UNIVERSITY OF CALIFORNIA,
IRVINE

On Sparse and Efficient Deep Learning

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Statistics

by

Yadong Lu

2021

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Pierre Baldi for his guidance and a lot of insightful discussions on research ideas throughout the past few years. I always feel fortunate and enjoyable to work with him on a variety of exciting scientific problems. Without him this dissertation would not have become possible. I want to acknowledge Professor Michele Guindani for introducing me to Bayesian thinking which greatly benefited my research. His support and encouragement helped me through difficult times in research. I also want to thank Professor Babak Shababa, who taught me useful knowledge in statistical computing, which later became an important topic of my thesis.

In addition, I am grateful for all of my collaborators, especially Professor Daniel Whiteson, Professor Cristina V. Lopes, Julian Collado and Farima Farmahinifarahani. It has been wonderful to work together with them during my PhD. Lastly, I want to give special thanks to my dad Yingjie Lu, who led me into the fascinating world of math and computer science in my childhood, and my mom Xiaoying Deng, who constantly fills me with her unconditional love and encouragement.

VITA

Yadong Lu

EDUCATION

Doctor of Philosophy in Statistics **2021**
University name *Irvine, California*

Bachelor of Science in Mathematics **2016**
Sichuan University *Chengdu, China*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019-2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2016–2021**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Yadong Lu, Julian Collado, Daniel Whiteson, Pierre Baldi. "SARM: Sparse Autoregressive Model for Scalable Generation of Sparse Images in Particle Physics", *Physics Review D, 2021.*

REFEREED CONFERENCE PUBLICATIONS

Yadong Lu, Yinhao Zhu, Yang Yang, Amir Said, Taco S Cohen. "Progressive Neural Image Compression with Nested Quantization and Latent Ordering", *International Conference on Image Processing (ICIP), 2021.*

Yasaman Razeghi, Kalev Kask, **Yadong Lu**, Pierre Baldi, Sakshi Agarwal, Rina Dechter. "Deep Bucket Elimination", *International Joint Conference on Artificial Intelligence (IJCAI), 2021.*

Ying Wang, **Yadong Lu**, Tijmen Blankevoort. "Differentiable joint pruning and quantization for hardware efficiency", *European Conference on Computer Vision (ECCV), 2020.*

Siyu Shao, Ruqiang Yan, **Yadong Lu**, Peng Wang, Robert X. Gao. "DCNN-based multi-signal induction motor fault diagnosis", *IEEE Transactions on Instrumentation and Mea-*

surement, 2019.

Vaibhav Saini, Farima Farmahinifarahani, **Yadong Lu**, Di Yang, Pedro Martins, Hitesh Sajnani, Pierre Baldi, and Cristina Lopes. "Towards Automating Precision Studies of Clone Detectors", *International Conference on Software Engineering (ICSE), 2019.*

Vaibhav Saini, Farima Farmahinifarahani, **Yadong Lu**, Pierre Baldi, and Cristina Lopes. "Oreo: Detection of Clones in the Twilight Zone", *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2018.*

BOOK CHAPTERS

Yinsen Miao, Jeong Hwan Kook, **Yadong Lu**, Michele Guindani, Marina Vannucci. "Scalable Bayesian Variable Selection Regression Models for Count Data", *Flexible Bayesian Regression Modelling, Elsevier, 2019.*

ABSTRACT OF THE DISSERTATION

On Sparse and Efficient Deep Learning

By

Yadong Lu

Doctor of Philosophy in Statistics

University of California, Irvine, 2021

Distinguished Professor Pierre Baldi, Chair

Rapid and ongoing technology developments enable researchers to collect large scale, high-dimensional data in a wide range of areas in science and technology. The availability of these data, together with increasing computational resources, have led to the remarkable success of deep learning in a variety of tasks such as computer vision, natural language processing, and reinforcement learning. However, deep neural networks for modeling high-dimensional data can be hard to interpret and pose heavy computational burdens. One approach for addressing some of these issues is to learn sparse representations.

In my thesis, I present several approaches to facilitate efficient learning of sparse representations in deep learning. I first describe a scalable variational inference algorithm to perform variable selection in Bayesian settings using non-local priors. There, I show that our method approximates the posterior estimates with the same degree of precision in variable selection as traditional MCMC algorithms, while providing a one-order-of-magnitude speedup. Then I propose a deep auto-regressive generative model to efficiently learn sparse distributions and demonstrate its effectiveness on the problem of generating high-quality jet images in particle physics to speed up scientific discovery. Finally, I introduce an approach which adapts transformers, the current state-of-the-art deep learning models for natural language processing tasks, to software data. There I tackle the problem of learning sparse location information to identify exception prone segments of source code in software engineering.

Chapter 1

Introduction

Sparse distribution is becoming increasingly useful in high dimensional statistical analysis. In a general setting where we have collected data for a variable of interest, and a large set of covariates that might be associated with the variable of interest, a common question asked is: can we discover a handful of covariates and build a model using the selected covariates to predict the variable of interest considerably well? Further, we often ask: how certain do we believe the selected covariates are indeed significant in explaining the variable of interest. The second question is especially important, for example in bio-medical applications where it is crucial to obtain uncertainty estimates. In this case, the sparse distribution, with a point probability mass at zero and the rest of probability mass at non-zero values, can be used to reflect the belief on how certain we should include each individual covariate. In the Bayesian setting, we estimate the posterior of these sparse distributions given the data we observed. This procedure is typical in Bayesian variable selection literature [87, 45].

Another type of use case of sparse distribution is to model the sparseness of the data. In high energy physics experiments, it is common that two beams of particles are accelerated to near light speed and collide in a vacuum chamber with cylinder shape and the particle detectors are installed on its wall. When the products of the collision hit the detectors, the Mass and angle information are recorded. So that at a given time point when we unroll the cylinder, the detector's signal can be seen as a rectangular image, where it is often called *jet image* [95, 4]. At any given time, only a small part of the detectors get activated due to collision, hence the jet image is usually sparse. The jet image is useful for a variety of tasks [51, 113], however, generating it relies on Monte Carlo sampling, which is extremely slow considering the amount needed to be generated [39]. To generate the sparse jet images, a sparse distribution is needed to model each individual pixel distribution. We will see an example where a deep neural network is applied to learn the sparse distribution of each pixel as well as the dependency between all the pixels in the images.

1.1 Outline and Contributions

1.1.1 Variational Sparse Learning via Non-local Priors

Sparse learning methods have received increasing interest in recent years due to their wide range of applications in high-dimensional problems. Fully Bayesian methods provide a principled approach to learning sparse representations of the data, but they tend to be computationally intensive. In this chapter, I discuss an approach to leverage the flexibility of the transformation in the normalizing flows as variational approximations and propose a scalable variational inference algorithm for Bayesian sparse learning using non-local priors. Non-local priors refer to a family of prior distributions that assign negligible mass to the region near the origin, leading to faster shrinkage of spurious non-zero parameters compared to the commonly-used spike and slab priors. A Learned Hard-Concrete (LHC) relaxation is proposed as a learnable approximation to the discrete spike in the variable selection priors. Simulation results in high-dimensional regression problems show that our approach can match the performance of MCMC methods, while providing an order of magnitude improvement in speed. Furthermore, we demonstrate the effectiveness of our method in identifying informative neurons based on a rodent experiment involving an odor recognition task.

1.1.2 Sparse Autoregressive Models for Scalable Generation of Sparse Images in Particle Physics

Generation of simulated data is essential for data analysis in particle physics, but current Monte Carlo methods are very computationally expensive. Deep-learning-based generative models have successfully generated simulated data at lower cost, but struggle when the data are very sparse. I introduce a novel deep sparse autoregressive model (SARM) that explicitly learns the sparseness of the data with a tractable likelihood, making it more stable and interpretable when compared to Generative Adversarial Networks (GANs) and other methods. In two case studies, we compare SARM to a GAN model and a non-sparse autoregressive model. As a quantitative measure of performance, we compute the Wasserstein distance (W_p) between the distributions of physical quantities calculated on the generated images and on the training images. In the first study, featuring images of jets in which 90% of the pixels are zero-valued, SARM produces images with W_p scores that are $24 - 52\%$ better than the scores obtained with other state-of-the-art generative models. In the second study, on calorimeter images in the vicinity of muons where 98% of the pixels are zero-valued, SARM produces images with W_p scores that are $66 - 68\%$ better. Similar observations made with other metrics confirm the usefulness of SARM for sparse data in particle physics.

1.1.3 Detecting JAVA Runtime Exceptions using Sparse Location Information.

Runtime exceptions are inevitable parts of software systems. While developers often write exception handling code to avoid the severe outcomes of these exceptions, such code is most effective if accompanied by accurate runtime exception types. Predicting the runtime exceptions that may occur in a program, however, is a challenging task, as the situations that lead to these exceptions are complex. In this paper, we propose D-REX (Deep Runtime EXception detector), as an approach for predicting runtime exceptions of Java methods based on the static properties of code.

The core of D-REX is a machine learning model that leverages the representation learning ability of neural networks to infer a set of signals from code to predict the related runtime exception types. This model, which we call Location Aware BERT, adapts a state-of-the-art language model, BERT, to provide accurate predictions for the exception types, as well as interpretable recommendations for the exception prone elements of code. We curate a benchmark dataset of 200,000 Java projects from GitHub to train and evaluate D-REX. Experiments demonstrate that D-REX predicts exception types with 81% of Top 1 accuracy, outperforming multiple non-BERT baselines by a margin of at least 22%. Furthermore, it can predict exception prone code elements with 75% Top 1 precision.

Chapter 2

Variational Sparse Learning via Non-local Priors

2.1 Background

Sparse learning methods play an important role in a wide range of applications such as image processing [138, 82], natural language processing [126, 125], and for learning sparse and efficient neural network structures [53, 136]. They typically lead to more interpretable models and reduce computational burden. Sparsity inducing priors offer a general approach to sparse learning within the Bayesian framework, while providing a reliable method for uncertainty quantification. The "spike and slab" family of priors [87, 45] is the gold standard in Bayesian variable selection. It places a Dirac delta mass at the origin to explicitly induce sparsity and uses a zero-centered Gaussian with large variance to model the non-zero density of the parameter. Such priors allow to compute the posterior probability of whether a parameter is different than zero in Bayesian hypothesis testing and variable selection. Another commonly-employed family of sparsity inducing priors is provided by scale-mixtures of Gaussian distributions [15, 103] The resulting “shrinkage" priors are characterized by heavy tails and therefore lead to a more robust framework for handling large outlying signals [32, 17]. In fact, penalized likelihood approaches commonly used within the frequentist framework can often be re-interpreted as sparsity inducing priors, for example the LASSO method [127] is analogous to using a Laplace prior [100].

In recent years, there have been discussions, both philosophically and practically, on the appropriateness of the above methods since they tend to place a significantly high probability mass around a neighborhood of the null value. By definition, the alternative hypothesis should be disjoint from the null hypothesis. Therefore the prior density reflecting the alternative hypothesis should coherently be zero at the null value. In practice, excessive mass around the null value often fails to penalize the parameters that are small but not supported by the data [110].

Non-local priors have been proposed as an alternative to the previous approaches [67]. Consider the case where the null hypothesis is that a parameter, β, is equal to zero. In a sparsity learning context, we can briefly describe a non-local selection prior $p(\beta)$ as a discrete mixture of a point mass at zero (the null value) and a non-local density, i.e. a density that places negligible mass around zero. It has been shown that non-local priors provide shrinkage for spurious parameters either at fast polynomial or quasi-exponential rates as the sample size n increases [110].

Currently, inference for models involving non-local priors is based on MCMC methods, which are computationally demanding for high dimensional variable selection. Our goal is to propose a scalable and practical variational method for sparse learning problems using non-local priors. Our contributions can be summarized as follows: 1) we derive the evidence lower bound (ELBO) for a family of non-local priors and show that our variational algorithm is able to consistently approximate the correct gradient signal to shrink spurious parameters in linear regression settings, 2) we propose a learned hard-concrete relexation as a reparameterization scheme for the random binary selection indicators in the spike-and-slab prior, removing the need to manually choosing the scaling constants in the hard-concrete distribution [79], 3) we conduct an extensive simulation study to show that by employing a factorized normalizing flows, [109, 62] as flexible variational posterior, our method is able to match and even outperform the estimation performances of MCMC methods in an high-dimensional variable selection problem, and 4) we apply our method on neuronal spike-train data from a rodent experiment, successfully identifying informative neurons and tetrodes implied in odor recognition.

2.2 Non-local Priors for variable selection

In this Section, we provide general background on the use of non-local prior densities in Bayesian variable selection [68]. To set the stage, we consider a linear regression framework, where the interest is to investigate how a dependent variable y is associated to a small subset of available covariates $x_1, \ldots, x_p$, after collecting N measurements,

$$y_i = \sum_{j=1}^{p} x_i \beta_j + \varepsilon_i,$$

$i = 1, \ldots N$, where the β_j's denote real-valued regression coefficients and $\varepsilon_i \sim N(0, \sigma^2)$ captures measurement error. In model selection, the objective is to identify which covariates are relevant and should be included in the model, by conducting hypotheses tests on each regression coefficient. More formally, suppose the null hypothesis $\mathbf{H}_0$ assumes that a coefficient $\beta_j \in \Theta_0$ for some j, and the alternative hypothesis $\mathbf{H}_1$ is $\beta_j \in \Theta_1$. In variable selection problems, the hypotheses relate to the significance of the coefficients, i.e. $\mathbf{H}_0$: $\beta_j = 0$ versus $\mathbf{H}_1$: $\beta_j \neq 0$. In this paper, we will define a non-local prior for the regression coefficients as a prior such that $p(\beta_j|\mathbf{H}_1) = 0$ for all $\beta_j \in \Theta_0$ and $p(\beta_j|\mathbf{H}_1) > 0$ for all $\beta_j \in \Theta_1$, $j = 1, \ldots, p$. Correspondingly, we introduce an auxiliary variable z_j, that follows a Bernoulli distribution

$Bern(\pi_0)$. The indicator variable identifies the distributional assumptions on the parameter β_j under the null and alternative hypotheses, i.e. if $z_j = 0$ the data are assumed to be drawn from the model where $\beta_j = 0$. Thus, the non-local selection prior can be regarded as a mixture of a point mass at zero and a continuous non-local alternative distribution (i.e., a spike-and-slab non-local prior)

$$\beta_j \sim (1 - z_j)\,\delta_0 + z_j\, p(\beta_j; \tau)$$

where $z_j \sim \text{Bern}(\pi)$ and $p(\beta_j; \tau)$ is a non-local density indexed by a parameter τ and characterizing the prior distribution of β_j under the alternative hypothesis. Similarly as in the traditional spike-and-slab prior formulation, a non-local selection prior models the sparsity explicitly by assigning a positive mass $1 - \pi$ at the origin. However, differently than when using a flat Gaussian distribution $N(0, \tau)$, the density $p(\beta_j; \tau)$ does not place a significant amount of probability mass near the null value, thus properly reflecting the prior belief that the parameter is away from zero under H_1. An important motivation for the use of non-local priors has been that they balance the rates of convergence of Bayes factors under the null and alternative hypothesis [67]. On the contrary, the large sample properties of Bayes factors obtained by the traditional local alternative priors imply that, as the sample size n increases, evidence accumulates much more rapidly in favour of true alternative models than in favour of true null models. More specifically, here, we consider the inverse moment prior (iMOM) [67, 86], assuming independence of the β_j's. Thus, the joint density function is obtained as

$$p\left(\boldsymbol{\beta}|\tau_j\right) = \prod_{j=1}^{p} \frac{(\tau_j)^{\frac{1}{2}}}{\sqrt{\pi}\beta_j^2} \exp\left\{-\frac{\tau_j}{\beta_j^2}\right\}.$$

In practice, we use $\tau_j = \tau > 0$,$j = 1, \ldots, p$, where the parameter controls the magnitude of the modes and the thickness of the tail densities (see Figure 2.1). When τ increases, the prior becomes flatter, with heavier tails.

