-linear model function. The Gauss-Newton (GN) update for $\theta^{(b)}$ is

$$\begin{aligned}\theta^{(b+1)} &= \theta^{(b)} + (JJ^T)^{-1}J^T e^{(b)} \\ &= \theta^{(b)} + J^+ e^{(b)},\end{aligned} \tag{A.33}$$

where J is the Jacobian of f w.r.t. $\theta^{(b)}$ and J^+ is the left pseudo-inverse of J. Note the similarity to the closed form solution to the linear least squares. The parameters are iteratively updated until convergence or a near convergent state is reached.

It is also common to add a regularization term to the Gauss-Newton update as follows

$$\theta^{(b+1)} = \theta^{(b)} + (JJ^T + \lambda I)^{-1}J^T e^{(b)}, \tag{A.34}$$

where I is the identity matrix and λ is some scalar. Notice the similarity to the ridge regression solution. The regularization term helps prevent the change to θ from growing too large. Note that as $\lambda \to \infty$ the update approaches gradient descent: $\theta^{(b+1)} = \theta^{(b)} + J^T e^{(b)}$.

A.3.2 A Closed form Description of G-IL Local Targets

Consider a linear MLP trained with G-IL using a squared error global loss. Local targets at hidden layers of such networks are the result of the optimization process that attempts to find

$$\hat{h}_n^{(b)} = \arg\min_{\hat{h}_n}\left(\frac{1}{2}\|\hat{h}_{n+1}^{(b)} - W_n\hat{h}_n\|^2 + \frac{\lambda}{2}\|\hat{h}_n - p_n^{(b)}\|^2\right), \tag{A.35}$$

where $\lambda = \frac{\gamma_n}{\gamma_{n+1}}$ which represents the relative weighting of the two terms (see equation 5.1). Here we ignore the optional decay term in equation 5.1.

The $\hat{h}_n^{(b)}$ can be expressed in closed form by noting that at the minimum of the expression above $h_n = \hat{h}_n^{(b)}$, and the gradient of the expression equals 0. One can thus compute the gradient, set it equal to zero, and solve for $\hat{h}_n^{(b)}$:

$$\begin{aligned} 0 &= -W_n^T(\hat{h}_{n+1}^{(b)} - W_n\hat{h}_n^{(b)}) + \lambda\hat{h}_n^{(b)} - \lambda p_n^{(b)} \\ 0 &= -W_n^T\hat{h}_{n+1}^{(b)} + (W_n^TW_n + \lambda I)\hat{h}_n^{(b)} - \lambda p_n^{(b)} \\ \hat{h}_n^{(b)} &= (W_n^TW_n + \lambda I)^{-1}W_n^T\hat{h}_{n+1}^{(b)} + \lambda(W_n^TW_n + \lambda I)^{-1}p_n^{(b)} \end{aligned} \tag{A.36}$$

Notice the term on the left $(W_n^TW_n + \lambda I)^{-1}W_n^T\hat{h}_{n+1}^{(b)}$ is identical to the closed form solution for ridge regression (see previous section), which is a regularized version of a simple target propagation $W_n^+\hat{h}_{n+1}^{(b)}$. The term on the right $(W_n^TW_n + \lambda I)^{-1}p_n \approx \epsilon p_n$, where ϵ is a scalar, since $(W_n^TW_n + \lambda I)^{-1}$ is positive definite and approaches a scalar multiple of I as $\lambda \to \infty$. Thus, at the minimum $\hat{h}_n^{(b)}$ closely approximates a weighted average between p_n and the solution to the regularized squared error at layer $n+1$.

A.3.3 Relation to Gauss-Newton Updates

Proposition A.2. *In the limit where $\lambda \to 0$ (in equation A.36), $\hat{h}_n = h_n^{(b)} + W_n^+\Delta h_{n+1}^{(b)}$, which is a Gauss-Newton update on initial activities $h_n^{(b)}$ with residual $\Delta h_{n+1}^{(b)}$ and Jacobian $J = W_n$.*

Proof.

$$\begin{aligned}\lim_{\lambda\to 0}\hat{h}_n^{(b)} &= (W_n^T W_n)^{-1} W_n^T \hat{h}_{n+1}^{(b)} = W_n^+ \hat{h}_{n+1}^{(b)} = h_n^{(b)} + W_n^+ \hat{h}_{n+1}^{(b)} - W_n^+ h_{n+1}^{(b)} \\ &= h_n^{(b)} + W_n^+(\hat{h}_{n+1}^{(b)} - h_{n+1}^{(b)}) = h_n^{(b)} + W_n^+ \Delta h_{n+1}^{(b)},\end{aligned} \tag{A.37}$$

□

More generally, in the case where the influence of the top down error term $\|\hat{h}_n - p_n^{(b)}\|^2$ is negligible, G-IL targets converge to Gauss-Newton targets.

A.3.4 Gauss-Newton G-IL as a Closed Form Approximation of G-IL

In practice, one approximates $\arg\min_{\hat{h}_n} F$ by initializing $h_n = \hat{h}_n = p_n$ then minimizes F using some optimization process. At the beginning of this optimization process the top down error $\|\hat{h}_n - p_n^{(b)}\|^2$ is small since it is initialized to zero and grows slowly afterward. Additionally, if optimization is stopped early (which is typically done in practice) this error's effect on activities will remain negligible compared to the larger bottom-up error term. This fact along with proposition A.2 suggest Gauss-Newton updated activities are a good approximation of G-IL local targets. Let's define a network that computes $\hat{h}_n$ using Gauss-Newton updates as follows

Definition A.2. *Gauss-Newton G-IL (IL-GN). IL-GN is a closed-form approximation of G-IL for training MLPs which uses the same weight update as G-IL and computes local targets using a Gauss-Newton update. In linear networks the update is* $\hat{h}_n = h_n^{(b)} + \gamma W_n^+ \Delta h_{n+1}^{(b)}$, *where* $0 < \gamma \leq 1$.

IL-GN approximates G-IL when $\gamma = 1$ as shown by proposition A.37. A $\gamma < 1$, however, better captures the fact that in the true closed form solution (see equation A.36), $\hat{h}_n$ is

a value in between a regularized GN-target and top-down prediction. Simulations below provide further justification that IL-GN with $0 < \gamma < 1$ is a good approximation of G-IL (see figure A.3.

A.3.5 Some Properties of IL-GN

There are several lemmas concerning IL-GN that will either be used in subsequent proofs or are relevant for understanding IL generally.

The first lemma says that predictions p_n in IL-GN networks are a weighted average of h_n and $\hat{h}_n$.

Lemma A.2. *After all $\hat{h}_n$ are computed, local predictions will lie between local sub-targets and FF activity: $p_n = (1-\gamma)h_n + \gamma\hat{h}_n$.*

Proof. First, prediction p_n can be described as follows: $p_n = W_{n-1}(\hat{h}_{n-1} + \Delta\hat{h}_{n-1}) = W_{n-1}(h_{n-1} + \gamma W_{n-1}^{+}\Delta h_n)$, which implies

$$\begin{aligned} p_n &= W_{n-1}(h_{n-1} + \gamma W_{n-1}^{+}\Delta h_n) \\ &= h_n + \gamma(\hat{h}_n - h_n) \\ &= (1-\gamma)h_n + \gamma\hat{h}_n. \end{aligned} \tag{A.38}$$

□

It then follows that the error after the targets and predictions are updated, e_n, is smaller but proportional to the initial error:

Lemma A.3. *After all $\hat{h}_n$ are computed, local errors $e_n = (1-\gamma)\Delta h_n$.*

Proof.

$$\begin{aligned} e_n &= \hat{h}_n - p_n \\ &= \hat{h}_n - ((1-\gamma)h_n + \gamma\hat{h}_n) \\ &= (1-\gamma)(\hat{h}_n - h_n) \\ &= (1-\gamma)\Delta h_n, \end{aligned} \tag{A.39}$$

where the second line is computed using A.38. □

Lemma A.4. *After all $\hat{h}_n$ are computed, local errors $e_n = (1-\gamma)\gamma W_n^+ \Delta h_{n+1}$.*

Proof.

$$\begin{aligned} e_n &= (1-\gamma)\Delta h_n \\ &= \gamma(1-\gamma)W^+ \Delta h_{n+1} \\ &= \gamma W^+ e_{n+1}, \end{aligned} \tag{A.40}$$

where the first and third lines are computed using lemma A.3 and the second line using definition A.2. □

These lemmas can be used to show local errors are smaller than but proportional to the global loss GN update w.r.t. hidden layer activities. For example, e_n can be expressed as follows:

Proposition A.3. *Consider a linear MLP trained with IL-GN. Assume $(W_n^+ ... W_{N-1}^+) = (W_{N-1} ... W_n)^+$ for all n. In this case, $e_n = \gamma'_n (W_{N-1} ... W_n)^+ \frac{-\partial L}{\partial h_N}$, where $\gamma'_n = \gamma^{(N-n)}(1-\gamma)$.*

Proof. According to definition A.2, $\hat{h}_n = h_n + \gamma W_n^+ \Delta h_{n+1}$, which implies $\Delta h_n = \gamma W_n^+ \Delta h_{n+1}$. Applying this same operation recursively from the output layer backward we get $\Delta h_n =$

$\gamma^{N-n}(W_n^+...W_{N-1}^+)\Delta h_N$.

Now we substitute $\gamma^{N-n}(W_n^+...W_{N-1}^+)\Delta h_N$ for $\hat{h}_n$ in lemma A.3, which results in $e_n = \gamma^{N-n}(1-\gamma)(W_{N-1}...W_n)^+\Delta h_N$. Finally, we note that $\Delta h_N = \frac{-\partial L}{\partial h_N}$, and thus $e_n = \gamma^{(N-n)}(1-\gamma)(W_{N-1}...W_n)^+\frac{-\partial L}{\partial h_N}$. □

Thus, local errors e_n can be seen as scaled down version of the global GN update, which is $\Delta h_n(W_{N-1}...W_n)^+\frac{-\partial L}{\partial h_N} = J_{n,N}^+\frac{-\partial L}{\partial h_N}$. The scaling is such that the nearer e_n is to the input layer the more scaled down the error is.

A.4 Stability of IL versus BP

In this section, we analyze certain stability properties of IL and BP algorithms. We begin by briefly discussing the unconditional stability of IL-prox, and equivalent G-IL algorithms. These algorithms use the NLMS update rule, so we next analyze the case where the LMS rule is used in an IL algorithm to update weights, as this case is more easily compared to G-BP algorithms which perform and LMS update over weights. In particular, we analyze how the FF output layer values h_N change after weight updates are applied to a linear network using IL-GN with the LMS rule and using an analogous BP algorithm, which we call BP-GN. We show IL-GN, trained with the LMS rule, pushes output layer activities $\hat{h}_N$ toward the target y (down it loss gradient) *for any positive learning rate*, while BP-GN only does this for a small finite range of learning rates. We focus on linear networks because constraining neuron activities with non-linearites may improve the stability of an algorithm by limiting the possible values a neuron may take or, e.g., by decreasing the magnitude of weight updates. We want to separate the effects of learning rules/algorithms on stability from those imposed by architectural constraints, so we focus on linear networks.

These theorems show that the way IL-GN computes and uses targets in its weight updates is

a key mechanism that aids stability. In particular, IL-GN computes weight gradients for W_n using pre-synaptic target $\hat{h}_n$, while BP-GN uses pre-synaptic FF activities h_n. This is the main difference between the two algorithms. The LMS update in the IL algorithm in linear networks is $e_{n+1}\hat{h}_n^T = (\hat{h}_{n+1} - W_n\hat{h}_n^T)\hat{h}_n^T$, while the BP update is $e_{n+1}h_n^T = (\hat{h}_{n+1} - W_n h_n)h_n^T$. IL updates thus 'chain together' the local learning problems such that the local prediction target at one layer, is the input to the next local prediction problem. G-BP updates do not have this property. The input to one local prediction problem is the FF activity, which is computed independently from local prediction targets. These results show that the way IL chains together the local prediction problems plays an important role in its stability. In particular, it shows it plays an important role in ensuring that output layer FF activities h_N change in the desired *direction* across a large range of learning rates. If we update weights like BP and do not chain the local learning problems together in this way, we lose this stability guarantee.

Ensuring h_N changes in the desired direction for any learning does not guarantee the loss will be minimized, since, e.g., h_N may overshoot the global target significantly. As we explain next, however, one can ensure minimization across any positive learning rate using the strategy of IL-prox, which uses the NLMS rule and uses the learning rate to control the output layer target rather than to scale the weight updates.

A.4.1 The Unconditional Stability of IL-prox

Implicit SGD is *unconditionally stable* in the sense that the proximal algorithm (equation A.3), which is equivalent to performing an implicit SGD update, monotonically decreases the loss function L for $0 < \alpha$. This can be seen from the proximal update, equation A.3, from which it trivially follows $L(\theta^{(b+1)}) + \frac{1}{\alpha 2}\|\theta^{(b+1)} - \theta^{(b)}\|^2 \leq L(\theta^{(b)})$. Theorem A.2 shows that IL-prox and G-IL, under the γ settings specified in theorem 4.1, are equivalent to an

implicit SGD update, and thus are unconditionally stable. Our simulations find that our implementation of IL-prox indeed displays this property of unconditional stability (tables A.1 and 4.4). It is worth, however, briefly discussing the mechanics of how this unconditional stability comes about in the IL-prox algorithm (and equivalent G-IL algorithms). First, theorem 4.1, tells us that the learning rate α of IL-prox (and G-IL under certain γ settings), is only used to determine how much $\hat{h}_N$ is pushed toward global target y during the inference phase. More specifically the gradient of the proximal update w.r.t. $\hat{h}_N$ is

$$\frac{\partial Prox}{\partial \hat{h}_N} = \frac{\partial L}{\partial \hat{h}_N} + \frac{1}{\|\hat{h}_{N-1}\|^2 \alpha} e_N, \tag{A.41}$$

where $e_N = \hat{h}_N - p_N$ and $p_N = W_{N-1} f(\hat{h}_{N-1})$. Let's assume a linear network and $L = \frac{1}{2}\|y - \hat{h}_N\|^2$. One way to compute the update of $\hat{h}_N$ would be to take the negative of the gradient, set equal to zero and solve for $\hat{h}_N$:

$$\begin{aligned} 0 &= -(y - \hat{h}_N) - \frac{1}{\|\hat{h}_{N-1}\|^2 \alpha} e_N, \\ \hat{h}_N &= \frac{\|\hat{h}_{N-1}\|^2 \alpha}{1 + \|\hat{h}_{N-1}\|^2 \alpha} y + \frac{1}{1 + \|\hat{h}_{N-1}\|^2 \alpha} p_N. \end{aligned} \tag{A.42}$$

Thus, each iteration of the inference phase we update $\hat{h}_N$ to a weighted average between p_N and y. The learning rate is only used to determine the weighting between these two terms. As $\alpha \to \infty$, $\hat{h}_N \to y$ and as $\alpha \to 0$, $\hat{h}_N \to p_N$. According to lemma A.1, $\hat{h}_N$ becomes the FF output layer value after the weights are updated (since the NLMS rule is used by IL-prox). Thus, when $\alpha \to \infty$, IL-prox finds a set of weights that best minimize the loss and as $\alpha \to 0$, weight are unchanged, which is the exact same property the proximal algorithm has (equation A.3). Intuitively, we see that in any case the loss is either minimized or remains unchanged (when $\alpha = 0$). IL-prox, and the proximal algorithm, thus gain their stability from the fact they do not use the learning rate to scale weight updates (as explicit SGD does) but rather uses it to determine the relative weighting of the loss term and regularization term in

the proximal operator.

A.4.2 Gauss-Newton IL Updates are Minimum-Norm and Stable

Here we show that the IL-GN weight updates collectively minimizes global loss along its minimum norm path for any positive learning rate in linear networks. This theorem is proven true under the conditions where $\hat{h}_n^T p_n > 0$, $h_n^T \hat{h}_n > 0$, and $\hat{h}_n^T \hat{h}_n > \hat{h}_n^T p_n$ for all n. This means local predictions, p_n, and initial activities, h_n must be within 90 degrees of sub-targets. This is an easily satisfied condition as long as targets are not moved far from initial activities. Also, the inequality $\hat{h}_n^T \hat{h}_n > \hat{h}_n^T p_n$ is generally true since the magnitudes of $\hat{h}_n$ and p_n will be similar and $\hat{h}_n$ is always more similar to itself than p_n. Thus, these conditions generally hold in practice.

Theorem A.3. *Assume $\hat{h}_n^T p_n > 0$, $\hat{h}_n^T h_n > 0$, and $\hat{h}_n^T \hat{h}_n > \hat{h}_n^T p_n$ for all n. For a mini-batch of size 1, the IL-GN weight updates applied to a linear MLP at iteration b collectively push h_N down its global loss gradient toward the target for any positive learning rate α: $h_N^{(b+1)} = h_N^{(b)} - j^{(b)} \frac{\partial L^{(b)}}{\partial h_N^{(b)}}$ where $j^{(b)}$ is a positive scalar.*

Proof. We define the linear network after weight updates are applied using a recursive formulation. Let $\hat{W}_n^* = \hat{W}_{n+1}^*(W_n - \Delta W_n)$ for all $n < N-1$, and let $\hat{W}_{N-1}^* = W_{N-1} - \Delta W_{N-1}$. We can now express the output of this updated network given input h_0 as $\hat{W}_0^* h_0$. We assume a linear gradient update $\Delta W_n = -\alpha e_{n+1} \hat{h}_n^T$, where α is the learning rate and $e_{n+1} = \hat{h}_{n+1} - p_{n+1}$.

The feedforward pass of the updated MLP at iteration $b+1$ is described as follows:

$$\begin{aligned} h_N^{(b+1)} &= \hat{W}_0^* h_0^{(b)} = \hat{W}_1^*(W_0 - \Delta W_0) h_0^{(b)} \\ &= \hat{W}_1^* h_1^{(b)} + \alpha \hat{W}_1^* e_1^{(b)} \hat{h}_0^{T(b)} h_0^{(b)} \end{aligned} \tag{A.43}$$

Note that $\hat{h}_0 = h_0$ and is the same across all training iterations, but $\hat{h}_n \neq h_n$ for $n > 0$.

Let $c_n = \alpha \hat{h}_{n-1}^T h_{n-1}$ such that $\alpha \hat{W}_1^* e_1 \hat{h}_0^T h_0 = c_1 \hat{W}_1^* e_1$. Because $\alpha > 0$ and because $\hat{h}_n^T h_n > 0$ for all n, $c_n > 0$.

Notice that the first term $\hat{W}_1^* h_1$ can be expanded recursively using the same expansion of $\hat{W}_0^* h_0$ in equation A.43:

$$\begin{aligned} h_N^{(b+1)} &= \hat{W}_1^* h_1^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \\ &= \hat{W}_2^* h_2^{(b)} + c_1 \hat{W}_2^* e_2^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \\ &... \\ &= h_N + c_N e_N^{(b)} + c_{N-1} \hat{W}_{N-1}^* e_{N-1}^{(b)} + ... + c_2 \hat{W}_2^* e_2^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \end{aligned} \tag{A.44}$$

The leftmost terms $h_N + c_N e_N$ was computed as follows: $(W_{N-1} - \Delta W_{N-1}) h_{N-1} = h_N + \alpha e_N \hat{h}_{N-1}^T h_{N-1} = h_N + c_N e_N$.

Now we need to expand $\hat{W}_n^*$ for all n in equation A.44.

$$\begin{aligned} \hat{W}_n^* e_n^{(b)} &= \hat{W}_{n+1}^* (W_n - \Delta W_n) e_n \\ &= \hat{W}_{n+1}^* (W_n e_n - (\Delta W_n \hat{h}_n - \Delta W_n p_n)) \\ &= \hat{W}_{n+1}^* (W_n e_n + \alpha (e_{n+1} \hat{h}_n^T \hat{h}_n - e_{n+1} \hat{h}_n^T p_n)) \\ &= \hat{W}_{n+1}^* (W_n e_n + e_{n+1} \alpha (\hat{h}_n^T \hat{h}_n - \hat{h}_n^T p_n)) \\ &= \hat{W}_{n+1}^* (W_n e_n + k_n e_{n+1}), \end{aligned} \tag{A.45}$$

Where $k_n = \alpha(\hat{h}_n^T \hat{h}_n - \hat{h}_n^T p_n)$ and is a positive scalar, since we assume $\hat{h}_n^T \hat{h}_n > \hat{h}_n^T p_n$.

We now simplify using lemma A.4, which states $e_n = \gamma W^+ e_{n+1}$:

$$\begin{aligned}\hat{W}_n^* e_n^{(b)} &= \hat{W}_{n+1}^*(\gamma W_n W_n^+ e_{n+1}^{(b)} + k_n e_{n+1}^{(b)}) \\ &= d_n \hat{W}_{n+1}^* e_{n+1}^{(b)},\end{aligned} \tag{A.46}$$

where $d_n = k_n + \gamma$ and clearly $d_n > 0$.

We now apply the same derivation of $\hat{W}_n^* e_n^{(b)}$ to all $\hat{W}$ in equation A.44 to yield the following:

$$\begin{aligned}\hat{W}_n^* e_n^{(b)} &= (d_n d_{n+1} ... d_{N-2}) \hat{W}_{N-1}^* e_{N-1}^{(b)} \\ &= (\prod_{i=n}^{N-2} d_i)(W_{N-1} - \Delta W_{N-1}) e_{N-1}^{(b)} \\ &= (\prod_{i=n}^{N-2} d_i)(W_{N-1} e_{N-1}^{(b)} + \alpha e_N \hat{h}_{N-1}^{T(b)} e_{N-1}^{(b)}) \\ &= (\prod_{i=n}^{N-2} d_i)(W_{N-1} e_{N-1}^{(b)} + \alpha e_N (\hat{h}_{N-1}^{T(b)} \hat{h}_{N-1}^{(b)} - \hat{h}_{N-1}^{T(b)} p_{N-1}^{(b)}) \\ &= (\prod_{i=n}^{N-2} d_i)((1-\gamma) W_{N-1} W_{N-1}^+ e_N^{(b)} + k_{N-1} e_N^{(b)}) \\ &= (\prod_{i=n}^{N-1} d_i) e_N^{(b)}\end{aligned} \tag{A.47}$$

Note that $(\prod_{i=n}^{N-1} d_i)$ is a positive scalar since it is the product of positive scalars.

