UNIVERSITY OF CALIFORNIA, MERCED

DEEP LEARNING METHODS FOR ENGINEERING APPLICATIONS.

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Mechanical Engineering

by

David Laredo Razo

Committee in charge:
Professor Jian-Qiao Sun, Chair
Professor Dong Li
Professor YangQuan Chen

The thesis of David Laredo Razo is approved:

Jian-Qiao Sun, Chair Date

Dong Li Date

YangQuan Chen Date

University of California, Merced

DEDICATION

Sincere gratitude to all the people that helped me in their own way not only with my thesis, but with the overall process of graduate school.

To my advisor Dr. Jian-Qiao Sun for his guidance and support. For his trust in me and this work and for his great lessons not only in engineering but also in life. Thank you for caring for your students and treating us with so much respect and friendliness. Thank you for the opportunity to work in your team.

To my parents Angelica and Rogelio whom I profoundly love and whose support has been invaluable. Their teachings and lessons have made me who I am. This venture could note have been possible without them. To my beloved brother Hiram, who has always been a a great support.

To Dr. Oliver Schütze for his support while I was in Mexico recovering. Thank you for the time you devoted to our joint projects and for always being willing to help.

To all the people that worked with me during my time in the lab. To Zhaoyin (Vivian) Chen, Yulin Qin and Ghazalee Leylaz. Thank you for your effort and valuable work.

To my girlfriend Rosa Garcia who is an important part of my life. Thank you for listening to me, for caring for me, for sharing beautiful moments with me.

To my friends at the UC, specially Suryabhan Singh and Sai Hota. Thank you for the time spent, the moments and the good memories.

To my committee members Dr. Dong Li and Dr. YangQuan Chen for the time spent checking this thesis and their valuable observations of this document.

To CONACYT, UC Mexus and the UC for granting me the opportunity to pursue graduate studies and providing me with all the financial support and necessary tools to accomplish this goal.

To my god Jehova who is always with me and guides me through the right way. For his blessings and teachings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge the following institutions/individuals for the support provided for this thesis document

The research in this thesis was supported/partially supported by the CONACyT/UCMEXUS scholarship number 293991/438579.

The Chapter 3 of this thesis is reprinted from Neural Networks, 116, David Laredo, Zhaoyin Chen, Oliver Schütze and Jian-Qiao Sun, A neural network-evolutionary computational framework for remaining useful life estimation of mechanical systems, 178-187, Copyright (2019), with permission from Elsevier. Dr. Jian-Qiao Sun directed and supervised research which forms the basis for the thesis/dissertation. Zhaoyin Chen synthesized some of the results in the chapter. Dr. Oliver Schütze reviewed the optimization part and provided valuable suggestions to improve the quality of the work.

ABSTRACT

Neural networks and deep learning are changing the way that engineering is being practiced. New and more efficient deep learning models are having a large impact in many engineering fields. Common engineering applications of deep learning are prognostics and health assessment of mechanical systems, design of optimal control, and surrogate modeling for computational fluid dynamics.

However, for a deep learning application to be successful there are a number of key decisions that have to be taken. Some of these decisions include the choice of a certain model architecture or topology, hyper-parameter tuning, and a suitable data pre-processing method. Efficiently choosing these key components is a time-consuming task usually entailing a staggering number of possible alternatives. This thesis aims to develop methods that make this process less cumbersome.

In this thesis, we propose a framework for efficiently estimating the remaining useful life (RUL) of mechanical systems. The framework mainly focuses on the data pre-processing stage of a machine learning pipeline. Making use of evolutionary algorithms and strided time windows, the presented framework can help process the data in a way that even simple deep learning models can make good predictions on it. The framework is tested using the C-MAPSS dataset, which consists of data recorded from the sensors of simulated jet engines. The obtained results are competitive compared against some of the recent methods applied to the same dataset.

Furthermore, an algorithm for efficiently selecting a neural network model given a specific problem (classification or regression) is also developed. The algorithm, named Automatic Model Selection (AMS), is a modified micro-genetic algorithm that automatically and efficiently finds the most suitable neural network model for a given dataset. The major contributions of this development are a simple list based encoding for neural networks as genotypes in an evolutionary algorithm, new crossover and mutation operators, the introduction of a fitness function that considers both, the accuracy of the model and its complexity and a method to measure the similarity between two neural networks. AMS is evaluated on two different datasets. By comparing some models obtained with AMS to state-of-the-art models for each dataset we show that AMS can automatically find efficient neural network models. Furthermore, AMS is computationally efficient and can make use of distributed computing paradigms to further boost its performance.

Chapter 1

INTRODUCTION

Not all that long ago, engineering was a profession conducted with pencils and paper. Today, engineering is a discipline intensely involved with computational and software tools. Computer-assisted design, computational fluid dynamics, and finite-element analysis applications are some of the basic tools that engineers deploy when tackling difficult challenges.

In the recent years there has been an increasing interest in using artificial intelligence techniques to reach higher levels of product automation and accelerate innovation of new products. Advances in IA, combined synergistically with other technologies such as cognitive computing, Internet of Things, 3-D printing, advanced robotics, etc., are transforming what, where, and how products are designed, manufactured, assembled, distributed, serviced and upgraded [1].

Machine learning (ML) advances in the last few years have seen the blossom of a number of techniques that, while mainly applied at fields such as image processing, object detection and natural language processing, can be used to solve and/or gain a better understanding of some mechanical processes. Methods like reinforcement learning have been successfully applied for control [2–4] , fault detection in mechanical systems is often times solved using clustering methods [5], neural networks [6], or support vector machines [7], estimation of remaining useful life has been tackled through regression using neural networks [8,9] and convolutional neural networks [10], finally computational fluid dynamics has seen an increase in the use of neural networks [11, 12].

Flexibility is at the core of the success of many of the recent machine learning algorithms, nevertheless it is this flexibility that may make the use of machine learning techniques somewhat frustrating for practitioners not familiar with the foundations of the algorithms; among such practitioners we can find economist, doctors, and, to a large extent, engineers. Indeed, an increasing number of non-machine learning experts require off-the-shelf solutions that hide the cumbersome details of building efficient machine learning models [13]. The machine learning community has aided these users by making available a variety of easy to use learning algorithms and feature selection methods as WEKA [5] and PyBrain [14]. Nevertheless, the user still needs to make some choices which not may be obvious or intuitive (selecting a learning algorithm, hyper-parameters, features, etc).

Two of the key decisions that have to be made when applying a machine learning model are the choice of a suitable algorithm and the fine tuning of the algorithm's hyper-parameters. A poor choice of any of these two factors usually leads to poor inference and sub-optimal results. This thesis aims to develop tools and methods for finding high performance machine learning models that are efficient and easy to use and implement.

1.1 Building Machine Learning Pipelines

Usually, the process of selecting a suitable machine learning model for a particular problem is done in an iterative manner. First, an input dataset must be transformed from a domain specific format to features which are predictive of the field of interest. Once features have been engineered users must pick a learning setting appropriate to their problem, e.g. regression, classification or recommendation. Next users must pick an appropriate model, such as support vector machines (SVM), logistic regression, any flavor of neural networks (NN), etc. Each model family has a number of hyper-parameters, such as regularization degree, learning rate, number of neurons, and each of these must be tuned to achieve optimal results. Finally, users must pick a software package that can train their model, configure one or more machines to execute the training and evaluate the model's quality. It can be challenging to make the right choice when we face so many degrees of freedom, leaving many users to select a model based on intuition or randomness and/or leave hyper-parameters set to default. Such an approach will usually yield sub-optimal results. Figure 1.1 shows a flowchart describing the usual pipeline used by most of the machine learning practitioners

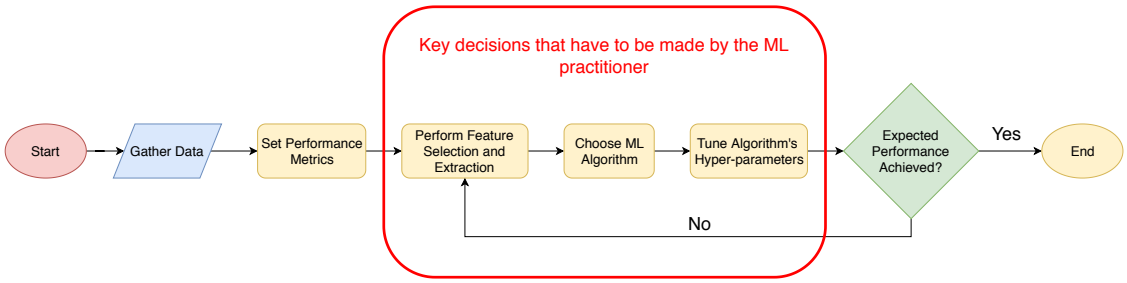


Figure 1.1: Machine learning pipeline.

The main focus of this thesis is on the red part of the process displayed in Figure 1.1. In this thesis we develop methods and guidelines for the feature selection, choice of machine learning algorithm and hyper-parameter tuning parts of the machine learning pipeline.

1.1.1 Algorithm Selection

One of the most critical parts of a machine learning pipeline is the selection of a suitable model. Given some data and some inference task (regression or classification), the machine learning practitioner must choose a proper algorithm based on the characteristics of the data. For instance, if the user is dealing with a classification task and the data is linearly separable, logistic regression [15] or support vector machine [15] may be a good choice. Nevertheless if the data is not linearly separable, if it entails nonlinear operations or if the dataset is too complex, neural networks [9, 10, 16] may be a better option.

There are several different types of neural networks. Among the most popular ones are multilayer perceptrons (MLPs) [17], convolutional neural networks (CNNs) [18], recurrent neural networks (RNNs) [18], and generative adversarial networks (GANs) [19]. The choice of what type of network depends mainly on the type of problem at hand; CNNs have demonstrated to be really efficient for image processing [20, 21], nevertheless some other applications such as remaining useful life have been successfully solved using CNNs [8, 10]. Applications of MLPs range from image classification [22, 23], to fault detection [6], to the estimation of remaining useful life [10], finally RNNs are extremely useful for tasks involving sequences such as natural language processing [24] or prognostic tasks [25].

Given the wide range of neural network types, an inexperienced machine learning practitioner can easily make a poor choice when selecting a neural network for its problem which, even for experienced users, can be time-consuming given the iterative nature of the process. It is therefore important to have tools that make this choice easier and, to some extent, reduce the effective time spent on it.

1.1.2 Hyper-parameter Tuning

Hyper-parameters are parameters that are defined before the learning process begins and they do not change during the learning process. Furthermore, hyper-parameters define the behavior of the machine learning algorithm and ultimately its inference power, common examples of hyper-parameters include: learning rate of neural network, number of clusters for K-means algorithm, tree depth for decision trees, etc. Some simple algorithms (such as least squares method) require none hyper-parameters while many of the most sophisticated algorithms (such as support vector machines, decision trees or neural networks) require several of them. In the following, an example to illustrate the influence of hyper-parameters in the outcome of a machine learning algorithm is provided. For the sake of the example the definitions and notation used will be rather informal.

Consider the C hyper-parameter used in support vector classifiers (SVCs), which are a type of support vector machines (SVMs) used for classification tasks. Informally, SVCs separate different classes by finding the hyper-plane that maximizes the margin between the classes (by maximizing the distance between the support

vectors). An example of such hyper-plane (in this case line since there are only two variables) is provided in Figure 1.2. In the example there are two linearly separable classes, cupcake and muffin. The SVC must classify each one of them based on the amount of sugar and butter used for its baking.

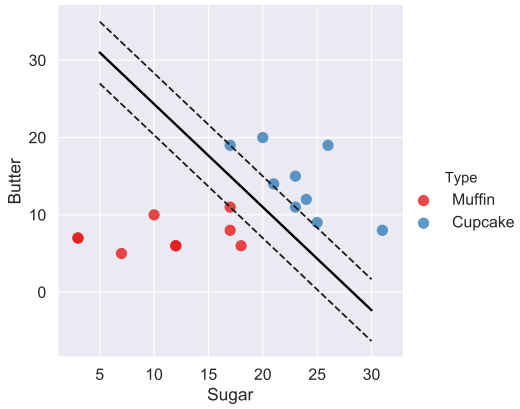
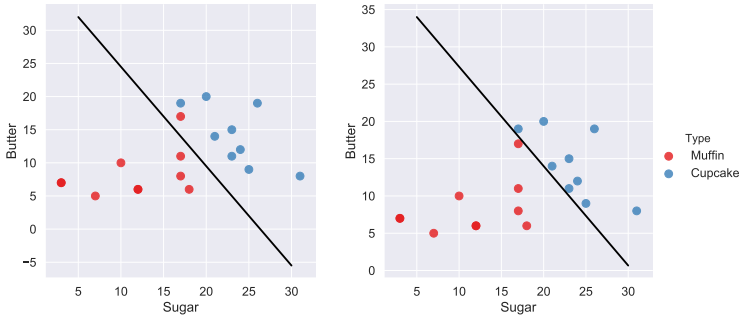


Figure 1.2: Intuition behind a support vector classifier. There are 2 linearly separable classes, blue and red. The hyper-plane is depicted as a black solid line and the confidence margin is depicted as dashed lines.

In SVMs the C hyper-parameter is used to balance the trade-off between a soft margin, i.e. prioritizing a simple model by loosely penalizing the miss-classifications, and a hard margin, which will make the algorithm try harder to minimize miss-classifications but at the same time making the model more prone to over-fitting the data. Figure 1.3 shows the effect of the C parameter on the behavior of a SVC trying to distinguish whether a piece of bread is a muffin or a cupcake based on the amount of sugar and butter used for each.



(a) SVC with $C = 2^{-5}$ (b) SVC with $C = 2^{5}$

Figure 1.3: SVC with different C parameters

As can be observed, when C is small the algorithm finds a boundary with a wider margin of confidence while affording some miss-classifications. On the other hand, if C is big the SVC will try to avoid miss-classifications leading to a very narrow confidence margin which may limit the ability of the model to generalize its results when subject to unseen data.

As the example shows, hyper-parameters play a very important role in the outcome of a machine learning algorithm. Furthermore, some of them are used to trade off between the complexity of the method and the performance, usually a good balance between these two is sought as very complex algorithms tend to require more computational power.

1.2 Algorithm Selection and Optimal Hyper-parameters

The problem of selecting an appropriate algorithm and tune its hyper-parameters can be modeled as an optimization algorithm. The objective can be defined as minimizing the error yielded by a machine learning algorithm on a given dataset. As mentioned in [26] the combined space of learning algorithm and hyper-parameters is very challenging to search: the response function is noisy and the space is high dimensional, involves categorical and continuous choices and contains hierarchical dependencies (e.g. hyper-parameters of the algorithm are only meaningful if that algorithm is chosen). Thus, identifying a high quality model is typically costly (in the sense that entails a lot of computational effort) and time-consuming.

Naive techniques such as grid search [15] and random search [27] result ineffective when dealing with algorithms as complex as neural networks. Gradient based optimization techniques such as line searchers or trust regions [28] are impossible to use, since gradient information is not known and also the selection of the model and its hyper-parameters is often times mixed-integer. Therefore, finding a suitable ML model for a given task requires a method that can handle a high dimensional search space, doesn't rely on gradient information and performs well even with the presence of noisy responses.

1.3 Evolutionary Algorithms for Finding Efficient Machine Learning Models

This thesis focuses on the development of methods that, given a dataset, an inference task (classification or regression) and a performance indicator, efficiently select a machine learning algorithm and tune its hyper-parameters, providing the user a reliable and efficient machine learning model. Since the number of machine learning algorithms available is extensive, this thesis will only focus on the three major neural network types, namely Multi-layer Peceptrons (MLPs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

Given the complexity of the search space we are left with two options for performing the optimization: Bayesian optimization [13, 29] and evolutionary algorithms [17, 30]. Nevertheless, given the robustness of evolutionary algorithms when dealing with noisy functions they are preferred over Bayesian optimization for this thesis.

One of the major disadvantages of evolutionary algorithms is that they usually require a high number of function evaluations in order to find optimal points. This thesis makes a big emphasis on the development of methods that are computationally efficient, therefore using few computations to estimate good ML models. On the other hand we exploit the parallelization properties of evolutionary algorithms to accelerate the optimization process.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 introduces basic concepts of machine learning and gives a brief introduction of deep learning and some of its more common architectural models. Chapter 3 presents a framework for the prognostics of mechanical systems. The framework makes use of genetic algorithms to perform feature extraction and then estimates the remaining useful life of jet engines using deep neural networks. In Chapter 4 we further extend the idea presented in Chapter 3. In Chapter 4 we develop a genetic algorithm for automatically selecting deep learning models and performing some hyper-parameter tuning. Conclusions and future work are presented in Chapter 5.

Chapter 2

MACHINE LEARNING BASICS AND NEURAL NETWORKS

This chapter introduces concepts required for a good understanding of this work. A brief introduction to machine learning is given, including a description of the process of building a machine learning solution (pipeline) and some basic concepts such as model tuning, performance assessment and model selection. Next, a brief description of neural networks is presented. Topics such as the training process of a neural network, neural network hyper-parameters and different neural network architectures are discussed. This chapter concludes with a review of some modern methods used for model selection and hyper-parameter tuning.

2.1 General Concepts of Machine Learning

Data is everywhere! Each of us is not only a consumer but also a generator of data. The gadgets we use everyday generate big amounts of data; the web pages we visit, the things we post in our social media, the places we visit, the things we buy. All of these actions generate valuable information that can say a lot about us. Things like what kind of movies we like, where are we likely to spend our next vacation and even who are we likely to vote for [31]. Machines also generate valuable data. Several machines come with lots of sensors installed to measure a wide variety of things; temperature, pressure, voltage, etc., are some of the many things things that are usually measured in machines. Such data can often give us insights on what the operational status of a machine is, if its has failed or is likely to fail in the near future [6, 32], and even help us apply better control techniques [33].

Machine learning (ML) is the sub-field of artificial intelligence that, by combining optimization, statistics and algorithm design, can find complex patterns (relationships) in data. As opposed to traditional algorithms (such as search or sorting algorithms), where a flow of steps on how to obtain the output from the input is clearly defined, machine learning algorithms do not depend on rules or steps defined by a human expert. Instead, an ML algorithm "learns" this rules by analyzing several examples of input and, in the case of supervised learning, output data. The process of analyzing the data and finding patterns in it is formally known training. As an example consider the famous "Handwritten Digit Recognition" task [34]. In

this task the algorithm is trained on thousands of handwritten digits images like the ones in Figure 2.1, the algorithm is also told which number each image represents. Once trained, the algorithm is given a new batch of images and must predict which is the number written on the image.

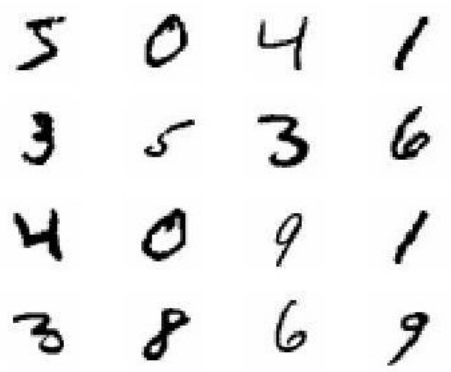


Figure 2.1: Handwritten digits examples extracted from MNIST dataset. Blurriness is part of the original dataset.

2.1.1 Training, Test and Cross-validation Sets

In this section we briefly introduce the sets of data used by machine learning algorithms. Consider a dataset $\mathcal{D}$ made up of points $\mathbf{d}_i = (\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{X} \times \mathbf{Y}$ where $\mathbf{X}$ is the set of data points and $\mathbf{Y}$ is the set of labels. Usually, the dataset $\mathcal{D}$ is split into three datasets. A training set $\mathcal{D}_t$ is used for training the machine learning algorithm while a test set $\mathcal{D}_e$ is used to assess the performance of the model. A third set, called the cross-validation set $\mathcal{D}_v$ is used to tune the hyper-parameters of the machine learning algorithm. It is common practice to split $\mathcal{D}$ into $\mathcal{D}_t/\mathcal{D}_v/\mathcal{D}_p$ using 70%/15%/5% of $\mathcal{D}$ for each subset respectively [35].

2.1.2 Machine Learning Paradigms

Broadly speaking, there are three types of machine learning paradigms. While all of them have in common that they learn patterns from data, the way they learn this patterns (the training process), and the kind of data they can deal with are different. In the following, a brief description of each type is given. This thesis deals only with supervised learning techniques, therefore a more in depth explanation of it is provided in later sections.

Supervised learning

Supervised learning algorithms require that the data consists of a target/outcome variable (or dependent variable) which is to be predicted from a given set of observations (independent variables). Mathematically speaking, let $\mathbf{X} \in \mathbb{R}^{n \times m}$ be the

matrix formed by n observations where each observation is a vector of dimension m and $\mathbf{Y} \in \mathbb{R}^{n \times p}$ be the matrix formed by the labels corresponding to the observations in $\mathbf{X}$. Supervised learning algorithms try to approximate a function $f(\mathbf{X}) \mapsto \mathbf{Y}$. This is usually accomplished by minimizing the error $E(\mathbf{Y}, \hat{\mathbf{Y}})$ between the real target values $\mathbf{Y}$ and the ones predicted by the algorithm $\hat{\mathbf{Y}}$. Examples of supervised learning algorithms include: Linear regression, decision tree, support vector machines, neural networks, etc.

Unsupervised Learning

In unsupervised algorithms only the data of the observations ($\mathbf{X} \in \mathbb{R}^{n \times m}$) is available, there are no target values. Unsupervised algorithms try to find patterns or relationships within this unlabeled data. Instead od responding to feedback, unsupervised learning identifies commonalities in the data and reacts based on the presence or absence of such commonalities in each new piece of data. Examples of unsupervised learning are: apriori algorithm, K-means clustering and generative adversarial networks (GANs).

Reinforcement Learning

Reinforcement learning algorithms deal with agents taking actions in an environment and what actions tend to maximize a cumulative reward. These are closed-loop problems because the learning actions of the system influence its latter inputs [36], that is, the algorithms learns from its previous experience and continues to improve over time. It differs from supervised learning in that labeled input/output pairs need not be present, and sub-optimal actions need not be explicitly corrected. A typical example of reinforcement learning are the Markov decision processes.

2.1.3 Regression and Classification Problems

Regression and classification are two of the most popular kind of problems solved using supervised learning. Typically, a regression/classification problem has labeled data $\mathbf{X}, \mathbf{Y}$ and a measurement of the error $E(\mathbf{Y}, \hat{\mathbf{Y}})$ made by the algorithm when inferring new instances. The task is to learn a mapping from the input $\mathbf{X}$ to the output $\mathbf{Y}$. In this section both kinds of problems are introduced to the reader, required notation and definitions for a good understanding of the rest of the thesis are also presented next.

Regression

Regression aims to find a mapping $f(\mathbf{X}) \mapsto \mathbf{Y}$ where $\mathbf{X} \in \mathbb{R}^{n \times m}$ and $\mathbf{Y} \in \mathbb{R}^{m \times p}$, that is the labels/target values are continuous and often $p = 1$. Usually, a mapping $f(.)$ that exactly maps the points of $\mathbf{X}$ to $\mathbf{Y}$ is impossible to find. Instead,

a curve $\phi(.)$ is fitted to the data. The curve $\phi(.)$ is chosen such that the error $E(\mathbf{Y}, \hat{\mathbf{Y}})$, where $\hat{\mathbf{Y}} = \phi(\mathbf{X})$, is minimized. Consider the following example.

We want to build a system that can predict the price of a house. Inputs are the house attributes: size, zip code, number of bedrooms, median income in the block, etc. The output is the price of the house (in U.S. dollars). Let $\mathbf{X} \in \mathbb{R}^{506\times 8}$ be the data of 506 different houses with 8 attributes each, also let $\mathbf{Y} \in \mathbb{R}^{506}$ be the price for each house. The task is then to predict the price y for an unseen house $\mathbf{x}$. By training a regression model a machine learning algorithm can fit a function $\phi(.)$ to the observed data and then use this function to make predictions for new observations. Figure 2.1 shows a linear model fitted to the house pricing dataset, the plot is a projection of an 8-dimensional space onto the plane formed by the avg. number of rooms in a house and its price.

Figure 2.2: Linear regression for house pricing estimation.

Naturally, a lot of the data can not be fitted using simple linear models as the one shown in 2.2, thus polynomial and even non-linear models are often required. Support vector machines [37, 38], decision trees [39], and neural networks [9, 18] are often used when dealing with regression problems .

Classification

Classification is the problem of identifying to which set of categories an observation belongs to. In classification we aim to to find a mapping $f(\mathbf{X}) \mapsto \mathbf{Y}$ where $\mathbf{X} \in \mathbb{R}^{n\times m}$ and $\mathbf{Y} \in \mathbb{N}^{m\times p}$, when $p > 1$ the problem is called a multi-label classification problem. Opposite to regression, in classification problems the labels/target values are discrete. Instead of fitting a curve $\phi(.)$ to the data as in regression, in

classification problems the curve $\phi(.)$ defines a decision boundary. Once again The curve $\phi(.)$ is chosen such that the error $E(\mathbf{Y}, \hat{\mathbf{Y}})$, where $\hat{\mathbf{Y}} = \phi(\mathbf{X})$ and $\mathbf{X}$ is a set of observations whose class is known, is minimized. As an example take the muffin vs cupcake problem described in Section 1.1.2.