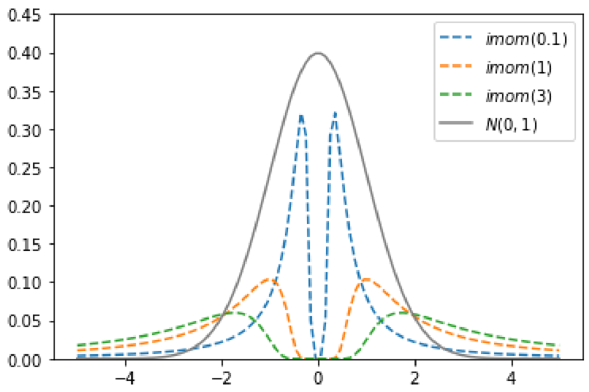


Figure 2.1: Comparison of the probability densities for a $N(0,1)$ local versus multiple non-local inverse moment priors for the alternative density of a discrete spike-and-slab selection prior. A point mass at zero is assumed for the null hypothesis (not shown). The inverse moment prior places negligible (i.e. close to zero) mass around the origin. When τ increases, the prior becomes flatter, with heavier tails.

2.3 Variational Inference for Non-local Prior selection

In a Bayesian framework, the posterior distribution of the β's, $p(\boldsymbol{\beta}|\boldsymbol{y},\boldsymbol{x})$, is obtained by employing MCMC sampling schemes, since the non-local selection prior does not lead to a posterior in closed form. However, the use of MCMC techniques may be computationally infeasible when dealing with a high dimensional p and a large number of samples. Variational inference (VI) [28] enables approximate posterior inference by introducing a variational distribution $q_\theta(\boldsymbol{\beta})$ with free variational parameter θ, to approximate the true posterior distribution $p(\boldsymbol{\beta}|\boldsymbol{y},\boldsymbol{x})$. In typical applications of VI, the variational distribution is simple and fully factorized, thus easy to sample from. The objective is to minimize the Kullback–Leibler (KL) divergence between $q_\theta(\boldsymbol{\beta})$ and the true posterior $p(\boldsymbol{\beta}|\boldsymbol{y},\boldsymbol{x})$ by using the gradient descent algorithm on the variational parameter θ. The so called *evidence lower bound* to the log-marginal distribution of $\boldsymbol{y}$, conditional on $\boldsymbol{x}$, is treated as the objective function,

$$\begin{aligned}\log p(\boldsymbol{y}|\boldsymbol{x}) &= \log \int q_\theta(\boldsymbol{\beta}) \frac{p(\boldsymbol{y},\boldsymbol{\beta}|\boldsymbol{x})}{q_\theta(\boldsymbol{\beta})} d\boldsymbol{\beta} \\ &\geq E_q \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\beta}) - E_q \log \frac{q_\theta(\boldsymbol{\beta})}{p(\boldsymbol{\beta})}\end{aligned} \tag{2.1}$$

Here we adopt the mean field assumption, i.e. we assume $q_{\pi,\eta}(\boldsymbol{\beta}) = \prod_{j=1}^{p} q_{\pi_j,\eta_j}(\beta_j)$, where $\boldsymbol{\pi}$ indicates the vector of parameters of the Bernoulli probability of the selection indicators $\boldsymbol{z}$, and $\boldsymbol{\eta} = (\eta_1, ..., \eta_p)$ parameterizes the variational density for the alternative hypothesis, $q_{\eta_j}(\beta_j|z_j = 1)$. Thus, the evidence lower bound becomes:

$$ELBO = E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{z}) - E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log \frac{q_{\pi,\eta}(\boldsymbol{\beta}, \boldsymbol{z})}{p(\boldsymbol{\beta}, \boldsymbol{z})}, \tag{2.2}$$

and the objective is to perform posterior inference on $\boldsymbol{\pi}$, $\boldsymbol{z}$, and the $\boldsymbol{\beta}$'s.

In this paper, we propose to use a factorizable normalizing flow generative model as the variational distribution $q_{\pi,\eta}(\boldsymbol{\beta}, \boldsymbol{z})$ [109, 62]. A normalizing flow is a composition of invertible mappings parameterized by neural networks, which transforms a simple distribution, e.g. uniform, into any complex multi-modal distribution, due to the universal approximation capability of neural networks. The density function of the resulting distribution can be obtained by change of variable techniques. The analytical form of the transformed density and its ability to model high dimensional multi-modal distributions make normalizing flows a promising candidate as a flexible variational family [132, 129].

2.3.1 Learned Hard Concrete Relaxation

In (A.1), the Bernoulli random variable z is non-differentiable with respect to its parameter π. Several recent manuscripts have investigated differentiable reparameterizations to allow gradient-based optimization for discrete random variables [65, 83, 130]. The underlying idea is to consider a differentiable continuous relaxation of a Bernoulli variable $z \sim \text{Bern}(\pi)$, by defining an auxiliary random variable $\tilde{z} = f_\lambda(u, \alpha)$, as a function of a positive "temperature" parameter λ, a random variable $u \sim \text{Uniform}(0, 1)$, and a positive parameter α. The "temperature" parameter λ controls the amount of relaxation: when λ decreases to zero, the auxiliary continuous $\tilde{z}$ converges in probability to a discrete Bernoulli random variable.

The real-valued parameter α characterizes the probability of success π, since $P(\tilde{z} > 0.5) = P(\lim_{\lambda \to 0} \tilde{z} = 1) = \alpha/(1+\alpha)$. Thus, gradient descent can be performed on α to learn the underlying Bernoulli probability π in eq (A.1). For details, we refer to **(author?)** [83], who consider the Binary Concrete relaxation defined by the transformation $\tilde{z} = \frac{1}{1+\exp(-(\log \alpha + L)/\lambda)}$ where $L = \log(u) - \log(1-u)$ with $u \sim \text{Uniform}(0,1)$.

However, the continuous relaxation defined above does not allow estimating exact zeroes and ones as it requires to choose a threshold for variable selection. **(author?)** [79] have proposed to use a hard-concrete distribution as a relaxation for a Bernoulli random variable, to obtain a low-variance reparameterized gradient and simultaneously allowing for exact zeroes and ones. More specifically, they consider a relaxation defined through a sigmoid function,

$$\begin{aligned} s &= \text{Sigmoid}((\log u - \log(1-u) + \alpha)/\lambda), \\ \bar{s} &= s(\zeta - \delta) + \delta \end{aligned} \tag{2.3}$$

where $u \sim \text{Uniform}(0,1)$ and λ is a temperature parameter. As $\lambda \to 0$, $\bar{s}$ reduces to z. The two parameters $\gamma < 0$ and $\zeta > 1$ define a scaling factor $(\zeta - \delta) > 1$ and allow to "stretch" the distribution to the $(\gamma, \zeta) \supset (0,1)$ interval, so that the distribution of s is “folded" to a point mass at zero and a point mass at one, respectively. See Figure 2.2. In practice, the hard concrete reparameterization effectively corresponds to a thresholded version of the concrete relaxation in [83], obtained by manually choosing a lower threshold at $\frac{-\delta}{\zeta-\delta}$, and a upper threshold at $\frac{1-\delta}{\zeta-\delta}$. These two scaling factors jointly determine the range of $\bar{s}$ that can receive gradient signals from the likelihood term. When $s < \frac{-\delta}{\zeta-\delta}$, $\bar{s} = 0$ thus the gradient $\frac{\partial \bar{s}}{\partial \alpha} = 0$. Our experience suggests that the performance of the variable selection is sensitive to these scaling factors.

Therefore, to alleviate the difficulty of choosing hyperparameter δ and ζ, we propose to optimize the truncation threshold for truncation by learning also the parameters ζ, δ using a gradient-descent algorithm. By learning the scaling factor, we achieve a more flexible form

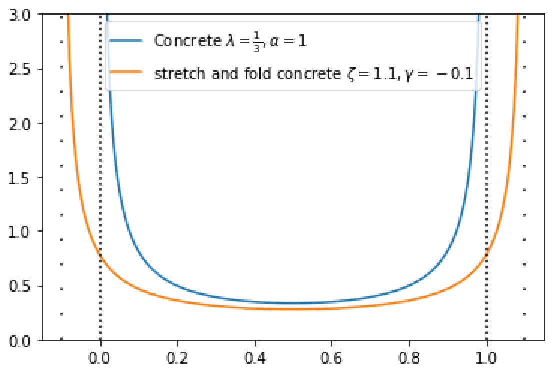


Figure 2.2: Hard Concrete distribution with parameter δ and ζ, initialized at $\zeta = 1.1, \delta = -0.1$

of the relaxation function $\bar{s}$ than the hard concrete distribution. The experiment results suggest that the Learned Hard Concrete relaxation (LHC) we propose leads to comparable performance to a manually optimal tuning of ζ.

2.3.2 A Factorizable Normalizing Flow as Flexible Variational Distribution

Since the non-local alternative density typically is bimodal, using a unimodal Gaussian family as the variational approximate posterior may lead to underestimate the posterior variance, since also the posterior will be multi-modal. We leverage the flexibility of the normalizing flow in density estimation to model multi-modal posterior distributions. However, the state of the art flow-based models often result in a large computational burden [62, 73] $\dot{\text{F}}$or computational and speed consideration, we maintain the mean field assumption of our variational method and apply a factorizable flow model based on the deep sigmoidal flow (DSF) in **(author?)** [62]. More specifically, for each j, we model the variational distribution $q_{\eta_j}(\beta_j|z_j = 1)$ using the distribution incurred by $f_{\eta_j}(\epsilon_j)$, where f_{η_j} is defined as the composition of K sigmoidal flow (SF) transformations:

$$f_{\eta_j}(\epsilon_j) = f_{j,k} \circ \cdot f_{j,k-1} \circ \ldots \circ f_{j,1}(\epsilon_j) \tag{2.4}$$

$$f_{j,k}(z_k) = \text{Logit}\left(w_{j,k}^T \cdot \text{Sigmoid}\left(a_{j,k} \cdot z_{j,k-1} + b_{j,k}\right)\right) \tag{2.5}$$

where $w_{j,k}^T = (w_{jk1}, w_{jk2}, ..., w_{jkd})$. To ensure invertibility of $f_{j,k}$, we require $w_{jkl} \in (0,1)$ and $\sum_{l=1}^{d} w_{jkl} = 1$, and $a_{j,k}, b_{j,k} \in \mathbb{R}^{+d}$ are all d dimensional column vectors. In practice, we optimize an underlying d dimensional unconstrained parameter for w'_{jkl}, and apply a softmax function to ensure $\sum_{l=1}^{d} w_{kl} = 1$:

$$w_{jkl} = \frac{e^{w'_{jkl}}}{\sum_{l=1}^{d} e^{w'_{jkl}}} \tag{2.6}$$

We found that the factorizable sigmoidal flow with $K = 2$ and $d = 2$ provide a sufficiently good capacity for modeling $q(\beta_j|z_j)$.

For each sampled β_j, we can compute its likelihood by change of variable formula:

$$p_{\eta_j}(\beta_j) = \left|\frac{\partial f_{\eta_j}(\epsilon_j)}{\partial \epsilon_j}\right|^{-1} p(\epsilon_j) \tag{2.7}$$

The non-linearities in the transformation f_k can produce multi-modal distribution as demonstrated in [62].

2.3.3 Derivation and Optimization of ELBO

Based on the previous discussion, we can rewrite the evidence lower bound in (A.1) by factorizing $q(\boldsymbol{\beta}, \boldsymbol{z}) = q_\eta(\boldsymbol{\beta}|\boldsymbol{z})q_{\boldsymbol{\pi}}(\boldsymbol{z})$. We adopt the mean field assumption, i.e.

$$q_{\pi,\eta}(\boldsymbol{\beta}, \boldsymbol{z}) = \prod_j q_{\eta_j}(\beta_j|z_j)q_{\pi_j}(z_j) \tag{2.8}$$

Thus, the evidence lower bound can be written as

$$\begin{aligned}\text{ELBO} =& E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log f(\boldsymbol{x}|\boldsymbol{\beta}, \boldsymbol{z})) - \left[\sum_{j=1}^{p} q_{\pi_j}(z_j = 0) \cdot \log \frac{q_{\pi_j}(z_j = 0)}{p_{\pi_j}(z_j = 0)}\right. \\ &+ q_{\pi_j}(z_j = 1)\ \log[\frac{q_{\pi_j}(z_j = 1) q_{\eta_j}(\beta_j|z_j = 1)}{p_{\pi_j}(z_j = 1) p_{\eta_j}(\beta_j|z_j = 1)}\Bigg] \\ =& E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} log f(\boldsymbol{x}|\boldsymbol{\beta}, \boldsymbol{z})) - [\sum_{j}^{p} \text{KL}(q_{\pi_j}(z_j||p(z_j)) \\ &+ \sum_{j}^{p} q_{\pi_j}(z_j = 1) \cdot \text{KL}(q_{\eta_j}(\beta_j|z_j = 1)||p(\beta_j|z_j = 1))] \end{aligned} \tag{2.9}$$

where KL indicates the Kullback–Leibler divergence for two distributions. We can apply the learned hard concrete reparameterization discussed in section 2.3.1 and approximate $\pi_j = q_{\pi_j}(z_j = 1)$ using

$$q(\bar{s}_j \neq 0) = \text{Sigmoid}(\alpha_j - \lambda \ \log \frac{-\delta_j}{\zeta_j}), \tag{2.10}$$

since $\lim_{\lambda \to 0} q(\bar{s}_j \neq 0) = \pi_j$. To reduce the variance in the gradient, we approximate the KL divergence $\text{KL}(q_{\pi_j}(z_j)||p(z_j))$ as

$$\text{KL}(q_{\pi_j}(z_j)||p(z_j)) = \sum_{j}^{p} q_{\alpha_j}(z_j \neq 0) \cdot log \frac{q_{\alpha_j}(z_j \neq 0)}{\pi_{0j}} + q_{\alpha_j}(z_j = 0) \cdot log \frac{q_{\alpha_j}(z_j = 0)}{1 - \pi_{0j}} \tag{2.11}$$

rather than estimating it via Monte Carlo expectation, where π_{0j} denotes the prior probability of $P(z_j = 1)$. The other KL divergence term $\text{KL}(q_{\eta_j}(\beta_j|z_j = 1)||p(\beta_j|z_j = 1))$ is instead estimated using Monte Carlo samples, i.e., we draw m number of samples from the $q_{\eta_j}(\beta_j|z_j = 1)$ and average over their log density ratio. In practice, we first draw from a standard normal distribution ϵ_j and then apply two consecutive SF transformation on it to get the output: $\beta_j = f_{\eta_1} \circ f_{\eta_1}(\epsilon_j)$, where η_1, η_2 are parameters in the SF transformations.

In practice we choose to optimize a symmetric learned hard concrete distribution , where

$\zeta_j = 1 - \delta_j$. So equivalently we have $\bar{s}_j = \phi_j s_j - \frac{\phi_j - 1}{2}$, where $\phi_j > 0$. Since we require both ϕ_j, α_j to be greater than zero, we optimize for the logarithm of both parameters during training and transform it back during inference.