According to lemma A.3 $e_N = (1-\gamma)\Delta h_N$ where we assume $\Delta h_N = -\frac{\partial L}{\partial h_N}$. Let $g_n = (1-\gamma)(\prod_{i=n}^{N-1} d_i)$, and notice $g_n > 0$. We can now rewrite equation A.44 in terms of $\frac{L}{h_N}$:

$$
\begin{aligned}
h_N^{(b+1)} &= h_N + c_N e_N^{(b)} + c_{N-1} g_{N-1} e_N^{(b)} + ... + c_2 g_2 e_N^{(b)} + c_1 g_1 e_N^{(b)} \\
&= h_N - j \frac{\partial L}{\partial h_N}^{(b)},
\end{aligned}
\tag{A.48}
$$

where $j = c_N + \sum_{i=n}^{N-1} c_i g_i$ and $j > 0$ is both $c > 0$ and $g > 0$. Hence, the weight updates applied at iteration b will collectively move the FF output layer values h_N down its loss gradient $h_N - j e_N^{(b)}$ for any positive value of α. □

It should be noted that moving h_N down its loss gradient will not necessarily minimize the loss e_N, given that j may be large and could cause h_N to significantly overshoot $\hat{h}_N$ preventing convergence.

A.4.3 Gauss-Newton Back Propagation is Minimum-Norm but Only Conditionally Stable

In this section, we show that, unlike IL-GN, weight updates performed by an analogous BP based network, we call BP-GN, are not guaranteed to collectively move the global prediction down its loss gradient each training iteration. Meulemans et al. [74] showed that each individual weight update in a BP-GN network will push the global prediction h_N down its loss gradient when its effects on h_N are considered independently of other weight updates in the network. Here we show this property does not hold for any learning rate when the weight updates are considered collectively.

Definition A.3. *Gauss-Newton backpropagation (BP-GN). Assume* $e_n = \hat{h}_n - h_n$. *The weight update of BP-GN is the general BP weight update:* $\Delta W_n = -e_{n+1} h_n^T$. *Local targets are computed at each hidden layer are computed using the following equation:* $\hat{h}_n = h_n + W_n^+ e_{n+1} = h_n + W_n^+ \hat{h}_{n+1} - W_n^+ h_{n+1}$.

We can now apply the same analysis we did for IL-GN to BP-GN, which proves the following.

Theorem A.4. *Consider a linear MLP trained with BP-GN. For a mini-batch of size 1, the BP-GN weight updates applied at iteration b only push the global prediction down its global loss gradient* $h_N^{(b+1)} = h_N^{(b)} - j^{(b)} \frac{\partial L^{(b)}}{\partial h_N^{*(b)}}$ *for a finite range of* α.

Proof. Here we follow the same notation and analysis as the proof for theorem A.3. Let $\hat{W}_n^* = \hat{W}_{n+1}^*(W_n - \Delta W_n)$ for all $n < N-1$, and let $\hat{W}_{N-1}^* = W_{N-1} - \Delta W_{N-1}$. We assume a linear gradient update $\Delta W_n = -\alpha e_{n+1} h^T$ (as in definition A.3), where α is the learning rate.

First, we can describe the FF pass of the updated MLP trained with GN-TP at iteration $b+1$ as we did above for IL-GN (see equation A.43):

$$h_N^{(b+1)} = \hat{W}_1^* h_1^{(b)} + \alpha \hat{W}_1^* e_1^{(b)} h_0^{T(b)} h_0^{(b)} \tag{A.49}$$

Let $c_n = \alpha h_{n-1}^T h_{n-1}$ such that $\alpha \hat{W}_1^* e_1 h_0^T h_0 = c_1 \hat{W}_1^* e_1$. Because $1 > \alpha > 0$ and because $h_n^T h_n > 0$ for all n, $c_n > 0$. We can see this equation is the same as equation A.43, except e_1 and ΔW_0 were defined according to GN-TP rather than IL-GN.

As above, the first term $\hat{W}_1^* h_1$ can be expanded recursively:

$$\begin{aligned} h_N^{(b+1)} &= \hat{W}_1^* h_1^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \\ &= \hat{W}_2^* h_2^{(b)} + c_1 \hat{W}_2^* e_2^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \\ &... \\ &= h_N - c_N e_N^{(b)} + c_{N-1} \hat{W}_{N-1}^* e_{N-1}^{(b)} + ... + c_2 \hat{W}_2^* e_2^{(b)} + c_1 \hat{W}_1^* e_1^{(b)} \end{aligned} \tag{A.50}$$

Equation A.44 shows that for IL-GN, each of these recursively computed terms push the initial prediction down the global loss gradient. However, the same is not true here of GN-TP:

$$\begin{aligned}
\hat{W}_n^* e_n^{(b)} &= \hat{W}_{n+1}^*(W_n - \Delta W_n)e_n \\
&= \hat{W}_{n+1}^*(W_n e_n - (\Delta W_n \hat{h}_n - \Delta W_n h_n)) \\
&= \hat{W}_{n+1}^*(W_n e_n + \alpha(e_{n+1} h_n^T \hat{h}_n - e_{n+1} h_n^T h_n)) \\
&= \hat{W}_{n+1}^*(W_n e_n + \alpha(h_n^T \hat{h}_n - h_n^T h_n)e_{n+1}).
\end{aligned} \tag{A.51}$$

$W_n e_n$ can be rewritten in terms of e_{n+1} as follows: $W_n e_n = W_n W_n^+ e_{n+1}$. Notice that it will often be the case that $h_n^T \hat{h}_n < h_n^T h_n$, since generally $h_n \neq h_n$. Thus, $\alpha(h_n^T \hat{h}_n - h_n^T h_n)$ will often be a *negative* scalar. This is distinct from the corresponding term in equation for IL-GN $\alpha(h_n^T h_n - \hat{h}_n^T h_n)$, which will generally be a positive scalar. The difference is due to the IL-GN update being conditioned on the presynaptic sub-target $\hat{h}_n$ rather than presynaptic FF activity h_n. Also notice there is no further dampening term $\gamma(1-\gamma)$ as there is with IL-GN.

Let $k_n = \alpha(h_n^T \hat{h}_n - h_n^T h_n)$ and $d_n = (1 + k_n)$. For reasons just noted, d_n is not guaranteed to be positive for all $1 > \alpha > 0$. Continuing the derivation above:

$$
\begin{aligned}
\hat{W}_n^* e_n^{(b)} &= \hat{W}_{n+1}^* (W_n W_n^+ e_{n+1} + k_n e_{n+1}) \\
&= \hat{W}_{n+1}^* (1 + k_n) e_{n+1} \\
&= d_n \hat{W}_{n+1}^* e_{n+1}^{(b)} \\
&= (d_n d_{n+1} ... d_{N-2}) \hat{W}_{N-1}^* e_{N-1}^{(b)} \\
&= (\prod_{i=n}^{N-1} d_i) e_N^{(b)}.
\end{aligned}
\tag{A.52}
$$

Because d_n is not guaranteed to be positive, then $(\prod_{i=n}^{N-1} d_i)$ is not guaranteed to be a positive scalar. Thus, the product of all d_n may or may not be positive for a given $1 > \alpha > 0$. Let $g_n = (\prod_{i=n}^{N-1} d_i)$. Also, note that $e_N = -\frac{\partial L}{\partial h_N}$. We can now rewrite equation A.49 in terms of e_N:

$$
\begin{aligned}
h_N^{(b+1)} &= h_N + c_N e_N^{(b)} + c_{N-1} g_{N-1} e_N^{(b)} + ... + c_2 g_2 e_N^{(b)} + c_1 g_1 e_N^{(b)} \\
&= h_N - j \frac{\partial L}{\partial h_N}^{(b)},
\end{aligned}
\tag{A.53}
$$

We can see that if enough g_n are negative it is possible that j is also negative, especially given that g_n are scaled by c_n, which will always be positive and may be larger than 1. Thus, GN-TP, unlike IL-GN, does not guarantee that for any positive value of α, the weight updates at iteration b will collectively move h_N down it loss gradient at $b+1$: GN-TP does not guarantee $j > 0$ for any $\alpha > 0$ or even any $1 > \alpha > 0$. □

In order for $j > 0$, a large number of g_n must be positive to offset any negative g_n. In order for $g_n > 0$, an even number of d_n in $g_n = (\prod_{i=n}^{N-1} d_i)$ must be negative. The simplest way to guarantee this is the case is to ensure $d_n > 0$ for all n. In order for all d_n to be positive in

each g_n we can see that alpha must be small since $d_n = 1 + k_n = 1 + \alpha(h_n^T \hat{h}_n - h_n^T h_n)$, and α must be small enough such that $k_n > -1$ for all n. This implies learning rate must be small: $\alpha < \frac{-1}{(h_n^T \hat{h}_n - h_n^T h_n)}$.

A.5 G-IL Avoids Interference Effects

The last section showed the IL-GN will push output layer values toward the global target for any positive learning rate, while BP-GN will only do so for a finite range of learning rates. Here we provide mathematical intuition for why this occurs. Specifically, we describe how BP weight updates necessarily interfere with each other's ability to minimize the global loss and this interference generally grows with learning rate, whereas IL weight updates do not necessarily interference and may actually improve each other's ability to minimize loss. In this sense, IL updates are more compatible with each other than BP updates. In line with this intuitive description, we show that under certain assumptions IL-GN updates (at hidden layers) are often *most effective* at pushing the output layer values toward the target when applied in conjunction with other updates than when applied alone, while the opposite is true of BP-GN. Figure A.4 provides evidence these theoretical results hold true in practice for IL-SGD and BP-SGD networks when small learning rates are used.

A.5.1 A Mathematical Intuition for Interference Effects

Consider the BP weight update: $\Delta W_n = \alpha_n e_{n+1}^{(b)} f(h_n)^{T(b)}$, where $h_n = W_{n-1} f(h_{n-1})$ and $n > 0$. Now, consider the effect of this weight update on the FF values at hidden layer $n+1$

given the same input $x^{(b+1)} = x^{(b)}$

$$\begin{aligned} h_{n+1}^{(b+1)} = W_n^{(b+1)} f(h_n^{(b+1)}) &= (\Delta W_n^{(b)} + W_n^{(b)}) f(h_n^{(b+1)}) \\ &= \Delta W_n^{(b)} f(h_n^{(b+1)}) + W_n^{(b)} f(h_n^{(b+1)}) \\ &= \alpha_n (f(h_n^{(b)})^T f(h^{(b+1)})_n)) e_{n+1}^{(b)} + h_{n+1}^{(b)}. \end{aligned} \tag{A.54}$$

We can see that the ability of the weight update to drive its output down the error gradient (where $e_{n+1}^{(b)} = \frac{-\partial L}{\partial h_{n+1}^{(b)}}$, see section 3.2) depends directly on the similarity between pre-synaptic activities before and after the weight update. In particular, if the two are similar, i.e. $f(h_n^{(b)})^T f(h_n^{(b+1)}) > 0$, then the weight update will affect FF values in the desired direction. However, if the two are orthogonal, i.e. $f(h_n^{(b)})^T f(h_n^{(b+1)}) = 0$, there will be no effect, and if dissimilar, i.e. $f(h_n^{(b)})^T f(h_n^{(b+1)}) < 0$, they will affect FF values in an undesired direction (up rather down the loss gradient).

The challenge for the BP update is that if all weights are updated at iteration b then FF activities will change, which will generally make $f(h_n^{(b)})$ and $f(h_n^{(b+1)})$ less similar. Additionally, presynaptic activities will generally become less similar as α_n is increased because changes to weights, and thus changes to FF activities, at previous layers will increase. Only when learning rates at previous layers $\to 0$ does $f(h_n)^{(b)} = f(h_n)^{(b+1)}$ because an α of zero means no change in weights or FF activities. We can describe this as a sort of interference effect: non-zero weight updates elsewhere in the network reduce the ability of ΔW_n to drive FF activities at layer $n+1$ in the desired direction.[1]

Now, consider the G-IL weight update: $\Delta W_n = \alpha_n e_{n+1}^{(b)} f(\hat{h}_n^{T(b)})$, where $n > 0$. The effect of this weight update on the FF values at hidden layer $n+1$ given the same input, $x^{(b+1)} = x^{(b)}$,

[1]Interference might be reduced by using non-linearities that keep activities positive, e.g., ReLU, which guarantees $f(h_n^{T(b)}) f(h_n^{(b+1)}) \geq 0$. However, for non-zero learning rates it would still be the case that updates to weights at layers before n would reduce similarity, i.e., generally $f(h_n^{T(b)}) f(h_n^{(b)}) > f(h_n^{T(b)}) f(h_n^{(b+1)})$, and thus reduce the magnitude change to post-synaptic FF values.

can be described as follows:

$$\begin{aligned} h_{n+1}^{(b+1)} = W_n^{(b+1)} f(h_n^{(b+1)}) &= (\Delta W_n^{(b)} + W_n^{(b)}) f(h_n^{(b+1)}) \\ &= \Delta W_n^{(b)} f(h_n^{(b+1)}) + W_n^{(b)} f(h_n^{(b+1)}) \\ &= \alpha_n (f(\hat{h}_n)^{T(b)} f(h_n^{(b+1)})) e_{n+1}^{(b)} + h_{n+1}^{(b)}. \end{aligned} \tag{A.55}$$

Unlike with G-BP we can see that the ability of the weight update to drive its output down the error gradient $e_{n+1}^{(b)}$ depends directly on the similarity between pre-synaptic *optimized* activities and the FF activities after the weight update. If the two are similar, i.e. $f(\hat{h}_n^{T(b)}) f(h_n^{(b+1)}) > 0$, then the weight update will affect FF values in the desired direction. If the two are orthogonal there will be no effect, and if they are dissimilar they will affect FF values in an undesired direction.

What's interesting about the G-IL update is that with a properly tuned α_n, $f(\hat{h}_n)^{(b)}$ and $f(h_n)^{(b+1)}$ will get *more* similar when weights prior to layer n are updated. As lemma A.1 shows, under a specific (typically) non-zero value of α_n it is actually the case that $f(\hat{h}_n^{(b)}) = f(h_n^{(b+1)})$. This value of α_n is the true solution to G-IL update $\arg\min_W F$ that the LMS learning rule above attempts to approximate. Intuitively, G-IL largely avoids interference because $f(\hat{h}_n^{(b)})$ accurately *anticipates* the value of $f(h_n^{(b+1)})$ and by doing so helps to ensure $f(\hat{h}_n)^{T(b)} f(h_n)^{(b+1)} > 0$.

A.5.2 G-IL Weight Updates are Most Effective when Applied Together

In previous sections we compared the direction of the effects of weight updates on a linear MLP output layer in an IL and BP network. We now assess the magnitude of the affects of IL and BP weight updates on the output layer of a linear MLP with a single hidden layer. In

sum, we show that under certain assumptions, updates in IL-GN should have greater effects on the output layer when applied together than when applied alone, while the opposite is true for BP-GN. This shows that interference effects not only can alter the direction the output layer changes with weight updates but also the magnitude of the change in output layer activities.

We analyze a linear MLP with a single hidden layer. Let $h_2^{(b)}$ be the output layer values at iteration b, $h_2^{(b+1/2),0}$ be the output layer values after W_0 is updated alone, $h_2^{(b+1/2),1}$ be the output layer values after W_1 is updated alone, and $h_2^{(b+1)}$ be the output layer values after both weight matrices are updated. Let c_n and g_n be defined as above in the proof of theorem A.3. We assume that $g_n > 1$ which will often (though not always) be true in practice for reasons noted above (see theorem A.3). We also assume $c_1 = c_2$ as it significantly simplifies the calculations. We test how well these theoretical results hold in practice and find they hold approximately for small learning rates (see figures A.4 for details).

Proposition A.4. *Consider a linear MLP with a single hidden layer trained with IL-GN, mini-batch size 1, at iteration b. Assume $g_1 > 1$, $c_1 = c_2$. Under these assumptions, and those in theorem A.3, each weight update, ΔW_0 and ΔW_1 has a larger effect on the output layer when applied in conjunction with each other than when applied alone: $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| > \|h_2^{(b+1/2),1} - h_2^{(b)}\|$ and $\|h_2^{(b+1)} - h_2^{(b+1/2),1}\| > \|h_2^{(b+1/2),0} - h_2^{(b)}\|$*

Proof. The effect of ΔW_0 on the output layer $h_N = h_2$ when applied alone is

$$\begin{aligned} h_2^{(b+1/2),0} = W_1^{(b)}(W_0^{(b)} + \Delta W_0^{(b)})h_0^{(b)} &= h_2^{(b)} + W_1^{(b)}\alpha e_1^{(b)} h_0^{(b)T} h_0^{(b)} \\ &= h_2^{(b)} + \alpha\gamma W_1^{(b)} W_1^{(b)+} e_2^{(b)} h_0^{(b)T} h_0^{(b)} \\ &= h_2^{(b)} + \alpha\gamma h_0^{(b)T} h_0^{(b)} e_2^{(b)} \\ &= h_2^{(b)} + \gamma c_1 e_2^{(b)}, \end{aligned} \tag{A.56}$$

where the second line is computed using A.4, which says $e_n = \gamma W_n^+ e_{n+1}$. Here $c_1 = \alpha h_0^{(b)T} h_0^{(b)}$ and is clearly a positive scalar.

The effect of ΔW_1 on the output layer, h_2, when applied alone is

$$\begin{aligned} h_2^{(b+1/2),1} &= (W_1^{(b)} + \Delta W_1^{(b)}) W_0^{(b)} h_0^{(b)} = h_2^{(b)} + \Delta W_1^{(b)} h_1^{(b)} \\ &= h_2^{(b)} + \alpha \hat{h}_1^{(b)T} h_1^{(b)} e_2^{(b)} = h_2^{(b)} + c_2 e_2^{(b)}, \end{aligned} \tag{A.57}$$

where $c_2 = \alpha \hat{h}_1^{(b)T} h_1^{(b)}$ and is a scalar.

Finally, when both updates are applied we get the same expression as that in equation A.48

$$h_2^{(b+1)} = h_2^{(b)} + c_2 e_2^{(b)} + g_1 c_1 e_2^{(b)}. \tag{A.58}$$

See theorem A.3 for details. Here c_1 and c_2 are defined as above. Since we make the same assumptions as theorem A.3, g_1 is a positive scalar: $g_1 = \gamma + \alpha(\hat{h}_1^{(b)T} \hat{h}_1^{(b)} - \hat{h}_1^{(b)T} p_1^{(b)})$.

Now we have

$$\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| = \|c_2 e_2^{(b)} + g_1 c_1 e_2^{(b)} - \gamma c_1 e_2^{(b)}\| = \|(1 + g_1 - \gamma) c_1 e_2^{(b)}\|, \tag{A.59}$$

which is simplifying using the assumption $c_1 = c_2$. Second, $\|h_2^{(b+1/2),1} - h_2^{(b)}\| = \|c_2 e_2^{(b)}\|$. Under our assumptions that $g_1 > 1$ and $c_1 = c_2$, it clearly follows that $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| > \|h_2^{(b+1/2),1} - h_2^{(b)}\|$.

Next, we can see that $\|h_2^{(b+1)} - h_2^{(b+1/2),1}\| = \|g_1 c_1 e_2^{(b)}\|$ and $\|h_2^{(b+1/2),0} - h_2^{(b)}\| = \|\gamma c_1 e_2^{(b)}\|$. Since we assume $g_1 > 1$ and $c_1 = c_2$ and $\gamma < 1$, it clearly follows that $\|h_2^{(b+1)} - h_2^{(b+1/2),1}\| >$

$\|h_2^{(b+1/2),0} - h_2^{(b)}\|$. □

A.5.3 BP-GN Updates are Most Effective When Applied Alone

Next, we perform the same analysis on BP-GN and find the opposite is true: BP-GN updates are most effective when applied alone. We assume $g_n < 1$, which is plausible given that g_n is typically a negative scalar for reasons noted in the proof for theorem A.4. We also make the same assumption as we did for IL-GN that $c_1 = c_2$.

Proposition A.5. *Consider a linear MLP with a single hidden layer trained with BP-GN, mini-batch size 1, at iteration b. Assume generally $g_n < 1$ and $c_1 = c_2$. The weight update to ΔW_0 and ΔW_1 either have a larger effect on the output layer when applied alone: $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| < \|h_2^{(b+1/2),1} - h_2^{(b)}\|$ and $\|h_2^{(b+1)} - h_2^{(b+1/2),1}\| < \|h_2^{(b+1/2),0} - h_2^{(b)}\|$ or they push the output layer activities up the loss gradient (i.e., away from global target).*

Proof. Following equation A.56 above, the effect of ΔW_0 on the output layer is $h_2^{(b)} + c_1 e_2^{(b)}$, except c_1 is now computed $c_1 = h_0^{T(b)} h_0^{(b)}$.

The effect of ΔW_1 on the output layer, h_2, when the update is applied alone is

$$\begin{aligned} h_2^{(b+1/2),1} &= (W_1^{(b)} + \Delta W_1^{(b)}) W_0^{(b)} h_0^{(b)} = h_2^{(b)} + \Delta W_1^{(b)} h_1^{(b)} \\ &= h_2^{(b)} + \alpha h_1^{(b)T} h_1^{(b)} e_2^{(b)} = h_2^{(b)} + c_2 e_2^{(b)}, \end{aligned} \tag{A.60}$$

where $c_2 = \alpha h_1^{(b)T} h_1^{(b)}$ and is generally a large positive scalar.