In the example there two are classes, cupcake and muffin. The mapping $\phi(.)$ must classify each one of them based on the amount of sugar and butter used for its baking. For such a simple task a linear classifier is enough. Indeed, both classes are linearly separable (see Figure 2.3). More complex datasets exhibit non-linear decision boundaries. Support vector classifiers [15], artificial neural networks [16,40] and convolutional neural networks [20, 41] are some of the most popular algorithms used in classification.

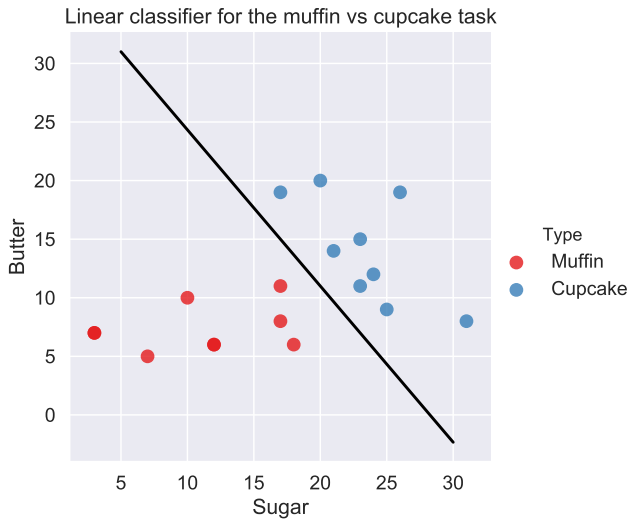


Figure 2.3: Classification of desserts into muffins and cupcakes.

2.1.4 Capacity, Over-fitting and Under-fitting

A key challenge in machine learning is that our model $\phi(.)$ must perform well on new, previously unseen inputs. The ability to perform well on previously unobserved inputs is called **generalization**.

When training a machine learning model we reduce the error E_t of the model $\phi(.)$ on the training set $\mathcal{D}_t$ by means of optimization. What separates machine learning from optimization is that we want the **generalization error** (also known as test error) to be low as well. The generalization error, E_p, is defined as the expected value of the error on unseen data (often referred as the test set D_p). Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

Typically, two assumptions about the data can be made: that the examples in each dataset are independent from each other, and that the training and test sets are identically distributed and drawn from the same probability distribution.

Based on the previous assumptions we can infer that the expected test error is greater than or equal to the expected value of training error. The factors that determine how well a machine learning algorithm performs depend on:

- Its ability to minimize the training error.
- Its ability to make the gap between training error and test error small.

These two factors are closely related with two central challenges in machine learning and statistical inference: **under-fitting** and **over-fitting**. Under-fitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Over-fitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to over-fit or under-fit by altering its **capacity**. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can over-fit by memorizing properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its **hypothesis space**, the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. In the case of neural networks the hypothesis space can be modified by increasing/reducing the number of layers and/or neurons in the network. Doing so modifies the model's capacity.

Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may over-fit.

Figure 2.4 shows this principle in action. We compare a linear, degree-4 and degree-15 predictor attempting to fit a problem where the true underlying function is non-linear. The linear function is unable to capture the curvature in the true underlying problem, so it under-fits. The degree-15 predictor is capable of representing a function that exactly passes through a lot of the sample points, thus over-fitting to the sample data and with very poor generalization performance. We have little chance of choosing a solution that generalizes well when so many wildly different

solutions exist. In this example, the degree-4 model is almost perfectly matched to the true structure of the task so it generalizes well to new data.

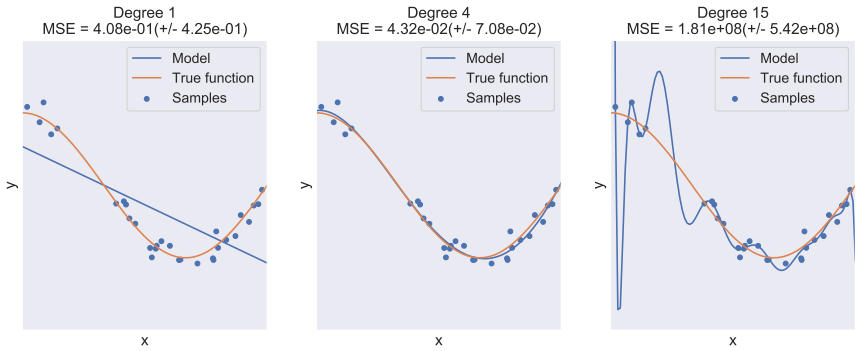


Figure 2.4: Three models fitted to the same data. It is observable that dependent on the model's capacity, it can over-fit, under-fit or adequately fit the data.

In practice, the learning algorithm does not actually find the best function for the data, but merely one that significantly reduces the training error. We must remember that while simpler functions are more likely to generalize (to have a small gap between training and test error) we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotically reaches the minimum possible error value as model capacity increases (assuming the error measure has a minimum value). Often, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in Figure 2.5.

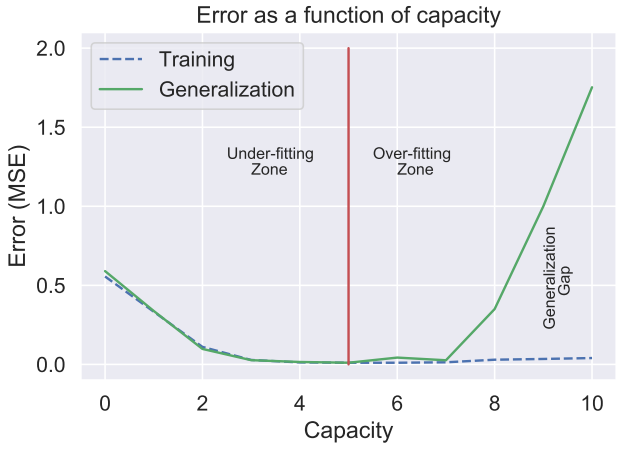


Figure 2.5: Typical relationship between capacity and error.

2.1.5 Regularization

We must design our machine learning algorithms to perform well on a specific task. This is achieved by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.

The behavior of machine learning algorithms is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. For example, linear regression would not perform very well if we tried to use it to predict $sin(x)$ from x. We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution in its hypothesis space to another. This means that both functions are eligible, but one is preferred. The un-preferred solution will be chosen only if it fits the training data significantly better than the preferred solution. One way to do so is by means of **regularization**.

We can regularize a model that learns a function $\phi(\mathbf{x}; \boldsymbol{\theta})$ by adding a penalty called a regularizer to the cost function. Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

As an example consider linear regression with weight decay. Let $\mathbf{X} \in \mathbb{R}^{n \times m}$ be the set of observations and $\mathbf{Y} \in \mathbb{R}^{n \times 1}$ the values corresponding each $\mathbf{x} \in \mathbf{X}$. A linear model $\phi(\mathbf{x}; \boldsymbol{\theta})$ parametrized by $\boldsymbol{\theta}$ is defined as

$$\phi(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{x}_i \boldsymbol{\theta} + \theta_0. \tag{2.1}$$

The cost function of a linear regressor $\phi(\mathbf{x}; \boldsymbol{\theta})$ for a set of parameters $\boldsymbol{\theta}$ is given by

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \sum_{i=0}^{n} \left(\hat{y}_i - y_i \right)^2, \tag{2.2}$$

where $\hat{y}_i = \phi(\mathbf{x_i}; \boldsymbol{\theta})$.

To perform linear regression with weight decay, we minimize a sum comprising both the mean squared error on the training and a criterion $\mathcal{L}(\boldsymbol{\theta})$ that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}, \boldsymbol{\theta}) = \frac{1}{2} \sum_{i=0}^{n} (\hat{y}_i - y_i)^2 + \lambda \boldsymbol{\theta}^T \boldsymbol{\theta}, \tag{2.3}$$

where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller. Minimizing $\mathcal{L}(\boldsymbol{\theta})$ results in a choice of weights that make a trade-off between fitting the training data and being small. This gives us solutions that have a smaller slope, or put weight on fewer of the features. As an example of how we can control a model's tendency to over-fit or under-fit via weight decay, we can train a high-degree polynomial regression model with different values of λ. See Figure 2.6 for the behavior of a polynomial model with different regularization terms α.

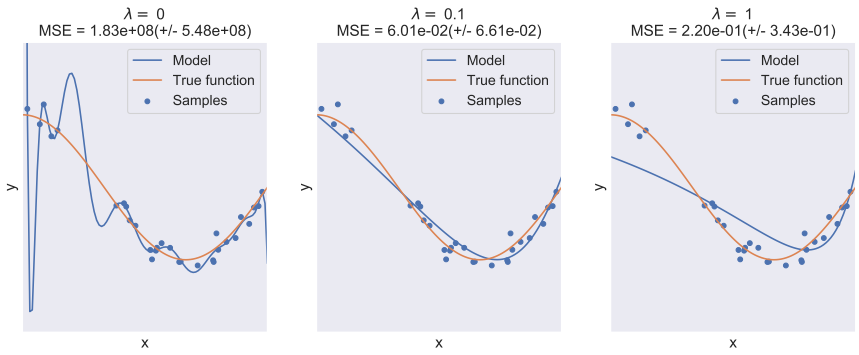


Figure 2.6: Behavior of a polynomial model when using different regularization values.

2.1.6 Hyper-parameters and Validation Set

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called hyper-parameters. The values of hyper-parameters are not adapted by the learning algorithm itself. An example of a hyper-parameter is the regularization term λ used in Equation (2.3), other examples include the number of neurons of a neural network.

Sometimes a setting is chosen to be a hyper-parameter that the learning algorithm does not learn because it is difficult to optimize. More frequently, the setting must be a hyper-parameter because it is not appropriate to learn that hyper-parameter on the training set. This applies to all hyper-parameters that control model capacity. If learned on the training set, such hyper-parameters would always choose the maximum possible model capacity, resulting in over-fitting. To solve this problem, we need a **validation set** $\mathcal{D}_v$ of examples that the training algorithm does not observe.

Earlier we discussed how a held-out test set $\mathcal{D}_v$, composed of examples coming from the same distribution as the training set $\mathcal{D}_t$, can be used to estimate the generalization error of a learner, after the learning process has completed. It is important that the test examples are not used in any way to make choices about the model, including its hyper-parameters [18]. For this reason, no example from the test set can be used in the validation set. Therefore, we always construct the validation set $\mathcal{D}_v$ from the training data. Specifically, we split the training data into two disjoint subsets $\mathcal{D}_t$ and $\mathcal{D}_v$. One of these subsets is used to learn the parameters. The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyper-parameters to be updated accordingly.

Dividing the dataset into a fixed training set and a fixed test set can be problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm $\phi_1(.)$ works better than algorithm $\phi_2(.)$ on the given task. When the dataset is too small, there are alternative procedures enable one to use all of the examples in the estimation of the mean test error, at the price of increased computational cost. The most common of these is the k-fold cross-validation procedure in which a partition of the dataset is formed by splitting it into k non-overlapping subsets. The test error may then be estimated by taking the average test error across k trials. On trial i, the i-th subset of the data is used as the test set and the rest of the data is used as the training set.

2.2 Neural Networks

In this section we introduce the basics of neural networks. A brief description of the logic and ideas behind them is given. Furthermore, we present different types of neural networks such as convolutional neural networks and recurrent neural networks.

2.2.1 Artificial Neurons

In general, artificial neural networks (ANNs) are systems vaguely inspired by the neurological networks in the brain [17, 42]. **Units** (also referred as artificial neurons) are the building blocks of any type of ANN. The unit (neuron), implements a nonlinear mapping $a : \mathbb{R}^n \to [c, d]$, where n is the number of inputs the unit receives and c and d depend on the chosen activation function. Usual combinations for c and d are $[0, 1]$ or $[-1, 1]$.

A unit receives a vector of n input signals, $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, either from the environment (input variables) or from other units and applies an activation function g to the net input signal to compute the output signal, a. The strength of the output signal is influenced by a set of weights θ_i and a bias value b. Mathematically speaking a unit performs the following operation

$$a = g(\sum_{i=1}^{n} \theta_i x_i + b). \tag{2.4}$$

Figure 2.7 presents an illustration of an artificial neuron. As can be observed, the net input signal to a unit is computed as the weighted sum of all input signals.

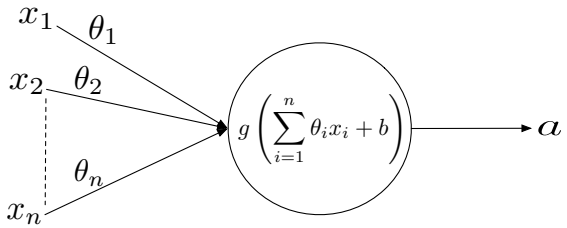


Figure 2.7: An artificial neuron

The function $g(.)$ receives the net input signal and bias, and determines the output of the neuron. This function is referred as the *activation function*. Different types of activation functions can be used [17], among the most popular ones we find the sigmoid function, tanh function and the Rectified Linear Unit (ReLU) function. The values of the weights θ_i and the bias b are adjusted through an optimization process called training [15, 17]. In supervised learning the neuron is provided with a dataset consisting of input vectors and a target (desired output) associated with each input vector. This data is referred as training set ($\mathcal{D}_t$). The aim is then to adjust the weight values and bias such that the error between the predicted output of the neuron, $a = g(\mathbf{x}; \boldsymbol{\theta})$, and the target output, $\mathbf{y} = f(\mathbf{x})$, is minimized.

Note that this approach is very similar to that of linear regression except for the application of the activation function g. Indeed, if g is the identity function, the neuron acts as a linear regressor, nevertheless most of the inference power of neural networks comes from the use of non-linear activation functions such as the ones described in [15, 17, 18]

2.2.2 Multi-layer Perceptron

Widely used, multi-layer perceptrons (MLPs) also known as feed-forward neural networks (FFNN), are the quintessential deep learning models [18]. The goal of a feed-forward network is to approximate some function $f(.)$. For example, for a classifier, $\mathbf{y} = f(\mathbf{x})$ maps an input $\mathbf{x}$ to a category $\mathbf{y}$. A feed-forward network defines a mapping $\hat{\mathbf{y}} = \phi(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

MLPs are formed by stacking together layers of artificial neurons. A layer is a set of units whose input is the same and whose output can be used as the input for upcoming layers. As such, MLPs compose together many different functions.

The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $\phi^{(1)}, \phi^{(2)}, \phi^{(3)}$ connected in a chain, to form $\phi(\mathbf{x}) = \phi^{(3)}(\phi^{(2)}(\phi^{(1)}(x)))$. These chain structures are the mathematical representation of the layers of a neural network. In this case, $\phi^{(1)}$ is called the **first layer** of the network, $\phi^{(2)}$ is called the **second layer** of the network and so on. The overall length of the chain gives the **depth** of the model. The final layer of a neural network is called the **output layer**, the layers between the output layer and the input layer are called **hidden layers**.

Each hidden layer in the network is typically vector-valued. The dimensionality of these layers determines the **width** of the model. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function $\phi^{(i)}$ where i denotes the layer number. Each unit resembles a neuron in the sense that it receives input from many others units and computes its own activation value. Figure 2.8 depicts a MLP with one hidden layer. Each unit of a layer is connected to every other unit of the next layer.

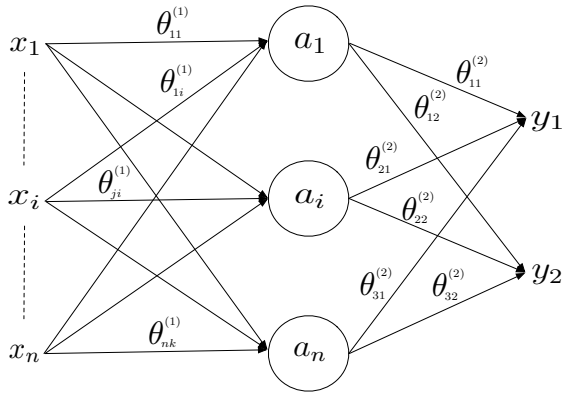


Figure 2.8: A MLP with input size of x_n and output size of 2. The hidden layer is made up of 3 units. $\boldsymbol{\theta}^{(1)}_{ji}$ and $\boldsymbol{\theta}^{(2)}_{ji}$ are the trainable parameters, $\boldsymbol{\theta}$, of the neural network $\phi(\mathbf{x}; \boldsymbol{\theta})$.

Activation Functions

Key to the concept of artificial neural networks are the activation functions. The activation function of a unit defines the output of such unit given a set of inputs (Equation (2.4)) by mapping the net sum of the unit into a range $[c, d]$ (depending upon the choice of the function). Being inspired by the action potential firing in a neuron [43], activation functions operate under the following principle: the stronger the input signal is, the stronger the output signal. Figure 2.9 plots some of the most common activation functions. Some desirable properties of an activation function are:

- Non-linear: A two-layer neural network can be proven to be a universal function approximator [44].
- Range: Gradient-based training methods tend to be more stable.
- Continuously differentiable: Enables gradient-based optimization methods.
- Monotonic: Error surface associated with a single-layer model is convex.
- Smooth functions with a monotonic derivative: Better generalization.
- Approximates identity near the origin: The neural network will learn efficiently when its weights are initialized with small random values.

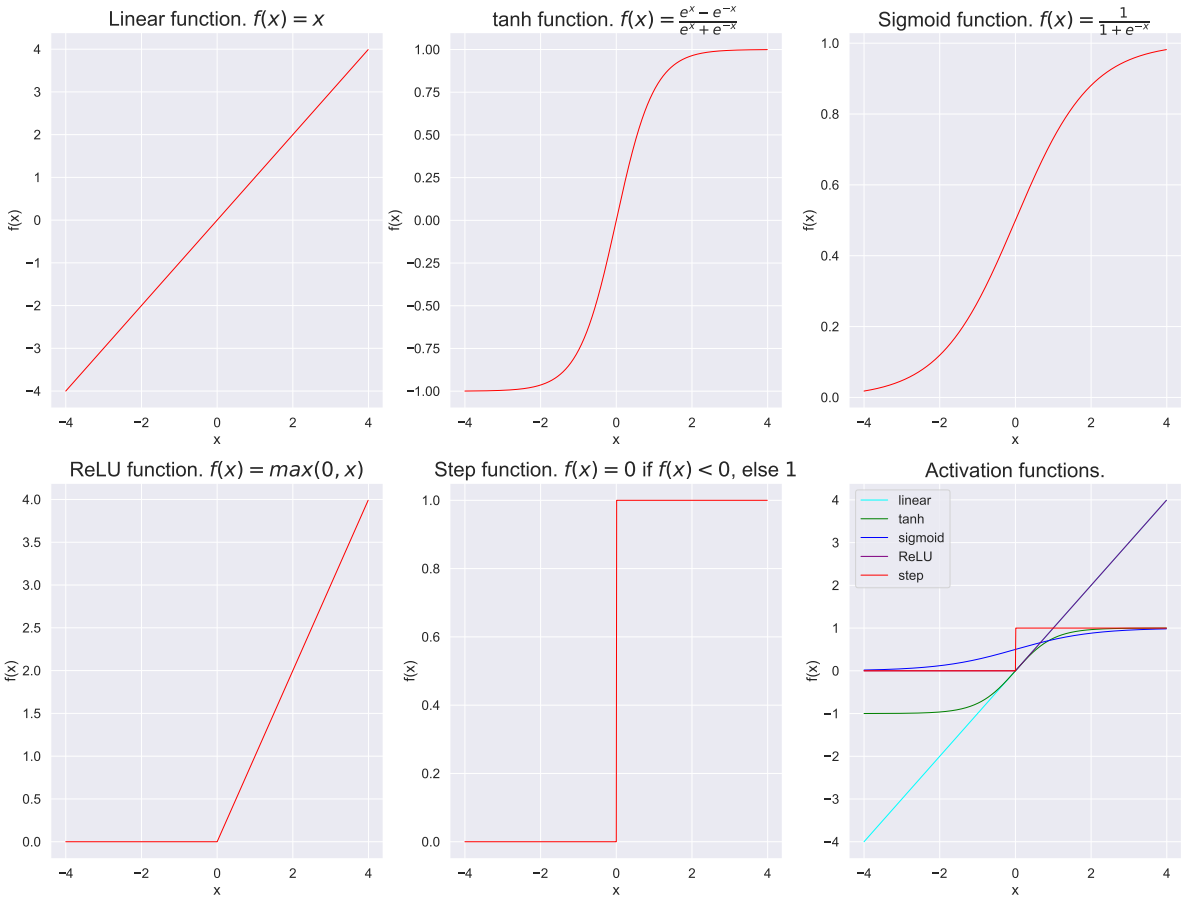


Figure 2.9: Common activation functions used in deep learning

2.2.3 Training the Neural Network

Before making accurate predictions, the MLP needs to be trained. The training process is the process of adjusting the set of parameters $\boldsymbol{\theta}$ of the neural network so that it minimizes the error between the predicted values and the real values. The training process uses a split of the original dataset $\mathcal{D}$ called the training set $\mathcal{D}_t$ (see Section 2.1.1). In order to perform training two things are needed: a loss function, that measures the error between the predicted and target values, $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$ and an optimization algorithm.

Loss Function

The measurement of the error depends on the type of problem at hand. For simplicity we will assume that $y \in \mathbb{R}$ in the case of regression and $y \in [, 1]$. That is regression problems are single-objective while for classification we only consider binary classification. Extended formulas for multi-variate regression and multi-class classification can be found in [45] and [15] respectively. For regression problems a usual measurement of the error is the mean squared error defined as:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2n} \sum_{i=1}^{n} \left(y_i - \hat{y_i}\right)^2, \tag{2.5}$$

Equation (2.5) measures the euclidean distance between the target values $\mathbf{y}_i$ and he predicted values $\hat{\mathbf{y}}_i = \phi(.)$. By minimizing the cost $\mathcal{L}$, we are effectively fitting the curve $\phi(.)$ to the data $\mathbf{X}$.

For classification problems, the cross-entropy function [15] is used. Binary cross-entropy loss function is defined as follows:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2n} \sum_{i=1}^{n} - \left(y_i \log \hat{y}_i + (1 - y_i) \log 1 - \hat{y}_i\right). \tag{2.6}$$

Given two classes C_1 and C_2. Equation (2.6) measures the performance of the model whose output is a probability value between 0 and 1. Cross-enthropy decreases as the predicted probability diverges from the actual label. So, for instance, predicting a probability $p(\mathbf{x}|C_1) = 0.12$ when $\mathbf{x}$ belongs to class C_1 would yield a high error. This behavior is illustrated in Figure 2.10 note how as the probability approaches 1, for a true label of 1, the cost (cross-enthropy) decreases rapidly. Cross-enthropy penalizes both types of errors, but specially those that are confident and wrong.

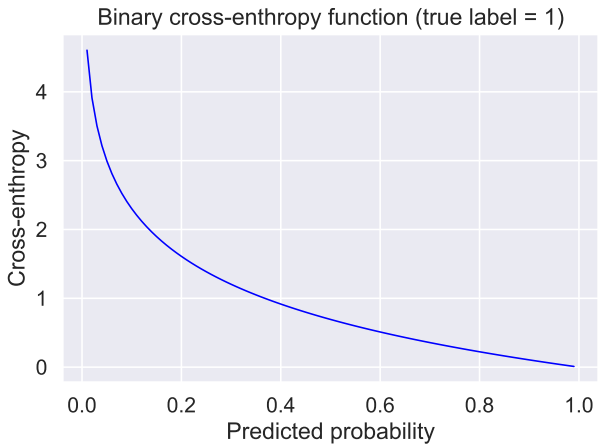


Figure 2.10: Binary cross-enthropy function.

Minimizing the Loss Function

The aim of the training process is to minimize the cost function $\mathcal{L}$. Hence, maximizing the similarity between the predicted and target values $\hat{\mathbf{y}}$ and $\mathbf{y}$. Since neural networks are often non-linear models, usually the cost function becomes non-convex [18]. Therefore, neural networks are usually trained using iterative, gradient-based optimizers [28] that drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.

The general form of any gradient-descent based algorithm is the following: Let $\mathbf{x} \in \mathbb{R}^n$ be a vector of parameters and $f : \mathbb{R}^n \mapsto \mathbb{R}$ a smooth function. To minimize f w.r.t. $\mathbf{x}$, a line searcher method computes, iteratively, a search direction $\mathbf{p}_k$ and then decides how far to move along that direction. The iteration is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \eta_k \mathbf{p}_k, \tag{2.7}$$

where the positive scalar η_k is called the step length. The success of a line search method depends on the effective choice of both the direction $\mathbf{p}_k$ and the step length η_k (in the machine learning literature η_k is often referred as the learning rate). In general $\mathbf{p}_k$ is chosen such that it is a descent direction, i.e. $\mathbf{p}_k^T \nabla f < 0$. The most popular descent direction for machine learning applications is gradient descent $\mathbf{p}_k = -\nabla f_k$. Almost every optimization method for neural network training uses gradient descent as descent direction. Popular optimization methods used for neural network training are: Adam, RMSProp, Momentum and Nesterov. A good survey of all of these methods can be found in [46].