For the likelihood term $E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log f(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{z}))$, again we draw Monte Carlo samples from $q_\pi(\boldsymbol{z})$ and $q_\eta(\boldsymbol{\beta}|\boldsymbol{z})$ to estimate this expectation. The full algorithm for updating the variational parameters are summarized in algorithm **??**.

2.3.4 Convergence Analysis

For the linear regression setting, we show that the gradient information of the objective function $\mathcal{L} = \frac{1}{N} ELBO$, consistently points α_j to the direction such that – under the null – it pushes $\lim_{\lambda \to 0} E(\bar{s}_j)$ to zero, i.e., the proposed variational algorithm can lead to a correct selection of the covariates.

Theorem 1. Consider a linear regression model, where $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{y} \in R^N$, $\boldsymbol{\beta} \in R^p$, $\boldsymbol{\epsilon} \sim N(\boldsymbol{0}, \phi^2 \boldsymbol{I})$. Further, let the covariates be generated as $\boldsymbol{X} \sim N(\boldsymbol{0}, \Sigma)$, where $\max(|\Sigma_{ij|}) < C_0$ for some constant $C_0 > 0$. If the true value of $\beta_{j,true} = 0$, denote $\max_j |\tilde{\beta}_j| < C_1$, where $\tilde{\beta}_j = \bar{s}_j \beta_j$ is the estimated value of β_j, $\bar{s}_j$ follows the definition in A.4, we have:

$$E_{\boldsymbol{X}} \frac{\partial \mathcal{L}}{\partial \alpha_{j'}} < -2 \frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}} [\sigma^2 \beta_{j'}^2 \bar{s}_{j'} - C_0 C_1 (M-1)] + \frac{C}{N}$$

where $\mathcal{L} = \frac{1}{N} ELBO$ and $C > 0$ is a positive constant defined in Lemma 1.

Remark: when $N > \frac{C}{2 \frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}} [\sigma^2 \beta_{j'}^2 \bar{s}_{j'} - C_0 C_1 (M-1)]}$, we have $E \frac{\partial \mathcal{L}}{\partial \alpha_j} < 0$.

It can be seen that the expected magnitude of the gradient signal increases when the absolute value of $\beta_{j'}^2$ is larger. Since a non-local alternative prior places more mass to the area away from zero, it encourages the absolute value of $\beta_{j'}^2$ to be larger compared to a local prior.

Therefore during optimization, the non-local prior often results in a stronger shrinkage of spurious $\beta_j s$. We note that the assumption $\boldsymbol{X} \sim N(\mathbf{0}, \sigma^2 \boldsymbol{I})$ can be relaxed to $\boldsymbol{X} \sim N(\mathbf{0}, \Sigma^2)$, where Σ has diagonal elements σ^2 and the off-diagonal elements ρ, with ρ satisfying certain regularity condition.

Proof of the theorem is detailed in the Supplementary materials. Theorem 2 requires careful calculation of gradient expectation These results show that if the number of covariates p grows with the number of samples N, the proposed gradient-based optimization of the variational algorithm is guaranteed to shrink the coefficient of unrelated covariates to zero.

Algorithm 1: Variational sparse learning with non-local prior

```
Input: data $\boldsymbol{x} \in \mathbb{R}^{N \times P}$
Initialize $\log \boldsymbol{\alpha} = 1$ and $\boldsymbol{\eta}$ with Xavier Initialization [48], set
while not converge do
  Draw S samples from the variational distribution $\boldsymbol{z} \sim q_{\boldsymbol{\alpha}(z)}$ and $\boldsymbol{\beta} \sim q_{\boldsymbol{\eta}}(\boldsymbol{\beta}|\boldsymbol{z} = 1)$.
  The terms in ELBO related to each dimension of the latent variables $\boldsymbol{z}$ and $\boldsymbol{\beta}$ are
  computed in parallel:
  -ELBO$_j = \frac{1}{N} \sum_{i=1}^{N} \sum_{s=1}^{S} f(x_i|\beta_{[s]} \odot z_{[s]}) - \frac{1}{N} \left[ \sum_{s=1}^{s} q_{\alpha_i}(\boldsymbol{z}_{[\boldsymbol{s}]}) \cdot \log \frac{q_{\alpha_j}(z_{[s]})}{p(\boldsymbol{z}_{[\boldsymbol{s}]}))} \right] - \frac{1}{N}$
  $\cdot q_{\alpha_j}(z_{[s]} = 1) \left[ \sum_{s=1}^{s} (q_{\boldsymbol{\eta_j}}(\beta_{[s]}|z_{[s]} = 1) \log \frac{q_{\boldsymbol{\eta_j}}(\beta_{[s]}|z_{[s]}=1)}{p(\beta_{[s]}|z_{[s]}=1)} \right]$
  Update the parameters $\alpha_j, \phi_j$, and $\eta_j$ in parallel:
  $\alpha_j = \alpha_j + \rho_t \nabla_{\alpha_j}(-\text{ELBO}_j)$
  $\phi_j = \phi_j + \rho_t \nabla_{\alpha_j}(-\text{ELBO}_j)$
  $\eta_j = \eta_j + \rho_t \nabla_{\eta_j}(-\text{ELBO}_j)$
end while
```

2.4 Related Work

There has been several contributions to the literature applying variational methods for posterior inference of sparsity induced priors. **(author?)** [128] have proposed spike-and-slab variational inference for multi-task and multiple kernel learning. They reparameterize the spike and slab prior $\beta \sim \pi N(\beta|0, \sigma^2) + (1 - \pi)\delta_0(\beta)$ as $\beta = \tilde{\beta} z$, where $\tilde{\beta} \sim N(\tilde{\beta}|0, \sigma^2)$ and $z \sim Bern(\pi)$. A EM algorithm is used in their variational method, whereas we employ a different reparameterization through a learned hard-concrete relaxation (LHC) to allow for direct gradient based optimization of the Bernoulli parameter. Differently from the hard concrete distribution in [79], the proposed LHC is able to optimize the scaling factor based on the available data and thus results in better performances (see Section 5). Furthermore, the proposed method doesn't depend on the specific form of the likelihood model, so it is readily generalizable to handling sparsity in generalized linear models.

Scale mixture of Gaussian priors are also commonly used to promote sparsity, e.g. the Horseshoe prior. [64] propose noncentered parameterizations of the horseshoe prior to allow for efficient mean-field variational inference. [47] use regularized Horseshoe prior for inference on the weights in Bayesian neural networks when the training data are limited [102]. Our method distinguishes from these contributions since we use non-local priors to model the sparsity, more specifically through a mixture containing a point mass at zero, thus providing larger separation between the null value and alternative values [110].

Finally, normalizing flow models have been previously considered for variational inference as an enrichment to the mean field Gaussian distribution [132, 109]. To exploit the trade-off between the computational efficiency of the mean field assumption, and the flexibility of the normalizing flow in density estimation, we use a factorizable deep sigmoidal flow in the variational distribution and show that leads to results comparable to those of MCMC methods in terms of posterior distribution estimation in high dimensional variable selection problem.

2.5 Experiments

2.5.1 High Dimensional Linear Regression

In this section, we assess the effectiveness of the proposed variational approach for variable selection in high dimensional linear regression settings, i.e. we assume a growing number of covariates p and a fixed number of samples n. We follow a similar set up as in [110], where they employ MCMC in posterior estimation using non-local priors. We fix $n = 100$, and set $p = 100, 500, 1000$. The covariates are generated using a Gaussian distribution, $\boldsymbol{x} \sim N(\mathbf{0}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$ is an exchangeable covariance matrix with variances all equal to 1 and off-diagonal correlations $\rho \in [0, 1)$. The data is generated according to a linear model, $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{y} \in \mathbb{R}^n$, $\boldsymbol{\beta} \in \mathbb{R}^p$, $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \phi^2 \boldsymbol{I})$ and $\boldsymbol{X}$ is an $n \times p$ matrix. In particular, we evaluate the performance of our procedure under different scenarios, with $\phi \in \{1, 4\}$, and $\rho \in \{0, 0.25\}$. In all cases, we consider only five truly non-zero coefficients $\beta_1, \beta_2, ..., \beta_5$, taking values $1, 2, 3, 4$,and 5 respectively.

from here on For model fitting, we considered three different versions of our variational approaches: the usual spike-and-slab prior with a Normal slab prior and the learned , and the proposed variations with an iMOM prior, either with or without the learned-hard concrete reparameterization. We denote the three variaional approaches as SaS-LHC and iMOM-LHC respectively). Also, we include comparison to the LASSO, AdaLASSO, and SCAD as Frequentist variable selection methods. The penalization parameter for LASSO, AdaLASSO, and SCAD are chosen by 10-fold cross-validation

we employed both a MCMC method with the iMOM prior and the proposed iMOM-LHCvariational methods, we empirically found if the temperature is low, the training is unstable in the beginning and are the estimated parameters are prone to degenerate to 0 value. Therefore we set the temperature to be a relatively high value 0.9. The initial value of $\log \alpha_j$ is set to zero, for all the covariates, so that the initial expected value of the learned hard concrete variable is $E\bar{s_j} = \frac{\alpha}{1+\alpha} = 0.5$.

For all of the variational methods, we set the prior probability to be 0.5, indicating un-informativeness on about whether each covariates should be included. And for MCMC inference of iMOM prior, we set a Beta-Binomial(1,1) on the model space. We cross-validate the hyperparameter τ of the prior on $\{1, 10\}$ and found that larger value of τ leads to a better log likelihood. For variational method, we found the models to converge in terms of likelihood score after 2000 iterations using Adam [70] optimizer with learning rate 0.05. All the variational methods are implemented in PyTorch [101] and run on a single card of TITAN X (Pascal) with 12 GB memory. And for MCMC method we use the R package 'mombf' with default choice of prior specification. More experimental details can be found in the Supplementary. All the experiments are repeated 50 times, we report the sum of square error between the estimated value coefficients and the ground truth value. We also report the precision and recall value of selection comparing to the ground truth non-zero coefficients. The results for $\rho = 0.25$ case are summarized in table **??**. More results can be found in supplementary material. We show that the flow based variational approach using iMOM prior (iMOM-LHC), is comparable to the iMOM-MCMC, which uses MCMC for posterior inference.

	$\phi = 1$						$\phi = 4$					
	p = 500		p = 1000		p=1500		p=500		p=1000		p=1500	
	SSE	P/R	SSE	P/R	SSE	P/R	SSE	P/R	SSE	P/R	SSE	P/R
LASSO	0.468	0.2 (1.0)	0.671	0.155 (1.0)	0.735	0.151 (1.0)	1.94	0.195 (1.0)	2.706	0.154 (1.0)	2.941	0.15 (0.98)
AdaLASSO	0.066	0.955 (1.0)	0.09	1.0 (1.0)	**0.112**	1.0 (1.0)	0.793	0.743 (0.98)	0.839	0.779 (0.94)	1.141	0.83 (0.92)
SCAD	0.247	0.971 (1.0)	0.186	1.0 (1.0)	0.293	0.967 (1.0)	0.691	0.532 (1.0)	0.835	0.578 (0.98)	1.053	0.553 (0.98)
iMOM-MCMC	0.071	1.0 (1.0)	0.073	1.0 (1.0)	0.074	1.0 (1.0)	1.299	1.0 (0.836)	1.302	1.0 (0.836)	1.224	1.0 (0.84)
SaS-LHC	0.566	0.668 (1.0)	0.134	0.957 (1.0)	0.061	0.997 (1.0)	6.335	0.258 (0.98)	3.77	0.421 (0.976)	2.703	0.58 (0.96)
iMOM	0.122	0.987 (0.996)	0.134	1.0 (0.996)	0.228	1.0 (0.976)	2.919	0.572 (0.948)	1.757	0.83 (0.928)	1.704	0.882 (0.908)
iMOM-LHC	0.097	1.0 (1.0)	0.108	1.0 (0.996)	0.166	1.0 (0.988)	2.083	0.781 (0.968)	1.183	0.916 (0.956)	1.185	0.92 (0.928)

Table 2.1: Comparison of Mean sum of square error of coefficient estimation (SSE) and precision/recall (P/R) of various methods: (a) iMOM prior with learned hard concrete reparameterization (iMOM-LHC) outperforms the iMOM prior without using LHC in both SSE and precision/recall, (b) in all of the settiings, iMOM-LHC has has higher precision than SaS-LHC, which indicates that it outperforms spike and slab prior in terms shrinking non-zero coefficients, (c) comparing to Frequentist methods, iMOM-LHC has higher precision especially at larger ϕ value, and (d) when $p = 1500$, it took 20x more time for MCMC algorithm to run 5000 iteration comparing to our variational algorithm for iMOM-LHC (1532.1 seconds on 2.3 GHz Intel Core i5 v.s. 75.6 seconds on single TITAN X (Pascal) GPU).

From Table **??**, we observe that in general the posterior estimate for non-local prior using our variational approach is comparable to the estimate provided MCMC methods. In additional the learned hard concrete reparameterization helps to improve both the variable selection precision/recall as well as parameter estimation especially when the signal to noise ratio $|\frac{\beta_j}{\phi}|$ is low. We show an empirical distribution of βs when $p = 1000, \rho = 0.25, \phi = 1$ in Figure 2.4.

Convergence speed We compare the convergence speed of iMOM-LHC and SaS-LHC. Figure 2.3 shows the sum of square errors (SSEs) between the estimated coefficients and the ground truth (both zero and non-zero) converges faster to zero using iMOM prior than using spike and slab prior. In particular, under the same setup, the SSE of coefficients with zero ground truth value converges to zero within 150 iterations using iMOM prior, compared to 1000 iterations using spike and slab prior, demonstrating the effectiveness of iMOM prior.

Comparison of posterior draws Figure 2.4 shows an example of the comparison between the variation posterior draws (iMOM-LHC) of the coefficients and the MCMC posterior draws (iMOM-MCMC), when $p = 500, \rho = 0.25$, and $\phi = 1$. Posterior draws from the five non-zero coefficients $\beta_1, \beta_2, ..., \beta_5$ displayed from left to right. It can be observed that the posterior draws of $\beta_i s$ from variational methods tend to have smaller variance than the MCMC draws. The underestimation of the variance is a consequence of the variational objective function which is commonly observed in variational infernce [28].

Comparison of SSEs Figure 2.5 shows the violin plots of the sum of square errors (SSEs) between the estimated coefficients and the ground truth coefficients, comparing SaS-LHC and iMOM-LHC. We observe that the iMOM-LHC outperforms SaS-LHC by having smaller SSE values. It is notable that when the signal to noise ratio $|\frac{\beta_j}{\phi}|$ is low (left plot), iMOM-LHC has a clear advantage compared to SaS-LHC across all the range of the number of the coefficients ps.