Finally, using the derivation from theorem A.4, when both updates are applied we get

$$h_2^{(b+1)} = h_2^{(b)} + c_2 e_2^{(b)} + c_1 g_1 e_2^{(b)} \tag{A.61}$$

where $g_1 = 1 + \alpha(h_1^{(b)T}\hat{h}_1^{(b)} - h_1^{(b)T}h_1^{(b)})$ (see theorem A.4 for details). As explained in theorem A.4, we can see that g_1 will often be negative since it will often be the case that $(h_1^{(b)T}\hat{h}_1^{(b)} - h_1^{(b)T}h_1^{(b)}) < 0$.

It follows from the assumption $c_1 = c_2$ that $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| = \|g_1 c_1 e_2^{(b)}\|$. Second, $\|h_2^{(b+1/2),1} - h_2^{(b)}\| = \|c_2 e_2^{(b)}\|$. Under our assumptions $g_1 < 1$ and $c_0 = c_1$, in the case where $-1 < g_1 < 1$, it follows $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| < \|h_2^{(b+1/2),1} - h_2^{(b)}\|$ and generally when $g_1 < -1$, and under assumption $c_1 = c_2$, we see from equation A.61 the output layer activities away from the target (i.e. up rather than down the loss gradient).

Similarly, $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\|^2 = \|g_1 c_1 e_2^{(b)}\|$ and $\|h_2^{(b+1/2),0} - h_2^{(b)}\| = \|c_1 e_2^{(b)}\|$. Under our assumptions $g_1 < 1$ and $c_1 = c_2$, in the case where $-1 < g_1 < 1$ it follows $\|h_2^{(b+1)} - h_2^{(b+1/2),0}\| < \|h_2^{(b+1/2),1} - h_2^{(b)}\|$ and again when $g_1 < -1$ it is the case the the output layer activities are pushed away from the target (i.e. up rather than down the loss gradient). □

Under the above assumptions, IL-GN weight updates do not interfere with one another, and in fact are more effective when applied together than alone. BP-GN weight updates, on the other hand, tend to be most effective when applied alone. We test how well these theoretical results are approximated by IL-SGD and BP-SGD through simulations on linear networks trained on regression tasks. Indeed, IL-SGD networks tend to have larger effects on the output layer when applied together than along, more often than their BP counterparts when small learning rates are used (see figure A.4).

A.6 Supplementary Results and Experiments

A.6.1 Cifar-10 Mini-batch Classification

In figure A.1, are training runs for Cifar-10 mini-batch size 64. We see that IL prox algorithms tend to discount the loss faster than BP-SGD and are comparable to BP-Adam in the first 2000 iterations. However, BP-SGD and BP-Adam converge to better accuracies. This suggests that IL algorithms tend to converge to shallower local minima that are nearer by the initial parameters than those minima reached by BP algorithms.

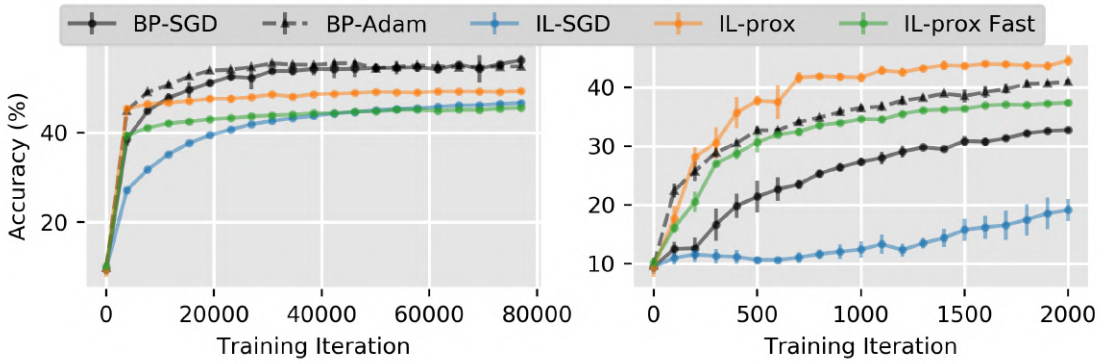

Figure A.1: Cifar-10 classification test accuracies during training with mini-batch size 64. Left, full training run. Right, a zoomed in look at test accuracies from the first 2000 training iterations.

A.6.2 Further Stability Analysis Results and Discussion

We run the same stability analysis as in table 4.4 for MNIST. BP-prox is unable to train in a stable manner for learning rates in the range of 2.5-100, and there is a clear decrease in accuracy as learning rate increases from .01 to 1. IL-prox's trains across all learning rates and even improves as learning rate increase from .01 to 1. As with the Cifar-10 results we again see IL-SGD is more stable than BP-SGD, and prox models are more stable than SGD models.

BP-prox uses the NLMS version of its update (equation 4.3) and uses the learning rate to adjust the output layer target. Like IL-prox, as $\alpha \to \infty$ then $\hat{h}_N \to y$ and as $\alpha \to 0$ then $\hat{h}_N \to h_N$. This use of the NLMS rule and learning rate improve stability. However, clearly these factors do not give BP-prox unconditional stability. The main difference between IL-prox and BP-prox is the way local targets are computed and used in the weight update. For example, while BP-prox uses presynaptic FF activity h_n in its update of W_n, IL-prox uses presynaptic target value $\hat{h}_n$ in its update of W_n. Clearly, these differences have a significant effect on stability.

Stability Test: MNIST Test Accuracy						
Model	lr=.01	lr=.1	lr=1	lr=2.5	lr=10	lr=100
BP-SGD	94.05(±.53)	–	–	–	–	–
IL-SGD	90.06(±.25)	94.49(±.21)	42.42(±44.67)	–	–	–
BP-prox	93.028(±.77)	90.86(±1.62)	55.87(±42.076)	–	–	–
IL-prox	89.30(±.32)	93.058(±.56)	93.57(±.32)	93.49(±.61)	93.49(±.94)	93.85(±.38)

Table A.1: Accuracy after 20,000 training iterations on MNIST mini-batch size 1. Fully connected networks with layer sizes 784-2x500-10. ReLU activations were used at hidden layers while softmax was used at output layer

A.6.3 MNIST Training with Mini-batch Size 1

Test Accuracy (mean±std.) w/ Best Learning Rate		
Model	MNIST	Fashion-MNIST
BP-SGD	97.096(±.131)	85.384(±.120)
BP-Adam	97.294(±.116)	85.896(±.081)
IL-SGD	96.938(±.102)	84.972(±.211)
IL-prox	96.650(±.091)	83.636(±.091)
IL-prox Fast	96.466(±.128)	83.402(±.169)

Table A.2: Test accuracies after 1 epoch of training with mini-batch size 1 (60,000 training iterations). Fully connected networks were used with dimensions 784-2x500-10.

The training runs for are plotted in figure A.2 below. Differences in performance between IL and BP algorithms are smaller with these data-sets than they are with CIFAR-10. IL-prox

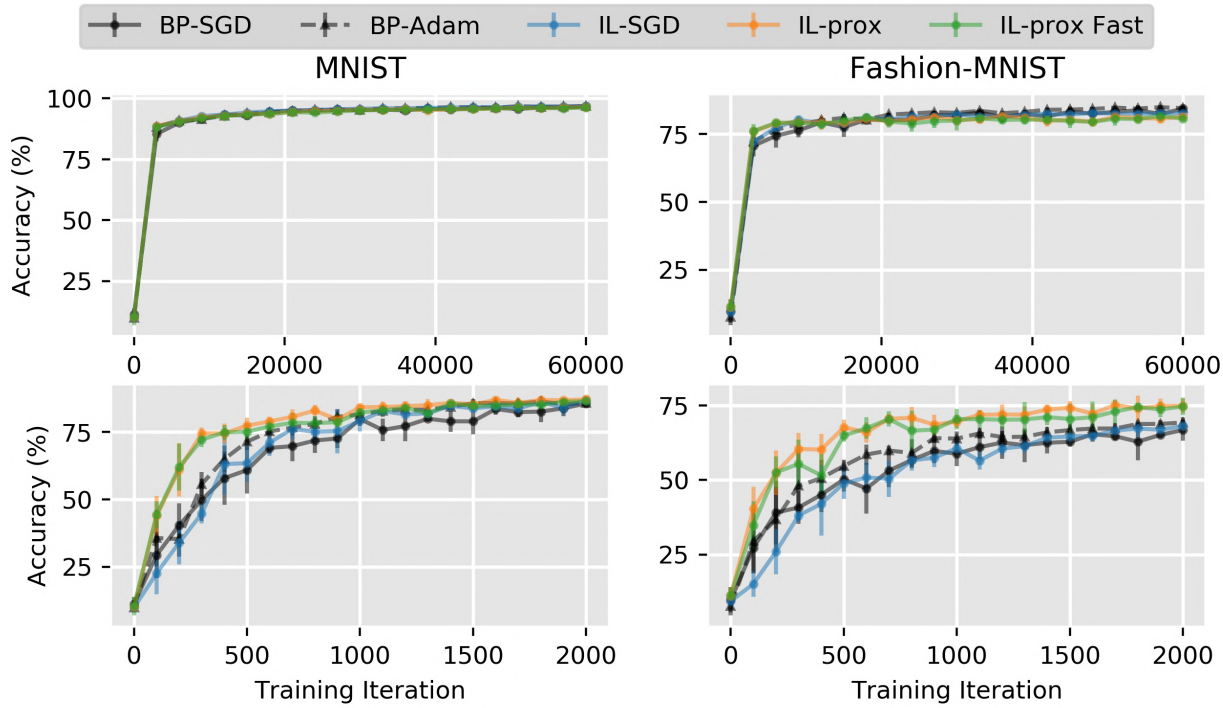

Figure A.2: Training runs for MNIST and Fashion-MNIST. Networks were fully connected dimensions 784-2x500-10. Models were trained mini-batch size 1 for one epoch (60,000 iterations). **Top Left** Full training run for MNIST. **Bottom Left** A zoomed in look at the first 2000 training iterations of MNIST. IL-prox algorithm improve accuracy slightly faster than other algorithms in the first 500 iterations. **Top Left** Full training run for Fashion-MNIST. **Bottom Left** A zoomed in look at the first 2000 training iterations of Fashion-MNIST. IL-prox algorithms improve accuracy faster than other algorithms for first 2000 training iterations.

improved accuracy slightly faster in the first few hundred or thousand iterations on both MNIST and Fashion-MNIST, but BP algorithms soon caught up.

A.6.4 Convolutional Networks

We also trained a small convolutional networks on CIFAR-10. Large mini-batches (size 64) were used and the Adam based algorithms were used for training (see 4.3), since IL algorithms worked best with Adam optimizers when large mini-batches were used. IL-prox Adam achieved essentially the same accuracy as BP-Adam, though IL-SGD did not achieve

comparable accuracy.

CIFAR-10 Test Accuracy (mean±std.) w/ Best Learning Rate		
BP-Adam	IL-Adam	IL-prox Adam
66.850(±.591)	54.802(±1.031)	66.704(±1.725)

Table A.3: Here we train on CIFAR-10 with mini-batches of size 64 for 77,000 iteration ($\approx$ 100 epochs). Runs were averaged over five seeds and the best score is shown. There were three convolutional layers followed by one fully connected layer. We used a similar architecture as [10]. Convolutions were (5x5, 64, 2), (5x5, 128, 2), (3x3, 256, 2). Fully connected layer was 1024x10. ReLU activations were used at hidden layers, and softmax at the output layer.

A.6.5 Data Constrained Tests

We test classification on F-MNIST and CIFAR-10 in the case where only 10, 100, or 500 data points are used from each category in the data set. Models are trained with mini-batch size 100. Grid search is used to find best learning rate. All models are trained to convergence and best accuracies, averaged across 5 seeds, are shown table A.4. Training runs can be seen in figure 4.4 and best accuracies.

Data Constrained Test						
	F-MNIST			CIFAR-10		
Model	n=10	n=100	n=500	n=10	n=100	n=500
BP-SGD	70.72(±.44)	79.41(±19)	84.00(±.10)	23.58(±.58)	33.18(±15)	44.54(±.08)
IL-SGD	72.94(±.19)	79.69(±.11)	83.63(±.24)	24.67(±.65)	34.63(±.18)	42.80(±24)
IL-prox	71.64(±.19)	78.90(±.24)	82.12(±.16)	24.00(±.56)	34.17(±.20)	39.80(±.45)
BP-Adam	70.99(±.16)	78.60(±.19)	84.09(±.11)	23.93(±.24)	34.11(±.18)	44.13(±.25)
IL-Adam	71.20(±.29)	80.52(±.16)	84.32(±.11)	24.07(±.13)	34.91(±.23)	44.41(±.13)
IL-prox Adam	70.96(±.47)	80.27(±.083)	84.16(±.10)	24.17(±.29)	34.58(±.22)	44.15(±.17)

Table A.4: Top test accuracies in constrained memory scenario. Networks were trained on subsets of F-MNIST and CIFAR-10 data sets, i.e., where only 10, 100, or 500 data-points were used from each category. Mini-batch size 100 was used. 5 seeds were trained to convergence for each model and the best score from each seed were averaged.

A.6.6 Output Layer Analysis

Theorem A.3 states that IL-GN, the closed form approximation of IL, updates parameters of linear MLPs in a way that changes FF output layer values ($h_N^{(b+1)} - h_N^{(b)}$) in the direction of the minimum-norm path of loss minimization for any positive learning rate. Theorem A.4 states that BP-GN updates parameters in a way that yields change in FF output layer values ($h_N^{(b+1)} - h_N^{(b)}$) in the direction of the minimum-norm path of loss minimization, but only for a finite range of positive learning rates. Here we test how well this result holds in practice in a network trained with IL-SGD. In particular, we test how closely change in FF output layer values $\Delta h_N = h_N^{(b+1)} - h_N^{(b)}$ align with the minimum norm path to the target $\Delta h_N^{min} = y - h_N^{(b)}$ using cosine similarity. For small linear networks trained on a regression task with, IL-SGD does indeed produce changes in the output (see A.3 left) very close in angle to the minimum norm path. This supports the claim that IL-GN is a useful approximation of G-IL. MLPs trained with BP-SGD produce changes in the output layer that are significantly less close to the minimum norm path. This suggests IL-SGD takes a shorter, more direct path toward local minimum than that of steepest descent. Similar results were found when ReLU activations were applied at hidden layers (see A.3 middle). Finally, we directly test theorem A.3 and theorem A.4 by exactly implementing IL-GN and BP-GN in a toy linear network. We measure the cosine similarity between the change in output FF values and the minimum norm change, as with the previous tests, and we do this over a range of learning rates. As predicted by theorem A.3 and A.4, IL-GN and BP-GN affect output layer values along minimum norm path toward the target. However, as predicted by theorem A.3 and A.4, we find BP-GN is significantly less stable than IL-GN (see A.3 right).

Propositions A.4 and A.5 show that under certain assumptions IL-GN weight updates should have a larger effect on h_N when applied together than when applied alone, and the opposite is generally true of BP-GN. We test how well this result holds in practice in IL-SGD and BP-SGD networks trained on a regression tasks. We compute the effect each individual

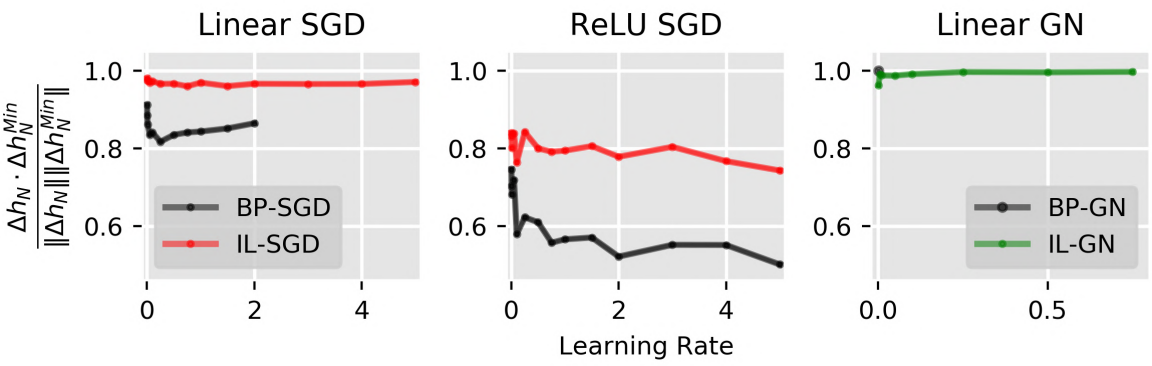

Figure A.3: (Left) Cosine similarity was measured between Δh_N and Δh_N^{Min} each training iteration across a range of learning rates in linear networks trained on a regression task. 95% confidence intervals are shown but are too small to be visible. (Middle) The same test was performed on non-linear networks that use ReLU at hidden layers. (Right) We run the same test for an actual IL-GN algorithm and BP-GN. As predicted by theorem A.3 and A.4 the IL-GN and BP-GN both push the output layer down its minimum-norm path, but BP-GN produces a minimum-norm Δh_N only for a tiny range of learning rates (see black dot upper left), while IL-GN is stable over a much larger range.

$\Delta W_n^{(b)}$ has on the output layer FF values when applied in conjunction with other weights: $\|h_N^{(b+1)} - h_N^{(b+1/2,\neg n)}\|$, where $h_N^{(b+1)}$ are the FF output values after updates are applied to all weights, and $h_N^{(b+1/2,\neg n)}$ are the output layer values and when all updates except ΔW_n are applied. We then compute $\|h_N^{(b+1/2,n)} - h_N^{(b)}\|$, where h_N^b are the FF output values before any updates are applied at the current training iteration, and $h_N^{(b+1/2,n)}$ are the output layer values after only ΔW_n is applied.

When $\|h_N^{b+1} - h_N^{(b+1/2,\neg n)}\|^2 \geq \|h_N^{(b+1/2,n)} - h_N^{(b)}\|$, we know ΔW_n had a larger affect on the output layer when applied in conjunction with other weights than when applied alone. When ΔW_n meets this condition we count it as compatible with the other weight updates. We then compute a compatibility score which stores a count of all compatible weight updates and uses it to compute the proportion of updates that were compatible.

$$CompatibilityScore = \frac{\#\text{Compatible } \Delta W_n}{\text{Total } \#\Delta W_n} \tag{A.62}$$

We find that less than 50% of weight updates in IL-SGD and BP-SGD networks are compatible though for smaller learning rates a greater proportion of IL updates are compatible than BP updates (see figure A.4). This supports the claim that IL updates interfere with one another less than do BP updates, both in their ability to minimize loss and in their ability to change the FF output layer values.

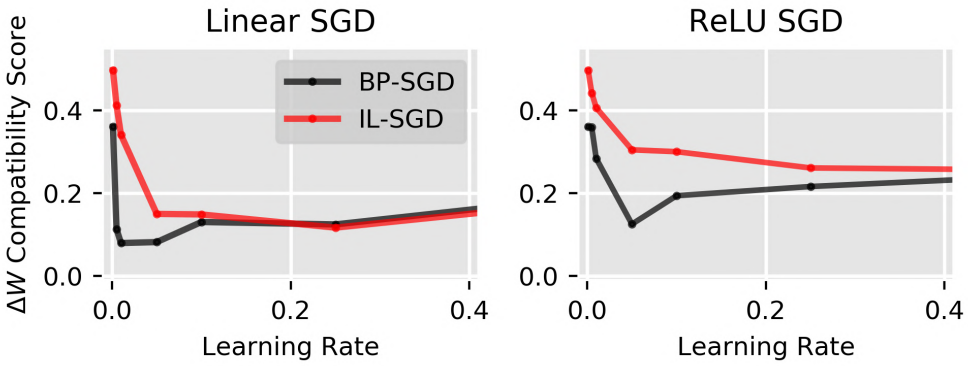

Figure A.4: (Left) In linear networks with small learning rates ($\alpha_n < .1$) IL weight updates have significantly larger compatibility scores (according to 2-sample T-test, 95% confidence intervals shown but are too small to see) than BP-SGD updates. (Right) The same trend exists in non-linear networks with ReLU at hidden layers for $\alpha_n < .4$.

A.6.7 Weight Norm Analysis

Here we report the mean and standard deviation of the parameter updates $\|\Delta\theta\|$ from the training runs reported in A.2. We can see that generally IL updates are on average smaller than BP-SGD updates and have smaller standard deviations. IL algorithms tended to improve accuracy faster than BP-SGD. This suggests that IL did not improve accuracy faster because the overall changes in parameters were larger. One possibility is that IL improved accuracy faster because it updated parameters along a more direct path toward local minima. This suggestion is consistent with the findings above, that IL updates push FF output layer activities along a more direct path toward global target activities y.

$\|\Delta\theta\|$ (mean±std.) for Supervised Classification Tasks			
Model	MNIST	Fashion-MNIST	CIFAR-10
BP-SGD	.0871(±.0194)	.0680(±.087)	.0807(±.0284)
BP-Adam	.0109(±.0082)	.0060(±.0019)	.0090(±.0014)
IL-SGD	.0074(±.0165)	.0092(±.0124)	.0036(±.0018)
IL-prox	.0059(±.0132)	.0361(±.012)	.0149(±.004)
IL-prox Fast	.0075(±.0167)	.1215(±.1102)	.0203(±.0161)

Table A.5: Average weight update norms for classification tasks.