Based on the amount of data used for every iteration gradient-descent methods can be further classified into: **Batch gradient descent**, which computes the

gradient of the cost function, w.r.t. $\boldsymbol{\theta}$, using the entire training set $\mathcal{D}_t$. **Stochastic gradient descent**, which performs a parameter update ($\boldsymbol{\theta}$) for each training example $\mathbf{x}_i, \mathbf{y}_i \in \mathcal{D}_t$. And **mini-batch gradient descent**, that performs a parameter update for every mini batch of n training samples. Stochastic and mini-batch gradient descent are the most widely used methods, since they speed up the training process and allow for online training.

Finally, opposite to convex optimization solvers, which converge from any initial parameter, gradient-descent methods applied to non-convex loss functions have no such convergence guarantee. For neural networks, it is important to initialize all weights, $\boldsymbol{\theta}$, to small random values. The biases, $\mathbf{b}$, may be initialized to zero or to small positive values [18].

The Backpropagation Algorithm

As mentioned in the last section, in order to minimize the cost function $\mathcal{L}$ the gradient of the function $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$ is needed. When we use a feedforward neural network to map an input $\mathbf{x}$ into an output $\hat{\mathbf{Y}}$, information **flows forward** through the network. The inputs $\mathbf{x}$ provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\mathbf{Y}}$. This is called **forward propagation**. During training, forward propagation continues onward until it produces a scalar cost $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$. The back-propagation algorithm [47], often simply called backprop, allows the information from the cost to flow backwards through the network, in order to compute the gradient.

Computing an analytic expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function.

In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$. The idea of computing derivatives by propagating information through a network is very general, and can be used to compute values such as the Jacobian of a function f with multiple outputs.

The backpropagation algorithm is a complex algorithm to explain in a couple of pages, and thus is out of the scope of this thesis. Good references for the backpropagation algorithm are [15, 18, 47].

The Training Process

We know have all the blocks to describe the training process of a neural network. This training process is generic and applies for any neural network (multi-layer perceptrons, convolutional neural networks, recurrent neural networks, etc.). Let $\phi(\mathbf{X};\boldsymbol{\theta}) \mapsto \mathbf{Y}$ be a mapping from input $\mathbf{X} \in \mathbb{R}^{n \times m}$ to a target value $\mathbf{Y} \in \mathbb{R}^{n \times k}$ parametrized by $\boldsymbol{\theta}$. Given a training set $\mathcal{D}_t$ of labeled data $d_i = (\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{X} \times \mathbf{Y}$ and a loss function $\mathcal{L}$, the training process of $\phi(.)$ is described in Algorithm 1.

Algorithm 1: Neural network training algorithm.

Data: A training set $\mathcal{D}_t$.
Input : A neural network model $\phi(\mathbf{X};\boldsymbol{\theta})$, a loss function $\mathcal{L}(\boldsymbol{\theta})$, learning rate η, batch size b and training epochs t.
Output: Trained neural network model $\phi(\mathbf{X};\boldsymbol{\theta})$.

Randomly initialize the parameters $\boldsymbol{\theta}$. Set $k = 0$.
for *epoch* $= 0 : t$ **do**
  Randomly split the training set $\mathcal{D}_t$ into b batches $\mathcal{B}$.
  for *Each mini-batch* $\mathcal{B}_i$. **do**
    Compute $\hat{\mathbf{Y}} = \phi(\mathbf{X};\boldsymbol{\theta})$ by doing a forward pass on $\phi(.)$ with batch $\mathcal{B}$.
    Compute the loss $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$.
    Compute $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$ using backprop.
    Update the parameters of the model using any gradient descent method for instance: $\boldsymbol{\theta_{k+1}} = \boldsymbol{\theta_k} - \eta_k \nabla_{\boldsymbol{\theta}}\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$.
  end
end

2.2.4 Convolutional Neural Networks

Convolutional networks [48], also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name convolutional neural network indicates that the network employs a mathematical operation called convolution, which is a kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Convolution

In general, convolutional layers operate over tensors, this in contrast with MLPs where the input is a vector. A standard convolutional layer takes as input a $D_F \times D_F \times M$ tensor $\mathbf{F}$ and produces a $D_G \times D_G \times N$ tensor $\mathbf{G}$ where D_F is the spatial width and height of a square input tensor, M is the number of input channels (input depth), D_G is the spatial width and height of the a square output tensor, and N is the number of output channels (output depth). Figure 2.11 illustrates a tensor of dimensions $32 \times 32 \times 3$.

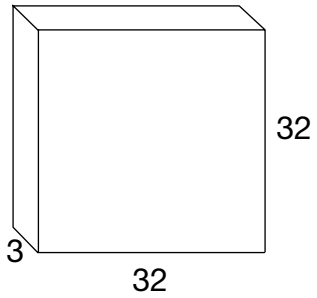


Figure 2.11: A tensor like the ones used by CNNs

The standard convolutional layer is parametrized by a convolution kernel $\mathbf{K}$ of size $D_K \times D_K \times M \times N$ where D_K is the spatial dimension of the kernel assumed to be square, M is the number of input channels and N is the number of output channels as previously defined.

In a convolution operation the input map $\mathbf{F}$ is convoluted with the filter $\mathbf{K}$ using a stride S and padding P. The output map's, G, width and height follow the formula:

$$D_G = \frac{\lfloor D_F - D_K + 2P \rfloor}{S} + 1. \tag{2.8}$$

Figure 2.12 gives a visual interpretation of the size of the output tensor after convoluting the input tensor with the kernel. In the example $D_F = 32$, $D_K = 5$, $S = 1$, and $P = 0$. By formula 2.8 $D_G = 28$. Since only one filter is used, the output is a tensor of rank 2 (a matrix instead of a volume).

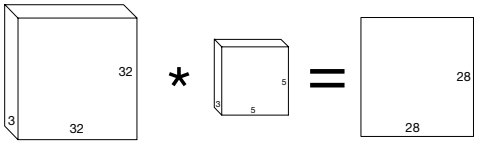


Figure 2.12: Convoluting the input volume D_F with a filter K gives a results an output volume D_G example.

The output tensor for standard convolution assuming stride one and valid padding is computed as:

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} * F_{k+1-1,l+j-1,m}. \tag{2.9}$$

Figure 2.13 gives a visual interpretation of the convolution operation. The filter moves along the input volume with stride of S and applies the convolution operation.

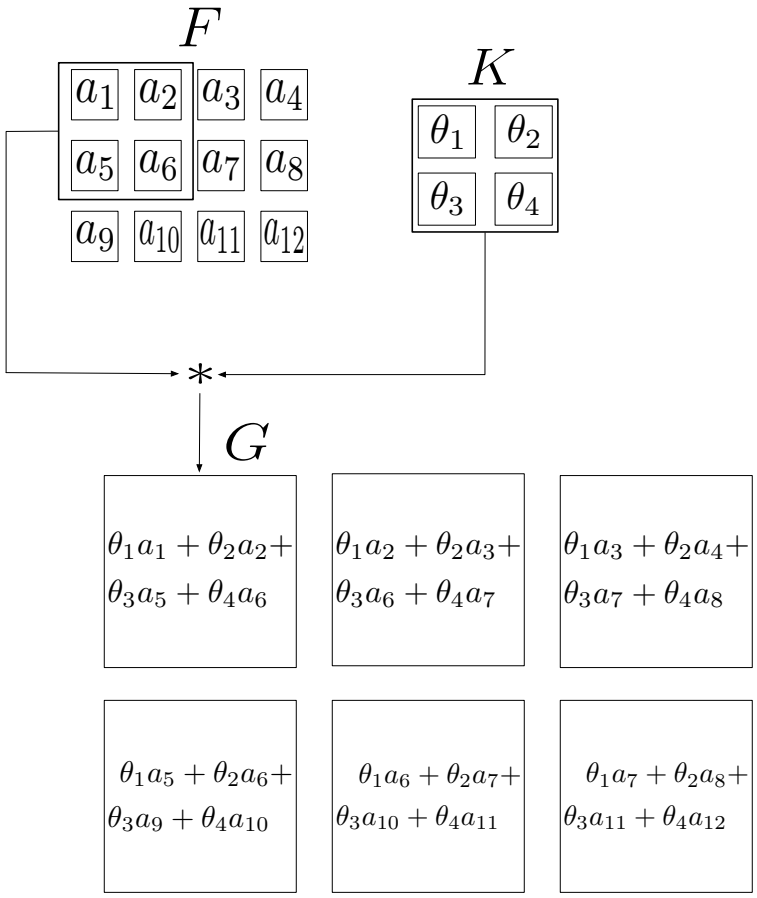


Figure 2.13: An example of 2-D convolution. In this case we restrict the output to only positions where the kernel lies entirely within the image, called valid convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

Finally, a bias b is added to each of the units in the output tensor, then a nonlinear activation function is applied to the output volume. Each entry of the output tensor is computed as:

$$G_{k,l,n} = g\left(\sum_{i,j,m}\left(K_{i,j,m,n} * F_{k+1-1,l+j-1,m}\right) + b\right). \tag{2.10}$$

Convolution leverages two important ideas that can help improve a machine learning system: sparse interactions and parameter sharing.

Traditional neural network units interact with every other unit in the next layer (by means of a matrix multiplication). Convolutional networks, however, typically have **sparse connections**. This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which in turn improves overall efficiency of CNNs over MLPs for some applications [18].

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary units). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the run-time of forward propagation but further reduces the storage requirements of the model.

Pooling

Pooling is one of the steps done in a convolutional layer. A pooling function replaces the output a convolutional layer with a summary statistic of the nearby outputs. For example, the **max pooling** [49] operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel [18].

There are two major types of pooling. Let's say we have a 4x4 tensor representing our initial input. Let's say, as well, that we have a 2x2 filter that we'll run over our input. We'll have a stride of 2 and won't overlap regions. **Max pooling** is done by applying a max filter to (usually) non-overlapping sub-regions of the initial representation. **Average pooling**, on the other hand, computes the average of the sub-region selected by the filter.

In max pooling, for each of the units in the output tensor is computed by taking the maximum element of the region of the input tensor selected by the kernel. For average pooling, each of the units of the output tensor is computed by taking the average of the region represented by the filter. Figure 2.14 depicts a graphical representation of max an average pooling.

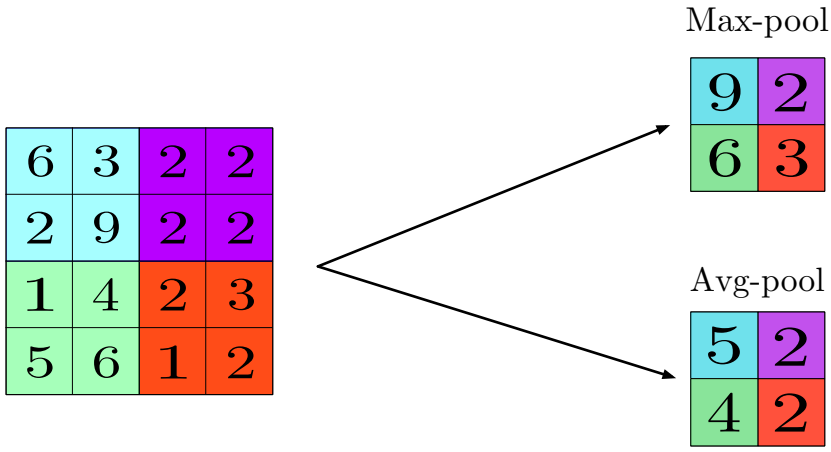


Figure 2.14: Max and average pooling.

In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to.

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k units apart rather than 1 unit apart. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process.

Building Convolutional Neural Networks

A typical convolutional layer consists of three stages (see Figure 2.15). In the first stage, the layer performs several convolutions to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear

activation function. This stage is sometimes called the detector stage. In the third stage, we use a pooling function to modify the output of the layer further.

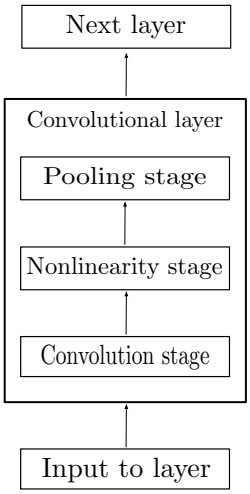


Figure 2.15: A convolutional layer composed of three "stages".

Convolutional neural networks (CNNs) are formed by stacking several convolutional layers together followed by a number of fully connected layers (See Section 2.2.2.). The convolutional layers act as feature extractor while the fully connected layers at the end act as the classifier/regressor. Consider the task of image classification. The convolutional and pooling layers will extract features from the pixels. For instance, the first layers may learn to identify simple features like edges, learning more complex shapes like eyes or faces in the last convolutonal layers. The fully connected layers will then take the learned features by the convolutional layers and learn the function to classify the different images. The training process for a CNN is analogous to that for the feedforward networks. Figure 2.16 shows a depiction of a CNN used for image classification.

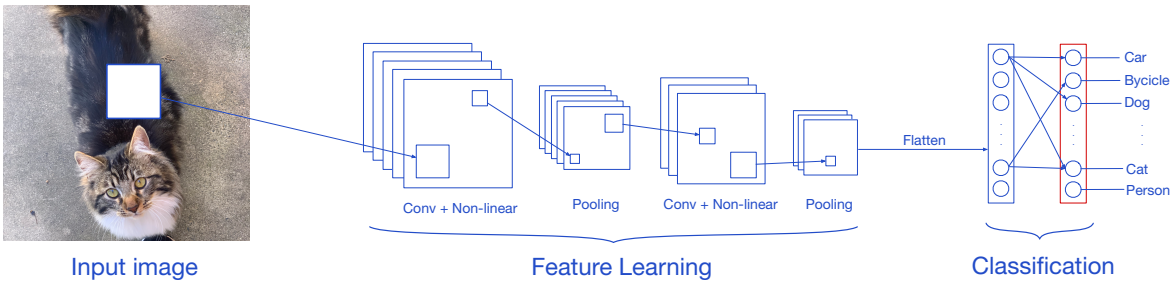


Figure 2.16: A convolutional neural network with two convolutional layers and two fully connected layers for image detection.

2.2.5 Recurrent Neural Networks

Recurrent neural networks or RNNs [50] are a family of neural networks for processing sequential data, that is, a sequence of values $\mathbf{x}^{<1>}, \ldots, \mathbf{x}^{<\tau>}$. Unlike standard feedforward networks, RNNs retain a state that can represent information from an arbitrarily long context window. Furthermore, they can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

Despite their power, standard neural networks have limitations when dealing with sequenced data. Most notably, they rely on the assumption of independence among the data samples. After each sample (data point) is processed, the entire state of the network is lost. If each example is generated independently, this presents no problem. But if data points are related in time, this represents a big problem. Frames from video, words in sentences and sensor information from different machines represent settings where the assumption of independence of the data fails.

The input to an RNN is a sequence. An input sequence can be denoted as $(\mathbf{x}^{<1>}, \mathbf{x}^{<2>}, \ldots, \mathbf{x}^{<\tau>})$ where each data point $\mathbf{x}^{<\tau>}$ is a real valued vector. Similarly, a target sequence can be denoted $(\mathbf{y}^{<1>}, \mathbf{y}^{<2>}, \ldots, \mathbf{y}^{<\tau>})$. Sequences may be of finite or countably infinite length. When they are finite, the maximum time index of the sequence is called τ.

Using temporal terminology, an input sequence consists of data points $\mathbf{x}^{<\tau>}$ that arrive in a discrete sequence of time steps indexed by τ. A target sequence consists of data points $\mathbf{y}^{<\tau>}$. We use superscripts with parentheses for time, and not subscripts, to prevent confusion between sequence steps and indices of nodes in a network. When a model produces predicted data points, these are labeled $\hat{\mathbf{y}}^{<\tau>}$.

The time-indexed data points may be equally spaced samples from a concontinuous real-world process. Examples include the still images that comprise the frames of videos or the discrete amplitudes sampled at fixed intervals that comprise audio recordings. The time steps may also be ordinal, with no exact correspondence to duration. In fact, RNNs are frequently applied to domains where sequences have a defined order but no explicit notion of time. This is the case with natural language. In the word sequence "David Laredo plays the guitar, $\mathbf{x}^{<1>} = \textit{David}$, $\mathbf{x}^{<2>} = \textit{Laredo}$, etc.

An Extension of Feedforward Networks

Recurrent neural networks are feedforward neural networks augmented by the inclusion of edges that span adjacent time steps, introducing a notion of time to the model. A recurrent neural network can be thought of as multiple copies of the same feedforward network, each passing a message to a successor (See Figure 2.17).

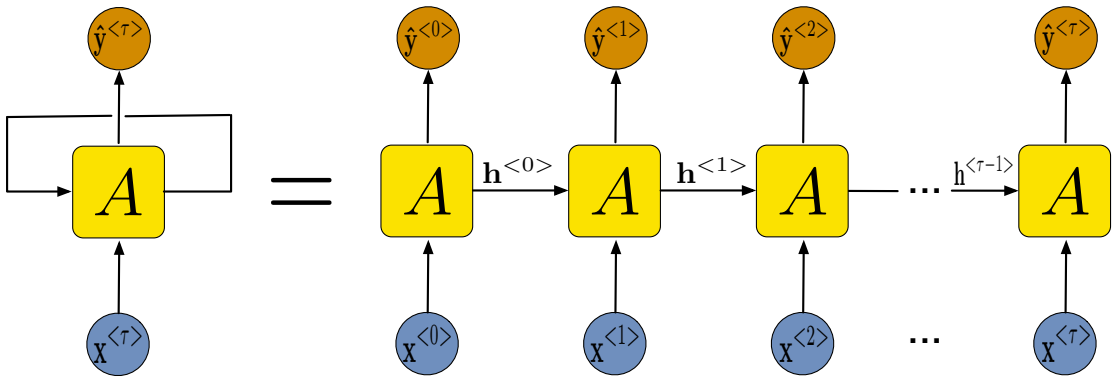


Figure 2.17: An unrolled recurrent neural network. At each time step τ, activation is passed along edges as in a feedforward network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

At time τ, nodes with recurrent edges receive input from the current data point $\mathbf{x}^{<\tau>}$ and also from hidden node values $\mathbf{h}^{<\tau-1>}$ in the network's previous state. The output $\hat{\mathbf{y}}^{<\tau>}$ at each time τ is calculated given the hidden node values $\mathbf{h}^{<\tau>}$ at time τ. Input $\mathbf{x}^{<\tau-1>}$ at time $\tau - 1$ can influence the output $\hat{\mathbf{y}}^{<\tau>}$ at time τ and later by means of the recurrent connections.

Two equations specify all calculations necessary for computation at each time step on the forward pass in a simple recurrent neural network as in Figure 2.17:

$$\mathbf{h}^{<\tau>} = g_h\left(\boldsymbol{\theta}_x\mathbf{x}^{<\tau>} + \boldsymbol{\theta}_h\mathbf{h}^{<\tau-1>} + \mathbf{b}_h\right) \tag{2.11}$$

$$\hat{\mathbf{y}}^{<\tau>} = g_y\left(\boldsymbol{\theta}_y\mathbf{h}^{<\tau>} + \mathbf{b}_y\right), \tag{2.12}$$

where $\boldsymbol{\theta}_x$ is the conventional matrix between the input and the hidden layer, $\boldsymbol{\theta}_h$ is the matrix of recurrent weights between the current layer and itself at adjacent steps and $\boldsymbol{\theta}_y$ is the matrix between the hidden layer and the output layer. The vectors $\mathbf{b}_h$ and $\mathbf{b}_y$ are bias parameters that allow each node to learn an offset. Finally, g_h and g_y are nonlinear functions. Usually g_h is the *tanh* function while g_y can either be the *sigmoid* (or $softmax$) function for classification or the linear function for regression.

The dynamics of the network depicted in Figure 2.17 across time steps can be visualized as a deep network with one layer per time step and shared weights across time steps. It is then clear that the unfolded network can be trained across many time steps using backpropagation. The algorithm called backpropagation through time (BPTT) [51] is used by all the recurrent neural networks to optimize its parameters $\boldsymbol{\theta}_x$, $\boldsymbol{\theta}_h$ and $\boldsymbol{\theta}_y$.

The architecture presented here is known as the basic RNN cell (unit), an illustration of it can be seen in Figure 2.18. In the diagram, each line carries an entire vector, from the output of one node to the inputs of others. The yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations. Stacking together several of this units gives rise to deep RNN models.

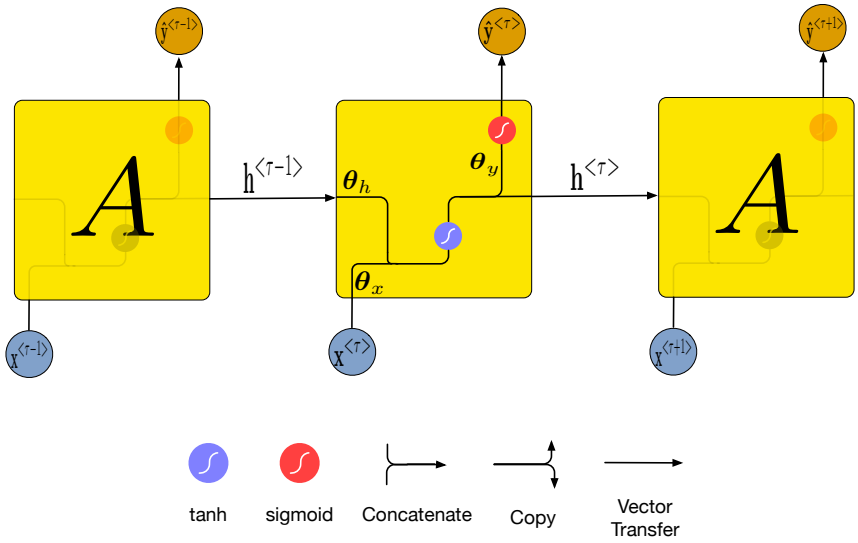


Figure 2.18: A basic RNN cell. This basic structure is repeated for every time step τ. The parameters $\boldsymbol{\theta}$ are shared across the network. σ stands for the sigmoid function.

Modern RNN Architectures

One major disadvantage of basic RNN cells is that they do not handle long-term time dependencies well. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the .." we don't need any further context, it's pretty obvious the next word is going to be "sky". In such cases, where the gap between the relevant information and the place that it's needed is small, basic RNN cells can learn to use the past information.

There are nevertheless cases where we need more context. Consider trying to predict the last word in the text I grew up in France I speak fluent French. Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Long short term memory (LSTM) [52] and Gated recurrent unit (GRU) [53] are RNN cell types that can handle very large time dependencies. The structure of both cells is similar to that of the basic RNN but they incorporate a new set of weights $\boldsymbol{\theta}_c$ that are related to a so-called "cell state" $\mathbf{c}^{<\tau-1>}$. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. LSTMs and GRUs also have have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point-wise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero lets nothing through the gate while a value of one lets everything flow through it.

LSTM and GRU cells mainly differ in the number of gates used to update the cell state. GRU cells introduce two gates to the basic RNN cell. An update gate controls how much of the current information to pass to the next cell state while at the same time controlling how much of the previous information to preserve. LSTM introduces two additional gates a separate forget gate to control how much of the previous information to preserve in the cell state and an output gate that controls what to output ($\hat{\mathbf{y}}^{<\tau>}$) based on the cell state. Figure 2.19 shows the diagrams of the three RNN cells.

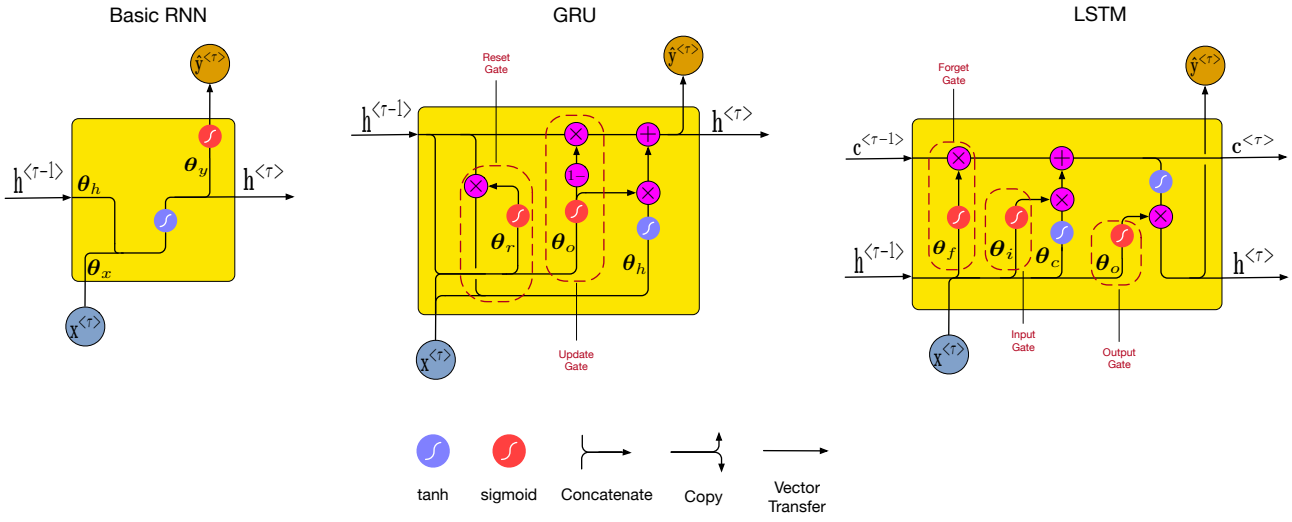


Figure 2.19: A comparison between basic RNN, GRU and LSTM cells.