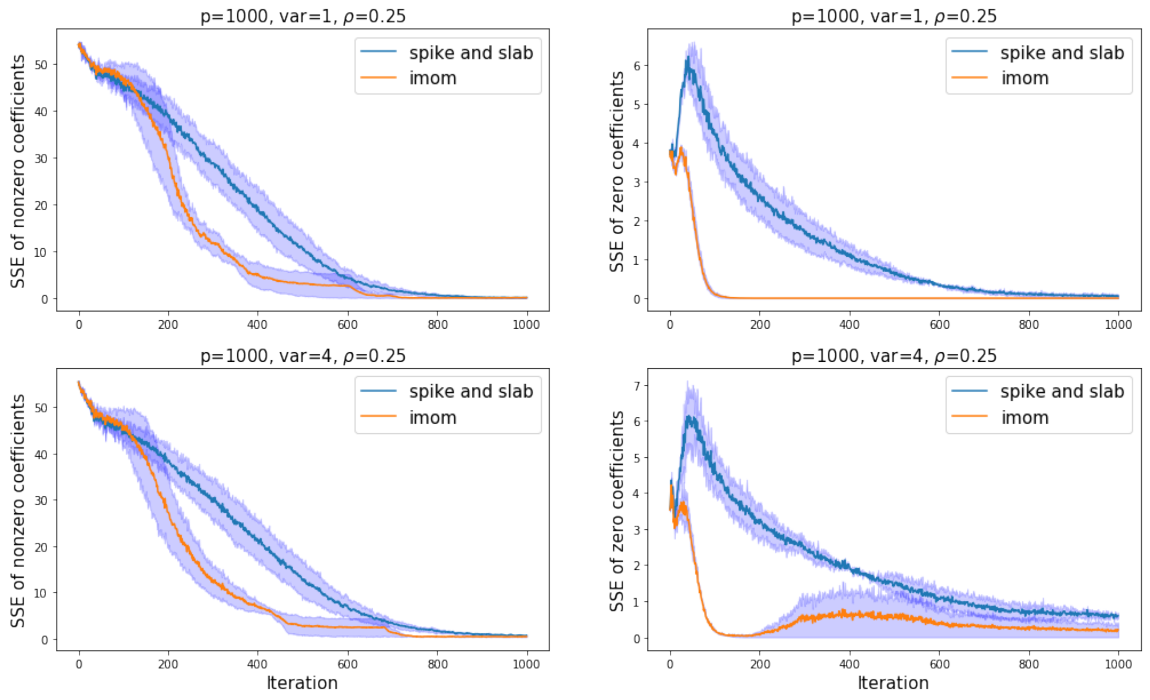


Figure 2.3: Comparing the convergence speed of the sum of square error for the parameter estimate during training in our variational algorithm. **Left column:** SSE of coeffcients with non-zero ground truth value. **Right column:** SSE of coeffcients with zero ground truth value. The error bar is showing for one standard deviation from the mean calculated using 50 repeated experiments. Using non-local prior results in faster convergence of both zero and nonzero parameters in all settings. And the variance of the estimates becomes larger when the residue variance ϕ increases

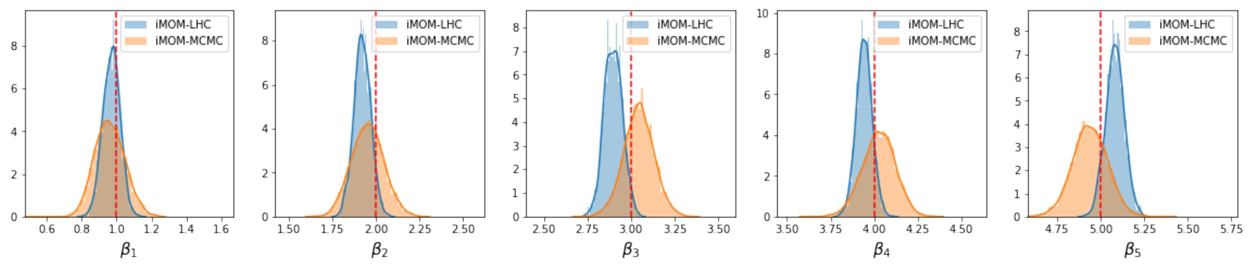


Figure 2.4: Histogram of 2000 variational posterior distribution draws (blue) using iMOM-LHC and the MCMC posterior draws (orange) using iMOM-MCMC. The red dashed vertical lines represent the ground truth values of the regression coefficients. Both iMOM-LHC and iMOM-MCMC are able to detect the non-zero coefficients and the posterior draws are close to the ground truth values. However, iMOM-LHC is much more scalable than iMOM-MCMC.

2.5.2 Identifying informative neurons and electrodes in rodent odor recognition

Dataset Description

We are particularly interested in associating neuron firing activity with rodent odor recognition. We used the data from a published experiment in which neural activity was recorded in the CA1 region of the hippocampus in rats as they performed a nonspatial sequence memory task [9, 121]. This task involves repeated presentations of a sequence of odors (e.g., odors ABCD) at a single port. On each trial, a single odor was presented and the rats were required to correctly identify whether each odor was presented "in sequence" (InSeq; by holding their nosepoke response until the signal at 1.2s) or "out of sequence" (OutSeq; by withdrawing their nose before the signal). Spiking activity was recorded using 14-24 tetrodes (bundles of 4 electrodes) in rats tested on well-trained or novel sequences. For each odor presentation (trial), the data typically features spike counts from 40-70 neurons and trial identifiers (e.g., neuron index, odor presented, InSeq/OutSeq, response correct/incorrect). Spiking activity from representative neuron is shown in Figure 2.7. The input data we use is the mean spiking rate of 46 neurons belonging to one rat (SuperChris) in the 0-500ms time window (relative to odor release). More experimental details can be found in the supplementary material.

Odor classification results at the neuron and electrode level

We first applied our method to identify neurons that are particularly informative in differentiating among the odors presented on each trial. The model used to predict the presented odor is a single hidden unit layer neural network with ReLU activation with input size 46 and output size 4. We then obtained the softmax of the output vector to produce the probability corresponding to the 4 odors. We compared the selection results of the model using spike

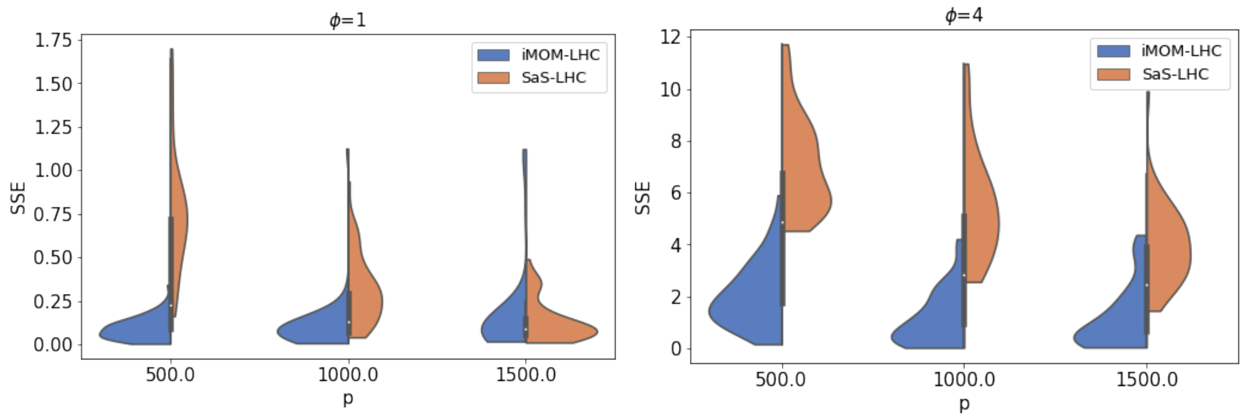


Figure 2.5: Comparison of sum of square errors (SSEs) between the estimated and the ground truth of the regression coefficients using iMOM-LHC (blue) and SaS-LHC (orange). The SSEs using iMOM-LHC is smaller than the SSEs using SaS-LHC.

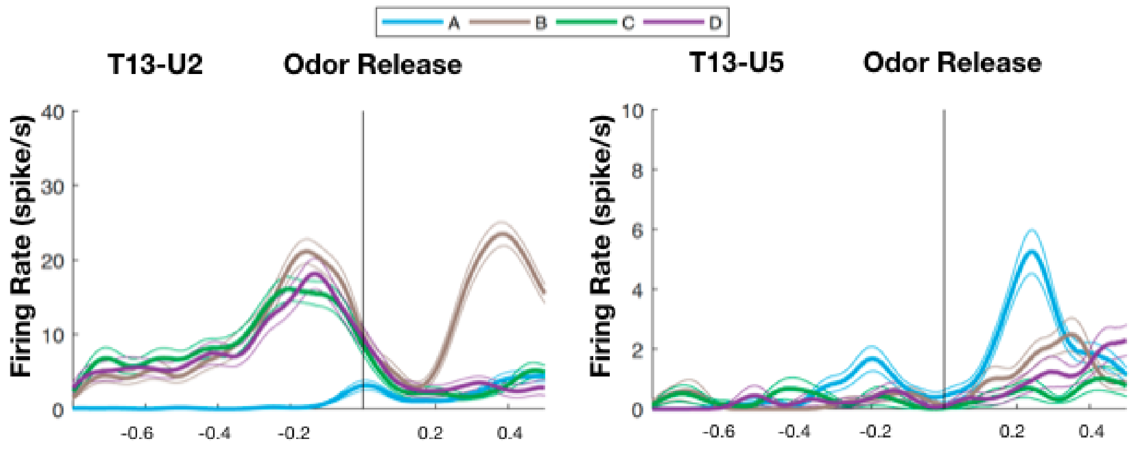


Figure 2.6: Dynamics of neuron spiking activity measured by spikes per second of the rodent named SuperChris. It shows the time window between -0.8 seconds and 0.5 seconds relative to odor release for Tetrode 13 unit 2 (T13-U2) and unit 5 (T13-U5).

and slab prior and non-local prior, and the prior probability of each neuron being activated is set to be 0.5. The initial values of α_js and the temparature are the same with linear regression settings. We performed a 5-fold cross validation and reported the mean predictive performance of our model. After training, the activation probability π_j for each neuron or tetrode is well separated, and in practice we choose to select with threshold at 0.01.

We found that, although the majority of neurons (37 out of 46) contributed very little information (probability $\leq .048$), 6 neurons were highly informative (.922, .926, 809, 0.879 .802, .685, .614,). A detailed examination of the activity of these 6 neurons using standard peri-event rasters and histograms confirmed that their activity differentiated across odor presentations (see example neuron in Figure (2.7), suggesting this model can be used as an efficient screening tool to identify informative neurons in large datasets. We then applied our model at the tetrode level by sharing the parameter α_j for neurons within the same tetrode group while allowing the parameters η_j in $q_{\eta_j}(\beta_j|z_j = 1)$ to be different for each neuron in the group. This allows us to enforce a tetrode-wise sparsity pattern in order to probe the spatial association of neural activity and odor recognition. As expected, we found that the tetrodes with the highest probabilities (T13, T14, T19, and T1) included the 6 neurons with the highest probability mentioned above. In addition, we also found that tetrodes T1 is moderately informative (probabilities of .325). This result suggests the model can also be used at the tetrode level to capture informative population activity patterns not detected at the level of individual neurons.

Finally, we examined the spatial distribution of informative electrodes within CA1. We found that high-information tetrodes appeared to cluster in the distal segment of CA1 (the bottom-right corner of Figure 2.8). This observation is consistent with evidence that this part of CA1 receives strong input from regions processing olfactory information [7, 137]. Applying the model to our experimental data allows us to confirm this finding. Further we are working on increase the number of units recorded in each CA1 sector to more extensively control for the potential influence of variations in the number of neurons per tetrode location (or in their individual firing rate). These results suggest that our model will be a useful tool to quantify the topographic organization of coding properties within and across brain regions.

2.6 Discussion

We present a practical approach for scalable approximate Bayesian inference for high-dimensional variable selection via non-local priors. To allow for a flexible variatinal distribution, we use the factorizable normalizing flow as our variational family. We propose a learned hard concrete relaxation scheme allowing the model to learn an optimal value of the scaling constant of the concrete distribution, hence allieviating the need for manual search of the scaling constants. We applied our method on variable selection in high dimensional linear regression, where it is shown to be comparable with MCMC estimation while can easily provide at least an order of magnitude of speed up. We test our sparse learning approach on rodent odor recognition data and successfully identify informative spatial patterns in neuron activity which is believed to be associated with odor recognition.

Our approach opens a route for further work on effectively tackle variable selection problem where Bayesian inference faces computational difficulty due to high dimensionality of the parameter space, such as brain connectivity and genetic study. Also, we are working on comparing our methods with more related feature selection methods and applying our method to neural network compression.

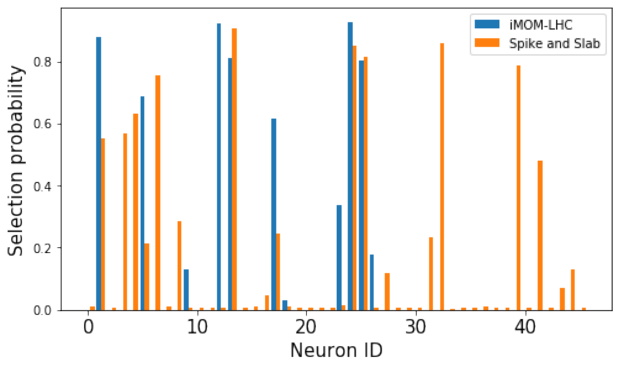


Figure 2.7: Neuron selection probability comparing spike and slab prior and iMOM prior. Using the iMOM prior results in a sparser solution while maintaining accuracy.

	Neuron Level	
	Acc	Avg num selected
SaS-LHC	57.58% (0.04)	12.6
iMOM-LHC	68.48%(0.09)	8.2

Table 2.2: Neural level selection results by spike and slab and iMOM prior.

	Tetrode Level	
	Acc	Avg num selected
SaS-LHC	61.82% (0.132)	8.0
iMOM-LHC	64.24% (0.08)	5.8

Table 2.3: Tetrode level selection results by spike and slab and iMOM prior.

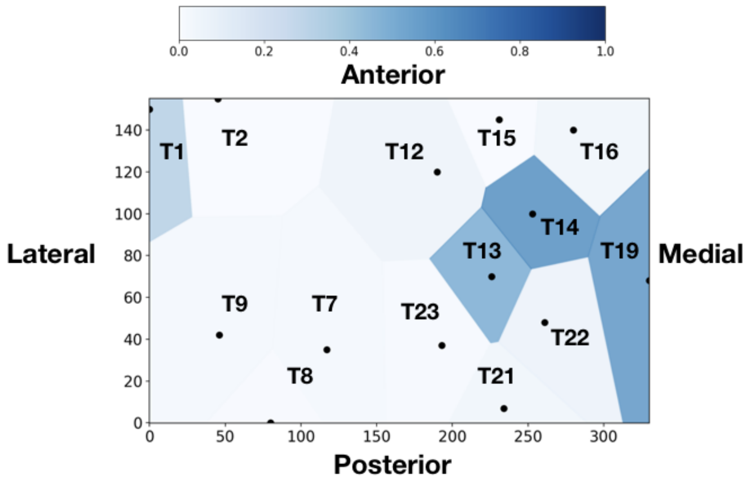


Figure 2.8: Voronoi diagram showing tetrode-wise selection probability (the darker the higher) by iMOM-LHC. Tetrode T13, T14 and T19 in the medial area (CA1) of the brain are selected with clearly higher probability than the rest of the tetrodes during rats' odor recognition process, corroborating with the evidence that CA1 receives strong input from regions processing olfactory information.

Chapter 3

SARM: Sparse Autoregressive Models for Scalable Generation of Sparse Images in Particle Physics

3.1 Background

Experiments in particle physics seek to uncover the building blocks of matter and their interactions, which determine the structure of the Universe from subatomic to cosmic distances. Analyses of the data produced by these experiments make extensive use of simulations to predict the experimental signature of particle interactions under various theoretical hypothesis. These simulations are used in likelihood-free inference as well as in the development of data selection and analysis strategies which optimize the statistical power of the data. Current state-of-the-art simulators apply Monte Carlo techniques to the microphysical processes governing individual particles' propagation and interaction [6], making them computationally expensive [5, 107].