Appendix B

Chapter 5 Supplementals

B.0.1 Derivation of Output Layer Update

The prediction error at the output layer is and $e_L = \hat{h}_L - p_L$. Let $\mathcal{L}(y, \hat{h}_L) = \frac{1}{2}\|y - \hat{h}_L\|^2$, which implies $-\frac{\partial \mathcal{L}(y,\hat{h}_L)}{\partial \hat{h}_L} = y - \hat{h}$, where y is the output target. To get the output layer update, we take the gradient of $-\frac{\partial F}{\partial \hat{h}_L}$ set to zero and solve for $\hat{h}_L$:

$$\begin{aligned} -\frac{\partial F}{\partial \hat{h}_L} = 0 &= -\gamma_L e_L - \frac{\partial \mathcal{L}(y, \hat{h}_L)}{\partial \hat{h}_L} \\ \hat{h}_L - y &= -\gamma_L(\hat{h}_L - p_L) \\ (1+\gamma_L)\hat{h}_L &= y + \gamma_L p_L \\ \hat{h}_L &= \frac{1}{(1+\gamma_L)} y + \frac{\gamma_L}{(1+\gamma_L)} p_L \end{aligned} \tag{B.1}$$

B.0.2 Theorem 1 Proof, Lemmas, and Preliminaries

Preliminaries

The proof of theorem 5.1 rests on one key insight, which is that the parameters, θ, after an IL update is performed at iteration b can be defined as a function of neuron activities, i.e., $\theta = j(\hat{h})$, since the IL update is a function of neuron activities. In particular, we can define $\Delta\theta^{(b)} = g(\hat{h})$, where $\Delta\theta^{(b)}$ and $g(\hat{h})$ are the set of IL updates (equation 5.4) at iteration b:

$$\Delta\theta^{(b)} = g(\hat{h}) = [\Delta W_0, \Delta W_1, ..., \Delta W_{L-1}] = [\alpha_0 e_1 f(\hat{h}_0)^\top, \alpha_1 e_2 f(\hat{h}_1)^\top, ..., \alpha_{L-1} e_L f(\hat{h}_{L-1})^\top], \tag{B.2}$$

where errors are just another function of $\hat{h}$: $e_l = \hat{h}_l - W_{l-1} f(\hat{h}_{l-1})$. Our updated parameters can then be defined as $\theta = j(\hat{h}) = \theta^{(b)} + g(\hat{h})$. Note that the range of function $j(\hat{h})$ covers all real valued vectors since $\hat{h}_l$ values may in principle be any real-valued vector, and therefore $g(\hat{h})$ and $j(\hat{h})$ output any real valued vector (see [8] theorem 4.2).

This insight allows us to express the proximal update as an optimization procedure over hidden/auxiliary variable $\hat{h}$, instead of over parameters θ. In particular, let $Prox(\theta) = \mathcal{L}(\theta) + \frac{1}{2\beta}\|\theta - \theta^{(b)}\|^2 = \mathcal{L}(\theta) + \frac{1}{2\beta}\|\Delta\theta\|^2$. We now have

$$\begin{aligned} Prox(\theta) = Prox(j(\hat{h}), \hat{h}) &= \mathcal{L}(j(\hat{h})) + \frac{1}{2\beta}\|j(\hat{h}) - \theta^{(b)}\|^2 \\ &= \mathcal{L}(j(\hat{h})) + \frac{1}{2\beta}\|g(\hat{h})\|^2. \end{aligned} \tag{B.3}$$

The proximal update now becomes

$$\theta^{(b+1)}_{prox} = \arg\min_{j(\hat{h})} Prox(j(\hat{h}), \hat{h}) = j(\arg\min_{\hat{h}} Prox(j(\hat{h}), \hat{h})), \tag{B.4}$$

where the equality on the right follows from the fact $j(\hat{h})$ is a function optimized through changes to $\hat{h}$. The IL update can be rewritten using the same notation:

$$\theta_{IL}^{(b+1)} = j(\arg\min_{\hat{h}} F(\theta^{(b)}, \hat{h})). \tag{B.5}$$

The proof for theorem 1 now follows from two lemmas:

Lemma B.1. *Under the assumptions of theorem 5.1, it is the case* $\frac{\partial Prox((j(\hat{h}),\hat{h}))}{\partial \hat{h}} \propto \frac{\partial F(\theta^{(b)},\hat{h})}{\partial \hat{h}}$.

The proof is shown in subsections B.0.2 and B.0.2.

Lemma B.2. *If* $\frac{\partial Prox((j(\hat{h}),\hat{h}))}{\partial \hat{h}} \propto \frac{\partial F(\theta^{(b)},\hat{h})}{\partial \hat{h}}$, *then* $\arg\min_{\hat{h}} Prox(j(\hat{h}), \hat{h}) = \arg\min_{\hat{h}} F(\theta^{(b)}, \hat{h})$

Proof. If the gradients of $Prox$ and F w.r.t. $\hat{h}$ are proportional they will be zero for the same values of $\hat{h}$ and therefore share the same stationary points, including minima. □

Proof of Theorem 1

The proof for Theorem 1 is given as follows:

Proof. Lemma 1 and 2 imply $\arg\min_{\hat{h}} Prox(j(\hat{h}), \hat{h}) = \arg\min_{\hat{h}} F(\theta^{(b)}, \hat{h})$. It follows trivially from this that $j(\arg\min_{\hat{h}} Prox(j(\hat{h}), \hat{h})) = j(\arg\min_{\hat{h}} F(\theta^{(b)}, \hat{h}))$, and therefore $\theta_{prox}^{(b+1)} = \theta_{IL}^{(b+1)}$. □

Proof Lemma 1 for Hidden Layer Updates

Proof. As with the minimization of F in PC networks, we consider the partial gradient of the terms of proximal objective using the terms that explicitly include $\hat{h}_l$. Since the loss term $\mathcal{L}(\hat{h})$ in $Prox(\theta)$ is only an explicit function of the output layer activities $\hat{h}_L$, we ignore

the gradient of the loss with respect to hidden layers. This means that hidden layers only minimize the regularization term $\frac{1}{2\beta}\|\Delta\theta\|^2$. Given the notation from the previous section, we can express the minimization of the proximal regularization term as follows

$$\arg\min_{\theta} \frac{1}{2\beta}\|\Delta\theta\|^2 = \arg\min_{j(\hat{h})} \frac{1}{2\beta}\|g(\hat{h})\|^2 = \arg\min_{j(\hat{h})} \sum_{l=0}^{L-1} \frac{1}{2\beta}\|\Delta W_l^{(b)}\|^2, \tag{B.6}$$

where β is a 'global' learning rate, not to be confused with matrix-wise step sizes α_l. Let the function $Prox(\hat{h}_l)$ be the components of equation B.6 that explicitly involve $\hat{h}_l$, which is

$$Prox(\hat{h}_l) = \underbrace{\frac{1}{2\beta}\|\Delta W_{l-1}\|^2}_{prox_1} + \underbrace{\frac{1}{2\beta}\|\Delta W_l\|^2}_{prox_2}. \tag{B.7}$$

Now we compute the gradient of $Prox(\hat{h}_l)$ w.r.t. $\hat{h}_l$ and show it is proportional to the gradient of F w.r.t. $\hat{h}_l$. The gradient can be expressed in terms of $prox_1$ and $prox_2$: $\frac{\partial Prox(\hat{h}_l)}{\partial \hat{h}_l} = \frac{\partial prox_1(\hat{h}_l)}{\partial \hat{h}_l} + \frac{\partial prox_2(\hat{h}_l)}{\partial \hat{h}_l}$. We first compute the gradient of $prox_1$.

$$prox_1 = \frac{1}{2\beta}\|\Delta W_{l-1}\|^2 = \frac{1}{2\beta}\|\alpha_{l-1} e_l f(\hat{h}_{l-1})^T\|^2, \tag{B.8}$$

where $p_l = W_{l-1} f(\hat{h}_{l-1})$ and $e_l = \hat{h}_l - p_l$. Using the chain rule

$$\frac{\partial prox_1(\hat{h}_l)}{\partial \hat{h}_l} = \frac{\partial prox_1}{\partial \Delta W_{l-1}} \frac{\partial \Delta W_{l-1}}{\partial e_l} \frac{\partial e_l}{\partial \hat{h}_l}, \tag{B.9}$$

where $\frac{\partial prox_1}{\partial \Delta W_{l-1}} = \frac{\alpha_{l-1}}{\beta} e_l f(\hat{h}_{l-1})^T$ and $\frac{\partial \Delta W_{l-1}}{\partial e_l}\frac{\partial e_l}{\partial \hat{h}_l} = \alpha_{l-1} f(\hat{h}_{l-1})$. Multiplying together we get

$$\frac{\partial prox_1(\hat{h}_l)}{\partial \hat{h}_l} = \frac{\alpha_{l-1}^2}{\beta}\|f(\hat{h}_{l-1})\|^2 e_l. \tag{B.10}$$

Now we derive $\frac{prox_2(\hat{h}_l)}{\hat{h}_l}$:

$$prox_2 = \frac{1}{2\beta}\|\Delta W_l\|^2 = \frac{1}{2\beta}\|\alpha_l e_{l+1} f(\hat{h}_l)^T\|^2, \tag{B.11}$$

where $p_{l+1} = W_l f(\hat{h}_l)$ and $e_{l+1} = \hat{h}_{l+1} - p_{l+1}$. Using the chain rule

$$\frac{\partial prox_2}{\partial \hat{h}_l} = \frac{\partial prox_2}{\partial \Delta W_l}\frac{\partial \Delta W_l}{\partial e_{l+1}}\frac{\partial e_{l+1}}{\partial p_{l+1}}\frac{\partial p_l}{\partial \hat{h}_l} + \frac{\partial prox_2}{\partial \Delta W_l}\frac{\partial \Delta W_l}{\partial f(\hat{h}_l)^T}\frac{\partial f(\hat{h}_l)^T}{\partial \hat{h}_l}. \tag{B.12}$$

The term on the left is composed of the following gradients: $\frac{\partial prox_2}{\partial \Delta W_l} = \frac{\alpha_l}{\beta} e_{l+1} f(\hat{h})_l^T$, $\frac{\partial \Delta W_l}{\partial e_{l+1}} = \alpha_l f(\hat{h}_l)$, and $\frac{\partial e_{l+1}}{\partial p_{l+1}}\frac{\partial p_l}{\partial \hat{h}_l} = -f'(\hat{h}_l)W_l^T$. The term on the right comes out to $\frac{\partial prox_2}{\partial \Delta W_l}\frac{\partial \Delta W_l}{\partial f(\hat{h})_l^T}\frac{\partial f(\hat{h}_l)^T}{\partial f(\hat{h})_l} = \alpha_{l-1}(e_{l+1}^T e_{l+1})f'(\hat{h}_l)\hat{h}_l$, where $f'(\hat{h}_l)$ is gradient of f. Substituting these terms back into the original equation, we get

$$\begin{aligned}\frac{\partial prox_2}{\partial \hat{h}_l} &= -\frac{\alpha_l^2}{\beta} f'(\hat{h}_l)W_l^T e_{l+1} f(\hat{h}_l)^T f(\hat{h}_l) + \frac{\alpha_l^2}{\beta}(e_{l+1}^T e_{l+1})f'(\hat{h}_l)\hat{h}_l \\ &= -\frac{\alpha_l^2}{\beta}\|f(\hat{h}_l)\|^2 f'(\hat{h}_l)W_l^T e_{l+1} + \frac{\alpha_l^2}{\beta}\|e_{l+1}\|^2 f'(\hat{h}_l)\hat{h}_l.\end{aligned} \tag{B.13}$$

Now we combine the gradients for $prox1$ and $prox2$ and use the assumptions of the theorem,

which are $\gamma_{l+1} = \alpha_l^2 \|f(\hat{h}_l)\|^2$, $\gamma_l = \alpha_{l-1}^2 \|f(\hat{h}_{l-1})\|^2$, and $\gamma_l^{decay} = \alpha_l^2 \|e_{l+1}\|^2$:

$$\begin{aligned}
\frac{\partial Prox(\hat{h}_l)}{\partial \hat{h}_l} &= -\frac{\alpha_l^2}{\beta}\|f(\hat{h}_l)\|^2 f'(\hat{h}_l) W_l^T e_{l+1} + \frac{\alpha_{l-1}^2}{\beta}\|f(\hat{h}_{l-1})\|^2 e_l + \frac{\alpha_l^2}{\beta}\|e_{l+1}\|^2 f'(\hat{h}_l)\hat{h}_l \\
&\propto -\alpha_l^2 \|f(\hat{h}_l)\|^2 f'(\hat{h}_l) W_l^T e_{l+1} + \alpha_{l-1}^2 \|f(\hat{h}_{l-1})\|^2 e_l + \alpha_l^2 \|e_{l+1}\|^2 f'(\hat{h}_l)\hat{h}_l \\
&= -\gamma_{l+1} f'(\hat{h}_l) W_l^T e_{l+1} + \gamma_l e_l + \gamma_l^{decay} f'(\hat{h}_l)\hat{h}_l \\
&= \frac{\partial F}{\partial \hat{h}_l}
\end{aligned} \tag{B.14}$$

□

Proof of Lemma 1 for Output Layer Updates

Proof. Let $\hat{h}_L^{IL}$ be the output layer value updated according to equation 5.3 and $\hat{h}_L^{prox}$ be the output layer activity updated according to the gradients of the proximal objective. Here we show $\hat{h}_L^{IL} = \hat{h}_L^{prox}$ in the limit where $\hat{h}_{L-1} \to h_{L-1}^{(b+1)}$ and $\gamma_L = \alpha_{L-1}(\frac{1}{\beta} + \|f(\hat{h}_{L-1})\|^2) - 1$.

The terms in $Prox(\theta)$ that explicitly contain $\hat{h}_L$ are $\mathcal{L}(j(\hat{h}, \hat{h}))$ and $\frac{1}{2\beta}\|\Delta W_{L-1}\|^2$. We know from the previous section that

$$\frac{\partial \frac{1}{2\beta}\|\Delta W_{L-1}\|^2}{\partial \hat{h}_L} = \frac{\alpha_{L-1}^2}{\beta}\|f(\hat{h}_{L-1})\|^2 e_L = \frac{c\alpha_{L-1}}{\beta} e_L, \tag{B.15}$$

where $c = \alpha_{L-1}\|f(\hat{h}_{L-1}^{(b)})\|^2$. The loss $\mathcal{L}(j(\hat{h}), \hat{h}) = \frac{1}{2}\|y^{(b)} - h_L^{(b+1)}\|^2$, where $h_L^{(b+1)}$ are the feed-forward activities after the weight updates are applied. We first derive an expression

for $h_L^{(b+1)}$ in the limit where $\hat{h}_{L-1} \to h_{L-1}^{(b+1)}$ then compute the gradient of the loss.

$$\begin{aligned}
\lim_{\hat{h}_{L-1} \to h_{L-1}^{(b+1)}} h_L^{(b+1)} = W_{L-1}^{(b+1)} f(\hat{h}_{L-1}^{(b)}) &= (W_{L-1}^{(b)} + \Delta W_{L-1}^{(b)}) f(\hat{h}_{L-1}^{(b)}) \\
&= W_{L-1}^{(b)} f(\hat{h}_{L-1}^{(b)}) + \alpha_{L-1} f(\hat{h}_{L-1}^{(b)})^\top f(\hat{h}_{L-1}^{(b)}) e_L^{(b)} \\
&= W_{L-1}^{(b)} f(\hat{h}_{L-1}^{(b)}) + c\hat{h}_L^{(b)} - cW_{L-1}^{(b)} f(\hat{h}_{L-1}^{(b)}) \\
&= (1-c) W_{L-1}^{(b)} f(\hat{h}_{L-1}^{(b)}) + c\hat{h}_L^{(b)} \\
&= (1-c) p_L^{(b)} + c\hat{h}_L^{(b)},
\end{aligned} \tag{B.16}$$

where $c = \alpha_{L-1} \|f(\hat{h}_{L-1}^{(b)})\|^2$. The second line follows from the assumption $\hat{h}_{L-1} \to h_{L-1}^{(b+1)}$. Now we can compute the gradient of the loss:

$$\begin{aligned}
\frac{\partial \mathcal{L}(y^{(b)}, h_L^{(b+1)})}{\partial \hat{h}_L^{(b)}} &= \frac{\partial \mathcal{L}(y^{(b)}, h_L^{(b+1)})}{\partial h^{(b+1)}} \frac{\partial h_L^{(b+1)}}{\partial \hat{h}_L^{(b)}} \\
&= c \frac{\partial \mathcal{L}(y^{(b)}, h_L^{(b+1)})}{\partial h^{(b+1)}} \\
&= c(((1-c) p_L^{(b)} + c\hat{h}_L^{(b)}) - y)
\end{aligned} \tag{B.17}$$

Under the assumption $\gamma_L = \frac{\alpha_L}{\beta} + c - 1 = \alpha_{L-1}(\frac{1}{\beta} + \|f(\hat{h}_{L-1})\|^2) - 1$ we have

$$\begin{aligned}
\frac{\partial Prox(\hat{h}_L)}{\partial \hat{h}_L} &= \frac{c\alpha_{L-1}}{\beta} e_L + c(((1-c) p_L^{(b)} + c\hat{h}_L^{(b)}) - y) \\
&\propto \frac{\alpha_{L-1}}{\beta} e_L + (1-c) p_L^{(b)} + c\hat{h}_L^{(b)} - y \\
&= \frac{\alpha_{L-1}}{\beta} \hat{h}_L - \frac{\alpha_{L-1}}{\beta} p_L + (1-c) p_L^{(b)} + c\hat{h}_L^{(b)} - y \\
&= (\frac{\alpha_{L-1}}{\beta} - c)\hat{h}_L - (\frac{\alpha_{L-1}}{\beta} + c - 1) p_L^{(b)} - y \\
&= (\frac{\alpha_{L-1}}{\beta} - c - 1)(\hat{h}_L - p_L^{(b)}) + \hat{h}_L - y \\
&= \gamma_L e_L + \frac{\partial \mathcal{L}(y^{(b)}, \hat{h}_L)}{\partial \hat{h}_L} \\
&= \frac{\partial F}{\partial \hat{h}_L}.
\end{aligned} \tag{B.18}$$

□

B.0.3 Extending to Mini-batches

Theorem 5.1 considers the case where mini-batch size 1 is used. We chose this condition because 1) it simplifies the mathematics of the proof, 2) it more realistically describes the learning scenario the brain faces (see appendix of [8]), and 3) because the extension of theorem 1 to larger mini-batches is straight-forward. In particular, let $\Delta\theta_{prox}^{(b)} = \theta_{prox}^{(b+1),n} - \theta^{(b)}$, be the proximal/implicit gradient update for the n-th data point in the mini-batch at iteration b. As long as the same γ setting and assumptions about the learning rate in theorem 1 hold for each data point in the mini-batch, it follows trivially that

$$\frac{1}{N}\sum_{n=0}^{N}\Delta\theta_{IL}^{(b),n} = \frac{1}{N}\sum_{n=0}^{N}\Delta\theta_{prox}^{(b),n}. \tag{B.19}$$

As long as the assumption of theorem 1 holds for each individual data point in the mini-batch, then mini-batching with IL is equivalent to computing the proximal update/implicit gradient for each individual data point in the mini-batch independently and then averaging over the updates/gradients. Thus, IL is equivalent to a kind of stochastic version of the proximal algorithm, where the proximal minimization problem is solved over individual data-points (then the solutions are averaged) rather than over entire (mini-)batches. This is why we claim IL approximates *stochastic* implicit gradient descent [127, 128], rather than the batch/non-stochastic version.

Theorem 5.2 extends to the mini-batch case in a similar way. In particular, for small β values, IL would approximate the process of computing the regularized Newton-Raphson update for each individual data point in the mini-batch and averaging those updates together.

B.0.4 Proof of Theorem 2

Proof. To prove this theorem we perform a Taylor expansion of the gradient around point $\theta^{(b)}$ to get an approximation of the implicit gradient. In particular, we approximate the function $f(\theta^{(b+1)}) = f(\theta^{(b)} + \Delta\theta_{imp}) = -\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}}$, where $\Delta\theta_{imp}$ is the change to parameters after an implicit SGD/proximal update, which is just $-\beta\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}}$. Let the current training iteration be iteration b and $H = \frac{\partial^2 \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)2}}$. We have

$$
\begin{aligned}
f(\theta^{(b)} + \Delta\theta_{imp}) &= f(\theta^{(b)}) + \frac{\partial f(\theta^{(b)})}{\partial \theta^{(b)}}\Delta\theta_{imp} + \mathcal{O}(\beta^2) \\
-\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} &= -\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}} - H\beta\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} + \mathcal{O}(\beta^2) \\
-\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} + \beta H\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} &= -\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}} + \mathcal{O}(\beta^2) \\
-(I + \beta H)\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} &= -\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}} + \mathcal{O}(\beta^2) \\
-\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} &= -(I + \beta H)^{-1}\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}} + \mathcal{O}(\beta^2)
\end{aligned}
\tag{B.20}
$$

From this and the assumption $\Delta\theta_{IL}^{(b)} = -\beta\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}}$, it follows that $\Delta\theta_{IL}^{(b)} = -\beta\frac{\partial \mathcal{L}(\theta^{(b+1)})}{\partial \theta^{(b+1)}} \approx -(I + \beta H)^{-1}\beta\frac{\partial \mathcal{L}(\theta^{(b)})}{\partial \theta^{(b)}}$, with error $\mathcal{O}(\beta^2)$. □

B.0.5 Methods

Below we outline details of each experiment with the intent of making the experiments replicable. Code for all experiments can be found at `https://github.com/nalonso2/PredictiveCoding-MQSeqIL/tree/main`. All simulations were run on the same laptop with a small GPU type NVIDIA GeForce RTX 270 with MAXQ design.