The steps for computing the output $\hat{\mathbf{y}}^{<\tau>}$, hidden value $\mathbf{h}^{<\tau>}$ and cell state $\mathbf{c}^{<\tau>}$ for LSTM and GRU cells are shown in Equations (2.14) and (2.15) respectively.

$$
\begin{aligned}
\mathbf{f}^{<\tau>} &= \sigma\left(\boldsymbol{\theta}_f \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_f\right) \\
\mathbf{i}^{<\tau>} &= \sigma\left(\boldsymbol{\theta}_i \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_i\right) \\
\mathbf{o}^{<\tau>} &= \sigma\left(\boldsymbol{\theta}_o \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_o\right) \\
\tilde{\mathbf{c}}^{<\tau>} &= tanh\left(\boldsymbol{\theta}_c \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_c\right) \\
\mathbf{c}^{<\tau>} &= \mathbf{f}^{<\tau>} * \mathbf{c}^{<\tau-1>} + \mathbf{i}^{<\tau>} * \tilde{\mathbf{c}}^{<\tau>} \\
\mathbf{h}^{<\tau>} &= \mathbf{o}^{<\tau>} * tanh(\mathbf{c}^{<\tau>}).
\end{aligned}
\tag{2.13}
$$

Here $\mathbf{i}^{<\tau>}$, $\mathbf{f}^{<\tau>}$, and $\mathbf{o}^{<\tau>}$ are called input, forget and output gates, respectively. The input gate defines how much of the newly computed state for the current input you want to let through. The forget gate defines how much of the previous state you want to let through. Finally, the output gate defines how much of the internal state you want to expose to the external network. All the gates have the same equations except that they use different parameter matrices ($\boldsymbol{\theta}_f$, $\boldsymbol{\theta}_i$, $\boldsymbol{\theta}_o$). The gates also have the same dimension d_h, which is the size of the hidden state.

$\tilde{\mathbf{c}}^{<\tau>}$ is a candidate hidden state that is computed based on the current input and the previous hidden state. $\mathbf{c}^{<\tau>}$ is the internal memory of the unit. It is a combination of the previous memory, multiplied by the forget gate, and the newly computed hidden state, multiplied by the input gate. Thus, intuitively it is a combination of how we want to combine previous memory and the new input. $\mathbf{h}^{<\tau>}$ the is output hidden state, computed by multiplying the memory with the output gate. Not all of the internal memory may be relevant to the hidden state used by other units in the network.

Intuitively, basic RNNs could be considered a special case of LSTMs. If fix the input gate all 1's, the forget gate to all 0's (say, always forget the previous memory) and the output gate to all 1's (say, expose the whole memory), it will almost behave as a basic RNN.

$$
\begin{aligned}
\mathbf{o}^{<\tau>} &= \sigma\left(\boldsymbol{\theta}_o \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_o\right) \\
\mathbf{r}^{<\tau>} &= \sigma\left(\boldsymbol{\theta}_r \cdot \left[\mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_r\right) \\
\tilde{\mathbf{h}}^{<\tau>} &= tanh\left(\boldsymbol{\theta}_h \cdot \left[\mathbf{r}^{<\tau>} * \mathbf{h}^{<\tau-1>}, \mathbf{x}^{<\tau>}\right] + \mathbf{b}_h\right) \\
\mathbf{h}^{<\tau>} &= (1 - \mathbf{o}^{<\tau>}) * \mathbf{h}^{<\tau-1>} + \mathbf{o}^{<\tau>} * \tilde{\mathbf{h}}^{<\tau>}.
\end{aligned}
\tag{2.14}
$$

For GRU cell $\mathbf{h}^{<\tau>}$ is the hidden state of the cell. Here $\mathbf{r}^{<\tau>}$ is a reset gate, and $\mathbf{z}^{<\tau>}$ is an update gate. Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If set the reset to all 1's and update gate to all 0's, it will arrive at the vanilla RNN model.

Chapter 3

ESTIMATION OF REMAINING USEFUL LIFE OF MECHANICAL SYSTEMS

This chapter presents a framework for estimating the remaining useful life (RUL) of mechanical systems. The framework consists of a multi-layer perceptron and an evolutionary algorithm for optimizing the data-related parameters. The framework makes use of a strided time window to estimate the RUL for mechanical components. Tuning the data-related parameters can become a very time consuming task. The framework presented here automatically reshapes the data such that the efficiency of the model is increased. Furthermore, the complexity of the model is kept low, e.g. neural networks with few hidden layers and few neurons at each layer. Having simple models has several advantages like short training times and the capacity of being in environments with limited computational resources such as embedded systems. The proposed method is evaluated on the publicly available C-MAPSS dataset [54], its accuracy is compared against other state-of-the art methods for the same dataset.

3.1 Related Work and Motivation

Traditionally, maintenance of mechanical systems has been carried out based on scheduling strategies. Such strategies are often costly and less capable of meeting the increasing demand of efficiency and reliability [55, 56]. Condition based maintenance (CBM) also known as intelligent prognostics and health management (PHM) allows for maintenance based on the current health of the system, thus cutting down the costs and increasing the reliability of the system [57]. Here, we refer to prognostics as the estimation of remaining useful life of a system. The remaining useful life (RUL) of the system can be estimated based on the historical data. This data-driven approach can help optimize maintenance schedules to avoid engineering failures and to save costs [58].

The existing PHM methods can be grouped into three different categories: model-based [59], data-driven [60, 61] and hybrid approaches [62, 63]. Model-based approaches attempt to incorporate physical models of the system into the estimation of the RUL. If the system degradation is modeled precisely, model-based approaches usually exhibit better performance than data-driven approaches [64]. This comes

at the expense of having extensive a priori knowledge of the underlying system and having a fine-grained model of the system, which can involve expensive computations. On the other hand, data-driven approaches use pattern recognition to detect changes in system states. Data-driven approaches are appropriate when the understanding of the first principles of the system dynamics is not comprehensive or when the system is sufficiently complex such as jet engines, car engines and complex machinery, for which it is prohibitively difficult to develop an accurate model.

Common disadvantages for the data-driven approaches are that they usually exhibit wider confidence intervals than model-based approaches and that a fair amount of data is required for training. Many data-driven algorithms have been proposed. Good prognostics results have been achieved. Among the most popular algorithms we can find artificial neural networks (ANNs) [55], support vector machine (SVM) [65], Markov hidden chains (MHC) [66] and so on. Over the past few years, data-driven approaches have gained more attention in the PHM community. A number of machine learning techniques, especially neural networks, have been applied successfully to estimate the RUL of diverse mechanical systems. ANNs have demonstrated good performance in modeling highly nonlinear, complex, multi-dimensional systems without any prior knowledge on the system behavior [10]. While the confidence limits for the RUL predictions cannot be analytically provided [67], the neural network approaches are promising for prognostic problems.

Neural networks for estimating the RUL of jet engines have been previously explored in [9] where the authors propose a multi-layer perceptron (MLP) coupled with a feature extraction (FE) method and a time window for the generation of the features for the MLP. In the publication, the authors demonstrate that a moving window combined with a suitable feature extractor can improve the RUL prediction as compared with the studies with other similar methods in the literature. In [10], the authors explore a deep learning ANN architecture, the so-called convolutional neural networks (CNNs), where they demonstrate that by using a CNN without any pooling layers coupled with a time window, the predicted RUL is further improved.

Despite the success some neural network models have exhibited in prognostics one challenge remains. Many of the state-of-the-art models [10,25,37] use very complex neural networks to perform the RUL estimation. Furthermore, architectures such as RNNs that may be more suitable for this kind of task are very computationally demanding. Here we present a novel framework for estimating the RUL of complex mechanical systems. The framework consists of a MLP to estimate the RUL of the system, coupled with an evolutionary algorithm for the fine tuning of data-related parameters. We refer to data-related parameters as parameters that define the shape, defined in terms of window size and window stride, and quality of the data, measured with respect to some performance indicators, used by the MLP. Please note that while this specific framework makes use of a MLP, the framework can in principle use several other learning algorithms, our main objective is the

treatment of the data instead of the choice of a particular learning algorithm. The publicly available NASA C-MAPSS dataset [54] is used to assess the efficiency and reliability of the proposed framework, by efficiency we mean the complexity of the used regressor and reliability refers to the accuracy of the predictions made by the regressor. This approach allows for a simple and small MLP to obtain better results than those reported in the current literature while using less computing power.

3.2 NASA C-MAPSS Dataset

The NASA C-MAPSS dataset is used to evaluate performance of the proposed method [54]. The C-MAPSS dataset contains simulated data produced using a model based simulation program developed by NASA. The dataset is further divided into 4 subsets composed of multi-variate temporal data obtained from 21 sensors.

For each of the 4 subsets, a training and a test set are provided. The training sets include run-to-failure sensor records of multiple aero-engines collected under different operational conditions and fault modes as described in Table 3.1.

	C-MAPSS			
Dataset	FD001	FD002	FD003	FD004
Training Trajectories	100	260	100	248
Test Trajectories	100	259	100	248
Operating Conditions	1	6	1	6
Fault Modes	1	1	2	2

Table 3.1: C-MAPSS dataset details.

The data is arranged in an $N \times 26$ matrix where N is the number of data points in each subset. The first two variables represent the engine and cycle numbers, respectively. The following three variables are operational settings which correspond to the conditions in Table 3.1 and have a substantial effect on the engine performance. The remaining variables represent the 21 sensor readings that contain the information about the engine degradation over time.

Each trajectory within the training and test sets represents the life cycles of the engine. Each engine is simulated with different initial health conditions, i.e. no initial faults. For each trajectory of an engine the last data entry corresponds to the cycle at which the engine is found faulty. On the other hand, the trajectories of the test sets terminate at some point prior to failure, hence the need to predict the remaining useful life. The aim of the MLP model is to predict the RUL of each engine in the test set. The actual RUL values of test trajectories are also included in the dataset for verification. Further discussions of the dataset and details on how the data is generated can be found in [68].

3.2.1 Performance Metrics

To evaluate the performance of the proposed approach on the C-MAPSS dataset, we make use of two scoring indicators, namely the Root Mean Squared Error (RMSE) denoted as $E_{RMS}(\mathbf{e})$ and a score proposed in [68] which we refer as the RUL Health Score (RHS) denoted as $E_{RH}(\mathbf{e})$. The two scores are defined as follows,

$$E_{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{N} e_i^2} \tag{3.1}$$

$$\begin{aligned} E_{RH} &= \frac{1}{n}\sum_{i=1}^{N} s_i \\ s_i &= \begin{cases} \exp(-\frac{e_i}{13}) - 1, & e_i < 0 \\ \exp(\frac{e_i}{10}) - 1, & e_i \geq 0, \end{cases} \end{aligned} \tag{3.2}$$

where n is the total number of samples in the test set and $\mathbf{e} = \hat{\mathbf{y}} - \mathbf{y}$ is the error between the estimated RUL values $\hat{\mathbf{y}}$, and the actual RUL values $\mathbf{y}$ for each engine within the test set. It is important to note that $E_{RH}(\mathbf{e})$ penalizes late predictions more than early predictions since usually late predictions lead to more severe consequences in fields such as aerospace.

3.3 Framework Description

In this section, the proposed ANN-EA based method for prognostics is presented. The method makes use of a multi-layer perceptron (MLP) as the main regressor for estimating the RUL of the engines in the C-MAPSS dataset. The choice of a MLP as the learning algorithm instead of any of the other choices (SVM, RNN, CNN, Least-Squares, etc) obeys to the fact that MLPs are in general good for nonlinear data like the one exhibited by the C-MAPSS dataset, but at the same time are less computationally expensive than some of the more sophisticated algorithms as the CNN or the RNN. Indeed, the RNN may be a more suitable choice for this particular problem since it involves time-sequenced data, nevertheless, we will show that by doing a fine tuning of the data-related parameters (and thus data processing), the inference power of a simple MLP can be competitive even when compared against that of an RNN. For the training sets, the feature vectors are generated by using a moving time window while a label vector is generated with the RUL of the engine. The label has a constant RUL for the early cycles of the simulation, and becomes a linearly decreasing function of the cycle in the remaining cycles. This is the so-called piece-wise linear degradation model [69]. For the test

set, a time window is taken from the last sensor readings of the engine. The data of the test set is used to predict the RUL of the engine.

The window-size n_w, window-stride n_s, and early-RUL R_e are data-related parameters, which for the sake of clarity and formalism in this study, form a vector $\mathbf{v} \in \mathbb{Z}^3$ such that $\mathbf{v} = (n_w, n_s, R_e)$. The vector $\mathbf{v}$ has a considerable impact on the quality of the predictions by the regressor. It is computationally intensive to find the best parameters of $\mathbf{v}$ given the search space inherent to these parameters. We propose an evolutionary algorithm to optimize the data-related parameters $\mathbf{v}$. The optimized parameter set $\mathbf{v}$ allows the use of a simple neural network architecture while attaining better results in terms of the quality of the predictions compared with the results by other methods in the literature.

3.3.1 The Network Architecture

After careful examinations of the C-MAPSS dataset, we propose to use a rather simple MLP architecture for all the four subsets of the data. The choice of a simple architecture over a more complex one follows the fact that simpler neural networks are less computationally expensive to train, furthermore, the inference is done also faster since less operations are involved. To measure the simplicity/complexity of a neural network we use the number of trainable parameters (weights) of the neural network, usually the more trainable parameters in a network the more computations that need be done, thus increasing the computational burden of the training/inference process. The implementations are done in Python using the Keras/Tensorflow environment. The source code is publicly available at the git repository `https://github.com/dlaredo/NASA_RUL_-CMAPS-` [70].

The choice of the network architecture is made by following an iterative process, our goal was to find a good compromise between efficiency (simple neural network models) and reliability (scores obtained by the model using the RMSE metric): We compared 6 different architectures (see Appendix A), training each for 100 epochs using a mini-batch size of 512 and averaging their results on a cross-validation set for 10 different runs. L1 (Lasso) and L2 regularization (Ridge) [71] are used to prevent over-fitting. L1 regularization penalizes the sum of the absolute value of the weights and biases of the networks, while L2 regularization penalizes the sum of the squared value of the weights and biases. The data-related parameters $\mathbf{v}$ used for this experiment are $\mathbf{v} = (30, 1, 140)$. Two objectives are pursued during the iterations: the architecture must be minimal in terms of the number of trainable parameters and the performance indicators must be minimized. Table 3.2 summarizes the results for each tested architecture.

	RMSE				RHS			
Tested Architecture	Min.	Max.	Avg.	STD	Min.	Max.	Avg.	STD
Architecture 1	15.51	17.15	16.22	0.49	4.60	7.66	5.98	0.91
Architecture 2	15.24	16.46	15.87	0.47	4.07	6.26	5.29	0.82
Architecture 3	15.77	17.27	16.15	0.45	5.11	8.25	5.93	0.94
Architecture 4	15.13	17.01	15.97	0.47	3.90	7.54	5.65	1.2
Architecture 5	16.39	17.14	16.81	0.23	5.19	6.58	5.98	0.42
Architecture 6	16.42	17.36	16.87	0.30	5.15	7.09	6.12	0.62

Table 3.2: Results for different architectures for subset 1, 100 epochs.

Table 3.3 presents the architecture chosen for the remainder of this work (which provides the best compromise between compactness and performance among the tested architectures). Each row in the table represents a neural network layer while each column describes each one of the key parameters of the layer such as the type of layer, number of neurons in the layer, activation function of the layer and whether regularization is used, where L1 denotes the L1 regularization factor and L2 denotes the L2 regularization factor, the order in which the layers are appended from the table is top-bottom. From here on we refer to this neural network model as $\phi(.)$.

Layer	Neurons	Activation	Additional Information
Fully connected	20	ReLU	$L1 = 0.1, L2 = 0.2$
Fully connected	20	ReLU	$L1 = 0.1, L2 = 0.2$
Fully connected	1	Linear	$L1 = 0.1, L2 = 0.2$

Table 3.3: Proposed neural network architecture $\phi(.)$.

3.3.2 Shaping the Data

This section covers the data pre-processing applied to the raw sensor readings in each of the datasets. Although the original datasets contain 21 different sensor readings, some of the sensors do not present much variance or convey redundant information, our choice of the sensors is based on previous studies such as [9,10] where it was discovered that some sensor values do not vary at all throughout the entire engine life cycle while some others are redundant according to PCA or clustering analysis. These sensors are therefore discarded. In the end, only 14 sensor readings out of the 21 are considered for this study. Their indices are $\{2, 3, 4, 7, 8, 9, 11, 12, 13, 14, 15, 17, 20, 21\}$. The raw measurements are then used to create the strided time windows with window-size n_w and window-stride n_s. For the training labels, R_e is used at the early stages and then the RUL is linearly decreased.

Assuming $\mathbf{x} \in \mathbb{R}^m$ is the vector whose components are the sensor readings at each time stamp, then the min-max normalized vector $\hat{\mathbf{x}}$ can be computed by means of the following formula:

$$\hat{x}_i = 2 * \frac{x_i - min(x_i)}{max(x_i) - min(x_i)} - 1. \tag{3.3}$$

Time Window and Stride

In multivariate time-series problems such as RUL, more information can be generally obtained from the temporal sequence of the data as compared with the multivariate data point at a single time stamp. For a time window of size n_w with a stride $n_s = 1$, all the sensor readings in the time window form a feature vector $\mathbf{x} \in \mathbb{R}^{s*n_w}$, where s denotes the number of sensors being read. Stacking together m_w of this time windows forms feature vector $\mathbf{X} \in \mathbb{R}^{m_w \times s*n_w}$ while its corresponding RUL values are defined as $\mathbf{y} \in \mathbb{Z}^m$. It is important to mention that the shape of $\mathbf{X}$ defines the number of input neurons for the neural network, therefore changing the shape of $\mathbf{X}$ effectively changes the number of inputs to the neural network. This approach has successfully been tested in [9, 10] where the authors propose the use of a moving window with sizes ranging from 20 to 30. We propose not only the use of a moving time window, but also a *strided* time window that updates more than one element ($n_s > 1$) at the time. A graphical depiction of the strided time window is shown in Figure 3.1. For Figure 3.1 the numbers and time-stamps are just illustrative, the window size exemplified is of 30 time-stamps while the stride is of 3 time-stamps.

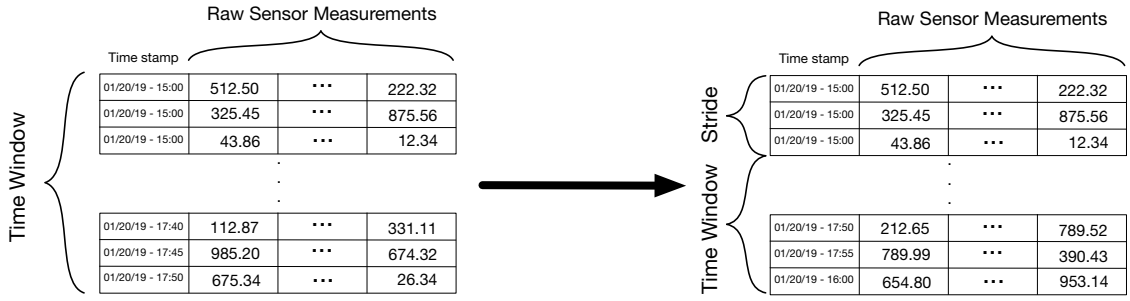


Figure 3.1: Graphical depiction of the time window used in this framework.

Figure 3.2 shows an example of how to form a sample vector, in this example $s = 14$, $n_w = 5$ and $n_s = 4$. Each one of the plotted lines denotes the readings for each of the fourteen chosen sensors. The dashed vertical lines (black lines) represent the size of the window, in this case we depict a window size of 5 cycles. For the next window (red dashed lines) the time window is advanced by a stride of 4 cycles, note

that some sensor readings may be overlapped for different moving windows. For every window, the sensor readings are appended one after another to form a vector of $14 * 5$ features, for this specific case the unrolled vector will be of 70 features.

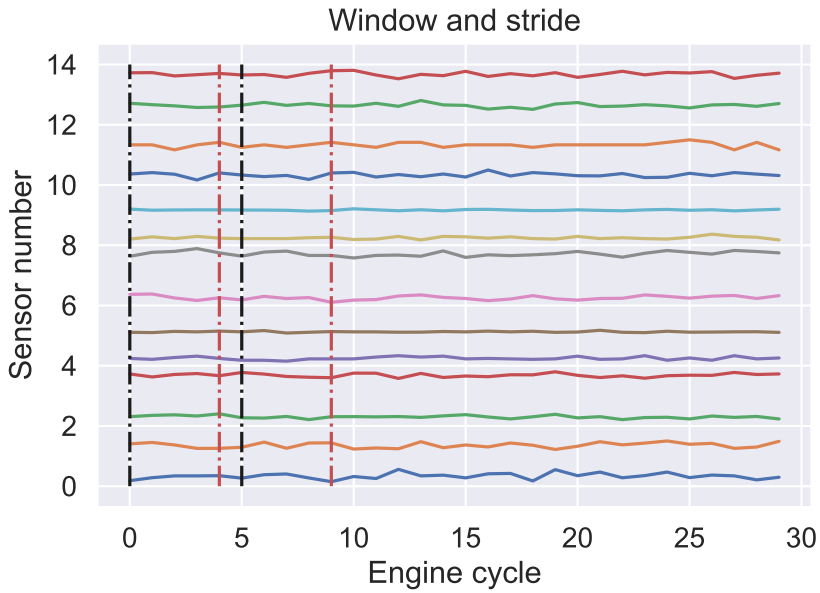


Figure 3.2: Window size and stride example.

The use of a strided time window allows for the regressor to take advantage not only of the previous information, but also to control the ratio at which the algorithm is fed with new information. With the usual time window approach, only one point is updated for every new time window. The strided time window considered in this study allows for updating more than one point at the time for the algorithm to make use of the new information with less iterations. Our choice of the strided time window is inspired by the use of strided convolutions in Convolutional Neural Networks [72]. Further studies of the impact of the stride on the prediction should be done in the future.

Piece-wise Linear Degradation Model

Different from common regression problems, the desired output value of the input data is difficult to determine for a RUL problem. It is usually impossible to evaluate the precise health condition and estimate the RUL of the system at each time step without an accurate physics based model. For this popular dataset, a piece-wise linear degradation model has been proposed in [69]. The model assumes that the engines have a constant RUL label in the early cycles, and then the RUL

starts degrading linearly until it reaches 0 as shown in Figure 3.3. The piece-wise linear degradation assumption is used in this work. We denote the value of the RUL in the early cycles as R_e. Initially, R_e is randomly chosen between 90 and 140 cycles which is a reasonable range of values for this particular application. When the difference between the cycle count in the time window and the terminating cycle of the training data is less than the initial value of R_e, R_e begins the linear descent toward the terminating cycle.

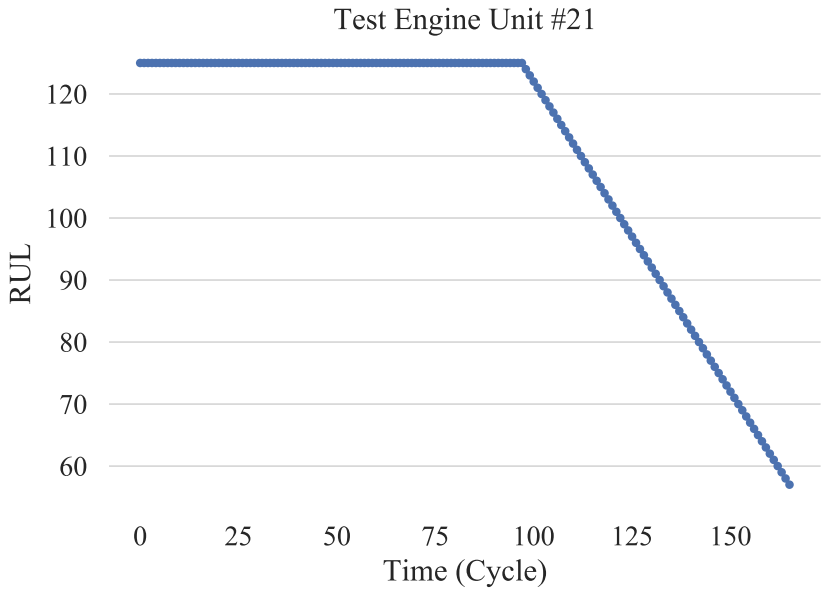


Figure 3.3: Piece-wise linear degradation for RUL.

3.3.3 Optimal Data Parameters

As mentioned in the previous sections the choice of the data-related parameters $\mathbf{v}$ has a large impact on the performance of the regressor. In this section, we present a method for picking the optimal combination of the data-related parameters n_w, n_s and R_e while being computationally efficient.

Vector $\mathbf{v} = (n_w, n_s, R_e)$ components specific to the C-MAPSS dataset are bounded such that $n_w \in [1, b]$, $n_s \in [1, 10]$, and $R_e \in [90, 140]$, where all the variables are integer. The value of b is different for different subsets of the data, Table 3.4 shows the different values of b for each subset.