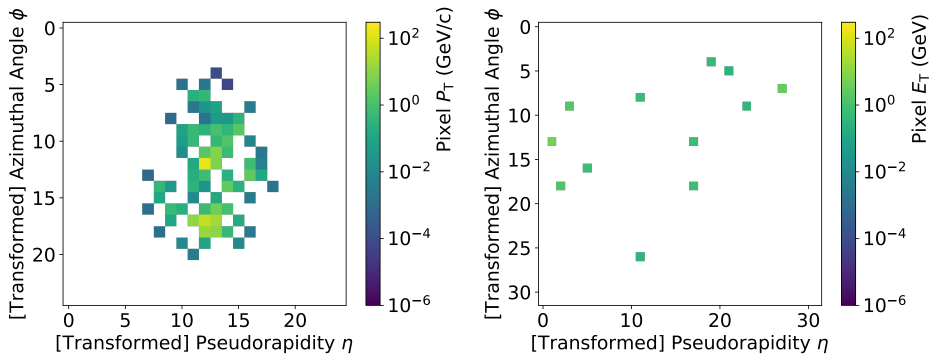


Figure 3.1: Calorimeter images in particle physics are often very sparse, where most of their pixels have very small values. **Left**: Typical signal image of a hadronic jet from [39] **Right**: Typical signal image of the vicinity of a muon from [81].

Detectors in particle physics experiments have a multi-layer architecture which produces highly structured data. One essential layer, the calorimeter, measures the energy of passing particles, and is subdivided into small cells to ensure spatial resolution. In collider experiments, the calorimeter is typically cylindrical [95], while in fixed-target experiments it may be a surface [4]. In both cases, the data can be represented as an image, allowing for the application of image-processing methods initially developed for natural images. However, in contrast to natural images, pixels in calorimeter images (figure 3.1) are very sparse, where usually 90% or more of the pixel values are zero. In addition, these images are not as uniform as natural images, featuring clusters in the center and noise in the periphery.

Recently, deep generative models [49, 72, 97] have produced high-quality artificial natural images [140, 30, 73] at a relatively low computational cost. The successful application of machine learning in high energy physics [22, 41, 122, 21, 52, 114, 119, 19], and generative models in natural images has inspired the use of these models for generating image-like data in physical sciences applications [39, 89, 88, 139, 104, 13, 40, 31, 120, 36, 54, 98], often employing Generative Adversarial Networks (GAN) [49] or, less frequently, Variational Auto-encoders (VAE) [72]. However, the extreme sparsity of the images in particle physics and other areas of the physical sciences [35] presents unique challenges for generative models.

The leading applications of GAN-based generative models for sparse image synthesis in high-energy physics, LAGAN [39] and CALOGAN [99], make use of the ReLU activation function in the final layer to induce sparsity in the output image. The flat portion of the ReLU activation function can lead to many error gradients being zero at the output layer, creating challenges [34] for stochastic gradient descent [29, 70] methods. In addition, GANs are notoriously unstable during training [16] and can suffer from mode collapse, which restricts the diversity of events in the generated data [90, 105]. Despite these difficulties, GANs have been one of the most popular deep generative models in particle physics.

However, other generative models may be better suited for sparse data. For example, deep autoregressive models (ARMs) have also demonstrated impressive performance for generating natural images among likelihood based generative models [116, 97]. In this paper, we develop *sparse* autoregressive models (SARM), a class of ARMs specifically tuned to produce sparse images. We present a systematic approach for designing SARMs and demonstrate their effectiveness through multiple experiments. SARMs are stable during training with respect to hyperparameter variations and weight initializations. SARMs are also interpretable in the sense that it is possible for these model to produce an analytic likelihood for any given sample. We then evaluate SARMs on two benchmark data sets. Given their flexibility, SARMs may be applicable to areas beyond particle physics where sparse images must be generated.

3.2 Datasets

An important statistical task in the analysis of particle physics data is identifying the particle source of a particular detector signature. Below, we describe two datasets, one which distinguishes between the detector signatures of single quarks and collimated pairs of quarks, and a second which distinguishes between muons produced in isolation and those produced as part of a shower of particles.

3.2.1 Jet Substructure Study

Quarks or gluons produced in collisions leave a particular detector signature: a *jet*, or shower of collimated particles, which deposit most of their energy in a tight core. In many applications, it is important to distinguish the signatures of a single quark or gluon from that of a collimated pair of quarks, which may leave two potentially overlapping cores. This task is a natural setting for image-recognition algorithms, and has been the focus of many deep learning studies [35, 10, 38, 27, 74] which rely on simplified calorimeter simulations due to the cost of generating realistic samples. Thus, an inexpensive generation of realistic datasets would be very valuable as a classification training sample.

We use a set of benchmark jet images from Ref. [39], where a full description of this dataset can be found as well as the code to generate it. In this dataset, quark pairs from W-boson decay are labeled as signal and single quark or gluon jets are labeled as background images. The intensity of each pixel value represents the sum of the momenta transverse to the beam (P_{T}) over the particles which strike a particular cell. The images are generated using PYTHIA 8.219 [124] simulations of proton collisions at a center-of-mass energy $\sqrt{s} = 14$ TeV, selecting jets with $250 < P_{\mathrm{T}} < 300$ GeV. Instead of a realistic detector simulation, the calorimeter response is mimicked via a regular 0.1×0.1 grid in the η and ϕ coordinates. The jet images are constructed and preprocessed as described in [38], including the centering and rotations of the images. The resulting images are 25×25 pixels, with intensity values in the [0,276] range. We divide them into a training set containing 400,000 images for the signal and 400,000 images for the background, and a testing set containing 36,000 images for the signal and 36,000 images for the background. A typical image from this dataset is shown in figure 3.1. This dataset has a high degree of sparseness: more than 90% of its pixels are zero valued.

3.2.2 Muon Isolation Study

Muons leave a very clear detector signature which is difficult to mimic. However, physicists must distinguish between two modes of muon production: a rare mode in which muons are produced from the decay of a heavy boson and are isolated in the detector, and a second prolific mode in which muons are produced inside a jet, surrounded by other particles. Fluctuations in the jet can occasionally produce apparently-isolated muons.

We use a set of benchmark calorimeter images from [81], where muons from heavy bosons are labeled as signal and muons produced within jets are labeled as background. The signal muons are generated with the process $pp \to Z' \to \mu^+\mu^-$ with a Z' mass of 20 GeV/c^2. Background muons are generated with the process $pp \to b\bar{b}$. Both signal and background datasets are generated at a center of mass energy $\sqrt{s} = 13$ TeV. The collisions and immediate decays are simulated with MADGRAPH5 2.3.3 [14], SHOWERING AND HADRONIZATION WITH PYTHIA 6.428 [124], AND DETECTOR RESPONSE WITH DELPHES 3.4.0 [37] USING THE DELPHES ATLAS DETECTOR MODEL. Additional proton interactions are overlaid on top of the primary process, at a rate of 50 additional interactions per event. This dataset only considers muons with P_T in the range: $P_\mathrm{T} \in [10, 15]$ GeV/c. The signal events are weighted to match the transverse muon momentum distribution of the background events. The calorimeter images in the vicinity of the muon are created from the calorimeter deposits within $\eta - \phi$ radius of $R < 0.4$, where each pixel represents the momentum transverse to the beam axis. The deposits are preprocessed by centering the image on the coordinates of the identified muon propagated to the calorimeter. The images are pixelated using a 32x32 grid to roughly match the granularity of the calorimeters of ATLAS and CMS, and the pixels have values in the range [0, 172]. The training set contains 41250 signal images and 41246 background images, and the testing set contains 41344 signal images and 41151 background images. A typical image from this dataset is shown in figure 3.1. This dataset has an even greater level of sparsity: more than 98% of its pixels have zero-value.

3.3 Autoregressive Models (ARMs)

Autoregressive models (ARMs) approximate a high dimensional data distribution $P_{\text{data}}(\mathbf{x})$ with $P(\mathbf{x})$, the distribution induced by the model where $\mathbf{x} \in \mathbb{R}^D$. For example, when working with images, $P_{\text{data}}(\mathbf{x})$ represents the distribution of the values of D pixels in the image. ARMs are generative models that create outputs sequentially, where each new output is conditioned on the previous output [75]. Formally, ARMs transform the problem of learning the joint distribution $P_{\text{data}}(\mathbf{x})$ into learning a sequence of tractable conditional distributions $P(x_i|x_{j<i})$. The ordering of the pixels can influence the model's performance and will be discussed later in the paper. ARMs rely on the basic factorization:

$$\begin{aligned} P(\mathbf{x}) &= P(x_0, x_1, \dots, x_D) \\ &= P(x_0)P(x_1|x_0)P(x_2|x_0, x_1)\dots P(x_{D-1}|x_0 \dots x_{D-2}) \end{aligned} \tag{3.1}$$

The conditional densities $P(x_i|x_{j<i})$ can be parameterized by deep neural networks [46, 97, 116, 131] so that: (1) $P(x_i|x_{j<i}) = P(x_i|\theta_i)$, where θ_i represents the parameters of a distribution (e.g. mean and standard deviation); (2) $\theta_i = f_i(x_0, \dots, x_{i-1})$, such that θ_i depends on previous output; and (3) the function f_i is implemented by a neural network. At generation time, the pixel values x_i are generated sequentially by sampling in order from the distributions $P(x_i|\theta_i)$. A simplified implementation of this process using a single neural network is depicted in figure 3.2. The weights of the neural networks that compute the θ_i's are shared across different values of i, for regularization [131] purposes and to reduce computational costs, hence the zero-padding of the input vector.

A common concern with ARMs is that by generating pixels in sequence, conditioning only on previously visited pixels, the model may not be able to take into account the dependence of a current pixel on subsequent pixels. However, this is not the case because the weights are trained using all the data (i.e. "past" and "future" pixels) and the model always learns to

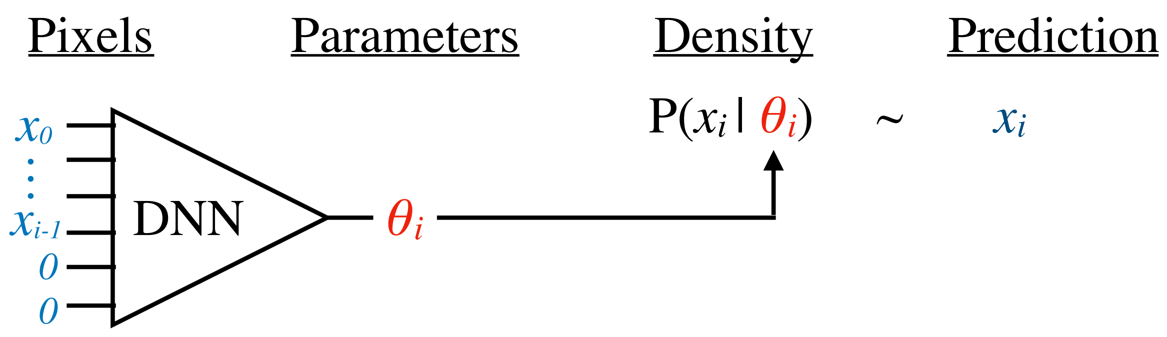


Figure 3.2: Pixel generation process by a deep ARM to create an image with D pixels. For the pixel x_i, a deep neural network (DNN) is evaluated on a vector with values $x_0, \ldots, x_{i-1}$, zero-padded to length D. The output of the network are the parameters θ_i of a parametric probability density $P(x_i|\theta_i)$, from which x_i is sampled.

generate the joint marginal distribution of previous and current pixels. This idea is further illustrated with a toy example in Appendix B.1.

Learning in ARMs is different from learning in other generative models such as GANs and VAEs. ARMs directly minimize the discrepancy, in terms of KL divergence, between the data distribution $P_{\text{data}}(\mathbf{x})$ and the model distribution $P(\mathbf{x})$ which is produced explicitly. In contrast, neither GANs nor VAEs produce a tractable marginal likelihood model $P(\mathbf{x})$ and, as a result, they have to resort to approximations for minimizing the KL divergence between the data and model distributions. ARMs avoid this issue by sequentially modeling each conditional probability distribution, allowing them to minimize the KL divergence directly with a tractable likelihood $P(\mathbf{x})$. Leveraging the flexibility of deep neural networks to learn each conditional probability, ARMs are able to approximate a large family of continuous distributions in $\mathbb{R}^D$ [63].

The implementation of ARMs for images can follow several approaches [46, 50, 97, 116]. For scalability during training and generation, we use a single neural network to model the parameters of the conditional probabilities at each step, where some connections are

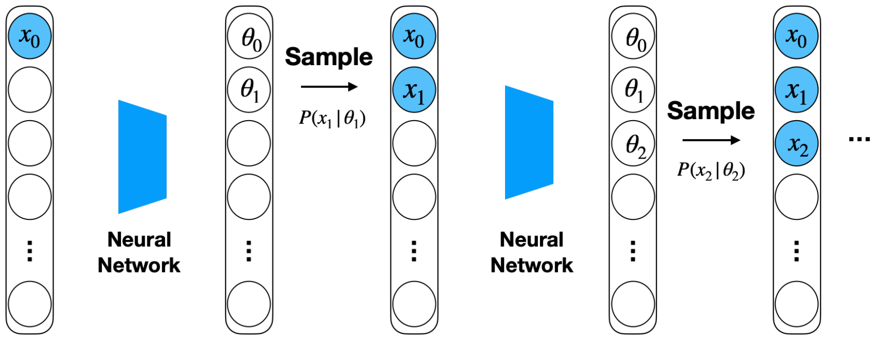


Figure 3.3: Generation process of a deep autoregressive model. During generation, the first pixel x_0 is sampled from $x_0 \sim P(x_0|\theta_0)$. Next, the pixel x_0 is zero-padded to a D dimensional vector and passed to the neural network ARM model, which evaluates the parameters $\boldsymbol{\theta} = \{\theta_0, \ldots, \theta_{D-1}\}$, though only θ_1 is needed to sample the next pixel $x_1 \sim P(x_1|\theta_1)$. The pixels x_0 and x_1 are again zero-padded to create a D dimensional vector which is passed into the neural network to generate the next pixel. This process is repeated until all pixels are generated. Note that the same neural network is used at each generation step, and part of its weight connections are disabled to preserve the autoregressive structure.

intentionally disabled to preserve the autoregressive structure (see Appendix B.2), similar to the structure used in [46]. Given a training image, this makes it possible to calculate *all* the parameters $\theta_0, \ldots, \theta_{D-1}$ in parallel, instead of calculating each θ_i sequentially. During generation, the model generates the output elements one-by-one as illustrated in figure 3.3.

3.4 Sparse Autoregressive Models (SARMs)

To deal with sparsity in images, we introduce sparse ARMs (SARMs) in which each conditional distribution is a mixture comprising a Dirac delta distribution at the zero pixel value, as one of its components. The probability associated with the zero-pixel value is learnable by gradient descent, providing a flexible and efficient way of modeling and fitting highly sparse datasets. The other components of the mixture can be modeled in different ways, as described below.