Details on Implementation of IL High level summaries of sequential and standard IL are given in algorithms 3 and 4 and in figure B.1. We provide a more detailed description of the implementation of IL and SeqIL in algorithms 5 and 6. In the simulations and figure,

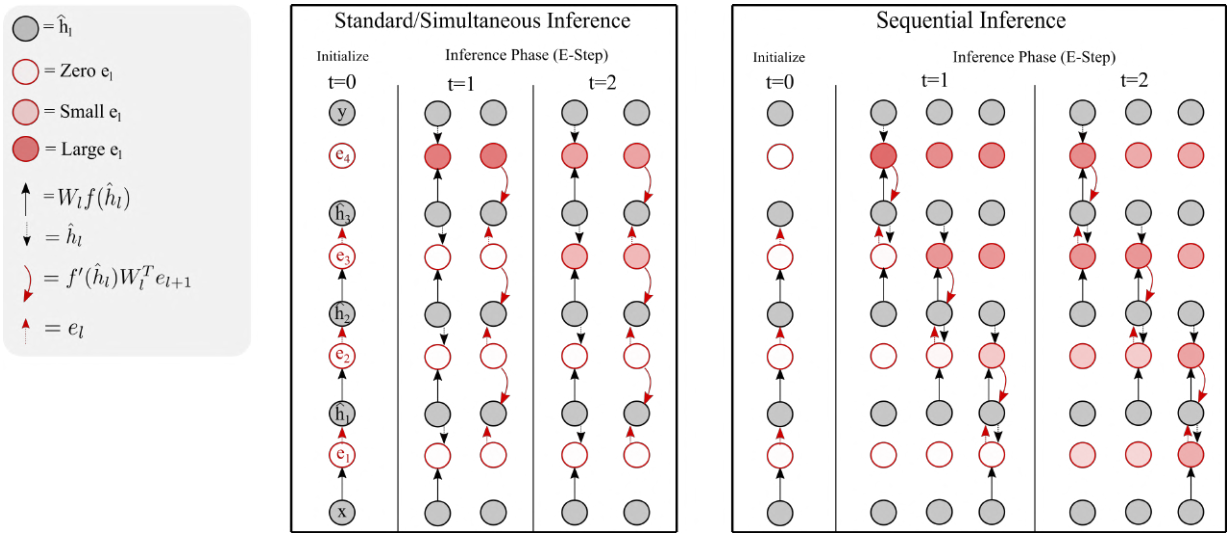

Figure B.1: Description of the standard/simultaneous inference versus sequential inference method. Initialization (t=0) and the first two inference steps (t=1 and t=2) are shown. Gray nodes are activities $\hat{h}_l$. Red nodes are errors e_l. At initialization input layer is clamped to x, and we assume the fully clamped scenario where the output layer, top nodes, are clamped to the target: $\hat{h}_L = y$. Hidden layers are initialized to their feed forward activities: $\hat{h}_l = h_l$. Each inference iteration, simultaneous inference begins by computing and storing errors at all layers (left column under t=1 and t=2), then uses the stored errors to update activities (right column under t=1 and t=2). We can see that this leads to a delay in the way errors are propagated through the network. Sequential inference begins by computing errors e_4 and e_3, then uses those to update $\hat{h}_3$ (left column under t=1 and t=2). Then it recomputes error e_3, which will now be non-zero, and e_2, then uses those to update $\hat{h}_2$. This process repeats until all activities are updated. We can see sequential inference propagates errors through the entire network in a single iteration.

we fully clamp the output layer and set $\gamma_l = 1$ and $\gamma_l^{decay} = 0$, which is common in practice.

Algorithm 3: Simultaneous Inference

```
begin
    // Initialize:  ĥ = h
    for t = 1 to T do
        // Compute and store
           errors at layers 1 to
           L
        // Update ĥ at layers 1
           to L using stored
           errors
    end
end
```

Algorithm 4: Sequential Inference

```
begin
    // Initialize:  ĥ = h
    for t = 1 to T do
        for reversed(l = 1 to L) do
            // Compute e_l and e_{l+1}
            // Update ĥ_l
        end
    end
end
```

Details on MQ Optimizer Implementation In our tests of the MQ optimizer, we use hyperparameter settings $\alpha_{min} = .001$, $r = .000001$, and $\rho = .9999$ for fully connected and $\rho = .999$ for convolutional networks. These hyper-parameter settings perform well generally across models, and using the same hyper-parameter settings across simulations significantly reduced hyper-parameter search. We only needed to then perform grid searches to find learning rates, α_l. We find initializing $v_l = \alpha_l$ works well. The variable v_l, which controls the adaptive learning rate in MQ is a slow moving average of the weight update magnitude:

$$v_l^{(b+1)} = (1-\rho)v_l^{(b)} - \rho\frac{1}{ij}|\frac{\partial F}{\partial W_l^{(b)}}|, \tag{B.21}$$

where $|\frac{\partial F}{\partial W_l^{(b)}}|$ is the L-1 norm and i and j are the row and column sizes respectively. This moving average will be biased early in training toward the initial value of v_l, making v_l and MQ highly sensitive to initialization. To prevent this bias we use a modified ρ^* to more

Algorithm 5: IL ($\gamma_l = 1$,$\gamma_l^{decay} = 0$, σ =softmax or sigmoid)

begin
  // Initialize
  $\hat{h}_0 \leftarrow x^{(b)}$
  for $l = 0$ **to** $L - 1$ **do**
    $\hat{h}_{l+1} \leftarrow W_l f(\hat{h}_l)$
  end
  $\hat{h}_L \leftarrow y^{(b)}$
  // Inference Phase
  for $t = 0$ **to** T **do**
    $e_L \leftarrow \hat{h}_L - \sigma(W_{L-1} f(\hat{h}_{L-1}))$
    for $l = 1$ **to** $L - 1$ **do**
      $e_l \leftarrow \hat{h}_l - W_{l-1} f(\hat{h}_{l-1})$
    end
    for $l = 1$ **to** L **do**
      $\hat{h}_l \leftarrow \hat{h}_l - \epsilon(e_l - f'(\hat{h}_l) W_l^T e_{l+1})$
    end
  end
  // Update Weight Matrices
  Equation 5.4
end

quickly update v_l early in training:

$$\rho^* = \min(\rho, 1/(b+2)), \tag{B.22}$$

where b is the training iteration. v_l is then updated using ρ^* instead of ρ. After a finite number of iterations $\rho^* = \rho$, but early in training $\rho^* > \rho$ allowing for faster updating of v_l. We find initializing $v_l = \alpha_l$ works well.

Comparison of Standard and Sequential Inference (Figure 5.3) We measure the mean of the squared errors at each layer of a MLP with three hidden layers (3072-3x1024-10). ReLUs are used at hidden layers and softmax at output layer. The same step size of .02 is used to update layer activities in both models. Weights are randomly initialized so training cannot skew the results. CIFAR-10 dataset is used. Results are averaged over 10

Algorithm 6: SeqIL ($\gamma_l = 1$,$\gamma_l^{decay} = 0$, σ =softmax or sigmoid))

begin
  // Initialize
  $\hat{h}_0 \leftarrow x^{(b)}$
  for $l = 0$ **to** $L - 1$ **do**
    $\hat{h}_{l+1} \leftarrow W_l f(\hat{h}_l)$
  end
  $\hat{h}_L \leftarrow y^{(b)}$
  // Inference Phase
  for $t = 0$ **to** T **do**
    $p \leftarrow \sigma(W_{L-1} f(\hat{h}_{L-1}))$
    for *reversed*($l = 1$ **to** L) **do**
      $e_{l+1} \leftarrow \hat{h}_{l+1} - p$
      $p = W_{l-1} f(\hat{h}_{l-1})$
      $e_l \leftarrow \hat{h}_l - p$
      $\hat{h}_l \leftarrow \hat{h}_l - \epsilon(e_l - f'(\hat{h}_l) W_l^T e_{l+1})$
    end
  end
  // Update Weight Matrices
  Equation 5.4
end

seeds. For each seed we test on 10000 images. MSE is computed each inference iteration and averaged across images, then these averaged runs are averaged across seeds.

Weight Update Analysis (Figure 5.2 and B.3) We train MLPs with three hidden layers, dimensions 3072-3x1024-10, on CIFAR-10. Five seeds are trained for each kind of algorithm. We train using standard IL (IL), standard IL with MQ (IL-MQ), and standard IL with Adam (IL-Adam). ReLU activations are used at hidden layers and softmax at the output layer. We train the network for 10000 iteration. After 1000 iterations we begin measuring the average update magnitude (we have this delay to allow the adaptive learning rates in Adam and MQ optimizers to 'warm up'). The average update magnitude is computed $\frac{1}{IJ} \sum_i \sum_j |\Delta W_{l,ij}|$, where i and j are the row and column indexes and I and J are the size of row and columns, respectively, of weight matrix l. The update magnitudes for each matrix are averaged over training iterations and seeds to obtain the final averages shown in figure 5.2. This same

procedure is used to compute the weight updates for figure B.3.

Proximal Objective Tests (Figure 5.1 and B.2) Fully connected networks with 3 hidden layers and ReLU activations are first trained for 1000 iterations on CIFAR-10 then the proximal objective value is measured during inference over 10000 data points and averaged. These averaged runs are then averaged across 5 seeds to get the final values.

For the tests in figure 5.1 and B.2, we test the IL algorithms under the typical settings of γ (rather than those in theorem 1), where $\gamma_n = 1$ for all n, and $\gamma_n^{decay} = 0$ for all n. We then measure the proximal objective during inference under the β values $[.01, .1, 1, 10, 100]$. How the proximal objective changes during inference under each of these β values is shown in figure B.2. How it changes under the values $\beta = 1$ is shown in figure 5.1 (with the exception of IL-MQ which is shown with $\beta = 100$, since this is the value it best minizes).

The proximal objective is measured each inference iteration, t, as follows: model parameters are updated each inference iteration using the IL learning rule and associated learning rate. The loss is then computed after the update (the $\mathcal{L}$ in the proximal objective) and the squared l-2 norm of the parameter update (the regularization term in the proximal objective) is computed. Parameter updates are then erased, another inference step is performed and the proximal objective value is recomputed.

Classification with IL versus SeqIL (Figure 5.4) We trained a four layer MLP, dimensions 3072-4x1024-10 with standard IL and sequential IL using Adam optimizers, since Adam has becme the standard optimizer used in previous works. Models are trained on CIFAR-10, with ReLU at hidden layers, softmax output layer. Grid searches were used to find the learning rate and step size for activity updates, ϵ, for each value of T. We searched over learning rates $[.0001, .00005, .00001]$ and step sizes for the activity updates $[.02, .05, .1, .3]$ for each value of T. After the grid search we trained 5 seeds at each value of T and show the mean and standard deviation over the best accuracies achieved by each seed.

Classification and Autoencoder Training (Figures 5.4-5.6) For classification tasks on SVHN, CIFAR-10, and Tiny Imagenet, we train fully connected MLPs dimension 3072-3x1024-10 on SVHN and CIFAR-10, and convolutional networks on SVHN, CIFAR-10, and Tiny Imagenet, with ReLU activitation at hidden layers and softmax at output layers. Due to limited compute, we used small convolutional networks, similar to previous work on biologically inspired algorithms [10], where max pools are removed and filters with stride of 2 are used to reduce the number of channels each layer. SVHN and CIFAR-10 use three convolutional layers (5x5-64, 5x5-128, 3x3-256) followed by one fully connected layer. Tiny imagenet used a network with four layers (5x5-32, 5x5-64, 5x5-128, 3x3-256) with one fully connected layer. All of the IL algorithms used a highly truncated inference phases, where T=3. In addition to sequential IL with MQ (SeqIL-MQ), we test SeqIL with no optimizer (SeqIL) and SeqIL with Adam (SeqIL-Adam). We compare to the performance of BP-SGD and BP with Adam (BP-Adam). Learning rates were first hand tuned to find ranges that yield good performance, then grid searches over 5-10 learning rates were performed to find the learning rate that the best test performance at convergence. The same approach for finding learning rates is used for the auto-encoder tasks. Autoencoders are fully connected networks with layers sizes 3072-1024-256-20-256-1024-3072, with ReLU at hidden layers and sigmoid at the output layer.

Real Time Training (Figure B.4) To get the training time in second for models trained on CIFAR-10, we trained a model of each type (FC classifier, convolutional classifier, and FC autoencoder) on CIFAR-10 without testing and without storing any values, so that testing and changes in available RAM from storing training data, do not affect training time. All models were trained on the same laptop with a small GPU type NVIDIA GeForce RTX 270 with MAXQ design. Built in python timer functions were used to get the training time for one training run for each model type. Training times were then plotted against the test losses, which were computed from the simulations described in the previous section.

Determining Computation Requirements (Table B.2) Although matrix multiplies do not account for all of the operations performed, it is the most numerous operation in BP and IL algorithms, so it is a good indicator of computation cost for each algorithm. Each training iteration, BP must compute a feed forward signal, which requires L matrix multiples, where L is the number of matrices in the network. To propagate feedback signals, L-1 matrix multiplies are needed, and to perform weight updates another L matrix multiplies are needed, totaling 3L-1. IL algorithms similarly perform L matrix multiplies to compute the predictions (forward signals) and errors, and another L-1 to update activities, totaling 2L-1 matrix multiplies each inference iteration, totaling T(2L-1) for T inference iterations. Then another L matrix multiplies are needed to update weights, totaling T(2L-1) + L matrix multiplies.

B.0.6 Supplementary Results

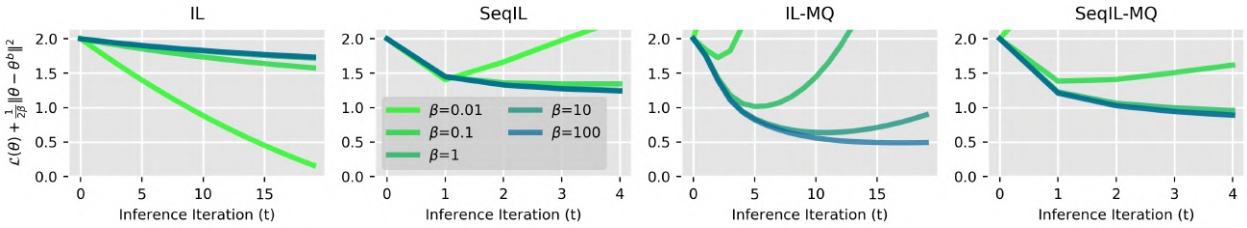

Figure B.2: Measurement of the proximal objective during the inference phase of IL in a fully connected networks trained on CIFAR-10. Algorithms are tested under settings $\gamma_l = 1$ and $\gamma_l^{decay} = 0$ for all l, which is the standard in previous work (e.g., [100, 132, 9, 111, 8]). We test IL with a fixed learning rate (IL), SeqIL with a fixed learning rate (SeqIL), and these algorithms with the MQ optimizer. The proximal objective is measured under different values of the β term. For easier comparison each inference run is shifted so its starting value is 2. For a wide range of β values the proximal objective is reduced during inference. *According to theorem 5.1, performing inference with a fully clamped output layer, as we do here, approximates minimizing the proximal objective under a large β value. All models significantly reduce the proximal objective for the largest β value we tested of 100. Reduction of the proximal objective is less reliable for very small β values. These results are therefore highly consistent with theorem 5.1.*

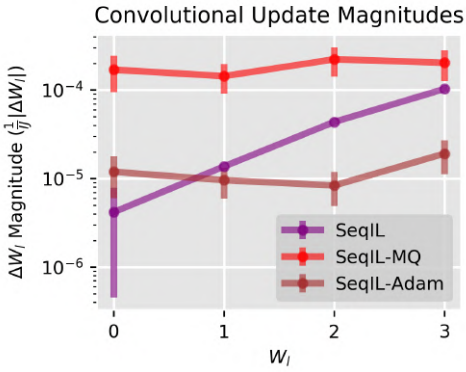

Figure B.3: Average update magnitude of each weight matrix/kernal in convolutional network with three hidden layers trained on CIFAR-10. The same network that was trained above was used: three convolutional layers (5x5-64, 5x5-128, 3x3-256), stride 2, followed by one fully connected layer, ReLUs at hidden layers, softmax output. The learning rates that achieved best test accuracy at convergence are used. In this convolutional network, we find that SeqIL-Adam equalized weight update magnitudes well, but tends to produce very small weight updates overall compared to SeqIL-MQ. This is different than the fully connected case, where the two produce very similar update magnitudes (figure 5.2). Weight update magnitudes can be increased by increasing the learning rate parameter in the Adam optimizer, but we find learning quickly becomes unstable when we do this. The fact that weight updates are so small may help explain why SeqIL-Adam falls into shallow local minima, i.e., because it is unable to push parameters far enough from the initial values to find deep regions of the loss landscape, leading parameters to get caught in shallow local minima near the initial parameter values. SeqIL-MQ, on the other hand, equalizes weight update magnitudes but produces much larger updates on average which likely helps parameters find deeper minima. The reason SeqIL-MQ yields large weight updates likely has to do with the fact that it uses a single scalar learning rate for each matrix, which prevents any adaptive learning rates from becoming too small, whereas Adam seems to end up producing many very small, per-parameter learning rates.

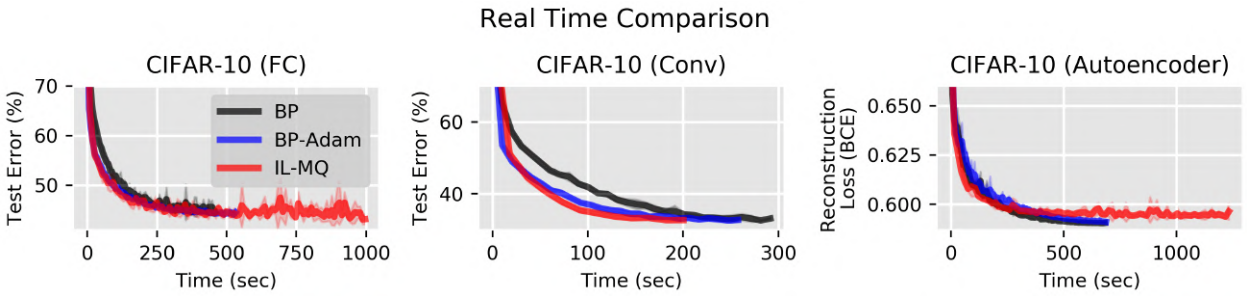

Figure B.4: MLP, convolutional network, and AE training runs shown for Cifar-10 with x-axis showing real time in seconds. Training runs shown to convergence or max number of epochs.

Memory Requirements	
Algorithm	Memory
BP	$\mathcal{O}(M)$
BP-Adam	$\mathcal{O}(3M)$
IL-MQ	$\mathcal{O}(M)$
IL-Adam	$\mathcal{O}(3M)$

Table B.1: Memory computed as model parameters (M) + optimizer/algorithm parameters, in the limit where $M \to \infty$ with finite network layers.

Computation Requirements			
	BP	SeqIL	IL(T=15)
MLP	$3L-1$	$7L-1$	$31L-1$
AutoEnc.	$3L-1$	$13L-1$	$31L-1$

Table B.2: The number of matrix multiplies performed each training iteration. L is the number weight matrices. SeqIL computation is computed given the value T used in our simulations above. We compare this to standard IL that uses a more common value of T (15) (see app. B.0.5 for details).

Computation Requirements	
IL/SeqIL	iPC
$T(2L-1)+L$	$T(2L-1)+LT$

Table B.3: The number of matrix multiplications per training iteration, where L is the number weight matrices and T the number of inference iterations. Each inference iteration, 2L-1 matrix multiplies are needed to update each $\hat{h}_l$. With T inference iterations, this means T(2L-1) matrix multiplies are needed to perform inference. Since IL and SeqIL only update weights once per training iteration, another L matrix multiples are needed to compute weight updates for each weight matrix, resulting in a total of T(2L-1) + L matrix multiplies. Incremental PC (iPC) [112] updates all weights each inference iteration requiring TL matrix multiplies, for a total of T(2L-1) + TL. Note, however, that [112] et al., point out that iPC typically requires fewer matrix multiplies to perform on non-zero update on each matrix, since weights are updated each iteration. This may be a useful property for scenarios where, e.g. the input is presented in close to real time and weight must be updated rapidly.

Appendix C

Chapter 6 Supplementals

C.1 Methods

C.1.1 Code

Code will be publicly released upon publication of the paper. Code was written in Python 3.7.6 using Pytorch version 1.10.0 to implement models and perform differentiation for models that use backpropagation. All simulations were run on a small GPU type NVIDIA GeForce RTX 270 with MAXQ design.

C.1.2 Datasets and Hyperparameters

Image values ranged between 0 and 1, unless otherwise noted. Images are converted to pytorch tensors, but no normalization or other alterations where made unless otherwise specified. MNIST, Fashion-MNIST, and E-MNIST are image data sets with images sized 1x28x28. SVHN, CIFAR-10, and CIFAR-100 are natural image data sets with images sized

3x32x32. Tiny ImageNet is a natural image data set with images sized 3x64x64. Finally, CalTech256 is a natural image data set with images of various sizes. We cropped all CalTech256 images to 3x128x128. For all models, we use a grid searches to find hyper-parameters.

Unless otherwise specified, we use a recall threshold, γ, of .01. Prior works (e.g., [111, 139]) use smaller thresholds of .005 or .001. We use a slightly larger threshold here, since we find one of our main comparison models, MHN, is unable to recall any images while training with BP, and we wanted to show this was not simply a result of an arbitrarily small recall threshold.

C.1.3 SQHN Implementation Details

All SQHN architectures are set up to have a tree structure. One can think of the structure as being similar to locally connected networks or convolutional networks without weight sharing. Thus, we can talk about SQHNs as having a certain number of channels at each hidden layer (equivalent to the number of neurons at each node) and the receptive field size of each channel/node. To simplify the architectures, we design SQHNs so that nodes have non-overlapping receptive fields, which means each node has one parent. More complex versions of SQHNs can be used for more complex vision tasks, but we found these simpler architectures performed very well and eased scaling in associative memory tasks. Here we explain how inference is implemented. Inference involves a single feed-forward and feedback sweep through the network. The goal of inference is to maximize energy, i.e. the sum of conditional probabilities of input and hidden node values (equation 6.1).

Let $h_{l,j}$ be the internal state value of the jth neuron in the lth node. Let the lth node be in the first hidden layer, which has one child node. Let M_l be the matrix from node l to its child node, which is visible and clamped in image patch x_{c_l}. The matrix M_l can be decomposed in to columns or 'memory vectors': $M_l = [m_{l,0}, m_{l,1}, ..., m_{l,J}]$. During the feed forward pass

$h_{l,j}$ receives the signal

$$h_{l,j} = \frac{.5(m_{l,j}^\top - .5)(x_{c_l} - .5)}{\|(m_{l,j}^\top - .5)\|\|(x_{c_l} - .5)\|} + .5. \tag{C.1}$$

This operation is a kind of shifted cosine similarity operation between in input and each memory vector $m_{l,j}$. We show in supplementals this operation can be interpreted as a weighted, normalized average of the probability of each pixel value, under the assumption each pixels value is a binary variable (supp. C.2.3). In matrix form this operation only involves a vector matrix multiply, like a neural network, with an extra shift operation and normalization operation:

$$h_l = \frac{1}{2Z}(M_l^\top - .5)(x_{c_l} - .5) + .5, \tag{C.2}$$

where $\frac{1}{Z}$ is the neuron-wise normalization. In practice, one could also store a separate feedback matrix with the shifted, transposed, and normalized version of matrix M_l.