	FD001	FD002	FD003	FD004
b	30	20	30	18

Table 3.4: Allowed values for b per subset.

Let $\mathbf{X}(\mathbf{v})$ be the training/cross-validate/test sets parametrized by $\mathbf{v}$ and $\hat{\mathbf{y}}(\mathbf{v}) = \phi(\mathbf{X}(\mathbf{v}); \boldsymbol{\theta})$ be the predicted RUL values of our model $\phi(.)$ with $\mathbf{X}(\mathbf{v})$. Thus, E_{RMS} depends directly on the choice of $\mathbf{v}$ since $\phi(.)$ is fixed (See Section 3.2.1). Here we propose to solve the following optimization problem

$$\min_{\mathbf{v} \in \mathbb{Z}^3} E_{RMS}(\mathbf{v}). \tag{3.4}$$

The problem to find optimal data-related parameters has no analytic descriptions. Therefore, no gradient information is available. An evolutionary algorithm is the natural choice for this optimization problem. Nevertheless, since the computation of the error $E_{RMS}(\mathbf{v})$ requires re-training $\phi(.)$ an strategy to make the optimization process computationally efficient must be devised.

True Optimal Data Parameters

The finite size of C-MAPSS dataset and finite search space of $\mathbf{v}$ allow an exhaustive search to be performed in order to find the true optimal data-related parameters. We would like to emphasize that although exhaustive search is possible for the C-MAPSS dataset, it is in no way a possibility in a more general setting. Nevertheless, the possibility to perform exhaustive search on the C-MAPSS dataset can be exploited to demonstrate the accuracy of the chosen EA and of the framework overall. In the following studies, we use the results and computational efforts of the exhaustive search as benchmarks to examine the accuracy and efficiency of the proposed approach.

We should note that the subsets of the data FD001 and FD003 have similar features and that the subsets FD002 and FD004 have similar features. Because of this, we have decided to just optimize the data-related parameters by considering the subsets FD001 and FD002 only. An exhaustive search is performed to find the true optimal values for $\mathbf{v}$. The MLP is only trained for 20 epochs. Table 3.5 shows the optimal as well as the worst combinations of data-related parameters and the total number of function evaluations used by the exhaustive search. It is important to notice that for this experiment the window size is limited to be larger than or equal to 15.

Dataset	argmin $\mathbf{v}$	min $E_{RMS}(\mathbf{v})$	argmax $\mathbf{v}$	max $E_{RMS}(\mathbf{v})$	Function evals.
FD001	$[24, 1, 127]$	15.11	$[25, 10, 94]$	85.19	8160
FD002	$[16, 1, 138]$	30.93	$[17, 10, 99]$	59.78	3060

Table 3.5: Exhaustive search results for subsets FD001 and F002.

Numerical experiments seem to suggest that, at least for CMAPSS dataset, the window size plays a big role in terms of the meaningful information used for prediction by the MLP. It also reflects the history-dependent nature of the aircraft engine degradation process. Furthermore, overlapping in the generated time windows seems to benefit the generated sequences of sensors.

3.3.4 Evolutionary Algorithm for Optimal Data Parameters

Evolutionary algorithms (EAs) are a family of methods for optimization problems. The methods do not make any assumptions about the problem, treating it as a black box that merely provides a measure of quality given a candidate solution. Furthermore, EAs do not require the gradient when searching for optimal solutions, making them very suitable for applications such as neural networks.

For the current application, the differential evolution (DE) method is chosen as the optimization algorithm [73]. Though other meta-heuristic algorithms may also be suitable for this application, the DE has established as one of the most reliable, robust and easy to use EAs. Furthermore, a ready to use Python implementation is available through the scipy package [74]. Although the DE method does not have special operators for treating integer variables, a very simple modification to the algorithm, i.e. rounding every component of a candidate solution to its nearest integer, is used for this work.

As mentioned earlier, evolutionary algorithms such as the DE use several function evaluations when searching for the optimal solutions. It is important to consider that, for this application, one function evaluation requires retraining the neural network from scratch. This is not a desirable scenario, as obtaining the optimal data-related parameters would entail an extensive computational effort. Instead of running the DE for several iterations and with a large population size, we propose to run it just for 30 iterations, i.e. the generations in the literature of evolutionary computation, with a population size of 12, which seems reasonable given the size of the search space of $\mathbf{v}$.

During the optimization, the MLP is trained for only 20 epochs. The small number of epochs of training the MLP is reasonable in this case because a small batch of data is used in the training, because we only look for the trend of the scoring indicators. Furthermore, it is common to observe that the parameters leading to lower score values in the early stages of the training are more likely to provide better

performance after more epochs of training. The settings of the DE algorithm to find the optimal data-related parameters are listed in Table 3.6.

Population Size	Generations	Strategy	MLP epochs
12	30	Best1Bin [17]	20

Table 3.6: Differential evolution hyper-parameters.

The optimal data-related parameters for the subsets FD001 and FD002 found by the DE algorithm are listed in Table 3.7. As can be observed, the results are in fact very close to the true optimal ones in Table 3.5 for both the subsets of the data. The computational effort is reduced by one order of magnitude when using the DE method as compared to the exhaustive search for the true optimal parameters. From the results in Table 3.7, it can be observed that the maximum allowable time window is always preferred while, on the other hand, small window strides yield better results. For the case of early RUL, it can be observed that larger values of R_e are favored.

Dataset	argmin $\mathbf{v}$	min $E_{RMS}(\mathbf{v})$	Function evals.
FD001	$[24, 1, 129]$	15.24	372
FD002	$[17, 1, 139]$	30.95	372

Table 3.7: Data-related parameters for each subset obtained with differential evolution.

3.3.5 The Estimation Algorithm

Having described the major building blocks of the proposed method, we now introduce the complete framework in the form of Algorithm 2.

Algorithm 2: ANN-EA RUL estimation framework.

Data: Training/testing data $\mathbf{X} \in \mathbb{R}^{m_w \times s * n_w}$

Input : Initial set of data-related parameters $\mathbf{v} \in \mathbb{Z}^3$, , training labels $\mathbf{y} \in \mathbb{Z}^m$ and number of training epochs for each evaluation of $\phi(\mathbf{v})$.

Output: Optimal set of data-related parameters $\mathbf{v}^* \in \mathbb{Z}^3$.

Choose regressor architecture (ANN, SVM, linear/logistic regression, etc).
Define $\phi(\mathbf{v})$ as in Section 3.3.3.
Optimize $\phi(\mathbf{v})$, by means of an evolutionary algorithm, using the proposed guidelines from Section 3.3.4.
Use $\mathbf{v}^*$ to train the regressor for as many epochs as needed.

3.4 Evaluation of the Proposed Method

3.4.1 Experimental Settings

In this section, we evaluate the performance of the proposed method. The architecture of the MLP is described in Table 3.3. We define an experiment as the process of training the MLP on any of the subsets (FD001 to FD004) and evaluate its performance using the subset's test set. The combinations of the optimal window size n_w, window stride n_s and early RUL R_e are presented in Table 3.8. We perform 10 different experiments for each data subset, the MLP is trained for 200 epochs using the train set for the corresponding data subset and evaluated using the subset's test set. The results for each dataset subset are averaged and presented in Table 3.9. Furthermore, the best model is saved and later used to generate the results presented in Section 3.4.2. All of the experiments were run using the Keras/Tensorflow framework, an NVIDIA GeForce 1080Ti GPU was used to speed up the training process.

Dataset	n_w	n_s	R_e	Input size (neurons)
FD001	24	1	129	336
FD002	17	1	139	238
FD003	24	1	129	336
FD004	17	1	139	238

Table 3.8: Data-related parameters for each subset as obtained by DE.

3.4.2 Experimental Results

The obtained results for $\phi(\mathbf{v})$ using the above setting are presented in Table 3.9. Notice that the performances obtained for datasets FD001 and FD002 are improved as compared with the results in Table 3.7. This is due to the fact that the MLP is trained for more epochs, thus obtaining better results.

	E_{RMS}				E_{RH}			
Data Subset	min	max	avg	STD	min	max	avg	STD
FD001	14.24	14.57	14.39	0.11	3.25	3.58	3.37	0.11
FD002	28.90	29.23	29.09	0.11	45.99	53.90	50.69	2.17
FD003	14.74	16.18	15.42	0.50	4.36	6.85	5.33	0.95
FD004	33.25	35.10	34.74	0.53	58.52	78.62	74.77	5.88

Table 3.9: Scores for each dataset using the data-related parameters obtained by DE.

We now compare the predicted RUL values versus the real RUL values for each of the datasets. For Figures 3.4 to 3.7 we plot on the top sub-figure the predicted RUL values (red lines) vs the real RUL values (green lines) while on the bottom sub-figure we plot the error between real RUL and predicted RUL.

Figure 3.4 shows the comparison for subset FD001, it can be observed that the predicted RUL values closely follow the real RUL values with the exception of a pair of engines. The error remains small for most of the engines. Meanwhile, for FD002 it can be observed in Figure 3.5 that the regressor overshoots several of the RUL predictions, especially in the positive spectrum. That is, the method predicts a RUL when in reality the real RUL is less than the predicted value. This is more evident in the second subplot where the maximum error is 138 at the magenta peak in the leftmost part of the plot.

Figure 3.6 shows that for FD003 the predictions follow closely the real RUL values. The behavior for FD004 is similar to FD002 as depicted in Figure 3.7, with most of the error such that the predictions of the RUL are larger than the real value.

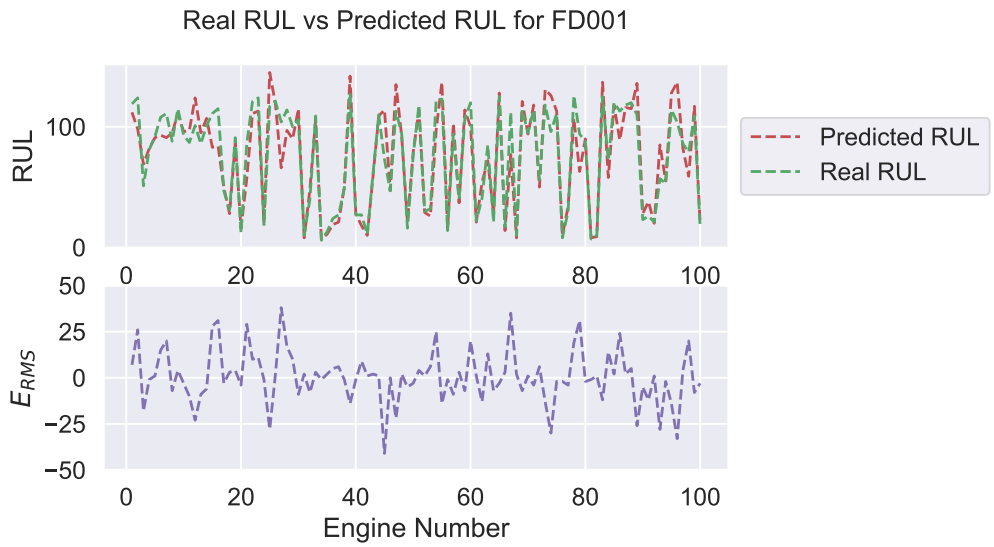


Figure 3.4: Comparison of predicted RUL values vs real RUL values for dataset FD001

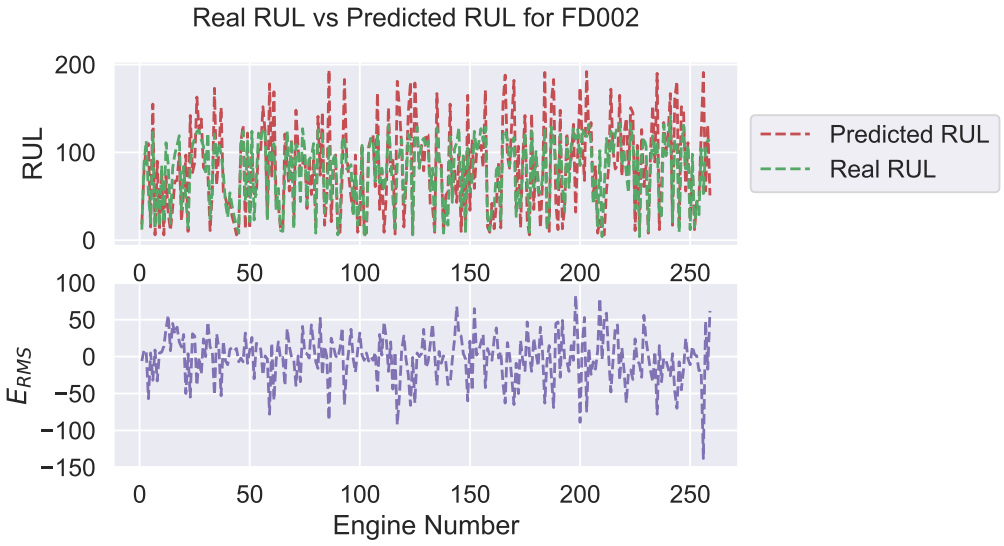


Figure 3.5: Comparison of predicted RUL values vs real RUL values for dataset FD002

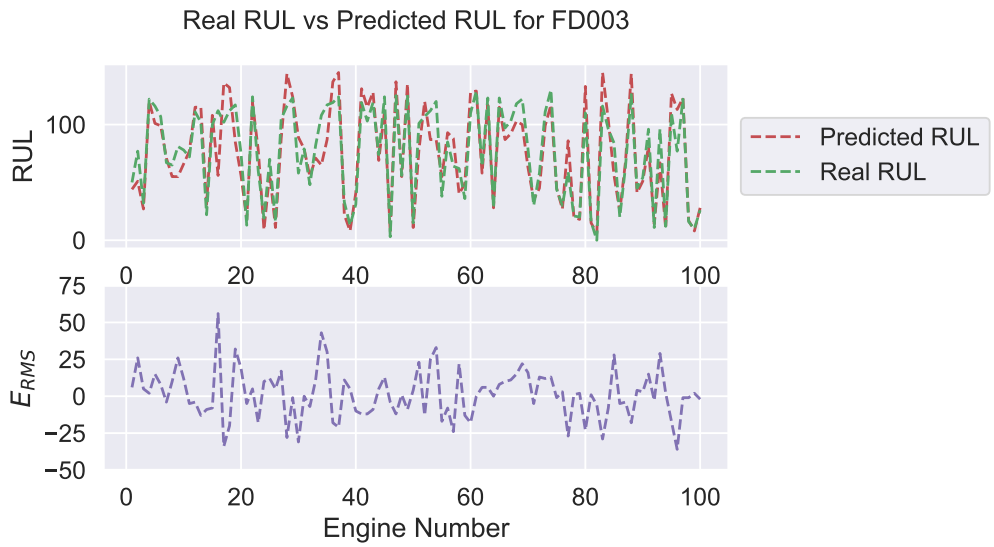


Figure 3.6: Comparison of predicted RUL values vs real RUL values for dataset FD003

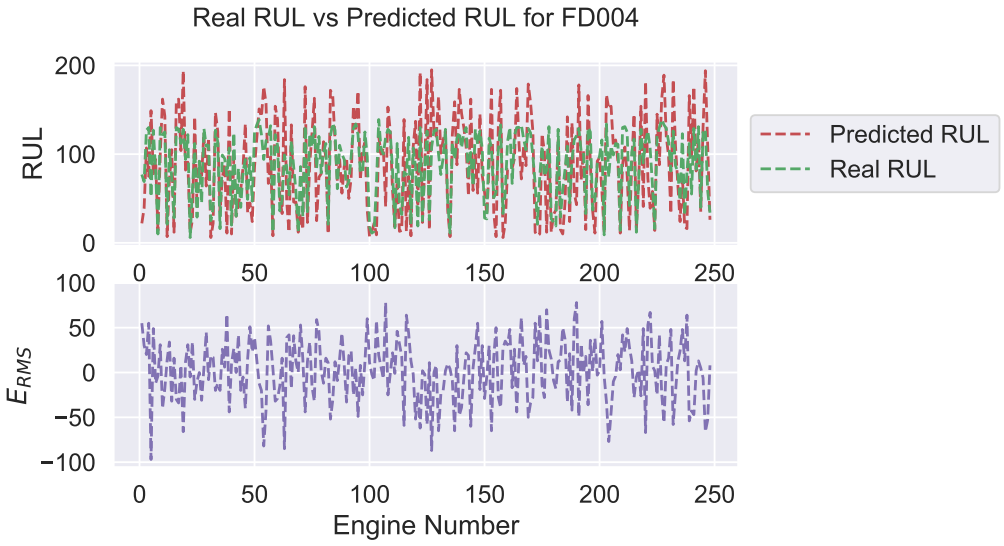


Figure 3.7: Comparison of predicted RUL values vs real RUL values for dataset FD004

Finally, Figure 3.8 shows that most of the RUL predictions by the method are highly accurate. In the case of FD001, the predictions of 50% of the engines is smaller than 6 cycles. In the case of FD002, the predictions are acceptable with the error of first quartile being lower than 6 cycles and the error for 50% of the engines being less than 19 cycles. The cases for FD003 and FD004 are similar to FD001 and FD002, respectively.

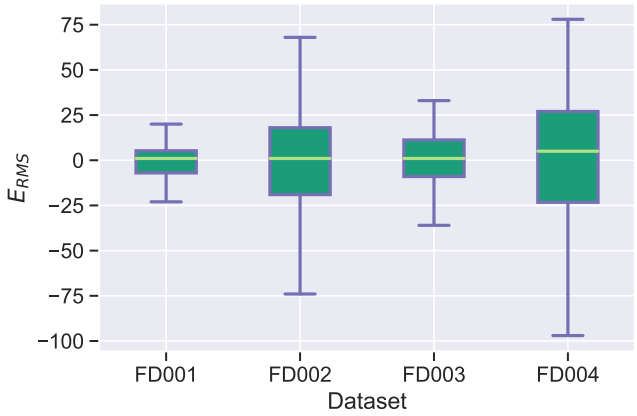


Figure 3.8: Prediction error of the MLP for each dataset.

Two conclusions can be drawn from the previous discussions. First, it can be observed that the number of operating conditions has a larger impact on the complexity of the data than the number of fault modes. This is because the subsets FD002 and FD004 exhibit larger errors than the subsets FD001 and FD003, although the general trend of prediction errors is similar among those two groups of data sets. Second, for the subsets FD002 and FD004, most of the error is related to the predictions larger than the real RUL values. Another observation is that larger window sizes usually lead to better predictions. This may be related to the history-dependent nature of the physical problem.

3.4.3 Comparison to Other Approaches

The performance of the proposed method is also compared against other state-of-the-art methods. The methods chosen for the comparison obey to two criteria: 1) that the method used is a machine learning (or statistical) method and 2) that the method is recent (6 years old at most). Most of the methods chosen here have only reported results on the test set FD001 in terms of E_{rms}. The results are shown in Table 3.10. The E_{rms} value of the proposed method in Table 3.10 is the mean value of 10 independent runs. The values of other methods are identical to those reported in their respective original papers.

Method	E_{rms}
ESN trained by Kalman Filter [75]	63.45
Support Vector Machine Classifier [37]	29.82
Time Window Neural Network [9]	15.16
Multi-objective deep belief networks ensemble [76]	15.04
Deep Convolutional Neural Network [8]	18.45
Proposed method with $n_w = 30$, $n_s = 1$ and $R_e = 128$	**14.39**
Modified approach with classified sub-models of the ESN [75]. Only 80 of 100 engines predicted	7.021
Deep CNN with time window [10].	13.32
RNN-Encoder-Decoder [25].	12.93

Table 3.10: Performance comparison of the proposed method and the latest related papers on the C-MAPSS dataset.

From the comparison studies, we can conclude that the proposed method performs better than the majority of the chosen methods when taking into consideration the whole dataset FD001. Two existing methods come close to the performance of the proposed approach here, namely the time window ANN [9] and the Networks Ensemble [76]. While the performance of these two methods comes close to the results of the proposed method, our method is computationally more efficient. We believe that the use of the time window with a proper size makes the difference. Notice that three of the methods presented in Table 3.10 perform better than the proposed method, namely, the Convolutional Neural Network [10], the RNN-Encoder-Decoder [25] and the modified ESN [75]. Nevertheless in the case of [75], the method can only predict 80 out of the 100 total engines while the deep CNN [10] and RNN [25] approaches are much more computationally expensive than the MLP used in this work. In the case of the method proposed in [10], the neural network model has four layers of convolutions and two more layers of fully connected layers. On the other hand, the RNN-Encoder-Decoder [25] makes use of a much more complicated scheme of two RNNs, one for encoding the sequences and one for decoding them. While specialized libraries such as TensorFlow or Keras make RNNs easier to use, they still remain up to date as some of the most computationally expensive architecture to train given their sequential nature. Finally, we would like to emphasize that the used MLP for this approach is one of the simplest ones in the reviewed literature. Furthermore, the framework proposed is simple to understand and implement, robust, generic and light-weight. These are the features important to highlight when comparing the proposed method against other state-of-the-art approaches.

Chapter 4

AUTOMATIC SELECTION OF DEEP LEARNING MODELS

Neural networks and deep learning are changing the way that artificial intelligence is being done. Efficiently choosing a suitable network architecture and fine tune its hyper-parameters for a specific dataset is a time-consuming task given the staggering number of possible alternatives. In this chapter, we address the problem of model selection by means of a fully automated framework for efficiently selecting a neural network model for a given task: classification or regression. The algorithm, named Automatic Model Selection, is a modified is a modified micro-genetic algorithm that automatically and efficiently finds the most suitable neural network model for a given dataset. The main contributions of this method are: a simple list based encoding for neural networks as genotypes in an evolutionary algorithm, new crossover and mutation operators, the introduction of a fitness function that considers both, the accuracy of the model and its complexity and a method to measure the similarity between two neural networks. AMS is evaluated on two different datasets. By comparing some models obtained with AMS to state-of-the-art models for each dataset we show that AMS can automatically find efficient neural network models. Furthermore, AMS is computationally efficient and can make use of distributed computing paradigms to further boost its performance.

4.1 Motivation

Machine learning studies algorithms that improve themselves through experience. Given the large amount of data currently available in many fields such as engineering, bio-medical, finance, *etc.* and the increasingly more computing power available, machine learning is now practiced by people with very diverse backgrounds. More users of machine learning tools are non-experts who require off-the-shelf solutions. Automated Machine Learning (AutoML) is the field of machine learning devoted to developing algorithms and solutions to enable people with limited machine learning background knowledge to use machine learning models easily. Tools like WEKA [26], PyBrain [14] and MLLib [77] follow this paradigm. Nevertheless, the user still needs to make some choices, which may not be obvious or intuitive in selecting a learning algorithm, hyper-parameters, features, *etc.* and thus leads to the selection of non-optimal models.

Recently, deep learning models such as CNN, RNN and Deep NN have gained a lot of attention due to their improved efficiency on complex learning problems and their flexibility and generality for solving a large number of problems including regression, classification, natural language processing, recommendation systems, *etc.* Furthermore, there are many software libraries which make their implementation easier. TensorFlow [78], Keras [79], Caffe [80] and CNTK [81] are some examples of such libraries. Despite the availability of such libraries and tools, the tasks of picking the right neural network model and its hyper-parameters are usually complex and iterative in nature, specially among non-computer scientists.

Usually, the process of selecting a suitable machine learning model for a particular problem is done in an iterative manner. First, an input dataset must be transformed from a domain specific format to features which are predictive of the field of interest. Once the features have been engineered, users must pick a learning setting which is appropriate to their problem, e.g. regression, classification or recommendation. Next, users must pick an appropriate model, such as support vector machines (SVM), logistic regression or any flavor of neural networks (NNs). Each model family has a number of hyper-parameters, such as regularization degree, learning rate, number of neurons, *etc.* and each of these must be tuned to achieve optimal results. Finally, users must pick a software package that can train their model, configure one or more machines to execute the training and evaluate the model's quality. It can be challenging to make the right choice when facing so many degrees of freedom, leading many users to select a model based on intuition or randomness and/or leave hyper-parameters set to default. This approach will usually yield sub-optimal results.

This suggests a natural challenge for machine learning: given a dataset, to automatically and simultaneously choose a learning algorithm and set its hyper-parameters to optimize performance. As mentioned in [26], the combined space of learning algorithm and hyper-paremeters is very challenging to search: the response function is noisy and the space is high dimensional involving both, categorical and continuous choices and containing hierarchical dependencies, e.g. hyper-parameters of the algorithm are only meaningful if that algorithm is chosen. Thus, identifying a high quality model is typically costly in the sense that it entails a lot of computational effort and time-consuming.

To address this challenge, we propose Automatic Model Selection (AMS), a flexible and scalable method to automate the process of selecting artificial neural network models. The key contributions of the method are: 1) a simple, list based encoding of neural networks as genotypes for evolutionary computation algorithms, 2) new crossover and mutation operators to generate valid neural networks models from an evolutionary algorithm, 3) the introduction of a fitness function that considers both, the accuracy of the model and its complexity and 4) a method for

measuring the similarity between two neural networks. All these components together form a new method based on an evolutionary algorithm, which we call AMS, and that can be used to find an optimal neural network architecture for a given dataset.