3.4.1 Sparse Images Likelihood Models

In SARM, the likelihood function for the i-th pixel x_i is formulated as:

$$p(x_i|\theta_i) = \gamma_i \cdot \delta_{x_i=0} + (1-\gamma_i) \cdot \delta_{x_i \neq 0} \cdot p(x_i|\phi_i) \tag{3.2}$$

where the parameters $\theta_i = \{\gamma_i, \phi_i\}$ are predicted by the underlying neural network taking $x_0, \ldots, x_{i-1}$ as its inputs. Since the pixel values in the calorimeter images represent the physical deposition of energy, they must be non-negative, i.e. $p(x_i|\phi_i) > 0$ only when $x_i > 0$. To satisfy this constraint, we explore two options. First, we use a mixture of a Dirac delta distribution at zero with a discrete distribution for the non-zero pixels (D+D). Second, we use a mixture of Dirac delta distribution at zero with a continuous distribution for the non-zero pixels (D+C).

Discrete Mixture Model (D+D): We discretize each pixel value x_i by rounding it to the nearest value in a pre-determined grid with points $\{0, g_1, \ldots, g_N\}$, where $g_j > 0$ for j from 1 to N, and g_N corresponds to the largest pixel value after rounding. The model learns the probability of each discrete value as a categorical distribution:

$$p(x_i|\theta_i) = \gamma_{i,0} \cdot \delta_{x_i=0} + \sum_{j=1}^{N} \gamma_{i,j} \cdot \delta_{x_i=g_j} \tag{3.3}$$

where each $\gamma_{i,j}$ is predicted by the parameter $\theta_i = (\theta_{i0}, \ldots, \theta_{iN})$ using a softmax function. When the grid is uniform, this likelihood is the same as the discretized softmax likelihood used by Pixel RNN [97], which has achieved state-of-the-art results on benchmark datasets of natural images. [42]. However, in particle physics the distribution of pixel values is typically far from uniform. In many typical cases, there is a large number of pixels with small values, and a few pixels with large values, as seen in figure 3.5. To better represent the pixel distribution and minimize the error due to quantization, we assign more grid points to the region of low pixel values. We achieve this by using a power transformation $\hat{x} = x^{1/p}$ on the pixel values, where p is a hyperparameter such that $p \geq 1$.

Discrete and Continuous Mixture Model (D+C): The pixel values of natural images are usually represented by unsigned integer values between 0 and 255. However, in particle physics images, the pixel values are typically real-valued. To avoid explicit rounding, SARM (D+C) is built with a truncated logistic distribution that models the non-zero distribution component of each pixel. To generate the D+C mixture, we reparameterize each pixel as $x_i = \tilde{x}_i \cdot z_i$, where $\tilde{x}_i$ follows a truncated logistic distribution $TL(\mu_i, s_i)$ with mean μ_i and scale parameter s_i. Here $z_i \sim \text{Bern}(\gamma_i)$ is a Bernoulli random variable with probability $p(z_i = 1) = \gamma_i$, which controls the sparsity level. By assuming independence of $\tilde{x}_i$ and z_i, the likelihood function of x_i becomes:

$$p(x_i|\theta_i) = \gamma_i \cdot \delta_{z_i=0} + (1 - \gamma_i) \cdot \delta_{z_i \neq 0} \cdot p(\tilde{x}_i|\mu_i, s_i) \tag{3.4}$$

where $\theta_i = \{\mu_i, s_i, \gamma_i\}$ are functions of the previous pixel values $x_{0:i-1}$, to ensure the autoregressive structure. In order to allow for unconstrained optimization, we treat $\log(s_i)$ as the learning parameter and take its exponential in the likelihood equation 3.4. Since the pixel distribution could be multi-modal, we use a mixture of truncated logistic (MTL) distributions for $\tilde{x}_i$ which is more flexible.

The mixture of truncated logistic likelihood differs from the discretized logistic mixture used in Pixel CNN++ [116] in the way it handles continuous pixel values. Pixel CNN++ requires discretizing x_i and then maximizing the probability on the discretized grid. In contrast, SARM can directly maximize the probability density function of x_i, allowing it to handle continuous pixel values without incurring quantization errors.

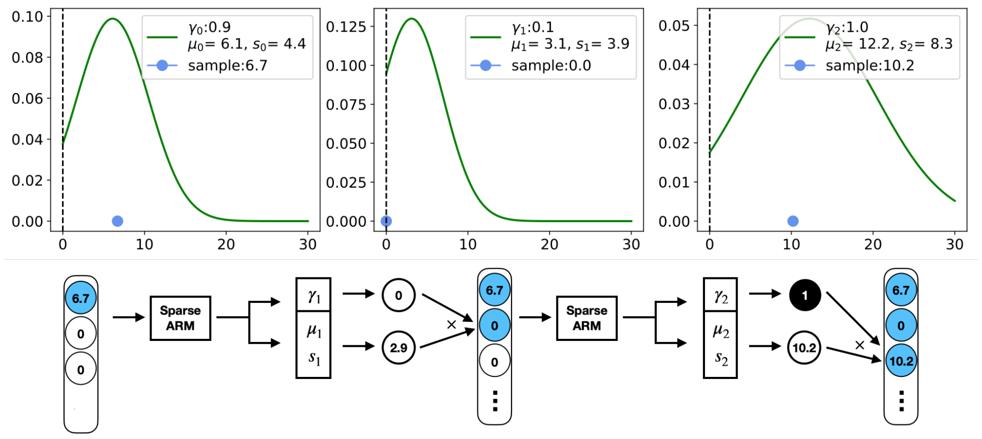


Figure 3.4: Generation process for the D+C model. The blue circle dots represent the value sampled for each pixel. For example, given the first pixel value of 6.7, sampled from the empirical distribution of the dataset, the neural network outputs the distribution parameters $\gamma_1 = 0.1, \mu_1 = 3.1, s_1 = 3.9$ to generate the second pixel. Then a Bernoulli random variable is sampled from $z_1 \sim \text{Bern}(\gamma_i)$ and a logistic random variable is sampled from $\tilde{x}_i \sim \text{Logistic}(\mu_i, s_i)$. The value of the second pixel x_i is produced by the product of these two variables as: $x_i = z_i \cdot \tilde{x}_i = 0 \cdot 2.9 = 0$. This sequential process is repeated until every pixel is generated.

There are several differences between the D+D and the D+C models. The D+D model allows enough flexibility to represent multi-modal distributions, as each grid point has its own learnable probability. However, there is a price for this flexibility. It is significantly more time-consuming to generate an $(N + 1)$-way softmax vector and sample from a discrete mixture (D+D) than it is to generate the parameters of γ, μ, s and then sample from a discrete and continuous mixture (D+C). Other constrained domain distributions such as the exponential and the gamma distributions were also considered but led to inferior results. The exponential distribution suffers from a lack of flexibility due to having only one learnable parameter.

3.4.2 Multi-Stage Generation for Heterogeneous Areas

In many ARM applications, a single network is used to predict the parameters θ_i of the conditional probability distribution $P(x_i|\theta_i)$. This approach works well if the distribution of

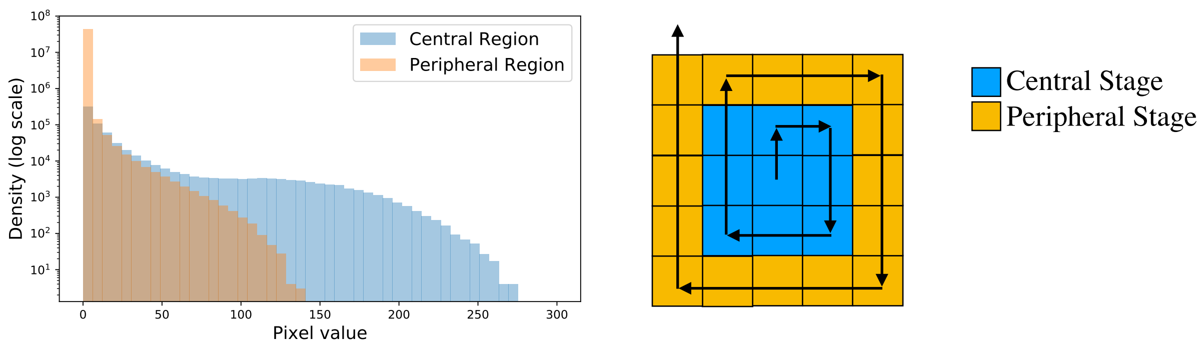


Figure 3.5: **(Left)** Distribution of pixel values in the jet substructure dataset for the 9 pixels in the center of the images (central region) and the rest of the pixels (peripheral region). Note that the majority of the pixels in the peripheral region are zero-valued and in general have lower variance than pixels in the central region. **(Right)** Two-stage generation for the central and peripheral regions using a spiral path and two different SARM modules. Using different networks for each region improves performance.

pixel values is similar across pixels, as is often the case in natural images. However, as shown in figure 3.5 (left), the pixel value distribution in the central square of a calorimeter image containing a jet is very different from the distribution in the rest of the image (see also [38]). In order to handle these heterogeneous regions, we use a two-stage approach by stacking two distinct deep SARM modules, one for the center and one for the periphery. When the model generates the image from the inside out, the outer module generates pixels conditioned on the outputs of the center module, as illustrated in figure 3.5 (right). We refer to this model as SARM-2 while the single stage model is SARM-1. Since the center may not have a clear border, we treat the size of the center relative to the periphery as a hyperparameter during training. Note that in general the number of stages depends on the structure of the data and is not limited to two. Furthermore, it is possible to learn the SARMs associated with each region in any order.

Thus, in summary, through the experiments to be presented, we show that a good heuristic approach for SARM design is to: (1) decompose the images into relevant regions (e.g. center vs background); (2) use a different SARM for each region type; and (3) within each region type, preferably choose a systematic and congruent order for generating the pixels, as these compare favorably to random generation orders. By systematic and congruent orders we mean orders that have some kind of continuity for the location of the pixels being generated–subsequent generated pixels should be close in the image–while respecting the geometry of the highly activated region (e.g. a spiral order for a globular region, a linear order for a linear region).

3.5 Evaluation Methods

The goal is to train generative models which create images indistinguishable from images created by the slower Monte Carlo methods. We compare the performance of our models, both in terms of image quality and generation time, against two other generative models: LAGAN [39], the current state-of-the-art generative model for sparse images in particle physics; and Pixel CNN++ [116], a widely used autoregressive model for natural images not tuned for sparse images. We evaluate all models on both datasets described above; note that LAGAN was designed to handle images typically found in the jet substructure dataset, while the muon dataset features extreme sparsity in comparison. We measure the quality of the generated images both qualitatively and quantitatively.

Qualitative Evaluation: We examine typical images generated by each model, as well as the pixel-wise average intensity of the generated images, using the images produced by the Monte Carlo methods, which in the jet substructure study are referred to as the Pythia images. Additional qualitative comparisons are described in the Appendix B.3 and B.4.

Quantitative Evaluation: Comparisons of distributions in high-dimensional datasets should focus on the scientific context and potential applications. In particle physics, the calorimeter information is typically used to calculate physical quantities, such as invariant mass or transverse momentum (P_{T}), which are especially revealing as metrics because they have not been explicitly optimized by the models. In addition, calorimeter images are used to train classifiers which can identify particles from their patterns of depositions.

One-dimensional distributions of mass and P_{T} can be evaluated in comparison to the distributions from Monte Carlo generators using the Wasserstein distance, the minimum cost to transform one distribution into the other one, expressed by:

$$W_p(P,Q) = \left(\inf_{J \in \mathcal{J}(P,Q)} \int \|x-y\|^p dJ(x,y) \right)^{1/p} \tag{3.5}$$

where $\mathcal{J}(x,y)$ is the family of joint probability distribution of x and y; P and Q are marginal distributions, and $p \geq 1$. When $p = 1$, this metric is also known as the Earth Mover's Distance [112]. To match the results in [39], we computed $W_1(P,Q)$, where P represents one of jet observable distributions from the Pythia images, and Q represents the corresponding jet observable distribution from the generated images.

An important motivation for developing generative models for fast simulations is to provide a computationally inexpensive method to augment existing datasets in classification task [38, 20]. The jet substructure dataset was generated to train classifiers to distinguish between jets from W boson decays (signal) and those from single quarks and gluons, a well-known classification task [38, 20]. The muon isolation dataset was generated to train classifiers to distinguish isolated muons from those due to heavy-flavor jet production. Therefore, an essential test for the quality of the generated images is whether they can be used in these classification tasks. To quantify this, the generated images were used as training sets to develop a classifier whose performance was assessed using the Monte Carlo images. The same convolutional neural network architecture was trained with the same hyperparameters on five different data sets: Monte Carlo images, images generated by SARM-2 (D+C) images generated by SARM-2 (D+D), images generated by LAGAN, and images generated by Pixel CNN++. Because higher quality images should lead to improved classification of the Monte

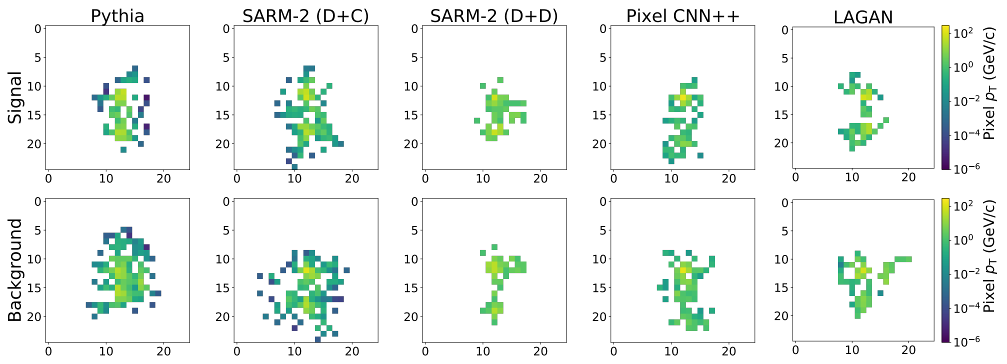


Figure 3.6: Example jet images generated from each model. Notice that SARM-2 (D+C) is able to produce small value pixels in the periphery of the images. The intensity of each pixel is shown on a log scale, where the white space represents pixels with value zero.

Carlo images, we used the classification performance as the evaluation metric.

Speed: Each generative model was used to generate batches of images multiple times to measure the average speed of image generation.

3.6 Results

3.6.1 Jet Substructure Study

Qualitative Analysis

An example image from each generative model and from the Pythia Monte Carlo generator is shown in figure 3.6. It is clear that SARM-2 (D+C) excels at generating pixels with small values around the periphery in comparison to the other models. Additional samples for each model can be seen in Appendix B.7. To assess the overall quality of the generated images, figure 3.7 shows the pixel-wise average of each dataset. The autoregressive models, SARMs and Pixel CNN++, are able to model the peripheral radial region around the center more accurately. This region has higher degree of sparseness than the center region, making it

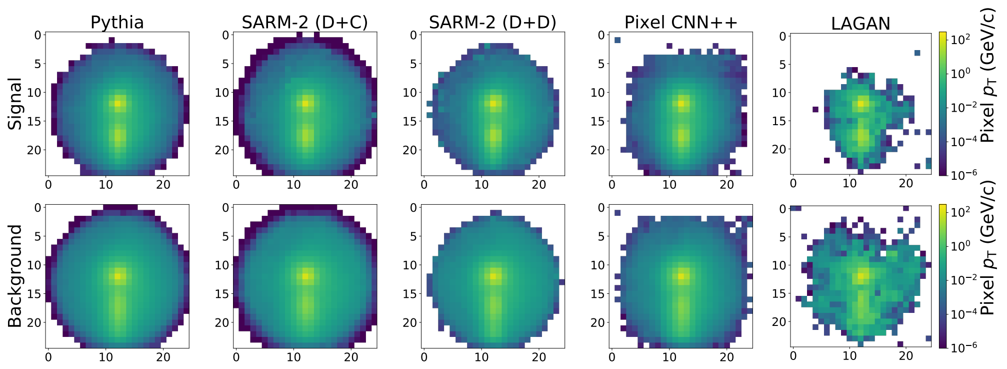


Figure 3.7: Pixel-wise average of the images generated by each model. Notice that LAGAN struggles to capture the distribution of low value pixels in the periphery of the images and has a non-smooth radial transition compared to the autoregressive models. The intensity of each pixel is shown on a log scale, where the white space represents pixels with value zero.

more challenging for the generative models to accurately capture. We note that the images from the SARM-2 (D+C) model appear to be most similar to the Pythia images, while the other models are less able to generate the peripheral region faithfully. In addition, Pixel CNN++ struggles to achieve the radial structure present in the Pythia images and creates a square-like structure instead. In general, the images from figure 3.7 generated by the autoregressive models show a smooth transition from the highly activated center to the sparse border, similar to that seen in the Pythia dataset. In contrast, the border of the LAGAN images is irregular, which could be due to its reliance on the ReLU activation function to induce the sparsity, making the model unable to estimate the sparseness level directly.