Hidden nodes at the 2nd layer and higher have children nodes that represent discrete/categorical variables. Computing the sum of probabilities of each child therefore requires a different operation. Let h_{c_n} be the vector of internal neuron values of the nth child node of node l, and let $h_{c_n}^{max} = max(h_{c_n})$, where max is an activation function that return a vector of all zeros except for the maximum value, e.g., $max([.1, .8, .4]) = [0, .8, 0]$. Let h_c^{max} be the concatenation of all N children node max values: $h_c^{max} = [h_{c_0}^{max,\top}, h_{c_1}^{max,\top}, ...h_{c_N}^{max,\top}]^\top$, and let M_l be the concatenation of the matrices lead from node l to its children: $M_l = [M_{l,0}^\top, M_{l,1}^\top, ...M_{l,N}^\top]^\top$. The update for a hidden node is

$$h_l = \frac{1}{Z} M_l^\top h_c^{max}, \tag{C.3}$$

where $Z = N\|h_c^{max}\|$. We show in the supplemental (section C.2.3) this is equivalent to taking the weighted average of the conditional probability of child node assignments, where

each conditional probability is weighted by the weighted average of the probability of its children values, which are a weighted probability of their children values, and so on. Using this weighting thus conveys information about the probabilities of all descendents.

After signals propagate to the root/memory node, signals are propagated down the tree, and one-hot values assigned to each node. Let $argmax$ be the activation function that assigns a one-hot to the maximum value, e.g., $argmax([.1, .8, .4]) = [0, 1, 0]$. Let h_l^* be the one-hot assignment for the lth node. At the memory node the assignment is simply

$$h_L^* = argmax(h_L). \tag{C.4}$$

Signals are then propagate down according to

$$h_l^* = argmax(\lambda h_l + (1 - \lambda) M_{pa_l,l} h_{pa_l}^*), \tag{C.5}$$

where pa_l is the parent node of l and λ a hyper-parameter. For all recall tasks we set $\lambda = .5$.

C.1.4 Associative Recall Comparison

For our initial comparison to SoTA associative memory models we tested our SQHN models on the same associative memory task as [139]. Each SQHN model had the same number of neurons at its nodes as there are images. SQHN L2 had kernal sizes 4x4 and 8x8 for CIFAR and tiny image net respectively at its bottom layer. It had kernal size 4x4 at second layer for both data sets. SQHN L3 had kernal sizes 2x2 and 4x4 for CIFAR and tiny image net respectively at its bottom layer. It had kernal size 4x4 at both its second and third layer for both data sets. In the moderate corruption task, 1024 images are presented. In the high corruption task, 128 are presented. The auto-associative task used white noise corruption with variance .2 and .8 for moderate and high corruption task, respectively. For hetero-

associative tasks, 25% and 75% of pixels are masked for moderate and high tasks. Masked pixels are treated as missing in these tasks and are therefore ignored by the bottom hidden layer of SQHN (rather than being treated as 0 values, which affects the normalization term). MSE is only computed in hetero-associative tasks between the output and the original pixels that were missing in the input.

Yoo et al. [139] tests the generative predictive coding network (GPCN) of [111], which trains offline. Yoo et al. develops an version for online training, called BayesPCN, and BayesPCN with forgetting, which prevents learning from slowing too much.

We also compared to MHN models. Following previous works (e.g., [75, 139], we update MHNs with a simple batch update where all the input vectors are stored in columns of a 'memory matrix' M and we set the temperature, β to 10000, essentially treating it as a argmax operator. The original model of [99] performs retrieval via the operation

$$x^{(new)} = Msoftmax(\beta M^{\top}\tilde{x}), \tag{C.6}$$

where β is the temperature. Millidge et al. [75] showed this recall process is a kind of nearest neighbor operation, where a weighted average of memory vectors are returned, and memory vectors more similar to the input, according to a dot product measure, are given more weight. Recall is typically best when β is large, and the nearest memory vector is returned. Millidge showed that other similarity measures work better than dot products. We show the best performing model of Millidge et al. which uses the Manhattan distance. Finally, we also report the results of the gradient-based MHN (MHN grad) listed in [139] that performs recall via a gradient-based inference procedure instead of the one-shot recall process shown above.

An important caveat is that the results reported by [139] used images normalized to -1 and 1. Our SQHN model was designed for non-normalized images with values 0 to 1. To make the

comparison fair, we multiplied the MSE scores for the SQHN by 4 since normalized images (ranging from -1 to 1) have an error range twice as large as unnormalized images (ranging from 0 to 1), which is then squared in the squared error: let $e = x - x^{new}$ be the unnormalized image and output error. If the image and output were normalized before computing the error we would be $2e$. If this error is then squared we get $4e^2$. Further, standard deviation must be multiplied by two.

C.1.5 Online Continual Auto-Associative Memory Tests (Figure 6.2)

In the online-continual learning task, images were presented online (one at a time, and single pass through the data) in a non-i.i.d. fashion. In particular, images from the first task are presented online, then images from the second task are presented online, and so on. We tested grouping images by class (online class incremental or OCI) and by visually distinct data sets (online domain incremental or ODI). Each class and domain group had an equal number of images. In the ODI setting, the one layer models trained one four different data sets/domains: MNIST, MNIST with flipped pixels, FMNIST, and FMNIST with flipped pixels. One layer models were tested with 300, 1300, and 2300 on MNIST data sets, and 700, 2000, 3300 on CIFAR-100. The three layer models trained on CIFAR-10 with dark pixels ($x^{dark} = x * .5$), CIFAR-10 with bright flipped pixels ($x^{light} = (-1 * x + 1) * .5 + .5$), SVHN with dark pixels, and SVHN with bright flipped pixels. The three layer SQHN model had kernal sizes 4x4, 4x4, 2x2, at the first, second, and third hidden layers and was tested with node sizes 200, 600, 1000. During test time training images with noise variance .2 presented.

We compare to another auto-associative memory model known as the modern Hopfield network (MHN) [52, 99]. We compare to the MHN of [99], in particular, because it is the most

similar architecturally to SQHN, and has a very high capacity relative to other MHNs [99]. The one layer model performs recall as:

$$x_{new} = Msoftmax(\beta M^T x), \tag{C.7}$$

where x is the input vector and M with weight matrix, and β a scalar temperature. This same mechanism can be stacked into tree-like hierarchies as well, similar to the multi-level SQHN. For the multi-level MHN we essential use the same architecture (same kernal sizes and node sizes) but replace the argmax with softmax and remove the normalization.

One issue we ran into was there was no standard way to train this model online for auto-associative tasks like the ones used here. The original model of [99] was trained with BP, but not used for auto-associative memory. Instead, it was treated like the attention layer of a transformer, where M is generated by a separate set of weights. Thus, we train the model like the original MHNs of [52], which were used for auto-association, where BP is used to directly optimize M. Since we are training to reduce recall MSE we directly optimize MHNs to reduce recall MSE. We test the model trained with plain SGD, SGD with Adam optimization [48], EWC++ [19] which is an online version of EWC [49], and episodic replay with a small memory buffer [20]. EWC++ uses a moving average of the Fisher information matrix, F, to modulate parameter updates. Parameter updates for EWC are regularized loss gradients where the objective, $\tilde{L}$ is:

$$\tilde{L}^k = L^k + \lambda \sum_i F_{\theta_i^{k-1}} (\theta_i - \theta_i^{k-1})^2, \tag{C.8}$$

where θ_i^{k-1} is the ith value of the parameters from task $k-1$ and λ weight the regularization. In EWC++, the Fisher is computed as a moving average $F^t = \alpha F^t + (1-\alpha)F^{t-1}$, where t is the current training iteration, and a moving average of parameters is kept as well. At the end of each task, $F_{\theta_i^{k-1}}$ is set equal to F^t and the moving average of parameters set

equal to θ_{k-1}. One issue is that EWC++ needs to know task boundaries, so it can store the parameters and Fisher matrix from previous task. In our task, no such supervision is allowed. To deal with this, we just treat each data point as its own task and compute a moving average of the parameters, same as the fisher information matrix. This should still allow parameters and Fishers to maintain information about previous tasks. The λ and α hyper-params are found via grid search. Generally, we find the regularization provides little to no benefit, like because the training runs are short and therefore training speed is as important as preventing forgetting. EWC slowed down training speed too when λ was significant, so the regularization provided little help.

Episodic replay with a tiny memory buffer [20] stores a small sample of previously observed data-points, then uses SGD to update parameters each iteration using the whole sample as a mini-batch. Despite using a small buffer this approach has shown to be highly effectively in continual classification. Methods that sample a mini-batch from a larger buffer can work better [64]. However, we are interested in memory and compute efficient algorithms and are using relatively small data sets. Thus, we use the tiny memory buffer method. There are several algorithms for deciding when to store a data point in the buffer. We use the resevoir sampling method (see [20]), which is simple method that has shown to consistently be better than other methods on classification tasks. Resevoir method is simple: let n be the maximum number of data point that can be stored in the buffer, and t be the total number of data points observed. For the data point, x^t, at each iteration t, check if the buffer is full. If it is not full, add the data point to the buffer. If the buffer is full, randomly overwrite one data point in the buffer with x^t, with probability $\frac{n}{t+1}$.

C.1.6 Noisy Encoding

We compared one and three hidden layer models on EMNIST and CIFAR-100, respectively, in the noisy encoding task. The single layer models were tested with node size 300, and were presented with 300 images. Trees had node size 150 and were presented with 150 images. Therefore, any inability to recall images must be due to an inability to remove noise during learning, rather than capacity limitations. Images were either Gaussian samples (white noise added then clamped to range 0 to 1) or binary samples, were drawn from the original image values. For each image, some number of samples were drawn and presented to the models online in a sequence, before moving to the next image. For EMNIST images, we test 1, 5, 20, and 50 samples. For CIFAR-100 we tested 1, 10, 20 samples. At no point during training was the original, non-corrupted image presented to the model. At test time, the original, non-corrupted image was presented and the model had to reconstruct. We also tested an SQHN model with a slight alteration (SQHN+), where after the hidden states are computed for the first image sample the hidden states are held fixed for the remainder of the samples. This ensured that samples from the same image were encoded to the same latent state. Although SQHN was able to do this well on its own, it sometimes mapped samples from the same image to different latent state in the high noise, binary sample scenario. In these cases, SQHN+ performed better.

C.1.7 Episodic Recognition

The the episodic comparison between SQHN and MHN we use a one layer model and MNIST data sets. The train set is pulled from the MNIST training data. The in-distribution hold out set is from MNIST test data. The out of distribution set is from F-MNIST. We test two methods for performing recognition in an MHN with one hidden layer. The first method uses the activities at the hidden layer as a measure of familiarity/similarity. If that value

is above a threshold, ρ, then the model judges it has observed the data point. The second method keeps a moving average, μ, of the recall MSE during training, and uses a scalar multiple of this average, $\rho\mu$, as the decision threshold. All hyper-parameters found with grid search. Learning rates updated to achieve best performance in terms of recall MSE. ρ updated to achieve best recognition accuracy. We test SQHN and MHN networks with 300 neurons. 3000 data points from EMNIST are presented in an online and i.d.d. fashion. At each test point all previously observed train data are presented along wth an equal number of in-distribution and out-of-distribution data. We plot the MSEs of MHN and SQHN for each dataset to show that MHN generalize so well that they have no performance different between training and in-distribution data, making it impossible for the model to perform recognition.

C.1.8 SQHN Architecture Compare (Figures 6.5 B,D,E)

Auto-association with Varying Amounts of Corruption (Figure 6.5 B) All SQHN models had 1000 neurons at each node and are trained to memorize 1000 images. SQHNs are trained so that each image is encoded in a unique set of one hot vectors at hidden nodes, (which can be done in practice by setting α to a very large number). This means that all SQHN models are not over capacity, and any inability to recall images is due only to their inability to handle varying amounts or types of corruption. Importantly, only auto-associative, and not hetero-associative, recall is tested. The masks added to images are treated as corrupted pixels (i.e., their values are taken as input) rather than missing pixels. For noise tasks we add white noise to the images then clamp the images to values between 0 and 1. Noise variances tested are $[0, .05, .15, .25, .4, .5, .75, 1, 1.25, 1.5]$. Fractions masked that are tested are $[0, .1, .25, .5, .75, .875, .9375]$. For masking tasks a rectangle with random length and width (less than or equal to the width and height of image) is sampled and its position within the image is randomly sampled. The black mask sets pixels equal to

0. The color mask randomly selects an RGB value from a uniform distribution, and sets pixels in the mask equal to that value. Noise mask sets mask values equal to a white noise sample (variance 1), clamped to values between 0 and 1. The one level SQHN-L1 model has input layer kernal size equal to the size of the image. SQHN-L1 essentially performs nearest neighbor operations comparing each images to the set of training images, which stored in columns of its weight matrix, using mean-shifted cosine similarity. SQHN-L2 has an input layer kernal size of 8x8, 16x16, and 32x32 for CIFAR-10, Tiny Imagenet and Caltech 256, respectively. The second layer kernal across all data sets is sized 4x4. SQHN-L3 has an input layer kernal size of 4x4, 8x8, and 16x16 for CIFAR-10, Tiny Imagenet and Caltech 256, respectively. The second and third layer kernal sizes across all data sets are sized 4x4.

Online Auto-associative Learning (Figure 6.5 D) For this task a one, two, and three hidden layer were model were trained in the online i.i.d. scenario on CIFAR-100. Models were trained with 200, 600, 1000 neurons at hidden nodes. SQHN L1 model had input layer kernal size equal to the dimension of the image (32x32). SQHN L2 had input layer kernal size 4x4 (which worked better than the 8x8 kernal used in the experiment above), and second layer kernal size of 8x8. SQHN-L3 has an input layer kernal size of 4x4, 8x8, and 16x16 for CIFAR-10, Tiny Imagenet and Caltech 256, respectively. The second and third layer kernal sizes across all data sets are sized 4x4. α was set very large so networks memorized first J data points, up until capacity reached. We measured recall accuracy without any corruption, and noticed L2 and L3 architectures performed much better than L1. We suspected this was because the L2 and L3 architectures learn representation of small features that generalized widely across training set. To test this we also measured test MSE on the test set from CIFAR-100.

SQHN Episodic Recognition Comparison (Figure 6.5 E) For the recognition task, a train set, in-distribution hold-out set, and out of distribution set are needed. We use CIFAR-10 training images for the train set, a hold-out/test set of CIFAR-10 images as the

in-distribution set, and CIFAR-100 with flipped pixels as out of distribution set (i.e., each image is multiple by -1 then 1 is added). SQHN L1 is given 500 neurons at its hidden node. SQHN L2 and L3 are given 500 neurons at their memory node. Since only the memory node is used directly for recognition, we pre-train the lower layers of SQHN L2 and L3 for 1000 iterations on CIFAR-10 images. SQHN L2 has kernal sizes 4x4 and 8x8 at hidden layers, and channel size of 40 at the first hidden layer. SQHN L3 has kernal sizes 2x2, 4x4, 4x4 at the first, second, and third hidden layers, respectively and channel sizes of 40 and 200 at the first and second hidden layers. After pre-training during the training phase, images from the train set are presented online in and i.i.d. manner. Only the weights leading into the memory node are updated. During testing, models are presented with a set of images, which are composed of all of the training images observed so far, an equal number of in-distribution images, and an equal number of out-of-distribution images. The best guessing strategy is to guess all data points are new which yields 66% accuracy.

C.2 Supplementary Material

C.2.1 Related Works

In this subsection we do a more thorough comparison of SQHN to similar models and methods.

Bio-Inspired Deep, Sparse Networks The model most similar to SQHN is likely the Recursive Cortical networks (RCN) [36], which is another neural network/Bayesian network hybrid that uses an algorithm akin to MAP learning in a tree-structured architecture. SQHNs use distinct learning rules than RCNs. RCNs for example do not perform the averaging operation that SQHNs do and as a result RCNs cannot learn under noise (e.g., see supplemental from [36]). RCNs also have more complex architectures that utilize max-pools and factorize color and shape representations. RCNs also do not explicitly minimize an energy function. Further, as far as we know RCNs have not been tested on natural images, or on associative memory tasks from recent machine learning literature. Hierarchical temporal memory [3] models (HTMs) are another deep neural network model, which, like SQHN, use sparse coding and have similarities to Bayesian networks with tree-like architectures. However, HTMs unlike SQHNs as far as we know, do not explicitly try to implement a discrete graphical model. They also use a different inference procedure, based on thresholding activations rather than one-hots, and a distinct learning algorithm that was not designed to optimize any energy function. Finally, sparse Hopfield-like networks, like those of [92, 105], also have some architectural similarities to SQHN too, but these models do not explicitly encode discrete random variables, they use distinct energy functions, and they have not been applied to memory tasks.

Online-Continual Learning Following previous reviews [95, 47, 131, 96, 35, 64, 41], continual and online-continual learning approaches may be split into several types: 1) Regular-

ization based approaches constrain the way parameters are update to reduce catastrophic forgetting (e.g., [49, 103, 141, 4, 60]). 2) Memory-based approaches store or model previously observed data for the purpose of replaying the data during training (e.g., [20, 117, 5]). 3) Parameter isolation models avoid forgetting by allocating different parameters for each task, either by gating components or by dynamically adding new sub-networks as needed (e.g., [58, 140, 66, 65]). Our SQHN model differs from these approaches since it uses local learning rules, while these previous approaches us BP in some form (e.g., all the methods reviewed by [96, 64] use BP either in pre-training and/or during online training of the classifier). Our SQHN model can be described as using a kind of parameter isolation (via sparse neuron activity and neuron growth) combined with regularization (via learn rate decay schedules) to avoid forgetting, but as far as we can tell SQHNs particular strategy is novel. For example, whereas all the previous works on regularization and isolation use some form of *global* gradient information to regularize and isolate parameters, the SQHN does so based solely on information *local* to the neuron or synapse.

Associative Memory Like the SQHN, classic Hopfield networks [44], modern Hopfield networks (MHN) [52, 99], and predictive coding (PC) networks [100] are all energy based models that can perform auto-associative tasks. SQHN, however, is the only model to explicitly implement discrete graphical model and the SQHN optimizes a novel energy function. Further, unlike the SQHN none of these models (with maybe the exception of [139] which we compare to SQHN below), were designed specifically for online-continual learning.

In sum, although our SQHN model combines some elements from existing model, it utilizing novel, bio-plausible, backprop-free strategies for encoding memories for associative memory tasks. Further, it should be emphasized unlike previous works we are training networks on a largely novel unsupervised noisy encoding memory tasks and recognition tasks, which as far as we can find, have not been directly explored in the machine learning literature.

C.2.2 Derivation of Energy

Consider a directed tree-structured acyclic graph (DAG) where each hidden node represents a discrete random variable. The joint probability of the nodes values, given the learned parameters, is the product of the conditional probabilities of each node given the values of its parent nodes:

$$p(h_0^*, h_1^*, ...h_L^*) = \prod_{l=0}^{L} p(h_l^* | pa_l^t), \tag{C.9}$$

where pa_l refers to the values of the parents of nodes l and h_l^* is the integer value assigned to node l. We derive our novel energy (equation 6.1) function by performing inference using not just the learned parameters, but also taking into account uncertainty over the parameters using a prior probability distribution. As we explain, adding this prior allows us to treat learning as Bayesian inference, which is highly useful for online learning.

Let $M_{pa_l,l}$ be the matrix of learned conditional probabilities over node l given its parent. The values of the parent nodes are discrete integer values represented, in SQHN networks, by the one-hot $h_{pa_l}^*$. The conditional distribution according to the learned matrix is $p_l = M_{pa_l,l} h_{pa_l}^*$, which is equivalent to the column of M_{pa_l} indexed by $h_{pa_l}^*$. However, p_l does not take into account the uncertainty over our parameters. Accounting for uncertainty is important especially early in online training, since early in training parameters have been updated using only a small fraction of the data set, and therefore these parameters are less 'trustworthy' than

A common method to represent uncertainty over parameters is to treat parameters as a random variable and place a prior distribution over it. Learning then amounts to performing Bayesian inference, where the maximum likelihood (learned) parameters are combined with the prior distribution. The typical prior distribution over parameters for discrete graphical models, like Bayesian networks, are Dirichlet priors, which are the conjugate prior for discrete

(e.g., categorical and multinomial) distributions. Details of about this prior can be found in [42]. Here we simply point out that the most common Dirichlet prior is the uniform distribution. If node l is a discrete distribution then the distribution the represents no prior knowledge is the uniform distribution.

There are several ways to place this prior over parameters. First, a prior distribution may be place over each individual column of $M_{pa_l,l}$, where each column represents the conditional distribution for a different parent node value. In this case, uncertainty about the distribution encoded in each column of $M_{pa_l,l}$ is represented separately, and may differ between columns. We use a simpler approach, which is to have one measure of uncertainty over $M_{pa_l,l}$ as a whole. Let's call θ^0 the parameters that set each column of each matrix equal to the uniform distribution. In particular, if we assume that the parent of l is assigned value j, we compute the conditional probability distribution as

$$p(h_l^*|pa_l, \theta^0, \alpha) = \frac{\alpha\frac{1}{J_l} + tp(h_l^*|pa_l)}{t+\alpha} = \frac{\epsilon}{J_l} + (1-\epsilon)p(h_l^*|pa_l), \tag{C.10}$$

where t is the total number of data points observed so far and $\epsilon = \frac{\alpha}{t+\alpha}$. The value J_l is the number of values node l can take and thus $\frac{1}{J_l}$ represents the prior (uniform) probability over child node values. The probability $p(h_l^*|pa_l)$ is the learned conditional probability of h_l^* generated by $M_{pa_l,l}$. The conditional probability at iteration t then is a weighted average between the uniform distribution and the learned distribution generated by $M_{pa_l,l}$. The learned distribution is weighted increasingly heavily as the number of data points observed, t, increases. The weighting also depends on the hyper-parameter α: the larger α is the more heavily the uniform distribution is weighted and the slower its influence will decay.