4.2 Problem Statement

In this section we mathematically define the general neural network architecture search problem. Consider a dataset $\mathcal{D}$ made up of training points $d_i = (\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{X} \times \mathbf{Y}$ where $\mathbf{X}$ is the set of data points and $\mathbf{Y}$ is the set of labels. Furthermore, we split the dataset into training set D_t, cross validation set $\mathcal{D}_v$, and test set $\mathcal{D}_p$. Given a neural network architecture search space $\mathcal{H}$, the performance of a neural network architecture $\phi \in \mathcal{H}$ trained on $\mathcal{D}_t$ and validated with $\mathcal{D}_v$ is defined as:

$$p = Perf(\phi(\mathcal{D}_t), \mathcal{D}_v), \tag{4.1}$$

where $Perf(.)$ is a measurement of the generalization error attained by the learning algorithm $\phi(.)$ on the validation set $\mathcal{D}_v$. Common error indicators are accuracy error, precision error and mean squared error. Their definitions along with some other common error indicators are presented in Table 4.1.

Indicator name	Application	Definition
Mean Squared Error	Regression	$E_{MSE} = \frac{1}{2}\sum_{i=1}^{n}(\hat{\mathbf{y}} - \mathbf{y})^2$
Accuracy	Classification	$E_A = \frac{tp+tn}{tp+tn+fp+fn}$
Precision	Classification	$E_P = \frac{tp}{tp+fp}$
Recall	Classification	$E_R = \frac{tp}{tp+fn}$
F1	Classification	$F_1 = 2\frac{E_P E_R}{E_P+E_R}$

Table 4.1: Common performance metrics for neural networks. $\hat{\mathbf{y}}$ represents the predicted value of the model for a sample $\mathbf{x}$. tp stands for true positives count. tn stands for true negatives count. fp stands for false positives count. fn stands for false negatives count.

Finding a neural network $\phi^* \in \mathcal{H}$ that achieves a good performance has been explored in [82, 83] among others. While this task alone is challenging, usually the efficiency of ϕ^* is not measured. Indeed, it turns out that there can be several candidate models that can attain similar performances with improved efficiency. By

efficiency we mean, in practical terms, how fast is to train ϕ^* as compared to other possible solutions.

We aim at achieving neural network models ϕ^* that not only exhibit good performance on $\mathcal{D}$ as measured by p, but also achieve such performance by using a simple structure, which directly translates to improved efficiency of the model. To measure the complexity of the architecture, we make use of the number of trainable parameters $w(\phi)$ of the neural network. We will also refer to $w(\phi)$ as the "size" of the neural network.

The problem of finding a neural network ϕ that achieves a good performance on the dataset $\mathcal{D}$ while using a simple model can be mathematically stated as the following multi-objective optimization problem:

$$\min_{\phi \in \mathcal{H}} \quad (p(\phi), w(\phi)). \tag{4.2}$$

In this chapter we develop an algorithm to efficiently solve Problem (4.2).

4.3 Related Work and Motivation

Automated machine learning has been of research interest since the uprising of deep learning. This is no surprise since selecting an effective combination of algorithm and hyper-parameter values is currently a challenging task requiring both deep machine learning knowledge and repeated trials. This is not only beyond the capability of layman users with limited computing expertise, but also often a non-trivial task even for machine learning experts [84].

Until recently most state-of-the-art neural network architectures have been manually designed by human experts. To make the process easier and faster, researchers have looked into automated methods. These methods intend to find, within a pre-specified resource limit in terms of time, number of algorithms and/or combinations of hyper-parameter values, an effective algorithm and/or combination of hyper-parameter values that maximize the accuracy measure on the given machine learning problem and data set. Using an automated machine learning, the machine learning practitioner can skip the manual and iterative process of selecting an efficient combination of hyper-parameter values and neural network model, which is labor intensive and requires a high skill set in machine learning.

In the context of deep learning, neural architecture search (NAS), which aims at searching for the best neural network architecture for the given learning task and dataset, has become an effective computational tool in AutoML. Unfortunately, the existing NAS algorithms are usually computationally expensive where its time complexity can easily scale to $O(nt)$ where n is the number of neural architectures evaluated, and t is the average time consumption for each of the n neural networks. Many NAS approaches such as deep reinforcement learning [85–87] and evolutionary algorithms [30, 88–90] require a large n to reach good performance.

Other approaches, including Bayesian Optimization [91, 92] and Sequential Model Based Optimization (SMBO) [93, 94], are often as expensive as NAS while being more limited to the kind of models they can explore.

In the recent years, a number of tools have been made available for users to automate the model selection and/or hyper-parameter tuning. In the following, we present a brief description of some of them.

4.3.1 Auto-Keras

Auto-Keras [82] is a method to automatically generate model architectures. It defines an edit-distance kernel to measure the difficulty of transferring the current model to a new one. This kernel makes it possible to search the model structures in a tree-structured space constructed from the network morphism. Bayesian optimization is used along with the kernel to optimize the tree-structure of the model. The consistency of the input and output shapes is guaranteed throughout the process. The use of Auto-Keras does not require an extensive knowledge of machine learning, making it very accessible for starters. However, training the model is very time-consuming since a new model must be trained from scratch every time. Furthermore, acquiring the parameters in the middle steps is also computational expensive. An example on MNIST dataset may take several hours to converge (depending on the used equipment).

4.3.2 AutoML Vision

AutoML [95] uses evolutionary algorithms to perform image classification. Without any restriction on the search space such as network depth, skip connections, *etc.*, the algorithm starts from a simple model without convolutions and iteratively evolves the model into more a complex one. A massively-parallel and lock-free infrastructure is designed, and many computers may be used to search the large solution space. Communication between different nodes in the network is handled by using a shared file system that keeps track of the population. Among the main disadvantages of AutoML are that it requires extremely high computational power. Furthermore, since the candidate models start from a very simple structure poor performing models are likely to be obtained as a solution.

4.3.3 Auto-sklearn

Auto-sklearn [94] is a system designed to help machine learning users by automatically searching through the joint space of sklearn's learning algorithms and their respective hyper-parameter settings to maximize performance using a state-of-the-art Bayesian optimization method. Auto-sklearn addresses the model selection problem by treating all of sklearn algorithms as a single, highly parametric machine learning framework, and using Bayesian optimization to find an optimal instance

for a given dataset. Auto-sklearn also natively supports parallel runs on a single machine to find good configurations faster and save the n best configurations of each run instead of just the single best. Nevertheless, and to the best of our knowledge, Auto-sklearn does not provide support for neural networks and it does not take into consideration the complexity of the proposed models for assessing their optimality.

4.4 Proposed Evolutionary Algorithm for Model Selection

While there are a number of methods for automatic model selection and hyper-parameter tuning, many of them do not provide support for some of the most sophisticated deep learning architectures. For instance, Auto-sklearn does not provide good support for deep learning methods, support for distributed computing is also very limited. On the other hand Auto-Keras and AutoML Vision do provide support for deep learning methods, but they do not consider the model's complexity when assessing its overall performance. Indeed, Auto-Keras and AutoML require clusters of or hours of computing time to yield models with good accuracy. Furthermore, Auto-Keras and AutoML do not provide good support for regression problems.

We propose an efficient method that, for a given dataset $\mathcal{D}$, will automatically find a neural network model that attains high performance while being computationally efficient. The proposed model is capable of performing inference tasks for both classification and regression problems. Furthermore, the proposed system is scalable and easy to use in distributed computing environments, allowing it to be usable for large datasets and complex models. For this task, we make use of Ray [96], a distributed framework designed with large scale distributed machine learning in mind.

Our method provides support for three of the major neural networks architectures, namely multi-layer percepetrons (MLPs) [17], convolutional neural networks (CNNs) [97] and recurrent neural networks (RNNs) [50]. Our method can construct models of any of these architectures by stacking together a *valid* combination of any of the four following layers: fully connected layers, recurrent layers, convolutional layers and pooling layers. Our method does not only build neural networks for the aforementioned architectures, but also tunes some of the hyper-parameters such as the number of neurons at each layer, the activation function to use or the dropout rates for each layer. Support for skip connections is left for future work.

We say that a neural network architecture is *valid* if it complies with the following set of rules, which we derived empirically from our practice in the field:

- A fully connected layer can only be followed by another fully connected layer.
- A convolutional layer can be followed by a pooling layer, a recurrent layer, a fully connected layer or another convolutional layer.

- A recurrent layer can be followed by another recurrent layer or a fully connected layer.
- The first layer is user defined according to the type of architecture chosen (MLP, CNN or RNN).
- The last layer is always a fully connected layer with either a softmax activation function for classification or a linear activation function for regression problems.

4.4.1 Automatic Model Selection (AMS)

The key idea of our method is to develop an evolutionary algorithm (EA) which is capable of evolving different neural network architectures to find a suitable model for a given dataset $\mathcal{D}$, while being computationally efficient. EAs are chosen for this work since, contrary to the more classical optimization techniques, they do not make any assumptions about the problem, treating it as a black box that merely provides a measure of quality for given a candidate solution. Furthermore, they do not require the gradient, which is impossible to obtain for a neural network ϕ when searching for optimal solutions.

In the following, we describe the very basics of evolutionary algorithms as an introduction for the reader. Further reading can be found in [17, 98, 99].

Every evolutionary algorithm consists of a population of individuals which are potential solutions to the optimization problem such as the one described by the fitness function in Equation (4.5). Each individual in the population is a neural network model. Every individual has a specific genotype or encoding, in the evolutionary algorithm domain, that represents a solution to the given problem while the actual representation of the individual, in the specific application domain, is often referred as the phenotype. For the current application, the phenotype represents the neural network architecture while the genotype is represented by a list of lists. When assessing the quality of an individual, EA makes use of a so-called fitness function, which indicates how every individual in the population performs with respect to a certain performance indicator, establishing thus an absolute order among the various solutions and a way of fairly comparing them against each other.

New generations of solutions are created iteratively by using crossover and mutation operators. The crossover operator is an evolutionary operator used to combine the information of two parents to generate a new offspring while the mutation operator is used to maintain genetic diversity from one generation of the population to the next.

The basic template for an evolutionary algorithm is described in Algorithm 3.

Algorithm 3: Basic evolutionary algorithm.

Data: None
Input : An objective function $f(\mathbf{x})$
Output: A vector $\mathbf{x}^*$ such that $f(\mathbf{x}^*)$ is a local minimum.

Let $t = 0$ be the generation counter. Create and initialize an n_x-dimensional population, $\mathcal{C}(0)$, consisting of n individuals.
while *Stopping condition not true.* **do**
    Evaluate the fitness, $f(\mathbf{x}_i(t))$, of each individual, $\mathbf{x}_i(t)$ in the population.
    Perform reproduction to create offspring.
    Select the new population, $\mathcal{C}(t+1)$.
    Advance to the new generation, i.e. $t = t + 1$.
end

One of the major drawbacks of EAs is the time penalty involved in evaluating the fitness function. If the computation of the fitness function is computationally expensive, as in this case, then using any variant of EA may be very computationally expensive and in some instances unfeasible. Micro-genetic algorithms [100] are one variant of GAs whose main advantage is the use of small populations, for example, less than 10 individuals per population, in contrast to some other EAs like the genetic algorithms (GAs), evolutionary strategies (ES) and genetic programming (GP) [17]. Since computational efficiency is one of our main concerns for this work, we will follow the general principles of micro-GA in order to reduce the computational burden of the proposed method.

The pseudocode for our proposed method is described in Algorithm 4. Let ρ_c and ρ_m be the crossover and mutation probabilities, respectively. Let γ_g be the maximum number of allowed generations and γ_r the maximum number of repetitions for the micro-GA. Finally, let $\mathcal{B}$ be an archive for storing the best architectures found at every run of the micro-GA. Our algorithm Automatic Model Selection (AMS) is stated in Algorithm 4. In the following sections, we describe in detail each one of the major components of the AMS algorithm.

Algorithm 4: Automatic model selection algorithm.

Data: Training/cross-validation dataset $\mathcal{D}_t$, $\mathcal{D}_v$.
Input : Algorithm's hyper-parameters. See Table 4.8 for an example.
Output: The most suitable neural network ϕ^* according to the user preferences.

Let $t_e = 0$ be the experiments counter Create and initialize an n_x-dimensional population, $\mathcal{C}(0)$, consisting of n individuals.
while $t_e < \gamma_r$ **do**
    Let $t_g = 0$ be the generation counter.
    Create and initialize an initial population $\mathcal{C}(0)$, consisting of n individuals, where $n <= 10$. See section 4.4.4.
    while $t_g < \gamma_g$ *or nominal convergence not reached* **do**
        Check for nominal convergence in $\mathcal{C}(t)$. See section 4.4.8.
        Evaluate the cost, $c(\phi)$ of each candidate model f. See section 4.4.2.
        Identify best and worst models in $\mathcal{C}(t)$.
        Replace worst model in $\mathcal{C}(t)$ with best from $\mathcal{C}(t-1)$.
        Perform selection. See section 4.4.5.
        Perform crossover of models in $\mathcal{C}(t)$ with $\rho_c = 1$. Let $\mathcal{O}(t)$ be the offspring population. See section 4.4.6.
        For each model in $\mathcal{O}(t)$ perform mutation with ρ_m probability. See section 4.4.7.
        Make $\mathcal{C}(t+1) = \mathcal{O}(t)$.
        $t_g = t_g + 1$.
    end
    Append best solution from previous run to $\mathcal{B}$.
    $t_e = t_e + 1$.
end
Normalize the cost for each model in the archive $\mathcal{B}$. See section 4.4.2.
Final Solution is best existing solution in $\mathcal{B}$.

4.4.2 The Fitness Function

To establish a ranking among the different tested architectures, a suitable cost or fitness function is required. While Equation (4.2) can be used as the cost function, this would give rise to a multi-objective optimization problem (MOP). We leave this approach for a future revision of this work and instead make use of scalarization to transform the MOP into a single-objective optimization problem (SOP). The scalarization approach taken here is the well known weighted sum method [101]. Equation (4.2) is restated as

$$\min_{\phi \in \mathcal{H}} \quad (1-\alpha)p(\phi) + \alpha w(\phi). \tag{4.3}$$

The cost function associated with Equation (4.3) is

$$c(\phi) = (1 - \alpha)p(\phi) + \alpha w(\phi), \tag{4.4}$$

where $\alpha \in [0, 1]$ is a scaling factor biasing the total cost towards the size of the network or its performance. Equation (4.3) measures the cost in terms of performance and size of a given neural network ϕ.

Note that making an accurate assessment of the inference performance, $p(\phi)$, of a neural network involves training ϕ for a large number of epochs. Since the training process usually involves thousands of computations, training every candidate solution and then assessing its performance becomes unfeasible. Instead, we relax the training process for each of the candidate models by using a *partial training strategy* which, in short, is training the model for a very small number of epochs, for example, only a few tens of them. This approach was been successfully tested in Chapter 3. Even though the models are just partially trained, a clear trend in terms of whether a model is promising or not can be clearly observed.

Computing the fitness of individuals using the current definitions of $p(\phi)$ and $w(\phi)$ present a challenge because the performance indicator $p(\phi)$ and the number of trainable weights $w(\phi)$ are on entirely different scales. While $w(\phi)$ can range from a few hundreds up to several millions, the range of $p(\phi)$ depends on the type of scoring function used. Table 4.2 presents some common ranges for $p(\phi)$.

$Perf(.)$	Range	Common range
Accuracy	$[0, 1]$	$[0, 1]$
Precision	$[0, 1]$	$[0, 1]$
Recall	$[0, 1]$	$[0, 1]$
MSE	$[0, +\infty)$	$[0, 10^4]$
RMSE	$[0, +\infty)$	$[0, 10^2]$

Table 4.2: Common ranges for some neural network performance indicators.

For the present application, it is necessary that the range of $p(\phi)$ is consistent independent of the type of score $Perf(.)$. Thus, we normalize the values of $p(\phi)$ to be within the range $[0, 1]$. The normalization is done as follows. Assume that a population $\mathcal{C}$ has n individuals. Let $\mathbf{p} = [p_0(f), p_1(f), \ldots, p_n(f)]$ be the vector whose components are the scores p_i for the i^{th} element in the population. Then $\mathbf{p}^* = \mathbf{p}/norm_2(\mathbf{p})$ is the vector with the normalized values of $\mathbf{p}$, hence $p_i^* \in [0, 1]$ for any score $Perf(.)$.

Now we focus on $w(\phi)$. For the sake of simplicity, let us just consider the case of the MLP class of neural networks since this is usually the model where $w(\phi)$ is larger. Let $\mathcal{A}$ be the maximum number of possible layers for any model. For

this work, we limit $\mathcal{A} = 64$ since this is a reasonable number for most mainstream deep learning models. Table 4.3 shows that the maximum number of neurons at any layer is set to be 1024. Thus, the maximum number of $w(\phi)$ for any given model is $\mathcal{W} = 2^{26}$. Furthermore, we want neural networks that are similar to have the same score $w(\phi)$. Therefore, we replace the last 3 digits of the $w(\phi)$ with $0's$, and let $w^{+}(\phi)$ be this new size score. Finally, considering that $p^{*}(\phi) \in [0, 1]$, we make $w^{*}(\phi) = log(w^{+}(\phi))$, therefore $\mathcal{W}^{*} \approx log(2) * 26$. Hence, $w^{*}(\phi) \in [0, 7.8]$ which is in the same order of magnitude as $p^{*}(\phi)$.

Thus, we rewrite Equation (4.4) as

$$c(\phi) = 10(1 - \alpha)p^{*}(\phi) + \alpha w^{*}(\phi), \tag{4.5}$$

where we multiply $p^{*}(\phi)$ by a factor of 10 to make the scaling similar to that of $w^{*}(\phi)$. As can be observed, Equation (4.5) is now properly scaled. Therefore, it is a suitable choice as the fitness function for assessing the performance of a neural network model while also considering its size.

4.4.3 Neural Networks as Lists

In order to perform the optimization of neural network architectures, a suitable encoding for the neural networks is needed. A good encoding has to be flexible enough to represent neural network architectures of variable length while also making it easy to verify the *validity* of the candidate neural network architecture.

Array based encodings are quite popular for numerical problems. They often use a fixed-length genotype which is not suitable for representing neural network architectures. While it is possible to use an array based representation for encoding a neural network, this would require the use of very large arrays. Furthermore, verifying the validity of the encoded neural network is hard to accomplish. Tree-based representation as those used in genetic programming [17] enables more flexibility when it comes to the length of the genotype. Imposing constraints for building a valid neural network requires traversing the entire tree or making use of complex data structures every time a new layer is to be stacked in the model.

In this work, we introduce a list-based encoding. In this new list-based encoding, neural network models are represented as a list of arrays, where the length of the list can be arbitrary. Each array within the list represents the details of a neural network layer as described in Table 4.3. A visual depiction of the array is presented in Figure 4.1.

Layer Type	Number of Neurons	Activation Function	Number of Filters	Kernel Size	Kernel Stride	CNN Pooling size	Dropout Rate

Figure 4.1: Visual representation of a neural network layer as an array.

Cell name	Data Type	Represents	Applicable to	Values
Layer type	Integer	The type of layer. See table 4.4	MLP/RNN/CNN	$x \in \{1, \ldots, 5\}$
Neuron number	Integer	Number of neurons/units in the layer	MLP/RNN	$8 * x$ where $x \in \{1, \ldots, 128\}$
Activation function	Integer	Type of activation function. See table 4.5	MLP/RNN/CNN	$x \in \{1, \ldots, 4\}$
Number of filters	Integer	Number of convolution filters	CNN	$8 * x$ where $x \in \{1, \ldots 64\}$
Kernel size	Integer	Size of the convolution kernel	CNN	3^x where $x \in \{1, \ldots, 6\}$
Kernel stride	Integer	Stride used for convolutions	CNN	$x \in \{1, \ldots, 6\}$
Pooling size	Integer	Size for the pooling operator	CNN	2^x where $x \in \{1, \ldots 6\}$
Dropout rate	Float	Dropout rate applied to the layer	MLP/RNN/CNN	$x \in [0, 1]$

Table 4.3: Details of the representation of a neural network layer as an array.

Layer type	Layer name	Can be followed by
1	Fully connected	$[1, 5]$
2	Convolutional	$[1, 2, 3, 5]$
3	Pooling	$[1, 2]$
4	Recurrent	$[1, 4]$
5	Dropout	$[1, 2, 4]$

Table 4.4: Neural network stacking/building rules.

Index	Activation function
0	Sigmoid
1	Hyperbolic tangent
2	ReLU
3	Softmax
4	Linear

Table 4.5: Available activation functions.

The proposed representation is capable of handling different types of neural network architectures. In principle, the representation can handle multi-layer perceptrons (MLPs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). For any given neural network layer, the array will only contain values for the entries that are applicable to the layer, values for other types of layers are set to 0.

Let us illustrate the proposed encoding with an example. The following example considers an MLP. Layer type, Number of neurons, Activation function and Dropout rate entries are applicable values. Consider S_e as a model made up of several stacked layers as shown in Figure 4.1. The neural network representation of the presented model is shown in Table 4.6.

$$S_e = [[1, 264, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.65],$$
$$[1, 464, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.35],$$
$$[1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]]$$

Layer Type	Neurons	Activation Function	Dropout Ratio
Fully Connected	264	ReLU	n/a
Dropout	n/a	n/a	0.65
Fully Connected	464	ReLU	n/a
Dropout	n/a	n/a	0.35
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

Table 4.6: Neural network model.

Encoding the neural network as a list of arrays has two advantages. First, the number of layers that can be stacked is, in principle, arbitrary. Second, the validity of an architecture can be verified every time a new layer is to be stacked to the model. This is due to the fact that in order to stack a layer to the model, one only needs to check the compatibility between the previous and next layers. The ability of stacking layers dynamically and verifying its correctness when a new layer is stacked allows for a powerful representation. We can build several kinds of neural networks such as fully connected, convolutional and recursive. The rules for stacking layers together are described in Table 4.4.

4.4.4 Generating Valid Models

With the rules for stacking layers together in place, generating valid models is straightforward. An initial layer type has to be specified by the user, which can be fully connected, convolutional or recurrent. Defining the initial layer type effectively defines the type of architectures that can be generated by the algorithm. That is, if the user chooses fully connected as the initial layer, all the generated models will be fully connected. If the user chooses convolutional as initial layer, the algorithm will generate convolutional models only and so on.

Just as the initial layer type has to be user-defined, the final or output layer is also user-defined. In fact, all the generated models share the same output layer. The output layer is always a fully connected layer. Furthermore, it is generated based on the type of problem to solve, i.e. classification or regression. In the case of classification, the number of neurons is defined by the number of classes in the

problem and the softmax function is used as activation function. For regression problems, the number of neurons is one and the activation function used is the linear function.

Having defined the architecture type and the output layer, generating an initial model is an iterative process of stacking new layers that comply with the rules in Table 4.4. A user-defined parameter γ_l is used to stop inserting new layers. Every time a new layer is stacked in the model, a random number $\psi \in [0, 1]$ is generated using the following probability distribution

$$\rho_l = 1 - \sqrt{1 - U}, \tag{4.6}$$

where U is a uniformly distributed random number. If $\rho_l < \gamma_l$ and if the current layer is compatible with the last layer according to Table 4.4, then no more layers are inserted. Equation (4.6) is used to let the user choose the probability with which more layers are stacked to a neural network model. Thus, if the user wants that a new layer is inserted with 80% probability, the user must choose $\gamma_l = 0.8$.

With regards to layers that have an activation function, even though in principle any valid activation functions are allowed, for this application we choose to keep the same activations for similar layers across the model since this is usually the common practice.

4.4.5 Selection

In order to generate n offsprings, $2n$ parents are required. The parents are chosen using a selection mechanism which takes the population $\mathcal{C}(t)$ at the current generation and returns a list of parents for crossover. For our application, the selection mechanism used is based on the binary tournament selection [17, 100]. A description of the mechanism is given next.

- Select m parents at random where $m < n$.
- Compare the selected elements in a pair-wise manner and return the most fit individuals.
- Repeat the procedure until $2n$ parents are selected.

It is important to note in the above procedure that the larger m is, the higher the probability that the best individual in the population is chosen as one of the parents, this is not a desirable behavior, thus we warn the users to keep m small. Also, since our algorithm uses elitism best individual of a current generation remains unchanged in the next generation.

4.4.6 Crossover Operator

Since the encoding chosen for this task is rather peculiar, the existing operators are not suitable for it. We design a new crossover operator. In this section, we describe in detail the proposed crossover operator. The operator is based on the two point crossover operator for genetic algorithms [102] in the sense that two points are selected for each parent. The operator is more restrictive as to which pairs of points may be selected in order to ensure the generation of valid architectures.

The key concept behind our crossover operator is that of "compatibility between pairs of points". Consider two models S_1 and S_2 that will serve as parents for one offspring. Assume that the offspring will be generated by replacing some layers in S_1 from some layers in S_2. S_1 is thus the base parent. If we select any two pairs of points (r_1, r_2) from S_1 and (r_3, r_4) from S_4, it may happen that such pairs of points cannot be interchangeable because layer r_3 cannot be placed instead of layer r_1 or layer r_4 cannot be placed instead of layer r_2. Therefore, the selection mechanism must ensure that the interchange points, $(r_1, r_2), (r_3, r_4)$, are compatible. That is to say, layer r_3 is compatible with the layer preceding r_1 and the layer after r_2 is compatible with layer r_4. Compatibility is defined in terms of the rules described in Table 4.4. A selection mechanism that guarantees the compatibility between pairs of points is described in Algorithm 5. This method assume that the offspring will be generated by replacing some layers in S_1 from some layers in S_2.