Quantitative Analysis: Jet Observables as Metrics for Quality

To quantify the fidelity of the images generated by each model as compared with the original samples, we insert them into typical applications in particle physics. In the context of collisions that produce jets, it is common to calculate the invariant mass of the jet, and the transverse momentum. Distributions of jet mass and P_{T} are shown in figure 3.8 for all models, which all succeed in matching the general shape, though discrepancies are visible, and Wasserstein distances are shown in table 3.1.

Table 3.1: Comparison of images created by various generative models with original images from Pythia, evaluated using the Wasserstein distance (with $p = 1$) between one-dimensional distributions of physical quantities calculated from the images: jet P_{T} and invariant mass, also shown in figure 3.8. Smaller values indicate a closer match to the Pythia images. Four SARMs are evaluated, those with either one-stage (SARM-1) or two-stage (SARM-2) models, and those with either discrete and continuous distributions (D+C) or a mixture of discrete distributions (D+D).

	P_{T}		Mass	
Model	Signal	Background	Signal	Background
LAGAN	3.15	3.29	1.45	1.39
Pixel CNN++	3.46	3.59	1.09	1.56
SARM-1 (D+C)	2.33	2.46	1.07	1.54
SARM-2 (D+C)	2.32	2.71	1.06	1.39
SARM-1 (D+D)	1.95	2.52	1.34	2.45
SARM-2 (D+D)	**1.44**	**1.66**	**0.94**	**0.92**

All SARM variants achieve lower distances in the P_{T} distributions than LAGAN and Pixel CNN+, and comparable or better distances in jet mass. The best results in all categories are obtained by the SARM-2 (D+D). Compared to the best of Pixel CNN++ and LAGAN, SARM-2 (D+D) provides a 51.92% improvement for P_{T}, and a 23.79% improvement for mass, averaged over the signal and background sets. These results demonstrate the effectiveness of taking sparseness into account during learning and generation. Secondly, the SARM-2 models clearly outperform the SARM-1 models for both the (D+D) and (D+C) likelihoods, which shows the effectiveness of the multi-stage approach in modeling heterogeneous areas in the images.

Classification of Generated Images

An important application of generated calorimeter images is to augment training sets for networks learning to perform vital signal-background classification tasks. As a high-level test of the image quality, we train networks using images generated by each model (200k signal, 200k background), and evaluate the performance on the original images from Pythia

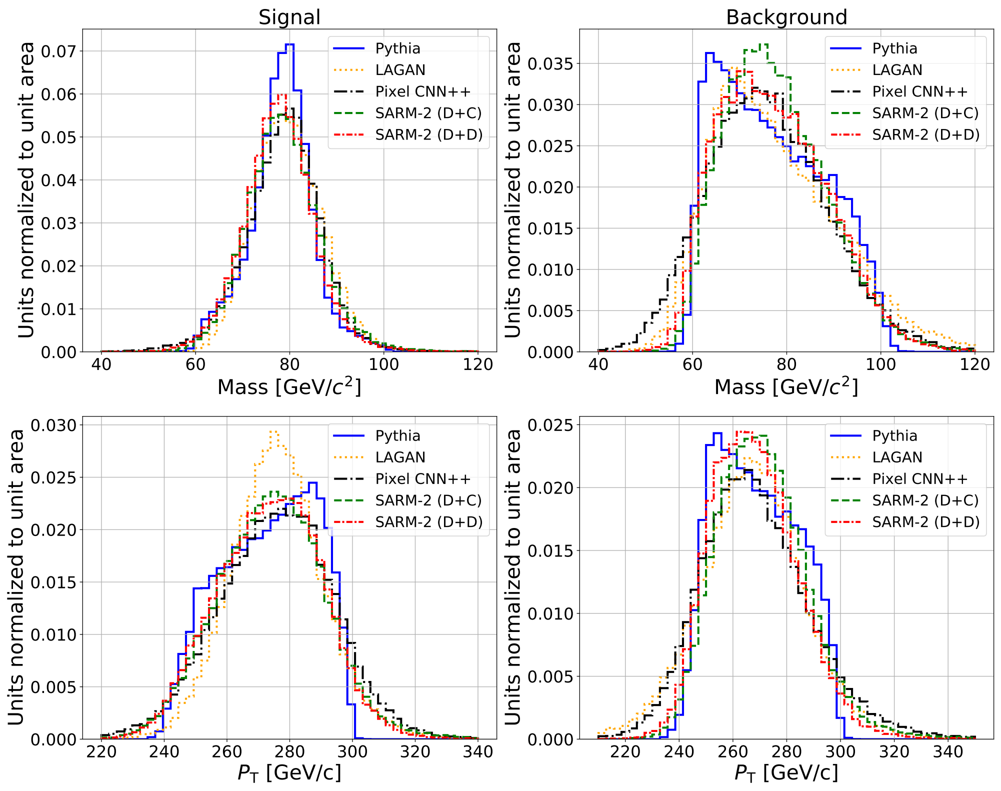


Figure 3.8: Distributions of jet observables (**Top**: Mass, **Bottom**: P_T) calculated from images generated by several generative models and from the original images generated by Pythia. Signal images, with two collimated quarks, are on the left; background images, with a single quark or gluon, are on the right.

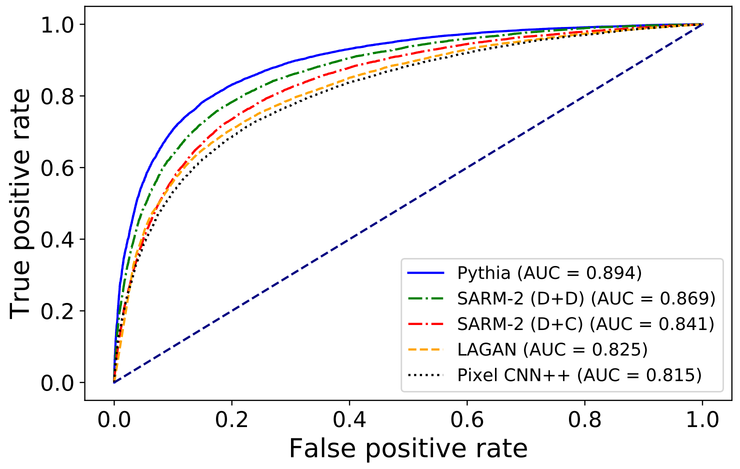


Figure 3.9: Evaluation of the fidelity of images generated by several models in the context of a classification task. Images generated by the model are used to train a network to discriminate between signal and background, but performance is measured using the original Pythia images.

(20k signal, 20k background). Training sets which best mimic the original Pythia images should lead to networks which most closely match the performance of a network trained on Pythia images. Detailed information about the classifier and training procedure are given in Appendix B.5. The receiver operating characteristic (ROC) curves for networks trained on images from Pythia , SARM-2 (D+C), SARM-2 (D+D), Pixel CNN++ and LAGAN are shown in figure 3.9. Classifiers trained on both SARM generated datasets have higher AUC (area under the ROC curve) scores than the classifiers trained on the LAGAN images and Pixel CNN++ images.

Generation Order

SARMs generate images pixel by pixel, conditioning each step on the previously generated pixels. The order of the pixel generation corresponds to a dependency decomposition in Equation 3.1, which may impact training performance. The traversal path is especially important for images containing heterogeneous areas. For natural images, [46] uses an ensemble of models with random paths, while Pixel CNN++ and other models [116, 97] use the row-by-row pixel ordering.

The average performance of various pixel orderings for SARM-1 (D+D) over 10 repeated runs is shown in table 3.2. Each order is evaluated by using the Wasserstein distance between the distributions of the generated signal images and the Pythia signal images for the jet P_{T} and invariant mass.

The spiral paths, clockwise (CW) and counterclockwise (CCW), achieve the stronger results. This could be understood in terms of mutual information between neighboring pixels. Unlike the other orderings, the spiral ordering is continuous, i.e. it always generates a pixel adjacent to the previously generated pixel. Furthermore, the spiral order is congruent with the globular shape of the highly activated region in the jet images, e.g. Fig. 3.7. Starting the spiral

Table 3.2: Quality of jet substructure signal images generated by SARM-1 (D+D) with various pixel-generation orderings. The quality is measured by the Wasserstein distance for the physical observables (P_{T} and mass) between the generated images and the original Pythia images. Spiral-in clockwise/counterclockwise (CW/CCW), spiral-out clockwise/counterclockwise (CW/CCW), column-wise, row-wise, and two random approaches are compared. The outward spiral orders show good performance due to the radial structure of the images.

	P_{T} (std)	Mass (std)
Spiral-out CCW	1.94 (0.09)	1.38 (0.10)
Spiral-out CW	2.47 (0.23)	1.53 (0.22)
Spiral-in CCW	3.64 (0.32)	1.62 (0.14)
Spiral-in CW	3.20 (0.22)	1.45 (0.16)
Row-wise	3.06 (0.30)	2.01 (0.11)
Column-wise	3.38 (0.39)	1.90 (0.08)
Random I	4.05 (0.51)	1.74 (0.53)
Random II	3.41 (0.33)	1.25 (0.26)

from the center outperforms inward spirals, indicating that it may be easier to learn the correlations between the pixels starting with pixels that are more active (more non-zero pixel values). The difference between CW and CCW is likely due to asymmetries generated by the rotation and centering steps in the preprocessing of the data. We use this asymmetric version of the data in order to enable direct comparison to the LAGAN model. These results confirm that non-random, systematic, generation orders that have good continuity and congruence properties perform well (and outperform random orders). A full exploration of the ordering dependency is beyond the scope of this work and computationally challenging due to the factorial number of possible orderings.

Computational Costs

Table 3.3 shows the speed of the generative models in comparison to the Monte Carlo method (Pythia). The SARM-2 models are five times slower than LAGAN, which is mainly due to the extra computational cost of the autoregressive structure. On the other hand, the SARM-2 models are two orders of magnitude faster than Pythia and Pixel CNN++. The forward pass of the Pixel CNN++ model is computationally expensive due to the ResNet blocks with convolutional layers and skip connections [116, 56]. In contrast, SARMs use a simple feed forward network with disabled connections to preserve autoregressive structure. The speed of the generative models is measured on a machine with 4 TITANX GPU cards each with 12G of memory. The speed of Pythia was assessed in [39] using Amazon Web Services (AWS) and an IntelR XeonR E5-2686 v4 at 2.30GHz CPU.

Table 3.3: Comparison of image generation speed between the Monte Carlo approach (Pythia) and various generative models. The SARM-2 models are slower than LAGAN, but still considerably faster than Pythia and Pixel CNN++.

Model	Speed (images/sec)
Pythia [39]	34
Pixel CNN++	50
SARM-2 (D+D)	1612
SARM-2 (D+C)	2480
LAGAN	10176

There is room to further optimize the speed of the SARM models. For instance, we find that reducing the size of the intermediate upsampling layer of the SARM (D+D) drastically reduces the memory requirements and improves the generation speed. Another direction is to explore model pruning and compression.

3.6.2 Muon Isolation Study

Qualitative Analysis: Average Generated Images

Typical calorimeter images in the vicinity of a muon generated by the standard Monte Carlo method, Pixel CNN++ as well as two SARMs are shown in figure 3.10. In this context, LAGAN suffered from mode collapse and failed to generate reasonable quality images (See figure B.5 in the Appendix). This is a well known problem when training GANs [16, 90, 39], especially with sparse data.

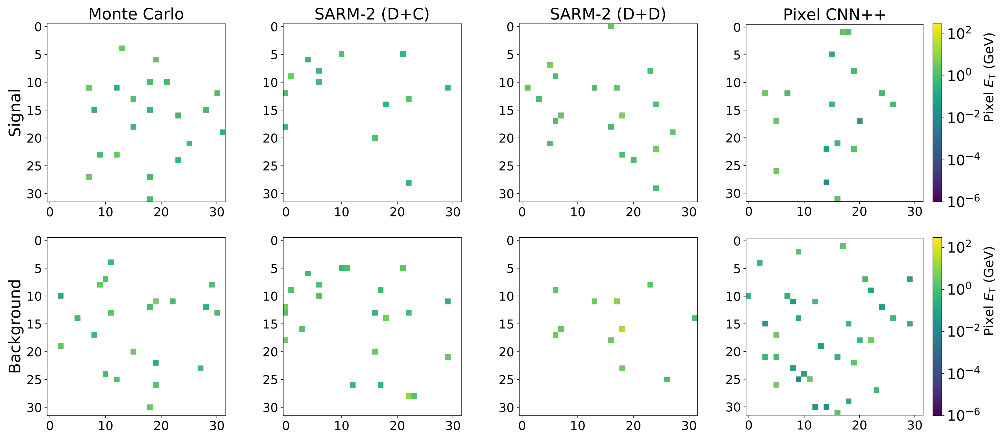


Figure 3.10: Example calorimeter images in the vicinity of a muon from the generative models as well as the original Monte Carlo generator. Top row shows isolated muons (signal), while the bottom shows muons produced in association with a jet (background). The intensity of each pixel is shown on a log scale, where the white space represents pixels with value zero.

Figure 3.11 shows the pixel-wise average images. The SARM-2 models and the Pixel CNN++ reproduce the radial symmetry seen in the original images. However, the average images produced by Pixel CNN++ contain noticeable artifacts, potentially due to the convolutional layers in the model [96].

Quantitative Analysis: Calorimeter Observables as Metrics for Quality

To assess the fidelity of the images quantitatively, we calculate physical quantities which summarize the content of the images and allow for comparison of one-dimensional distributions. While calorimeter images in the vicinity of a muon do not necessarily contain a clustered jet, the total P_{T} and invariant mass of the entire image do have physical meaning. Figure 3.12 shows the distributions of these quantities for the original Monte Carlo images, as well as for the generated images, and table 3.4 provides the corresponding Wasserstein distances.