Taking the joint of these conditionals we get

$$p(h_0^*, h_1^*, ..., h_L^*) = \prod_{l=0}^{L} p(h_l^*|pa_l, \theta^0, \alpha) = \prod_{l=0}^{L} (\frac{\epsilon}{J_l} + (1-\epsilon)p(h_l^*|pa_l)). \tag{C.11}$$

We derive an energy from this expression by approximating the maximization of the joint joint distribution. It is approximated by expanding the product and removing certain terms that are guaranteed to be small relative to other terms and by removing constants which do not affect the local maximum of the joint. To simplify notation, without loss of generality, assume all nodes have the same number of possible values J. Further, let $g = \frac{\epsilon}{J}$ and $p(h_l^*|pa_l) = p_l$ and L be the number of nodes. Now if we expand the equation above we get a combination of sums and products of probabilities:

$$\begin{aligned}\prod_{l=0}^{L}(g+(1-\epsilon)p_l) &= g^L + g^{L-1}(1-\epsilon)(\sum_{l=1}^{L} p_l) \\ &\quad + \sum_{l=2}^{L} g^{L-l}(1-\epsilon)^l(\prod_{k=1}^{l} p_k) + G \\ &= c_0 + c_1 \sum_{l=1}^{L} p_l + \sum_{l=2}^{L} c_2^l \prod_{k=1}^{l} p_k + G\end{aligned} \tag{C.12}$$

where c_0, c_1, c_2 are scalar coefficients, and G is a placeholder for a large number of similar terms we were unable to express concisely. These terms are similar in the sense they all involve a scalar x, a sum of some number k probabilities $y = \sum_{l=1}^{k} p_l$ multiplied by by a product of probabilities $z = \sum_{k}^{L-n} p_k$, where n is some number less than L, having the form xyz.

So we have four terms. A constant scalar c_0, a sum of the conditional probabilities $c_1 \sum_{l=0}^{L} p_l$, a product of probabilities, and a large sum of terms that multiply a sum of probabilities by a product of probabilities. Let's collapse G and the product term into the term S. In the

limit where $S \to 0$ we have

$$\begin{aligned}
\arg\max_{h,\theta} p(h_0^*, h_1^*, ..., h_L^*) &= \lim_{S \to 0} \arg\max_{h,\theta} c_0 + c_1 \sum_{l=0}^{L} p_l + S \\
&= \arg\max_{h,\theta} c_0 + c_1 \sum_{l=0}^{L} p_l \\
&= \arg\max_{h,\theta} \sum_{l=0}^{L} p_l \\
&= \arg\max_{h,\theta} E.
\end{aligned} \tag{C.13}$$

Thus, under this limit $E \approx p(h_0^*, h_1^*, ..., h_L^*)$.

What does it take for this limit to be well approximated in practice? Each in S has form xyz, where x is a non-zero scalar, y will typically be non-zero since it is a sum of probabilities, and z is a product of probabilities, which will be zero when at least one term in the product is zeroed out. The more terms in the product z the more likely the term will be zero, and the less terms the less likely it will be zero (assuming roughly equal probability any one term will be zero). When z has few terms, y is a sum of many probabilities (see above). Thus, those terms in S which have a product z of many probabilities and a sum y of few will be zeroed out more often on average than terms with products of a small number of probabilities and sums of many. This suggests in networks where it is often the case that multiple conditional probabilities are zero each iteration, a sum of probabilities is a good approximation of the joint

SQHNs generally have highly sparse weights, and therefore it will often be the case that multiple conditional probabilities will be zero during learning. For example, if a node l grows a new neuron at the current iteration, its conditional probability according to the parameters at the current iteration will be zero. If node l takes a value that was not previously observed in combination with its sibling node values, it (or at least one of its siblings) will have probability 0. *This suggests that maximizing our energy function in SQHNs is a decent*

approximation to maximizing the joint with the Dirichlet prior over parameters as described above.

C.2.3 Derivation of Inference Procedure

MAP inference in tree-structured graphs with discrete nodes can be implemented via the max-product algorithm [17], whose goal can be described as:

$$\textbf{Max-Product:} \quad \arg\max_{h^*} \prod_{h_l^*} p(h_l^* | pa_l^t), \tag{C.14}$$

where h^* is the set of integer value assignments for each node. The max-product algorithm works by performing a single FF and FB sweep through the network. Consider a tree structured graph where, without loss of generality, we assume the simple case where nodes take one of two value. In this case, the conditional probability of node l given its parent node are represented by matrix $M_{l,pa_l} = [m_0, m_1]$. The max-product algorithm first propagates a signal from the visible nodes up through hidden nodes to the root node using the operation (in our notation) $h_{pa_l} = [\max(m_0 \otimes h_l), \max(m_1 \otimes h_l)]$ [17], where max (not to be confused with the activation function max) outputs a single scalar. In the case of multiple children nodes inputs to h_l are multiplied element-wise. The root node value is set equal to this maximum value, $h_L^* = \arg\max h_L$, then an operation known as backtracking is performed, where a signal is propagated back from the root node down the tree to find the MAP values for hidden nodes (see [17] for details). An analogous algorithm may be used to maximize the SQHN energy E, which is a sum rather than product of conditional probabilities. The analogous operation is $h_2 = [\max(m_0 + h_1), \max(m_1 + h_1)]$. These operations may be used in the same procedure as the max-product to obtain the max E value at the root node, MAP hidden node values via a backtracking procedure.

The issue with this approach is that the main operation used in the FF sweep is not a

standard vector matrix multiply. Instead of multiplying each row/memory vector of $M_2^\top$ element-wise by h_1 then summing over elements, as in a matrix multiply, each row and h_1 are multiplied element-wise then the max value is returned. However, our goal is to create a neural network, that may be easily implemented in hardware that assumes vector matrix multiplies as the main operation (e.g., GPUs and memoristor based neuromorphic hardware). Therefore, we propose the following alternative approximate MAP inference procedure, which only uses vector matrix multiplies and a neuron-wise normalization operation. Like the MAP inference procedures described above, this procedure involves a feed forward (FF) and feedback sweep through the network. During the FF sweep, each node, starting at the lowest layer and working up, updates according to:

$$h_l = \frac{1}{Z} \sum_{c \in ch(h_l)}^{C} M_{l,c}^\top max(h_c), \tag{C.15}$$

where $ch(l)$ is the set of nodes that are children of h_l and $M_{l,c}$ is the matrix from node l to child node c and the italicized max operation to express an activation function that output a vector of all zeros, except for the max element: e.g., $max([.3, .8, .4]) = [0, .8, 0]$. The summed input is normalized by $Z = \frac{1}{C\sqrt{\sum_c \max(h_c)^2}}$.

We can also describe the activation at l's child node as $max(h_c) = argmax(h_c) \max(h_c) = h_c^* \max(h_c)$, where $argmax$ is the activation that returns a one-hot, and the un-italized max returns a scalar, e.g., $\max([.3, .8, .4]) = .8$. Further, for each each neuron j in node l it is the case that $M_{l,c,j}^\top h_c^* = p(h_c^* | h_l = j)$. This means we can express the input to each neuron j in node l as

$$h_{l,j} = \frac{1}{Z} \sum_{c \in ch(h_l)}^{C} p(h_c^* | h_l = j) \max(h_c). \tag{C.16}$$

Thus, the input to each neuron j, during the FF sweep is a sum of the probabilities of child nodes, given node l has value j, where each probability is weighted by the associated max

value at the child node. Therefore, $h_{l,j}$ is the *weighted energy of l's children node values*, when l takes value j. The term Z then takes the average, so that $h_{l,j}$ is a weighted average of the probabilities of child nodes given node l has value j. *Thus, the FF sweep sets nodes to the value that maximizes the weighted average of the probability (i.e., weight energy) of child node values.*.

Importantly, the weighting terms carry information about the energy of descendent nodes: the weighting term $\max(h_c)$ can be defined recursively via equation C.16, which shows that h_c is itself a weighted average of its children node probabilities (i.e., grandchildren of l), and so on. *Therefore, weighting terms act as a kind of bottom up attention mechanism, where inputs from children are weighted based on the weighted energy/probability of grandchildren, whose weight are based on the energy/probability of great-grandchildren, and so on.*

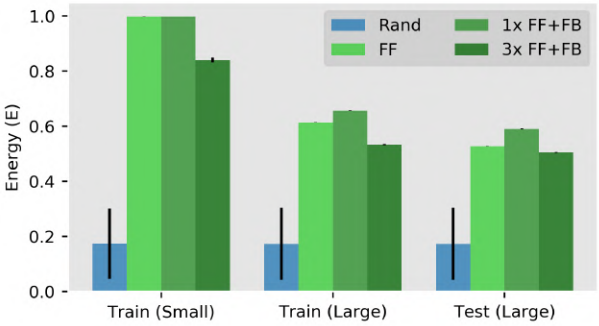

Figure C.1: Energy measurements during SQHN inference procedure. Energy after a single feed forward (FF) sweep, one FF and feedback sweep (1x FF+FB), and three FF FB sweep (3x FF+FB). Energy shown on small train set (200 images), and on large train set (1000 images) and test/hold-out data set. One FB sweep increases energy, but the network performs best only after a single FF and FB sweep.

Nodes at the first hidden layer, on the other hand, receive a signal from visible nodes with no children. It turns out this signal can still be interpreted as the a weighted average of the likelihood of each pixel value. The input to a node at the first hidden layer is

$$h_{l,j} = \frac{.5(m_{l,j}^{\top} - .5)(x_{c_l} - .5)}{\|(m_{l,j}^{\top} - .5)\|\|(x_{c_l} - .5)\|} + .5, \tag{C.17}$$

where x_{c_l} is a vector of input value (e.g., from an image patch). The input we use has values between 0 and 1. We treat each value as a binary variable whose probability is

$$\begin{aligned} p(x_{c_l,i}|h_l^*) &= x_{c_l,i}p_{c_l,i} + (1 - x_{c_l,i})(1 - p_{c_l,i}) \\ &= 2x_{c_l,i}p_{c_l,i} - x_{c_l,i} - p_{c_l,i} + 1 \end{aligned} \tag{C.18}$$

Next, consider that the prediction p_{c_l} just equals the memory vector $m_{l,j}$ indexed by h_l^*. Given this, we can rewrite the numerator of equation C.17 as

$$\begin{aligned} (m_{l,j,i} - .5)(x_{c_l,i} - .5) &= (p_{c_l,i} - .5)(x_{c_l,i} - .5) \\ &= p_{c_l,i}x_{c_l,i} - .5p_{c_l,i} - .5x_{c_l,i} + .25 \\ &= .5p(x_{c_l,i}|h_l^*) - .25, \end{aligned} \tag{C.19}$$

which is just the liklihood of the input with an affine shift applied. Thus, the numerator in equation C.17 computes an affine shifted version of the likelihood of each pixel and sums these together. The denominator ensures the output is between -1 and 1, which is rescaled to the range 0 and 1 by the multiplication and shift of .5. The advantage of computing the likelihood this way, rather than a direct computation of equation C.18 is that this operation can be implemented via a neural network like matrix vector multiply with neuron-wise normalization:

$$h_l = \frac{1}{2Z}(M_l^\top - .5)(x_{c_l} - .5) + .5, \tag{C.20}$$

where Z is a vector of the terms computed accord to the denominator in equation C.17. It is also possible to just store a separate feedback matrix, with shifted and normalized rows.

Finally, after activities h_l are computed via the FF sweep, the final h_l values are computed

using the top-down/FB sweep. Let h_{l+n} be the parent of h_l

$$h_l = (\lambda)h_l + (1 - \lambda)p_l, \tag{C.21}$$

where $p_l = M_{pa_l,l}h^*_{pa_l}$ and λ is a scalar between 0 and 1 that modulates the influence of the top-down signal, which we typically set to .5 during recall.

Algorithm 7: SQHN Recall Algorithm

begin
  for $t = 1$ **to** T **do**
    // Clamp visible nodes to x^t
    // Inference
    for $l = 0$ **to** L **do**
      // Compute bottom up input h_l, equation C.17, C.15
    end
    for *reversed*($l = 0$ **to** $L - 1$) **do**
      // Combine h_l and top-down input, equation C.21
      // Set $h^*_l = argmax(h_l)$
    end
    // Set each input patch $x_{c_l} = Mh^*_l$
  end
end

C.2.4 Neuron Growth and the Dirichlet Process Prior

The inference process described above assumes a fixed number of neurons at each node. It is assumed neuron numbers are fixed during recall, when we are performing inference over *old* data points, from the training set. However, intuitively, it may be desirable to

add new neurons and synapses dynamically as needed during learning, when the model performs inference over *new* data points. A common principled way to dynamically add new components to a discrete/categorical distribution is via the Dirichlet process prior. In parametric models with discrete variables (e.g., standard mixture models), the number of components J is treated as a hyper-parameter, which set by the modeler. In non-parameteric models, J is learned/inferred by the model according to a prior distribution over J. The Dirichlet process prior (DPP) is a common prior for discrete non-parameteric models. We do not go into details on the derivation of the DPP prior here. We only explain how it inspires our model. For good tutorials see [37, 59].

The DPP provides a method for determining when it is more probable that some input belongs to a new latent, integer value $(J+1)$, rather than an already existing value $(\leq J)$. It does this by representing the prior probability that the input belongs to a new integer value, given all previous assignments: $p(h_l^{*,t} = new | h_l^{*,0:t-1})$, which is the prior probability the input to node l at iteration t belongs to a new value given all of the previous value assignments up until current time t. The DPP sets this prior probability as

$$p(h_l^{*,t} = new | h_l^{*,0:t-1}, \alpha) = \frac{\alpha}{t+1+\alpha} \tag{C.22}$$

where α is a hyper-parameter. The DPP is order invariant [37], which means that the ordering the previous data points arrive does not affect the computation of the prior. The posterior probability that the node value is new, multiplies the DPP above with the likelihood of the input according to some prior distribution over parameters [59]. This posterior can then be compared to the posterior probabilities of existing values given a data point, x, and existing parameters. One can then make the decision to add a new value, if this is the most probable case.

Since the SQHN maximizes an energy function rather than the actual joint distribution

and performs approximate inference rather than exact MAP inferance over the energy, our algorithm for neuron growth is inspired by the DPP rather than an exact implementation of it. In particular, we use the following simple computation as an estimate of the energy value associated with adding a new neuron:

$$\epsilon = \frac{\gamma\alpha}{t+1+\alpha}, \tag{C.23}$$

where α and γ are hyper-parameters and $0 < \gamma \leq 1$ and $0 < \alpha$. One can think about γ as analogous to an estimate of the likelihood the input belongs to a new value (according to some prior distribution) and $\frac{\alpha}{t+1+\alpha}$ as analogous to the prior probability of the the input belonging to a new node value/neuron. During learning the SQHN performs inference before updating weights. If the maximum internal state value (energy), $\max(h_l)$, at node l is less than ϵ, a new neuron is grown. Otherwise, the node is set to the maximum existing value (see next section).

C.2.5 Derivation of Learning Algorithm

Under the SQHN algorithm, parameters are updated each iteration to solve the following optimization problem:

$$\begin{aligned} \theta^t &= \arg\max_{\theta} \sum_{i=0}^{t} E(\theta, h^{*,i}, x^i) \\ &\text{s.t.} \sum_j p_{l,j} = 1 \ \forall l^{hid} \text{ and } p_{l,j} \geq 0 \ \forall l, j \end{aligned} \tag{C.24}$$

where $h^{*,i}$ are one-hot value assignments for each node associated with data point x^i, t is the current training iteration, x^i is the data point presented at iteration i, and l^{hid} refers to hidden nodes. The weight update at iteration t sets parameters equal to the values that maximize the energy over the current and all previously observed data points, and associative

hidden states, under the constraint the prediction p_l at hidden nodes are properly normalized. Importantly, since this update is computed online, we assume we have to perform this update only provided the data point and hidden states at the current iteration (i.e., we are assuming there is no buffer storing previous data and hidden states). This leaves two questions: 1) How do we solve this constrained optimization problem, in the online setting, and 2) how should the values of h^* be set each iteration?

The same optimization above has been solved for the Bayesian networks [42]. Bayesian networks update parameters to maximize the *product* of conditional probabilities. SQHN networks update parameters to maximize the *sum* of conditional probabilities. Despite these differences, the goal ends up being equivalent w.r.t. to parameter updates: each matrix representing the conditional probabilities between parents and children must be updated to maximize the conditional probabilities of the observed child values (under normalization constraints). Thus, SQHN networks can utilize the solution for Bayesian networks as a basis for its online learning rule.

Consider the case where child node, l, is a discrete/categorical variable. The matrix containing the conditional probabilities between the parent of l, pa_l and l is M_{l,pa_l}. The columns of this matrix contain the conditional probability distributions given various values of the parent node: $m_{l,pa_l} = [m_{l,0}, m_{l,1}..., m_{l,J}]$, where $m_{l,j}$ is the conditional probability distribution over values of l, given $pa_l = j$. Each value of $m_{l,j}$ tells us the probability that l takes a particular value given the $pa_l = j$. In particular, $p(l = k|pa_l = j) = m_{l,j,k}$, where k is the kth value of $m_{l,j}$. The update for Bayesian networks tells us that this value is simply the number of times j and k co-occurred previously, divided by the total number of times parent to value j occurred [42]:

$$p(l = k|pa_l = j) = m_{l,j,k} = \frac{c_{l,j,k}}{c_{l,j}}, \tag{C.25}$$

where $c_{l,j,k}$ is the number of times $pa_l = j$ and $l = k$ at the same iteration. The term $c_{l,j}$ is the number of times $pa_l = j$. This solution entails the matrix column $m_{l,j}$ is an average over the one-hot values of the child node that were present when $pa_l = j$:

$$m_{l,j}^t = \frac{1}{c_{l,j}^t} \sum_{i \in t_j} h_l^{*,i}, \tag{C.26}$$

where t_j refers to the set of time steps where $pa_l = j$. This same average can be computed online:

$$m_{l,j}^t = m_{l,j}^{t-1} + \frac{1}{c_{l,j}^t}(h_l^{*,t} - m_{l,j}^{t-1}), \tag{C.27}$$

under the assumption $pa_l = j$. This same update can more generally be implemented as a local, Hebbian-like learning rule:

$$M_l^t = M_l^{t-1} + \frac{1}{c_l^{t,\top} h_{pa_l}^*}(h_l^{*,t} - p_l^{t-1}) h_{pa_l}^{*,t\top} \tag{C.28}$$

where the update uses the fact that $p_l = M_l h_{pa_l}^*$ which trivially entails that p_l equals the memory vector indexed by $h_{pa_l}^*$, e.g., if $pa_l = j$ then $p_l = m_{l,j}$. Second, the step size $\frac{1}{c_l^{t,\top} h_{pa_l}^*}$ uses the fact that $h_{pa_l}^*$ indexes the correct count, e.g. if $pa_l = j$ then $c_l^{t,\top} h_{pa_l}^* = c_{l,j}^t$.

This is the update rule if children nodes are are discrete/categorical variables. We use the same update for visible nodes, which we treat as vectors of binary variables. The properties remain the same in this case: $m_{l,j,k}$ ends up representing an average over previously observed values of input value k in image patch l.

Next, is the question of how to set the hidden node values during training? The weight updates above assume h_l^* values are *observed*. However, when h_l^* is a hidden node its values are, by definition, not observed. One option for setting hidden node values is to use MAP learning. MAP learning works by first performing MAP inference over hidden variables to

find the specific values of the hidden variables, h^*, that maximize the posterior $P(h^*|x,\theta) \propto P(h^*,x,\theta)$. Parameters are then updated to further increase the probability of the joint $P(h^*,x,\theta)$. We use a variant of MAP learning here. In particular, hidden node values are determined by an inference procedure where each node maximizes the probability of its children nodes: each node receives the FF signal from its children, then it is either assigned a one-hot indexing the existing neuron with the maximum internal state value, or if all existing neuron values are below the threshold ϵ, the node is set to a new value (i.e., a new neuron is grown) (see algorithm 8).

The resulting algorithm as a kind of hybrid between MAP learning and learning in the case where all node values are observed. Like MAP learning we set the hidden variables to a point estimate value with the highest probability. (This is opposed to computing a distribution over hidden nodes values, which is what the common expectation maximization [27, 42]) algorithm does.) However, unlike MAP learning, we do not compute the maximum of the posterior since we ignore FB/top-down signal in inference. Instead, similar to [36], our algorithm computes the values that maximize the conditional probability of its child node values, while ignoring parent node values. Nodes, therefore, maximize the likelihood of their child node values. Thus, like the case where all node values are observed, each parent node does not influence/determine the value of its child node, but instead only 'observes' its child's values and updates its conditional distribution accordingly. But unlike this fully observed case, hidden node values are set via inference.

We find this max-likelihood variant works better than MAP in the online setting. MAP inference ends up pushing hidden node values to the values that were inferred in previous iterations, making the model get 'stuck' reproducing the same hidden variable values rather than learning new combinations of values for different inputs. Removing the FB/top-down influence during learning allows nodes to assign new sets of values to new inputs, as desired.

Algorithm 8: SQHN Learning Algorithm

begin

  for $t = 1$ **to** T **do**

    // Clamp visible nodes to x^t

    // Max Likelihood Inference

    for $l = 0$ **to** L **do**

      // Compute bottom up input h_l, equation C.15, C.17

      // If h_l is less than ϵ, set h_l^* to new value.