Algorithm 5: Crossover method.

Data: None
Input : Two neural network string representations S_1 and S_2.
Output: A neural network string representation S_3.

Let S_1 and S_2 be the arrays containing the stacked layers of a neural network model in parents 1 and 2, respectively. Take two random points (r_1, r_2) from S_1 where $r_1 <= r_2$.
if $r_1 = r_2$ **then**
  $r_2 = len(S_1 - 1)$
end
else
  pass
end
Find all the pairs of points $(r_3, r_4)_i$ in S_2 that are compatible with (r_1, r_2) where $r_3 < r_4$ and $(r_4 - r_3) - (r_2 - r_1) < \mathcal{A}$.
Randomly pick any of pairs $(r_3, r_4)_i$.
Replace the layers in S_1 between r_1, r_2 inclusive with the layers in S_2 between r_3, r_4 inclusive. Label the new model as S_3.
Rectify the activation functions of S_3 to match the activation functions of S_1.

It is possible that the mechanism described in Algorithm 5 requires more than one attempt to find valid interchange points (r_1, r_2) and (r_3, r_4) for models S_1 and S_2. Based on our experience with the method and the obtained results, Algorithm 5 usually requires only one attempt to successfully generate a valid offspring. To prevent the crossover mechanism from getting trapped in an infinite loop, we limit the number of trials to γ_c where $\gamma_c = 3$ is the default and can be adjusted by the user. Let us illustrate Algorithm 5 with an example. Consider the following models

$$\begin{aligned} S_1 = [&[1, 264, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.65], \\ &[1, 464, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.35], \\ &[1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]] \end{aligned}$$

$$\begin{aligned} S_2 = [&[1, 56, 0, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.25], \\ &[1, 360, 0, 0, 0, 0, 0, 0], [1, 480, 0, 0, 0, 0, 0, 0] \\ &[1, 88, 0, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.2], \\ &[1, 10, 3, 0, 0, 0, 0, 0]] \end{aligned}$$

Let us take $r_1 = 1$ and $r_2 = 3$, since these points are going to be removed from the model we need to find the compatible layers with $S_1[r_1 - 1]$ and $S_1[r_2]$ according to the rules described in Table 4.4. Note that if $r_1 = 0$, i.e. the initial layer, only a layer whose layer type is equal to the layer type of $S_1[0]$ is compatible. Thus, for this example, the compatible pairs of points $(r_3, r_4)_i$ are

$$\begin{aligned} &[(0,0), (0,2), (0,4), (0,5), (1,2), (1,4), (1,5), \\ &(2,2), (2,4), (2,5), (4,4), (4,5), (5,5)] \end{aligned}$$

Assume that we pick at random the pair $(2, 4)$. The offspring, which we will call S_3, looks like

$$\begin{aligned} S_3 = [&[1, 264, 2, 0, 0, 0, 0, 0], [1, 360, 2, 0, 0, 0, 0, 0], \\ &[1, 480, 2, 0, 0, 0, 0, 0], [1, 88, 2, 0, 0, 0, 0, 0], \\ &[1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]] \end{aligned}$$

which is a valid model. The reader is encouraged to check the actual neural network representations for each of the models in Appendix B. Notice that all the activation functions of the same layer types are changed to match the activation functions of the first parent S_1. We call this process "activation function rectification". It basically implies changing all the activation functions of the layers that share the same layer type between S_1 and S_3 to the activation functions used in S_1.

Finally, one important feature of this crossover operator is that it has the ability to generate neural network models of different sizes, i.e. it can shrink or increase the size of the base parent. This behavior mimics that of machine learning practitioners, which will often start with a base model and iteratively shrink or increase the size of the base model in order to find the one that has the best inference performance.

4.4.7 Mutation Operator

The mutation operator is used to induce small changes to some of the models generated through the crossover mechanism. In the context of evolutionary computation, these subtle changes tend to improve the exploration properties of the current population, i.e. to keep genetic diversity, by injecting random noise to the current solutions. Although mutation is not needed in the micro-GA according to [100], we believe some sort of mutation is beneficial for our application to get more diverse models which could potentially lead to better inference abilities. Nevertheless, our mutation approach will be less aggressive in order to mitigate its effect. In the following, we discuss the core ideas of our mutation mechanism.

As stated above, our mutation approach is less aggressive than common mutation operators [17]. Our design follows two main reasons: First, the fact that usually micro genetic algorithms don't make use of the mutation operator since the crossover operator has already induced significant genetic diversity in the population, thus we want to minimize its impact. The second reason is related to the way neural networks are usually built by human experts. Usually, experts try a number of models and then make subtle changes to each of them in order to improve their inference ability. Such changes usually involve adjusting the parameters in a layer, adding or removing a layer, adding regularization or changing the activation functions. Based on these reasons, our mutation process randomly chooses one layer of the model, using a probability ρ_m, and then proceeds to make one of the following changes to it:

- Change a parameter of the layer chosen for a value complying the values stated in Table 4.3.
- Change the activation function of the layer. This would involve rectifying the entire model as described in Section 4.4.6.
- Add a dropout layer if the chosen layer is compatible.

These operations altogether provide a rich set of possibilities for performing efficient mutation while still generating valid models after mutation is performed. Furthermore, the original model is barely changed.

4.4.8 Nominal Convergence

Nominal convergence is one of the criteria used for early stopping of the proposed evolutionary algorithm. Some literature defines the convergence in terms of the fitness of the individuals [17], while in [100] the convergence is defined in terms of the genotype or phenotype of the individuals. Although convergence based on the actual fitness of the individuals may be easier to assess given that the fitness is already calculated, we believe that an assessment of convergence based on the actual genotype of the individuals suits the current needs better.

Since neural networks are stochastic in nature, we expect some variations in the fitness of the individuals at every different run. Furthermore since we are running the training process for only few epochs, the final performance of the networks can be quite different and would not be a reliable indicator of convergence. Instead, to assess the convergence we, look at the genotype of the actual neural network architecture and compute the layer-wise distance between the different individuals in population $\mathcal{C}$.

Let S_1 and S_2 be the genotypes representing any two different models such that where $len(S_2) >= len(S_1)$. Let $S_1[j]$ be the vector representation of the j^{th} layer of model S_1. The method for computing the distance $d(S_1, S_2)$ between any two models S_1 and S_2 is defined in Algorithm 6.

Algorithm 6: Layer-wise distance $d(S_1, S_2)$ between model genotypes.

Data: None
Input : Two neural network string representations S_1 and S_2.
Output: Distance between models S_1 and S_2

Let $\mathbf{d} \in \mathbb{R}$ be the distance between S_1 and S_2. Make $\mathbf{d} = 0$.
for *Each layer i in* S_1 *except last layer* **do**
  $d = d + norm_2(S_2[i] - S_1[i])$.
end
for *Each remaining layer i in* S_2 *except last layer* **do**
  $d = d + norm_2(S_2[i])$.
end
Return $\mathbf{d}$.

This method is computationally inexpensive since the size of the population is small. Furthermore, it helps accurately establish the similarity between two neural network models. Given two neural network models S_1 and S_2, if $d(S_1, S_2) = 0$, then $S_1 = S_2$. We say that AMS (Algorithm 4) has reached a nominal convergence if at least m_c pairs of models have $d(S_1, S_2) \leq d_t$, where both m_c and d_t are user defined parameters.

4.4.9 Implementation

AMS is implemented in about 700 lines of code in Python. The code can be found in `https://github.com/dlaredo/automatic_model_selection` [103]. We took an object oriented programming approach for its implementation. The models S_i generated by the algorithm are fetched to Keras [79] and then evaluated. The models can be fetched and evaluated in any other framework such as TensorFlow or Pytorch and even other programming languages including C++.

In order to boost performance of AMS, we make use of Ray [96] which is a distributed computing framework tailored for AI applications. In order to distribute workloads in Ray, developers only have to define Remote Functions by making use of Python annotations. Ray will then distribute these Remote Functions across the different nodes in the cluster. There are three different types of nodes in Ray: Drivers, Workers and Actors. A Driver is the process executing the user program, a Worker is a stateless process that executes remote functions invoked by a driver or another worker, and finally, an Actor is a stateful process that executes, when invoked, the methods it exposes.

For the implementation, we code the individual fetching to Keras and its fitness evaluation as Ray Remote Functions, i.e. Workers, while the rest of the algorithm is implemented within the Driver. Partial training of each neural network within the current population can therefore be performed in a distributed way, leading to a highly increased performance of the algorithm. Furthermore, since the only messages being sent over the cluster are arrays of the neural network representation S_i and the performance p of the neural network model, there is little chance that the interchange of data causes a bottleneck or increases latency in the system. Nevertheless, each node in the cluster has to maintain a local copy of the dataset.

4.5 Evaluation

We evaluate AMS with two different datasets, each of which represents a different type of inference problem. We also compare our results with the state-of-the-art neural network models for each problem. For the experiments in this section, each model generated by AMS is trained using the following parameters.

Dataset	Epochs	Learning Rate	Optimizer	Loss Function	Metrics
MNIST	5	0.001	Adam	Categorical cross-entropy	Categorical accuracy
CMAPSS	5	0.01	Adam	MSE	MSE

Table 4.7: Training parameters for each of the used datasets.

For the CMAPSS dataset, we use a larger learning rate since we intend to evaluate the model using very few epochs for this complex problem. In order to get

a clear idea of which individuals within the population may be promising, we make the learning process more aggressive during the first iterations of the algorithm.

All of the experiments were run using the Keras/Tensorflow GPU framework. A desktop PC with a Core i7-8700k processor and an NVIDIA GeForce 1080Ti GPU. Ray was not used for the results presented here.

4.5.1 MNIST Dataset - A Classification Problem

We first test our algorithm on the MNIST dataset [48]. The MNIST dataset of handwritten digits is one of the most commonly used datasets for evaluating the performance of neural networks. It has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image, while the size of each image is of 28x28 pixels. As a part of the data pre-processing, we normalize all the pixels in each image to be in the range of $[0, 1]$ and unroll the 28x28 image into a vector with 784 components.

We use MNIST dataset as a baseline for measuring the performance of the proposed approach. Furthermore, we use MNIST to analyze each one of the major components of AMS. Given the popularity of MNIST, several neural networks with varying degrees of accuracy have been proposed in the literature. Therefore, it is easy to find good models to compare with.

We start by running AMS to find a suitable fully connected model for classification of the MNIST dataset. The details for the parameters used in this test are described in Table 4.8. Each of the experiments carried out by AMS takes about 4 minutes in our test computer.

Parameter	AMS Value
Problem Type	1
Architecture Type	FullyConnected
Input Shape	$(784, M)$
Output Shape	$(10, M)$
Cross Validation Ratio	$\gamma_v = 0.2$
Mutation Probability	$\rho_m = 0.4$
More Layers Probability	$\gamma_l = 0.4$
Network Size Scaling Factor	$\alpha = 0.5$
Population Size	$n = 10$
Tournament Size	$n_t = 4$
Max Similar Models	$\gamma_c = 3$
Training epochs	$\gamma_t = 5$
Max generations	$\gamma_g = 10$
Total Experiments	$\gamma_r = 5$

Table 4.8: AMS Parameters for MNIST dataset.

We first take a look at the generated initial population. For the sake of space, we will only discuss the sizes of the models. Furthermore, we make a small

change in our notation for describing neural network models. For the remainder of this section, we denote a layer of a neural network as (n_e, a_f) where n_e denotes the number of neurons for fully connected layer or the dropout rate for a dropout layer, and a_f denotes the activation function of the fully connected layer. The initial five generated individuals are presented below. Fitness, accuracy and raw size of the models in the initial population are presented in Table 4.9.

$$
\begin{aligned}
S_1 &= [(64, 0), (0.4), (10, 3)] \\
S_2 &= [(760, 2), (0.5), (608, 2), (0.65), (10, 3)] \\
S_3 &= [(864, 0), (0.15), (536, 0), (928, 0), (10, 3)] \\
S_4 &= [(40, 0), (0.45), (912, 0), (10, 3)] \\
S_5 &= [(968, 1), (976, 1), (32, 1), (0.15), (808, 1), \\
&\quad (10, 3), (832, 2)]
\end{aligned}
$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1	91.7%	50890	2.7688
S_2	98.7%	1065378	3.0812
S_3	94.6%	1649506	3.3791
S_4	92.1%	77922	2.8427
S_5	96.6%	1771642	3.2867

Table 4.9: Scores for the initial population found by AMS for MNIST. $\alpha = 0.5$

Observe that the sizes of the models in the initial population are diverse with some models having as few as 1 hidden layer and some having more than 5 hidden layers. The number of layers of the models in the initial population is defined by the parameter n_r. We set this value to be small on purpose, since MNIST is a dataset that is easy even for simple neural network models. Also note that the initial population has models with different activation functions. The Sigmoid, Tanh and ReLU are all used by some models. This is beneficial to the search process as some activation functions may yield better results than others.

Finally, we note that some models in the initial population already yield decent accuracy (about 90%). They also have a large number of trainable parameters. It is decided that, in the case of MNIST dataset, the task for AMS is to find a model with an accuracy higher than 95% and a small number of trainable parameters.

For a value of the network size scaling factor $\alpha = 0.5$ in Equation (4.4), after 5 experiments and 10 generations for each experiment, AMS converges to the following five models. As a side note, for all of the experiments in this section we

denote the best model found at experiment $i \in \{1, \cdots, n\}$ as S_i^* and the best model out of n experiments for a given α as S_i^+. The fitness, accuracy and raw size of the models are presented in Table 4.10.

$$
\begin{aligned}
S_1^* &= [(56, 2), (10, 3)] \\
S_2^* &= [(168, 2), (10, 3)] \\
S_3^* &= [(40, 2), (0.2), (10, 3)] \\
S_4^+ &= [(48, 2), (48, 2), (10, 3)] \\
S_5^* &= [(312, 2), (10, 3)]
\end{aligned}
$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	93.9%	44530	2.6287
S_2^*	95.8%	133570	2.7736
S_3^*	93.4%	31810	2.5826
S_4^+	94.7%	40522	2.5693
S_5^*	95.0%	248050	2.9431

Table 4.10: Scores for the best models found by AMS for MNIST. $\alpha = 0.5$

Table 4.10 shows a clear preference for small models. Furthermore, there seems to be a preference for ReLU activation functions. It can also be observed that for $\alpha = 0.5$, a good balance between size of the network and its performance is obtained. In the following, we perform tests with $\alpha = 0.3$ and $\alpha = 0.7$ to further analyze the behavior of the algorithm with respect to the preference of the network size scaling factor. A smaller α value will prioritize a better performing network while a larger value of α instructs AMS to focus on light-weight models.

The best models for each experiment with $\alpha = 0.3$ are listed below. The fitness and raw size of the models are described in Table 4.11.

$$
\begin{aligned}
S_1^* &= [(32, 0), (10, 3)] \\
S_2^* &= [(32, 1), (32, 1), (10, 3)] \\
S_3^+ &= [(64, 1), (64, 1), (64, 1), (64, 1), (10, 3)] \\
S_4^* &= [(56, 1), (10, 3)] \\
S_5^* &= [(40, 1), (10, 3)]
\end{aligned}
$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	91.5%	25450	1.9126
S_2^*	95.2%	26506	1.6677
S_3^+	97.2%	63370	1.6358
S_4^*	94.7%	44530	1.7640
S_5^*	94.4%	31810	1.7412

Table 4.11: Scores for the best models found by AMS for MNIST, $\alpha = 0.3$.

The models presented in Table 4.11 exhibit, in general, better performance than those depicted in Table 4.10. This is due to the fact that $\alpha = 0.3$ prioritizes the model accuracy over model size. Surprisingly, the sizes of the models obtained when $\alpha = 0.3$ are on the same order of magnitude as those obtained when $\alpha = 0.5$. A discussion on this behavior is provided later in this section.

We repeat the experiment with $\alpha = 0.7$. The obtained models are listed below and their fitness, raw size and accuracy are shown in Table 4.12.

$$
\begin{aligned}
S_1^* &= [(24, 2), (10, 3)] \\
S_2^* &= [(104, 2), (0.2), (10, 3)] \\
S_3^+ &= [(16, 2), (24, 2), (10, 3)] \\
S_4^* &= [(56, 2), (10, 3)] \\
S_5^* &= [(208, 2), (10, 3)]
\end{aligned}
$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	92.2%	19090	3.2284
S_2^*	95.2%	82690	3.5856
S_3^+	92.1%	13218	3.1158
S_4^*	94.5%	44530	3.4200
S_5^*	95.8%	165370	3.7762

Table 4.12: Scores for the best models found by AMS for MNIST, $\alpha = 0.7$.

The results in Table 4.12 show that when the number of trainable parameters has a large impact on the overall fitness of the individuals, and the algorithm tends to prefer smaller networks, which is specially useful for cases where the computational power is limited, such as embedded systems. This brings a dilemma, namely that neural networks that exhibit a lower performance as compared to larger neural

networks may be preferred. Nevertheless, this trade-off can be controlled by the user by varying the α parameter.

As a side note, we point out that the difference in the fitness exhibited among the models of the three different experiments is due to the fact that the size of the neural network is scaled as described in Section 4.4.2. Thus, there is no fair way to compare the fitness of the models shown in Tables 4.10, 4.11 and 4.12 against each other. We are clearly dealing with a multi-objective optimization problem with conflicting objective functions.

Figure 4.2 shows the results obtained by all of the models for $\alpha \in \{0.3, 0.5, 0.7\}$, which are trained for 100 epochs using 5-fold cross-validation to assess their accuracy. It is observed that the models cluster around a model size less than 50000 and an error less than 0.15. As expected, the models obtained with $\alpha = 0.7$ yield, in general, the smallest sizes. Outliers are mostly due to the fact that for such experiments the algorithm is unable to find a smaller model, which is likely due to a bad initial population for such experiments. It could also be attributable to the scaling done to the network size as seen in Equation (4.5). Since AMS uses a logarithmic scale to measure the size of the networks, a few thousands of weights are unlikely to make a big difference in the fitness of a model.

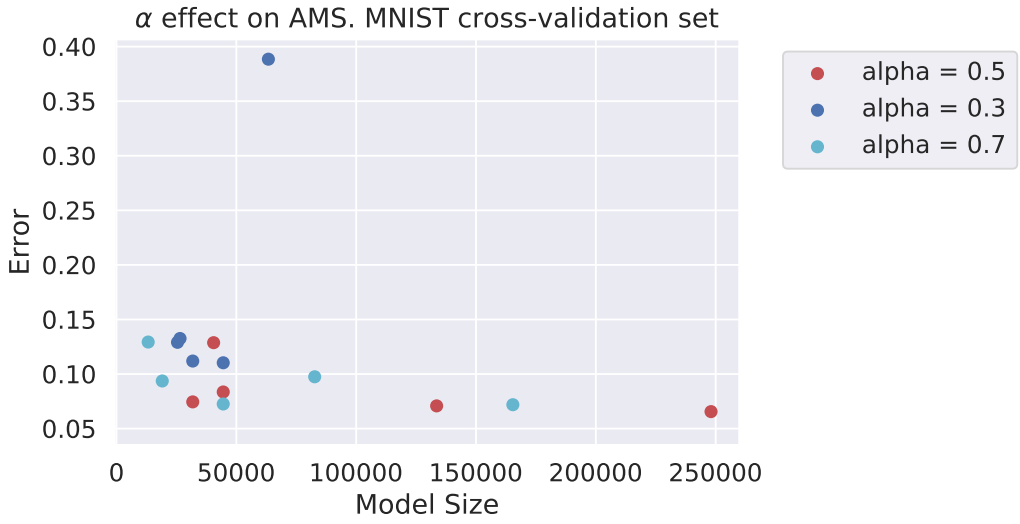


Figure 4.2: Cluster formed by the found models for different α values for MNIST cross-validation set.

Finally, we compare the best models for each value of α against each other. A 10-fold cross-validation process, with a training of 50 epochs per fold, is carried out for each one of the best models in order to obtain the mean measure of accuracy for each model. The three models are feed the same training data and are validated

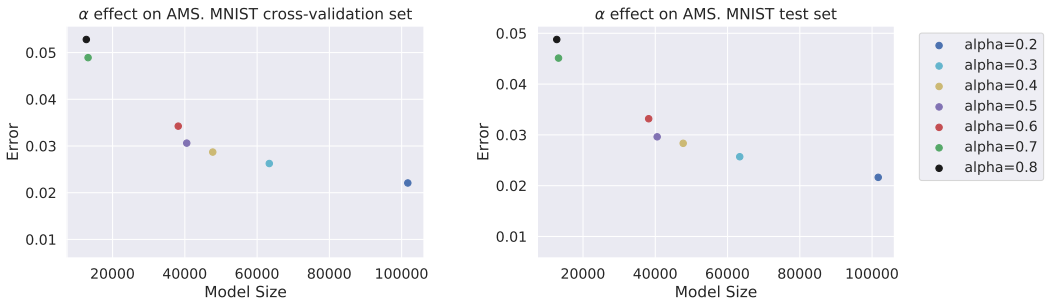
using the same folds for the cross-validation data. We also measure the accuracy of each models by using a test set that is never used during the training or hyper-parameter tuning processes of the models. The accuracy averages and size of the networks are summarized in Table 4.13.

Model	5-fold Avg. Score	Test Accuracy	Network size
$S^+_{0.3}$	97.3%	97.4%	63370
$S^+_{0.5}$	96.8%	97.0%	40522
$S^+_{0.7}$	95.0%	95.4%	13218

Table 4.13: Accuracy obtained by each of the top 3 models for MNIST dataset.

As expected, the model obtained for $\alpha = 0.7$ yields the smallest neural network, about 4 times smaller than the model obtained when $\alpha = 0.3$. Nevertheless, its accuracy is the worst of the three models, though only by a small margin. On the opposite, the resulting model for $\alpha = 0.3$ gets the best performance in terms of accuracy, but also attains the largest neural network model of the three. Finally, when $\alpha = 0.5$ AMS yields a model with a good balance between the model size and performance. It is important to highlight that to obtain the models presented in Table 4.13, at most 50 different models are tried. Nevertheless, since each model is trained for only 5 epochs, the total time spent by the algorithm to find the models in Table 4.13 is less than 4 minutes in our test computer.

Figure 4.3 plots the size of the model vs. the error. As can be observed, the resulting models for different α values form a so called Pareto front [28], i.e., none of the resulting models is better than the others in terms of the size and error. The trade-off between the size and performance of the model can be clearly seen in the figure. Although this is a nice property exhibited by the MNIST dataset, this need not always hold since it is known that the performance of a neural networks does not monotonically increase with the size of the model.



(a) Obtained models on cv set

(b) Obtained models on test set

Figure 4.3: Influence of α on the model's size and error on MNIST dataset

Table 4.14 shows three of the top hand-crafted models along with their obtained accuracy for the MNIST dataset, it can be observed that the accuracy obtained by AMS is close to that obtained by the fine tuned models.

Method	Test accuracy	Size
3 Layer NN, 300+100 hidden units [48]	96.95%	266610
2 Layer NN, 800 hidden units [104]	98.4%	636010
6-layer NN (elastic distortions) [22]	99.65%	11972510

Table 4.14: Top results for MNIST dataset.

By comparing the models obtained by AMS in Table 4.13 against the models in Table 4.14 we can observe that, even though AMS models don't attain the highest accuracy, they exhibit good inference capabilities with a much lesser number of trainable parameters, at least one order of magnitude smaller. This shows that AMS models have a good balance between the inference power of the model and its size. Furthermore, the score-size trade-off can be controlled by means of the parameter α, where a value closer to $\alpha = 0$ makes AMS prefer networks with higher accuracy and a value closer to $\alpha = 1$ makes AMS prefer networks with smaller sizes.

4.5.2 CMAPSS Dataset - A Regression Problem

Here we analyze the performance of AMS when dealing with regression problems. For testing regression, we use the C-MAPSS dataset [54]. The C-MAPSS dataset contains the data produced using a model based simulation program developed by NASA. A description of the C-MAPSS dataset was given in Section 3.2.

For this test, we follow the data pre-processing described in Section 3.3.2. Only 14 out of the total 21 sensors are used as the input data. Furthermore, we also use a strided time-window with window size of 24, a stride of 1 and early RUL of 129, to form the feature vectors for the MLP. Further details of the time-window approach can be found in Chapter 3.

We run AMS to find a suitable MLP for regression using the C-MAPSS dataset. The parameters used by the algorithm are described in Table 4.15.

Parameter	AMS Value
Problem Type	1
Architecture Type	FullyConnected
Input Shape	$(336, M)$
Output Shape	$(1, M)$
Cross Validation Ratio	$\gamma_v = 0.2$
Mutation Probability	$\rho_m = 0.4$
More Layers Probability	$\gamma_l = 0.7$
Network Size Scaling Factor	$\alpha = 0.8$
Population Size	$n = 10$
Tournament Size	$n_t = 4$
Max Similar Models	$\gamma_c = 3$
Training epochs	$\gamma_t = 20$
Max generations	$\gamma_g = 10$
Total Experiments	$\gamma_r = 5$

Table 4.15: Parameters for the CMAPSS dataset.