      // Else $h_l^* = argmax(h_l)$

    end

    // Update ϵ, equation C.23

    // Update Weights, equation C.28

  end

end

C.2.6 Derivation of Episodic Recognition Rule

Recognition tasks begin by presenting a sequence of items/data points from the training set X_{train} during a training phase. Then during a testing phase, the model is presented with a mixture of old data points from X_{train} and new data points from a similar (in-distribution) data set $X_{in-dist.}$ and from a dissimilar (out of distribution) data set $X_{out-dist.}$. An equal portion of data is presented from each data set during testing. The model must correctly judge if the data point is in the training set (old) or not (new). This task is based on classic and common tests of human memory [106].

Importantly, episodic recognition is distinct from the common OOD detection task in machine learning and will therefore require a different solution. OOD detection is the task of

detecting data drawn from a distribution distinct from the training set [137]. Solutions typically involve learning a generative model of the training data, then computing the likelihood of data under this model [137]. This is not the same as detecting data that was not present during training, since many unobserved/new data may still be from sampled from the same distribution as the training set, yet still be unobserved/new.

Inspired by probabilistic models of human episodic recognition (e.g. [116]), we instead use the following approach. We assume each training input x^t is mapped to, what we call, a global feature representation h^*. By global, we mean it represents features of the entire input (e.g., entire image) rather than just a sub-portion of it (e.g. image patch). Then each feature vector is stored in an itemized memory via a mixture model. Let M be the matrix that contains cluster means. Feature representations, $h^{*,t}$ at each training iteration t, get stored in the columns of M, where $M = [h^{*,0}, h^{*,1}, ...]$. If every data point from the train set gets stored separately then during testing the model can simply compare a feature representation h^* of the input to the stored feature representations. If the likelihood is around 1, then the model judges 'old' if less then one, it judges new.

Up until it reaches capacity, and with a high α, this is exactly what happens at the memory node of SQHN. The memory node has full receptive field and takes as input the activities from the values of its children, which represent the input's visual features. Thus, until capacity is reached, SQHN has a clear way of making old/new judgments: perform max-likelihood inference (see algorithm 8), and check if children of the memory node have probability/energy ≈ 1.

However, after capacity is reached, every data point is not stored in M_L in its original form. Instead, new inputs will be averaged with old ones. This raises the question of how judgments should be make after capacity is reached. We use the following strategy. Let $p(h^*_{c_L}|h^*_L = j)$ be the likelihood of the values of the children of the memory node given the memory node value $h^*_L = j$. Here for simplicity we assume this is a good estimate of the

likelihood $p(x^t|h_L^* = j)$. Let's say n data points from the train set have been assigned to j. Lets say we know the data point assigned to this value with the lowest likelihood is x_{min}. If we know this likelihood, we know any new data point assigned to j that has a likelihood lower than this minimum must be new (i.e., have 0 probability of being old). Thus, we propose approximating the probability that some data point x^t is old using the re-scaled likelihood

$$p(x^t = old|h_L^* = j) \approx p(x^t|h_L^* = j) * (1 - p(x_{min}|h_L^* = j)) + p(x_{min}|h_L^* = j) \tag{C.29}$$

Then to make a recognition judgments, the model checks if $p(x^t = old|h_L^* = j)$ is greater than or less than .5. Equivalently, we could set a threshold, μ, equal to the likelihood value at which $p(x^t = old|h_L^* = j) = .5$. In practice, we find the minimum and midpoint likelihood difficult to exactly compute online. Instead, we use a simple estimate of the mid-point value, which we find works well in practice. This estimate just keeps a moving average of the likelihoods:

$$\mu_{l,j}^t = \frac{1}{c_{l,j}^t} \sum_{n=1}^{t} h_{l,j}^{*,n} = \frac{c_{l,j}^t - 1}{c_{l,j}^t} \mu_{l,j}^{t-1} + \frac{1}{c_{l,j}^t} h_{l,j}^{*,t}, \tag{C.30}$$

where the equation on the right is how to compute this average online.

Algorithm 9: SQHN Episodic Recognition Algorithm

begin

  for $t = 1$ **to** T **do**

    // Clamp visible nodes to x^t

    // Max Likelihood Inference (w/o Neuron Growth)

    for $l = 0$ **to** L **do**

      // Compute bottom up input h_l, equation C.17, C.15

      $h_l^* = argmax(h_l)$

    end

    // If $\max(h_L) > \mu_j$ (where $\arg\max(h_L) = j$), judge old

    // Else judge new

  end

end

C.2.7 Theoretical Results

Capacity We analyze the properties of the recall accuracy of an SQHN with one hidden layer. We define the capacity of the network as the maximum number of data points the network is able to recall given some recall threshold γ. Let J be the number of neurons at the hidden layer.

Theorem C.1. *Assume no two data points in the training data set are scalar multiples of each other and during recall data points are not corrupted. The capacity of a single hidden layer SQHN network is at least J, i.e., the number of neurons at the hidden layer, for any* $\gamma \geq 0$.

Proof. Assume a single data point is stored in each memory vector. If the child node is a discrete variable then the input to each hidden neuron, j is the cosine similarity between

the input and the jth memory vector. If the child node is a binary variable then the input to each hidden neuron, j is the mean-shifted cosine similarity (equation C.1) between the input and the jth memory vector. Both will return the maximum possible value for two identical vectors, as long as no two vectors have the same angle. We assume no to vectors have the same angle (no two stored vectors are scalar multiples of each other). Therefore, the maximum value at the hidden layer is guaranteed to index the correct stored training data point, which will yield perfectly accurate MSE of 0 and thus perfect recall accuracy for any γ. If some memory vectors average over data points, it is not guaranteed (though possible depending on how small γ and how similar averaged vectors) returned memory vectors yield correct recall for obvious reason. Thus, the minimum capacity is at least the number of neurons at the hidden layer, J. □

Forgetting It is also important to understand what happens when the SQHN network is pushed passed capacity. Ideally, memory models should show 'graceful' forgetting in such cases, where performance decreases in a predictable and non-abrupt manner. We assume a kind of worst-case conditions where 1) data points are highly orthogonal such that averaging two or more data points in some memory vector m_j means that returning the memory vector m_j during recall does meet the condition for successful recall for any of the data points it was averaged over, and 2) each existing memory vector is equally likely to be accessed and updated during training.

Theorem C.2. *Under the worst case assumptions, the number of memory vectors that store a single data point decreases at an exponential rate of* $Je^{\frac{-t}{J}}$*, where* t *is the training iteration (starting after the network is at capacity) and* J *is the number of hidden units. This entails its recall accuracy will decay at rate* $\frac{Je^{\frac{-t}{J}}}{J+t}$.

Proof. The process we are characterizing is the training of a basic SQHN unit that begins at capacity (each memory vector is set equal to one previously observed data point). Each

training iteration, t, a memory vector is chosen with uniform probability and the data point observed at that iteration is averaged with the chosen memory vector. There are thus two kinds of memory vectors: those that store a single data point and those that store an average over more than one data point. We want to characterize how the number of memory vectors that store only a single data point decrease over time on average, where by 'on average' we mean that the decreases are measured and averaged over an infinite number of training runs.

Let $I(t)$ be the number of memory vectors, at time t, that store only a single data point. Under these assumptions, the decrease in $I(t)$ is clearly equal to the probability of choosing and updating a memory vector that store only a single data point, since this is equal to the proportion of training runs that will choose and update such a memory vector. Under the assumption of a uniform probability distribution, the probability of choosing and updating a memory vector that equals a single data point is just equal to the proportion of memories that store a single data point at time t:

$$\frac{\partial I(t)}{\partial t} = -\frac{I(t)}{N}. \tag{C.31}$$

A function $f(x)$ decays exponentially according to $f(x_0)e^{-\lambda x}$ with rate λ and initial value $f(x_0)$, if $\frac{\partial f(x)}{\partial x} = -\lambda f(x)$. Clearly, the rate of forgetting shown above is an exponential with rate $\frac{1}{N}$. At the initial iteration it is assumed $I(0) = N$. Therefore, the forget rate can be described as $Ne^{\frac{-t}{N}}$. □

Thus, the SQHN memory unit or one hidden layer model shows a graceful, exponentially decaying forget rate.

Learning and Parameter Isolation Let's consider a distance metric between the parameters before and after the update in terms of the amount of energy that is left for parameters

to reduce:

$$d(M_l^{new}, M_l^{old}) = E_{ch_l}(M_l^{new}, h_l^*, h_{ch_l}^*) - E_{ch_l}(M_l^{old}, h_l^*, h_{ch_l}^*), \tag{C.32}$$

where E_{ch_l} is the energy (averaged conditional probability) of the children of l, and M_l^{old} is the matrix of conditional probabilities from l to its children before the weight update, and M_l^{new} the conditional probabilities after. Thus, this distance measure tells us how different the conditional probabilities of the children are under the new and old matrix.

Here we analyze properties of the SQHN learning algorithm in a case where two simplifying assumptions are made:

1. Each column of M_l is updated with an equal, constant step size.
2. The weight are updated to maximize energy to a specific value: $\kappa = E_{ch_l}(M_l^{new}, h_l^*, h_{ch_l}^*)$.

From this the following theorem clearly follows.

Theorem C.3. *Under the assumptions above, the value of h_l^* computed during the training phase of the SQHN is equivalent to*

$$h_l^* = \arg\min_{h_l^*} d(M_l^{new}, M_l^{old}). \tag{C.33}$$

Proof. During learning, each node l takes value h_l^* that maximizes the energy of its children. Thus, during learning h_l^* is the value that maximizes $E_{ch_l}(M_l^{old}, h_l^*, h_{ch_l}^*)$. From assumption 2, we know after the update for any value of h_l^*, the energy of the children is κ. Thus, it clearly follows that the value h_l^* can be described as the value minimizing

$$\begin{aligned} \kappa - E_{ch_l}(M_l^{old}, h_l^*, h_{ch_l}^*) &= E_{ch_l}(M_l^{new}, h_l^*, h_{ch_l}^*) - E_{ch_l}(M_l^{old}, h_l^*, h_{ch_l}^*) \\ &= d(M_l^{new}, M_l^{old}). \end{aligned} \tag{C.34}$$

□

In this simplified case, setting h_l^* to the value that maximizes the energy of the children of l, is equivalent to setting h_l^* equal to the value that minimizes *the amount that conditional probabilities encoded in M_l change during the weight update*. By first maximizing the energy w.r.t. h_l^*, there is less 'work' for weight updates to do when they increase energy w.r.t. M_l. In other words, the SQHN algorithm seems to have a principled method for choosing which parameters to isolation during training, i.e., update the column in M_l that requires the least amount of change given the value of children nodes. This makes sense from the point of view of prevent catastrophic forgetting, where we generally want to reduce the amount parameters change. Future work could look to understand how this result applies to less simplified cases, better describing the SQHN.

C.2.8 Relations to Hopfield Network and Transformers

The SQHN has close relations to the continuous modern Hopfield network (MHN) [99]. The continuous MHN network is

$$x_{new} = Msoftmax(\beta M^T x), \tag{C.35}$$

where M is a matrix of data points/memory vectors, x is an input/query vectors, and β is the temperature. The MHN takes the dot product between each memory vector and the query vector and passes the output through a softmax softmax. These softmax values are then multiplied again by the weight matrix. It was shown by [99] that this model is identical to the attention layer of the transformer in the case where the transformer key (K) and value (V) matrices are tied/identical. One interpretation of this operation is that it is performing a kind of nearest neighbor computation (e.g., CITE), similar to memory retrieval

in a dictionary, where similarity values between memory vectors (which are like templates stored in the dictionary) and query vectors are computed using the dot product. Then a weighted average of memory vectors is returned, where those memory vectors more similar to the query vector are given more weight. An SQHN network with a single hidden layer also performs a nearest neighbor operation. This simple network performs recall given query vector x with the following equation

$$x_{new} = Margmax(\frac{1}{Z}M^T x), \tag{C.36}$$

where $\frac{1}{Z}$ normalizes each elements of the hidden layer input, $M^T x$, such that the values range between zero and one and the input, x, and memory vectors, m_j, are normalized. Clearly, this is a kind of nearest neighbor operation similar to the MHN, except 1) the similarity is computed using a kind of cosine similarity (instead of dot product) and 2) instead of a weighted average, SQHN returns the single memory vector that is most similar to query vector. MHN may equivalently also use this hard recall procedure in the case where $\beta \to \infty$. However, special kinds of normalization would need to be added to the standard MHN and transformer attention to yield equivalence to SQHN.

C.2.9 Supplementary Figures

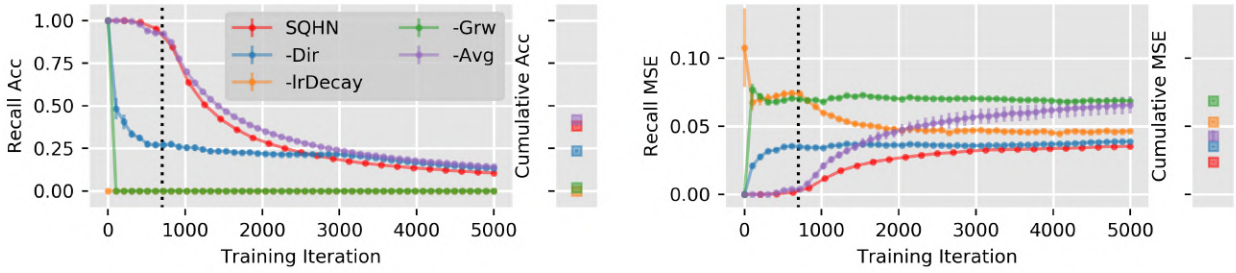

Figure C.2: Ablation study. We test the effects of removing various components of the SQHN learning algorithm in the online continual recall task on CIFAR-100 for the basic SQHN unit (figure C.2. We test the following ablations: 1) remove the Dirichlet based, exponentially decaying grow threshold and replace with a constant threshold (-Dir). 2) Remove the learning rate decay schedule and replace with a constant learning rate (-lrDecay). 3) Remove the grow operation and instead randomly initialize weights (-Grw). 4) Remove the averaging update and only grow new memory vectors (-Avg). The decaying grow threshold, learning rate schedule, and the grow operation are essential for high performance before the capacity (vertical dotted line) is reached. The grow operation, learning rate decay, and averaging operation are all essential for high performance after capacity is reached. Note that removing averaging helped slightly with the number of recalled images, however, doing so significantly worsened recall MSE. All components are necessary for a high cumulative performance under both measures.Recall accuracy and cumulative accuracy for various ablations is shown on the left. Recall MSE and cumulative MSE for the same ablations shown on right.

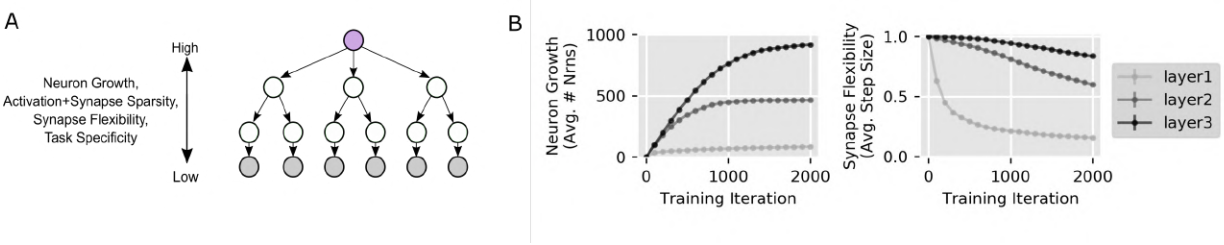

Figure C.3: Emergent properties. **A** Diagram summary of the emergent properties. **B** Measurements of neuron growth (left) and synapse flexibility (right) at each layer during training in online scenario in an SQHN with three hidden layers. Each node in the network has the same maximum number of neurons (1000), the same grow threshold, the same input kernal size (4x4), same learning rate decay, etc. The only difference between each node is its location in the network.

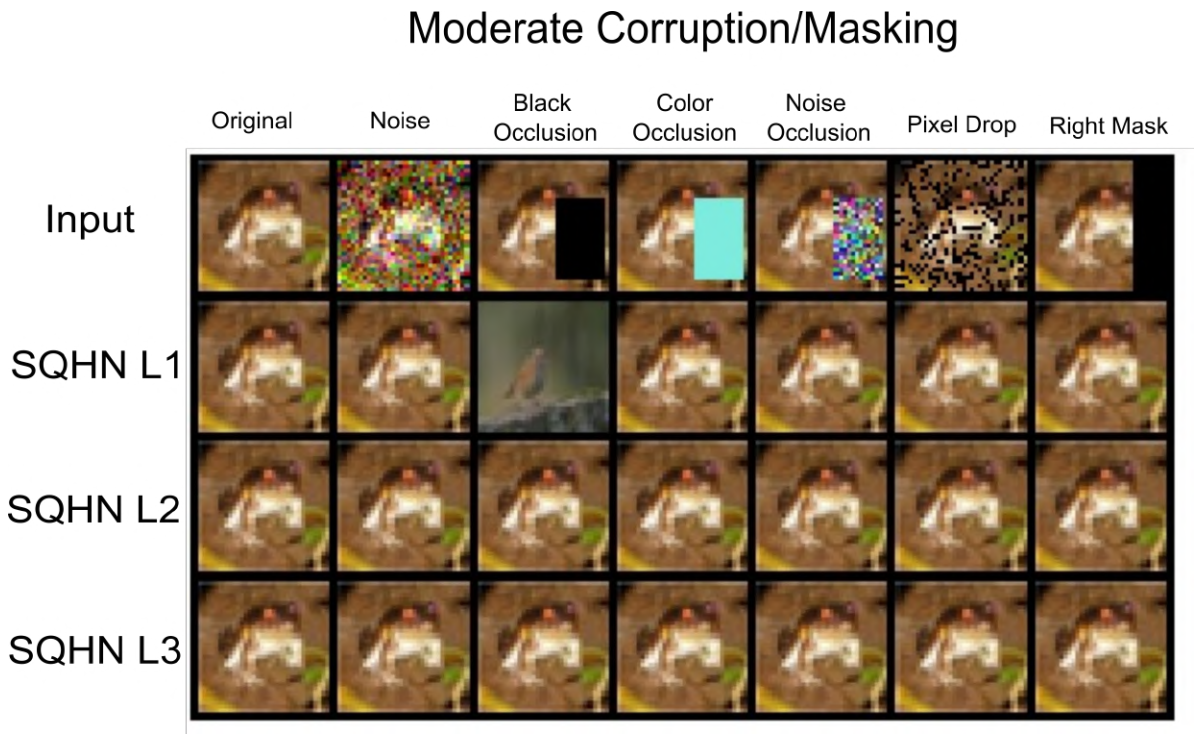

Figure C.4: Example outputs for moderate corruption scenario. Noise variance .2, masking and occlusion covers .25 of image. Trees are better at occlusion, but otherwise, with moderate corruption/masking all networks perform very well on all tasks.

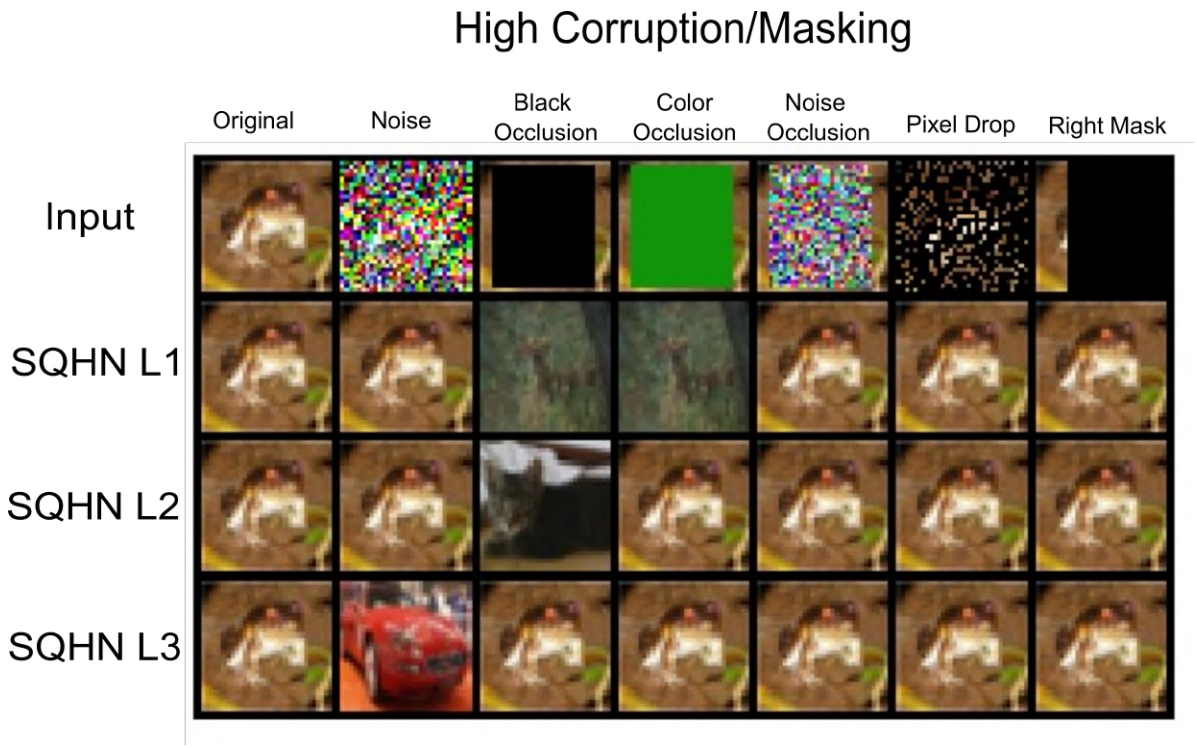

Figure C.5: Example outputs for high corruption scenario. Noise variance .8, masking and occlusion covers .75 of image. Taller trees, especially L3, are better at occlusion, but because they have small receptive fields at bottom layers they are more sensitive to noise. Smaller receptive fields mean smaller input vector per node, and smaller vector are more likely to overlap due to noise. The part-whole representation of the image, however, allows trees with multiple levels to better ignore occluded regions of the image.