We perform experiments for $\alpha \in \{0.3, \cdots, 0.7\}$ with an increment $\Delta_\alpha = 0.1$. For the sake of space, we only discuss the results obtained when $\alpha \in \{0.4, 0.5, 0.6\}$, which are the α values giving the best results. The best models out of the five experiments obtained for each of the α values by AMS are listed in Table 4.16 along with their RMSE scores and sizes. As with the MNIST dataset, AMS partially evaluates at most 50 different models for each α value. The experiments for each α take about 2 minutes in our test computer.

$$S^+_{0.4} = [(104, 1), (824, 1), (1, 4)]$$
$$S^+_{0.5} = [(264, 2), (1, 4)]$$
$$S^+_{0.6} = [(80, 1), (80, 1), (1, 4)]$$

Model	5-fold Avg. Score	Test RMSE	Network Size
$S^+_{0.4}$	14.87	14.99	122393
$S^+_{0.5}$	15.27	15.90	89233
$S^+_{0.6}$	14.78	15.74	33521

Table 4.16: RMSE of the top 3 models for the CMAPSS dataset.

The results presented in Table 4.16 further demonstrate the impact of the α parameter. As can be observed, the size of the networks grows as α is smaller. It can also be observed that the results obtained by the three proposed models in the cross-validation sets are very close to each other. Again, small networks are usually preferred. Table 4.17 presents some of the top results obtained by the hand-crafted

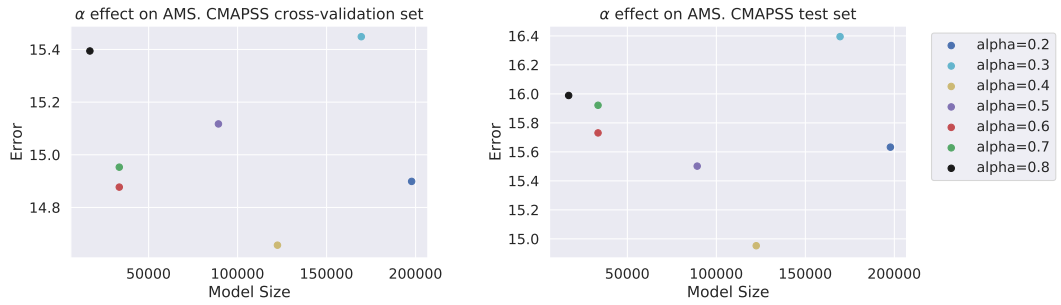
MLPs for the CMAPSS dataset. The performance of the models is measured in terms of the root mean squared error (RMSE) between the real and predicted RUL values.

Method	Test RMSE	Network Size
Time Window MLP [9]	15.16	6041
Time Window MLP with EA [70]	14.39	7161
Deep MLP Ensemble [76]	15.04	n/a

Table 4.17: Top results for the CMAPSS dataset.

The obtained models are also competitive when compared against some of the latest MLPs designed for the CMAPSS dataset as shown in Table 4.17. We compare the score obtained in the test set for all the models. It is shown that one of the three models obtained by AMS produces a better score than the two of the three compared models, i.e. Time Window MLP and Deep MLP ensemble. Although the sizes of the neural networks obtained by AMS are one order of magnitude bigger than the hand-crafted models, we can observe that AMS delivered compact models (having few layers with few neurons in each layer).

Finally, Figure 4.4 shows that all of the obtained models have a small model size with few hundred thousand parameters while all of them deliver a very good performance for this dataset. See references [70] and [10] for other models and their scores. Once again, it can be observed that the model size decreases with larger α values. One important observation is that, for the CMAPSS dataset, there does not seem to be a correlation between the model size and its performance. Indeed, the model size is just one simple indicator of a networks architecture. Trying to characterize a network only by its size may leave out valuable information about it. The information could be used in the search of new individuals with better traits for the dataset. We leave out further analysis of this behavior for future work.



(a) Obtained models on cv set

(b) Obtained models on test set

Figure 4.4: Influence of α on the model's size and error on CMAPSS dataset

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Concluding Remarks

We presented two methods for the application of deep learning. Though the methods are of general application, they were inspired by some common problems encountered by practitioners who are not experts in deep learning. In mechanical engineering the presented methods are applicable to a variety of problems; fault detection, prognostics, and control are some of the potential applications for the methods developed in this thesis.

The presented framework for the estimation of the remaining useful life of jet-engines is applicable to any kind of mechanical system so long as data describing the behavior of the system is available. The use of a strided moving time window to form a feature vector instead of independent samples is a powerful technique that better captures the dynamic behavior of the system. Nevertheless, the use of the strided time-window introduces additional parameters that have to be tuned by the practitioner.

Coupling the time-window strategy with a simple, yet efficient, evolutionary algorithm allows for practitioners with little experience to get results out of the box. Though the framework comes packed with a shallow MLP to make the inference, which proved to be enough in the experiments thanks to the data treatment with the time-window, the end user can easily change the model for one that better suits its needs. In fact, the user need not use a deep learning model at all, having the possibility to choose from some more conventional models as polynomial regression or SVMs. Furthermore, the compactness of the default model makes the framework suitable for applications that have limited computational resources such as embedded systems. It is important to note that the framework itself does involve some computations, nevertheless such computations are done off-line.

This thesis also presents AMS, a new evolutionary algorithm for efficiently finding suitable neural network models for classification and regression problems. Making use of efficient mutation and crossover operators, AMS is able to generate valid and efficient neural networks, in terms of both the size of the network and its performance. Furthermore, AMS design is highly parallelizable and distributable. With the use of frameworks such as Ray [96] or Spark [105], the performance of the algorithm can be greatly boosted.

By allowing the user to control the trade-off between the total size of the network and its performance, AMS is capable of finding small neural networks for applications with limited memory, mobiles or embedded systems that may have constraints on the size of the models or prioritizing the model performance. Although not every model found by AMS is a Pareto point, the found models yield a good balance between the performance and size.

Furthermore, AMS is computationally efficient since it only needs to evaluate a few tens of models to find suitable ones. As demonstrated in this Chapter 4, even a medium tier computing rig consisting of a modern, medium-range processor and a general purpose GPU can find good models in less than 5 minutes depending on the dataset. This is achievable mainly due to the partial train strategy, which is demonstrated to be an efficient method for assessing the fitness of a given model.

Overall, AMS provides an easy to use, efficient and robust algorithm for finding suitable neural network models given a dataset. We believe that the method can be easily used by somebody who has a basic knowledge of programming, making it possible for non-expert machine learning practitioners to obtain out-of-the-box solutions.

5.2 Future Work

Two major features of the framework for estimating remaining useful life are its generality and scalability. While for this study, specific regressors and evolutionary algorithms are chosen, many other combinations are possible and may be more suitable for different applications. Future work for the framework will consider the use of different machine learning algorithms. An analysis of the influence of the window stride parameter n_s is also considered for future work.

In the case of AMS future work will consider more complex neural network architectures such as RNNs and CNNs. Techniques for assembling entire neural network pipelines will also be explored in the future as well as the inclusion of more hyper-parameters in the tuning process. An analysis of what information can be used to better characterize a neural network is also left for future work. Finally, a better way of measuring distance between two neural network architectures will be explored, since this element is of high importance for applications such as visualization and evolutionary computation.

Finally, a graphical user interface that makes the algorithms easier to use is also considered for future versions of the presented methods.

BIBLIOGRAPHY

[1] A. K. Noor, AI and the future of the machine design, Mechanical Engineering Magazine Select Articles 139 (10) (2017) 38–43.

[2] S. G. Khan, G. Herrmann, F. L. Lewis, T. Pipe, C. Melhuish, Reinforcement learning and optimal adaptive control: An overview and implementation examples, Annual Reviews in Control 36 (1) (2012) 42–59.

[3] L. Busoniu, T. De Bruin, D. Tolic, J. Kober, I. Palunko, Reinforcement learning for control: Performance, stability, and deep approximators, Annual Reviews in Control 46 (2018) 8–28.

[4] D. Gorges, Relations between model predictive control and reinforcement learning, in: Elsevier (Ed.), 20th IFAC World Congress, 2017, pp. 4920–4928.

[5] S. Bahrampour, B. Moshiri, K. Salahshoor, Weighted and constrained possibilistic c-means clustering for online fault detection and isolation, Applied Intelligence 35 (2) (2011) 269–284.

[6] A. Papadimitropoulos, G. A. Rovithakis, T. Parisini, Fault detection in mechanical systems with friction phenomena: An online neural approximation approach, IEEE Transactions on Neural Networks 18 (4) (2007) 1067–1082.

[7] J. Liang, R. Du, Model-based fault detection and diagnosis of HVAC systems using support vector machine method, International Journal of Refrigeration 30 (6) (2007) 1104 – 1114.

[8] G. S. Babu, P. Zhao, X. Li, Deep convolutional neural network based regression approach for estimation of remaining useful life, in: S. I. Publishing (Ed.), 21st International Conference on Database Systems for Advanced Applications, 2016, pp. 214–228.

[9] P. Lim, C. K. Goh, K. C. Tan, A time-window neural networks based framework for remaining useful life estimation, in: Proceedings International Joint Conference on Neural Networks, 2016, pp. 1746–1753.

[10] X. Li, Q. Ding, J. Sun, Remaining useful life estimation in prognostics using deep convolution neural networks, Reliability Engineering and System Safety 172 (2018) 1–11.

[11] D. Pandya, B. Dennis, R. Russell, A computational fluid dynamics based artificial neural network model to predict solid particle erosion, Wear 378 (2017) 198–210.

[12] M. McCracken, Artificial neural networks in fluid dynamics: A novel approach to the navier-stokes equations, arXiv:1808.06604 (2018).

[13] C. Thornton, F. Hutter, H. H. Hoos, K. Leyton-Brown, Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms, arXiv:1208.3719 (2012).

[14] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rucksties, J. Schmidhuber, Pybrain, Journal of Machine Learning Research 11 (2010) 743–746.

[15] E. Alpaydn, Introduction to Machine Learning, 2nd Edition, The MIT Press, 2014.

[16] D. L., Y. Dong, Deep convex network: A scalable architecture for speech pattern classification, in: Interspeech, International Speech Communication Association, 2011, pp. 112–121.

[17] D. Engelbrecht, Computational Intelligence: An Introduction, Willey, 2007.

[18] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, et al., Generative adversarial nets, in: Advances in Neural Information Processing Systems 27, Curran Associates, Inc., 2014, pp. 2672–2680.

[20] J. Redmon, A. Farhadi, YOLO9000: Better, faster, stronger (2016).

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, et al., Mobilenets: Efficient convolutional neural networks for mobile vision applications, arXiv:1704.04861 (2017).

[22] D. C. Ciresan, U. Meier, L. M. Gambardella, J. Schmidhuber, Deep, big, simple neural nets for handwritten digit recognition, Neural Computation 22 (12) (2010) 3207–3220.

[23] M. D. McDonnell, T. Vladusich, Enhanced image classification with a fast-learning shallow convolutional neural network (2015).

[24] T. Young, D. Hazarika, S. Poria, E. Cambria, Recent trends in deep learning based natural language processing, arXiv:1708.02709 (2017).

[25] P. Malhotra, V. TV, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, G. Shroff, Multi-sensor prognostics using an unsupervised health index based on LSTM encoder-decoder, arXiv:1608.06154 (2016).

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, I. Reutemann, P.and Witten, The WEKA data mining software: an update, ACM SIGKDD Explorations Newsletter 11 (1) (2009) 10–18.

[27] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, Journal of Machine Learning Research 13 (2012) 281–305.

[28] J. Nocedal, S. J. Wright, Numerical Optimization, 2nd Edition, Springer, New York, NY, USA, 2006.

[29] J. Mockus, Bayesian Approach to Global Optimization, Springer, 1989.

[30] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, et al., Evolving deep neural networks, arXiv:1703.00548 (2017).

[31] T. Gebru, J. Krause, Y. Wang, D. Chen, J. Deng, E. L. Aiden, L. Fei-Fei, Using deep learning and google street view to estimate the demographic makeup of neighborhoods across the united states, in: Proceedings of the National Academy of Sciences, Vol. 114, National Academy of Sciences, 2017, pp. 13108–13113.

[32] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM Computational Survey 41 (3) (2009) 1–58.

[33] F. L. Lewis, D. Vrabie, Reinforcement learning and adaptive dynamic programming for feedback control, IEEE Circuits and Systems Magazine 9 (3) (2009) 32–50.

[34] Y. LeCun, C. Cortes, MNIST handwritten digit database, http://yann.lecun.com/exdb/mnist/ (2010).

[35] A. NG, Machine Learning Yearning, DeepLearning.AI, 2018.

[36] R. Sutton, A. Barton, Reinforcement Learning. An Introduction, MIT Press, 2018.

[37] C. Louen, S. X. Ding, C. Kandler, A new framework for remaining useful life estimation using support vector machine classifier, in: Conference on Control and Fault-Tolerant Systems, 2013, pp. 228–233.

[38] T. B. Trafalis, H. Ince, Support vector machine for regression and applications to financial forecasting, in: IEEE International Joint Conference on Neural Networks, Vol. 6, 2000, pp. 348–353.

[39] K. Yoo, H. Yoo, J. M. Lee, S. Shukla, J. Park, Classification and regression tree approach for prediction of potential hazards of urban airborne bacteria during asian dust events, Scientific Reports 8 (2018) 1–11.

[40] O. J. Nisha, S. M. S. Bhanu, Permission-based android malware application detection using multi-layer perceptron, in: International Conference on Intelligent Systems Design and Applications, Springer, 2018, pp. 362–371.

[41] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems 25, Curran Associates, Inc., 2012, pp. 1097–1105.

[42] S. Singh, M. Perpinan, Sampling the inverse set of a neuron, in: Bay Area Machine Learning Symposium, BayLearn, 2018.

[43] A. F. Huxley, A. L. Hodgkin, A quantitative description of membrane current and its application to conduction and excitation in nerve., Journal of Physiology 117 (4) (1952) 500–544.

[44] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (5) (1989) 359 – 366.

[45] A. J. Izenman, Modern Multivariate Statistical Techniques, Springer Texts in Statistics, 2013.

[46] S. Ruder, An overview of gradient descent optimization algorithms, arXiv:1609.04747 (2016).

[47] D. E. Rummelhart, G. E. Hinton, J. R. Williams, Learning representations by back-propagating errors, Nature. International Journal of Science 323 (1986) 533–536.

[48] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[49] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, A. Torralba, Learning deep features for discriminative localization, arXiv:1512.04150 (2015).

[50] Z. C. Lipton, J. Berkowitz, C. Elkan, A critical review of recurrent neural networks for sequence learning, arXiv:1506.00019 (2015).

[51] P. J. Werbos, Generalization of backpropagation with application to a recurrent gas market model, Neural Networks 1 (4) (1988) 339–356.

[52] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.

[53] K. Cho, B. van Merrienboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder-decoder approaches, arXiv:1409.1259 (2014).

[54] A. Saxena, K. Goebel, PHM08 challenge data set, https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ (2008).

[55] N. Z. Gebraeel, M. A. Lawley, R. Li, V. Parmeshwaran, Residual-life distributions from component degradation signals: A neural-network approach, IEEE Transactions on Industrial Electronics 51 (3) (2005) 150–172.

[56] M. Zaidan, A. Mills, R. Harrison, Bayesian framework for aerospace gas turbine engine prognostics, in: IEEE (Ed.), Aerospace Conference, 2013, pp. 1–8.

[57] Z. Zhao, L. Bin, X. Wang, W. Lu, Remaining useful life prediction of aircraft engine based on degradation pattern learning, Reliability Engineering & System Safety 164 (2017) 74–83.

[58] J. Lee, F. Wu, W. Zhao, M. Ghaffari, L. Liao, D. Siegel, Prognostics and health management design for rotary machinery systems - reviews, methodology and applications, Mechanical Systems and Signal Processing 42 (12) (2014) 314–334.

[59] W. Yu, H. Kuffi, A new stress-based fatigue life model for ball bearings, Tribology Transactions 44 (1) (2001) 11–18.

[60] J. Liu, G. Wang, A multi-state predictor with a variable input pattern for system state forecasting, Mechanical Systems and Signal Processing 23 (5) (2009) 1586–1599.

[61] A. Mosallam, K. Medjaher, N. Zerhouni, Nonparametric time series modelling for industrial prognostics and health management, The International Journal of Advanced Manufacturing Technology 69 (5) (2013) 1685–1699.

[62] M. Pecht, Jaai, A prognostics and health management roadmap for information and electronics rich-systems, Microelectronics Reliability 50 (3) (2010) 317–323.

[63] J. Liu, M. Wang, Y. Yang, A data-model-fusion prognostic framework for dynamic system state forecasting, Engineering Applications of Artificial Intelligence 25 (4) (2012) 814–823.

[64] Y. Qian, R. Yan, R. X. Gao, A multi-time scale approach to remaining useful life prediction in rolling bearing, Mechanical Systems and Signal Processing 83 (2017) 549–567.

[65] T. Benkedjouh, K. Medjaher, N. Zerhouni, S. Rechak, Remaining useful life estimation based on nonlinear feature reduction and support vector regression, Engineering Applications of Artificial Intelligence 26 (7) (2013) 1751–1760.

[66] M. Dong, D. He, A segmental hidden semi-markov model (HSMM)-based diagnostics and prognostics framework and methodology, Mechanical Systems and Signal Processing 21 (5) (2007) 2248–2266.

[67] J. Z. Sikorska, M. Hodkiewicz, L. Ma, Prognostic modelling options for remaining useful life estimation by industry, Mechanical Systems and Signal Processing 25 (5) (2011) 1803–1836.

[68] A. Saxena, K. Goebel, D. Simon, N. Eklund, Damage propagation modeling for aircraft engine run-to-failure simulation, in: International Conference On Prognostics and Health Management, IEEE, 2008, pp. 1–9.

[69] E. Ramasso, Investigating computational geometry for failure prognostics, International Journal of Prognostics and Health Management 5 (1) (2014) 1–18.

[70] D. Laredo, X. Chen, O. Schütze, J. Q. Sun, ANN-EA for RUL estimation, source code, [Online] (2018).
URL `https://github.com/dlaredo/NASA_RUL_-CMAPS-`

[71] P. Bühlmann, G. S., Statistics for High Dimensional Data. Methods, Theory and Applications, Springer, 2011.

[72] C. Kong, S. Lucey, Take it in your stride: Do we need striding in CNNs?, arXiv:1712.02502 (2017).

[73] R. Storn, K. Price, Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces, Journal of Global Optimization 11 (4) (1997) 341–359.

[74] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 06/2018] (2001).
URL `http://www.scipy.org/`

[75] Y. Peng, H. Wang, J. Wang, D. Liu, X. Peng, A modified echo state network based remaining useful life estimation approach, in: IEEE Conference on Prognostics and Health Management, 2012, pp. 1–7.

[76] C. Zhang, P. Lim, A. Qin, K. Tan, Multiobjective deep belief networks ensemble for remaining useful life estimation in prognostics, IEEE Transactions on Neural Networks and Learning Systems 99 (2016) 1–13.

[77] X. Meng, J. Bradley, B. Yavuz, B. Sparks, S. Venkataraman, D. Liu, et al., MLlib: Machine learning in apache spark, Journal of Machine Learning Research 17 (34) (2016) 1–7.

[78] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL `http://tensorflow.org/`

[79] C. Francois, Keras, `https://github.com/fchollet/keras` (2015).

[80] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, et al., Caffe: Convolutional architecture for fast feature embedding, arXiv:1408.5093 (2014).

[81] F. Seide, A. Agarwal, CNTK: Microsoft's open-source deep-learning toolkit, in: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 2135–2135.

[82] H. Jin, Q. Song, X. Hu, Auto-keras: An efficient neural architecture search system, arXiv:1806.10282 (2018).

[83] E. Real, A. Aggarwal, Y. Huang, Q. V. Le, Regularized evolution for image classifier architecture search, arXiv:1802.01548 (2018).

[84] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzales, Automated model search for large scale machine learning, in: SoCC, 2015, pp. 368–380.

[85] B. Zoph, Q. V. Le, Neural architecture search with reinforcement learning (2016).

[86] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, arXiv:1611.02167 (2016).

[87] Z. Zhong, J. Yan, W. Wu, J. Shao, C.-L. Liu, Practical block-wise neural network architecture generation, arXiv:1708.05552 (2017).

[88] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, et al., Progressive neural architecture search, arXiv:1712.00559 (2017).

[89] P. Angeline, G. Saunders, J. Pollack, An evolutionary algorithm that constructs recurrent neural networks, IEEE Transactions on Neural Networks 5 (1) (1994) 54–65.

[90] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, ACM, 2017, pp. 497–504.

[91] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, K. Leyton-Brown, Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in weka, Journal of Machine Learning Research 18 (25) (2017) 1–5.

[92] E. Brochu, V. M. Cora, N. de Freitas, A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, arXiv:1012.2599 (2010).

[93] F. Hutter, H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: Proceedings of the 5th International Conference on Learning and Intelligent Optimization, LION'05, Springer-Verlag, 2011, pp. 507–523.

[94] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: Advances in Neural Information Processing Systems 28, Curran Associates, Inc., 2015, pp. 2962–2970.

[95] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, et al., Large-scale evolution of image classifiers, arXiv:1703.01041 (2017).

[96] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, et al., Ray: A distributed framework for emerging ai applications, arXiv:1712.05889 (2017).

[97] J. Deng, W. Dong, R. Socher, L. Li, and, ImageNet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255.

[98] R. Ebehart, Y. Shi, Computational Intelligence, Morgan Kauffman, 2007.

[99] S. Sumathi, P. Surekha, Computational Intelligence Paradigms. Theory and Applications using MATLAB, CRC Press, 2010.

[100] K. Krishnakumar, Micro-genetic algorithms for stationary and non-stationary function optimization, in: SPIE Proceedings: Intelligent Control and Adaptive Systems, 1989, pp. 289–296.

[101] C. Hillermeier, Nonlinear Multiobjective Optimization, Springer, 2001.

[102] J. Holland, Adaptation in Natural and Artificial Systems, MIT Press, 1992.

[103] D. Laredo, Y. Quin, O. Schütze, J. Q. Sun, Automatic model selection for neural networks, source code, [Online] (2019).
URL `https://github.com/dlaredo/automatic_model_selection`

[104] P. Simard, D. Steinkraus, J. Platt, Best practices for convolutional neural networks applied to visual document analysis, in: the 7th International Conference on Document Analysis and Recognition (ICDAR 2003), 2-Volume Set, 3-6 August 2003, Edinburgh, Scotland, UK, 2003, pp. 958–962.

[105] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2010, pp. 10–10.

Appendix A

TESTED NEURAL NETWORK ARCHITECTURES

In this appendix we present the tested neural network architectures used in Chapter 3. Each table represents a neural network model. Each row in the table represents a neural network layer while each column describes each one of the key parameters of the layer such as the type of layer, number of neurons in the layer, activation function of the layer and whether regularization is used, where L1 denotes the L1 regularization factor and L2 denotes the L2 regularization factor, the order in which the layers are appended from the table is top-bottom.

Table A.1: Proposed neural network architecture 1.

Layer	Neurons	Activation	Additional Information
Fully connected	20	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	20	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Table A.2: Proposed neural network architecture 2.

Layer	Neurons	Activation	Additional Information
Fully connected	50	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	20	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Table A.3: Proposed neural network architecture 3.

Layer	Neurons	Activation	Additional Information
Fully connected	100	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	50	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Table A.4: Proposed neural network architecture 4.

Layer	Neurons	Activation	Additional Information
Fully connected	250	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	50	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Table A.5: Proposed neural network architecture 5.

Layer	Neurons	Activation	Additional Information
Fully connected	20	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Table A.6: Proposed neural network architecture 6.

Layer	Neurons	Activation	Additional Information
Fully connected	10	ReLU	L1 = 0.1, L2 = 0.2
Fully connected	1	Linear	L1 = 0.1, L2 = 0.2

Appendix B

GENERATED NEURAL NETWORK MODELS

In this appendix we present the generated neural network models generated in Chapter 4. Each table represents a neural network model. Each row in the table represents a neural network layer while each column describes each one of the key parameters of the layer. The order in which the layers are appended from the table is top-bottom.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	264	ReLU	n/a
Dropout	n/a	n/a	0.65
Fully Connected	464	ReLU	n/a
Dropout	n/a	n/a	0.35
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

Table B.1: Neural network model corresponding to S_1.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	56	Sigmoid	n/a
Dropout	n/a	n/a	0.25
Fully Connected	360	Sigmoid	n/a
Fully Connected	480	Sigmoid	n/a
Fully Connected	80	Sigmoid	n/a
Dropout	n/a	n/a	0.20
Fully Connected	10	Softmax	n/a

Table B.2: Neural network model corresponding to S_2.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	264	ReLU	n/a
Fully Connected	360	ReLU	n/a
Fully Connected	480	ReLU	n/a
Fully Connected	88	ReLU	n/a
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

Table B.3: Neural network model corresponding to S_3.