The datasets generated by both SARM-2 models have considerably smaller Wasserstein distances than the datasets generated by the Pixel CNN++ model for both signal and background. The distributions of all the generated datasets approximate the shape of the

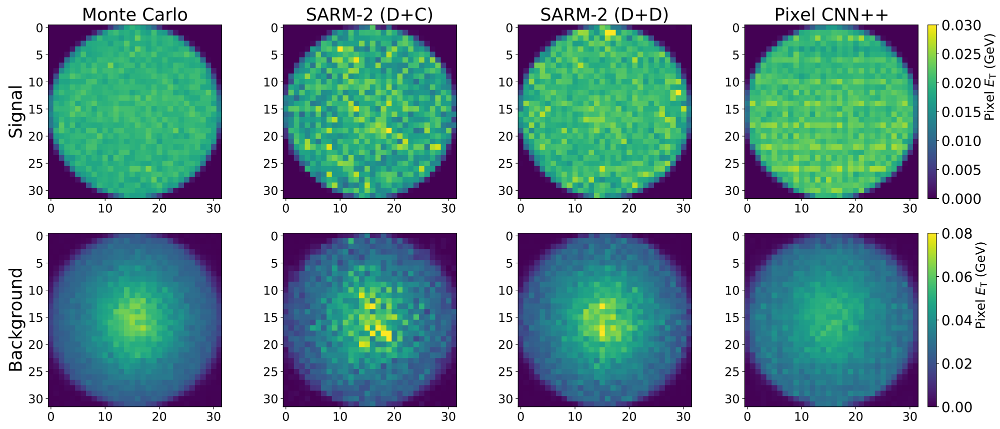


Figure 3.11: Pixel-wise averages of calorimeter images in the vicinity of a muon from the generative models as well as the original Monte Carlo generator. Top row shows isolated muons (signal), where little calorimeter activity is expected. The bottom row shows muons produced in association with a jet (background), which deposits significant energy near the muon. A linear scale is used to reveal the differences between signal and background images.

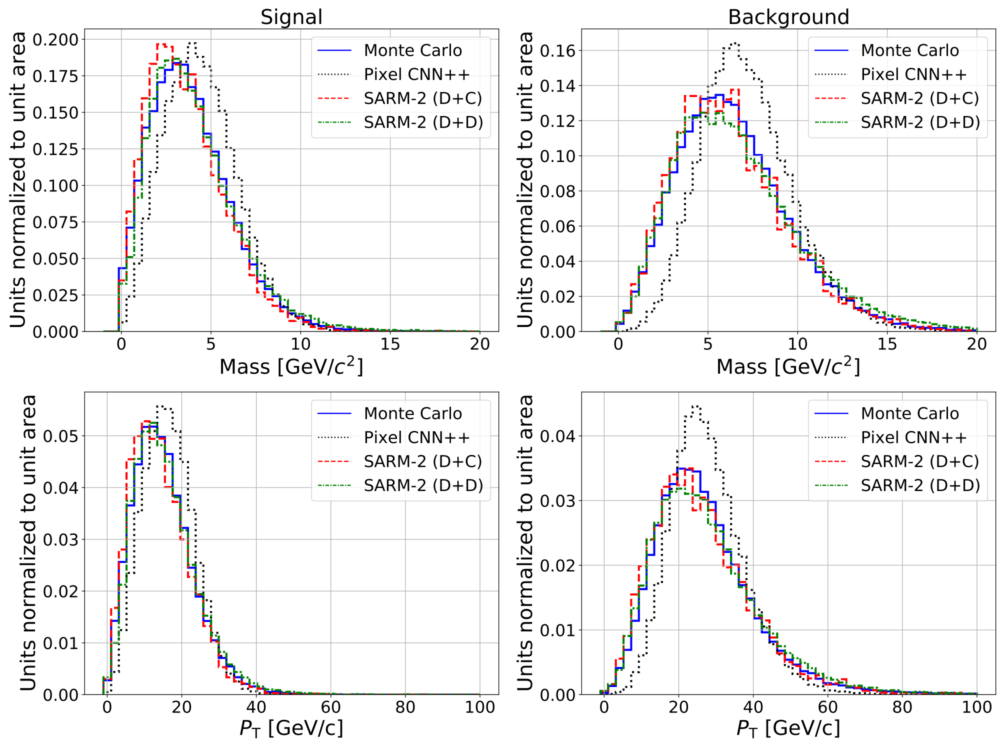


Figure 3.12: Distributions of calorimeter observables (top: invariant mass, bottom: total P_T) calculated from images generated by several generative models and the originals generated by a Monte Carlo generator. Signal images, in the vicinity of an isolated muon, are on the left. Background images, in the vicinity of a muon produced with an associated jet, are on the right.

Table 3.4: Comparison of images created by various generative models to the original Monte Carlo images using the Wasserstein distance (with $p = 1$) between one-dimensional distributions of physical quantities calculated from the images: P_{T} and invariant mass, also shown in figure 3.12. Smaller values indicate a closer match to the Monte Carlo images. Two SARMs are evaluated, with either discrete and continuous distributions (D+C) or a mixture of discrete distributions (D+D).

	P_{T}		Mass	
Model	Signal	Background	Signal	Background
PixelCNN++	1.75	2.92	0.58	0.82
SARM-2 (D+C)	0.79	0.97	0.25	**0.21**
SARM-2 (D+D)	**0.56**	**0.93**	**0.17**	0.31

Monte Carlo distributions quite well for P_{T} and mass, but the distributions of the Pixel CNN++ dataset have a small shift towards higher values, for both the signal and the background. In addition, for the background they are more concentrated around the mean. This is potentially due to the fact that Pixel CNN++ fails to model the right tail of the pixel distribution, where the pixels have higher values but appear much less frequently in the data (figure B.8 in Appendix). The SARM-2 (D+D) has the best overall performance, with improvements of 68.08% for P_{T} and 66.44% for mass, averaged over the signal and background datasets.

Classification of Generated Images

The fidelity of the images can be evaluated in the context of the data analysis task for which they were created, training a network to distinguish between signal (calorimeter images near isolated muons) and background (calorimeter images near non-isolated muons).

A convolutional neural network classifier was trained using images generated exclusively by each of the models (SARM-2 (D+C), SARM-2 (D+D), or Pixel CNN++); one additional network was trained using images from the Monte Carlo generator. The quality of the images is measured by comparing the classification performance of these networks on images from the Monte Carlo generator, see figure 3.13. The classifiers trained on each SARM dataset have higher AUC score than the classifier trained on the Pixel CNN++ dataset, providing additional evidence that the SARM datasets are more similar to the Monte Carlo images and thus better suited for downstream tasks such as data augmentation.

Generation Order

In this section, we discuss the impact of the pixel order for SARMs associated with the signal dataset of the muon isolation study. Similarly to section 3.6.1, we conducted 10 repeated experiments for each of the orders and summarized the results in Table 3.5.

In contrast to the jet substructure study, the muon isolation data is not rotated and the pixel value distribution is quite uniform. Therefore we see that different generation orders have similar performance in terms of mass and P_{T} distances. In addition, all the models trained using systematic orders that have some continuity in the sequence of pixels slightly outperform the models trained using random orders. In combination, these results confirm the validity of the heuristic strategy outlined at the end of Section IV, providing general guidelines for SARM design and pixel generation when applying these models to other datasets.

Computational Costs

Calorimeter image generation speeds in the context of the muon isolation study are shown in table 3.6 for the SARM models, Pixel CNN++ and the Monte Carlo generator. The SARM models are one to two orders of magnitude faster than Pixel CNN++, similar to the observation of the jet substructure study. The generation speed of each generative model is measured with the same hardware as described in Section 3.6.1. The speed for the Monte Carlo generator is measured on an Intel(R) Xeon(R) E5-2680 at 2.70GHz CPU.

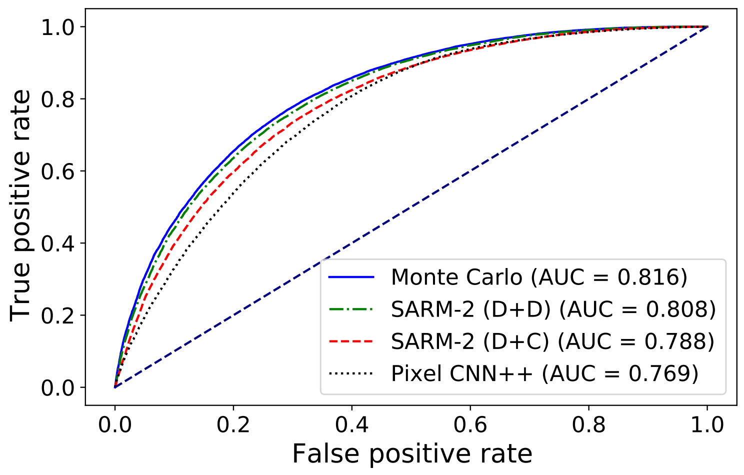


Figure 3.13: Evaluation of the fidelity of images generated by several models in the context of a classification task, distinguishing muons produced in isolation from those produced in association with a jet. Images generated by the model are used to train a network to discriminate between signal and background, but performance is measured using the original Monte Carlo images.

Table 3.5: Quality of images generated by SARM-1 models with various pixel-generation orderings for the muon isolation signal dataset. The quality is measured by the Wasserstein distance for the physical observables (P_{T} and mass) between the generated images and the original Monte Carlo images.

	P_{T} (std)	Mass (std)
Spiral-out CCW	0.99 (0.37)	0.27 (0.10)
Spiral-out CW	0.92 (0.33)	0.26 (0.09)
Spiral-in CCW	0.81 (0.23)	0.20 (0.05)
Spiral-in CW	0.95 (0.24)	0.24 (0.07)
Row-wise	0.99 (0.28)	0.20 (0.05)
Column-wise	0.90 (0.26)	0.22 (0.05)
Random I	1.17 (0.30)	0.32 (0.08)
Random II	1.34 (0.41)	0.37 (0.11)

Table 3.6: Comparison of image generation speed between the Monte Carlo approach and various generative models. The SARM-2 models are considerably faster than Pixel CNN++ and the Monte Carlo generator.

Model	Speed (images/sec)
Monte Carlo	5
Pixel CNN++	10
SARM-2 (D+D)	625
SARM-2 (D+C)	1136

3.7 Discussion

Sparse images, prevalent in particle physics datasets, present unique challenges for generative models. We have developed and applied a new class of models, deep sparse autoregressive generative models (SARMs), specifically designed to handle extreme sparseness. These compositional models are also able to take advantage of the structure present in particle physics images by using a multi-stage generation approach. Using several different metrics, we compared SARMs to other generative models, in particular to Pixel CNN++, a popular autoregressive model not adapted for sparsity, and to LAGAN, a state-of-the-art GAN for sparse images. The comparisons were carried using two benchmark data sets.

In the first case study on jet substructure, the adaptation to sparseness enables SARMs to produce qualitatively and quantitatively higher quality images than Pixel CNN++ and LAGAN. SARM are also orders of magnitude faster than traditional Monte Carlo methods and Pixel CNN++, but slower than the non-autoregressive model LAGAN, showing a trade-off between speed and quality. The second case study features extremely sparse images corresponding to calorimeter images in the vicinity of muons. While competing models produce artifacts or suffer from mode collapse, SARMs are able to handle and model extreme degrees of sparseness.

In sum, given the prevalence of sparse images in particle physics and beyond, SARMs can be expected to provide an important option for rapid, high-quality, image generation from training data. Because of their quality, the generated images in turn will be able to benefit a variety of downstream data analyses.

Chapter 4

Detection of JAVA Runtime Exceptions with Sparse Location Information [1]

4.1 Background

Exception handling is a crucial part of developing robust and reliable software systems. Exceptions can be defined as events (or errors) that happen during the runtime of a software system and result in the deviation of the execution of the program from its normal and expected behavior [25, 92], which subsequently can result in serious issues such as crashing the system [94]. In order to control the execution of the program in the event of exceptions, developers develop code specific to the occurrence of each exception, also referred to as *exception handling*. For instance, in Java, exception handling can be done by implementing *try-catch* blocks.

Most languages that support exceptions, treat all exceptions in the same way. Java, however,

[1]This Chapter is a joint work with Farima Farmahinifarahani,Vaibhav Saini, Pierre Baldi, and Cristina Lopes

has two categories of exceptions: check and unchecked. The conceptual difference between the two is subtle: checked exceptions are meant to capture errors that can be recovered from, while unchecked exceptions are meant to capture unrecoverable errors. Examples of the former include problems with the file system and network, such as file not found; examples of the latter include problems arising from fatally wrong inputs, such as division by a zero-valued variable and the dereferencing of null objects [123]. Operationally, Java treats checked and unchecked exceptions differently: the declaration and handling of checked exceptions is enforced by the Java compiler, while unchecked exceptions have no compile-time presence, and are left for the runtime to handle. All languages that support exceptions, and don't have the distinction between checked and unchecked exceptions, treat them all as unchecked runtime exceptions.

By their very nature, the consequence of unchecked exceptions is that developers can ignore them. As such, they can go undetected for a long time, until something in the runtime goes wrong, resulting in breaking the execution of the program while in use. However, predicting the types of exceptions that will happen during the runtime of a program can be difficult, and developers may not always come up with an inclusive set. Hence, they may ignore handling such exceptions, or simply resort to handling the most generic type of exception. Studies of the Java ecosystem have shown that the Exception class, which is at the top of the exception hierarchy, has a high frequency of appearance in try-catch blocks [91]. Handling the Exception class is considered a bad practice, and should be avoided, since it hides the real causes of problems [18, 1, 2]. Previous studies reveal that bad exception handling practices are prevalent [66], and that unchecked exceptions have a heavy toll with respect to bugs [66, 94]. Therefore, handling unchecked exceptions, or avoiding them in some way, is crucial.

In order to prevent such exceptions from happening, developers typically investigate their code, along with the documentation of the APIs it uses. However, rather than solely relying on the developers' knowledge of which exceptions to handle, it will be useful to develop techniques that make recommendations based on the concrete code at hand. A state-of-the-art tool in this context is FuzzyCatch [92, 93], developed for the Android ecosystem. In the absence of the runtime information before executing programs, FuzzyCatch makes predictions based on the co-occurrences of API method calls and runtime exceptions in the Android apps. However, an approach that only considers pairs of method calls and exception types, fails to account for the complex interplay between multiple code elements (such as method calls, numerical operations, or structural code constructs) and the exception types, which is important in predicting exceptions, because often there are multiple method calls and the use of certain operations and classes in a certain order that result in a runtime exception. Moreover, by only relying on API method calls, methods lacking API method calls (that are possibly exception prone) are totally missed.

In order to be able to estimate the probability of various exception types based on the interplay of different code elements, we need an approach that can infer these relationships from the code. A simple statistical model or human developed heuristics may not work well here since the existence of various code elements can lead to various scenarios of exceptions occurrence, making it infeasible to derive an inclusive set of rules. To address this challenge, we present D-REX (Deep Runtime Exception detector), a deep learning based approach that captures the correlations between certain code elements and a set of Java runtime exception types to predict the possible runtime exceptions. D-REX is also able to make predictions for *the exception prone code elements*. 'Code elements' here refers to various code constructs, such as method calls, type casting, or numerical operations, which are captured by D-REX in the form of special tokens (described in Section 4.4.1). By being informed about the part of the code that is causing an exception, developers can make more informed decisions on how to handle the exception.

The workflow of D-REX is divided into two stages: 1) building a special token sequence for each method which we call *Action-Context Token Sequence* (ACTS), containing tokens representing different code elements, and 2) joint training of a feature extractor model which produces semantically meaningful representation of the inputted ACTS, and two neural network classifiers which take the learned representation as input in order to predict the relevant exception types, as well as the exception prone ACTS tokens. In its second stage, we propose a deep neural network model, *Location Aware BERT* (LA-BERT), which adapts BERT (Bidirectional Encoder Representations from Transformers), a state-of-the-art language model, for predicting the possible exception types and exception prone tokens. LA-BERT is location aware in the sense that given a method's ACTS tokens, it leverages the information of which tokens are located inside the try block in its training phase. This extra piece of information helps it in making more accurate exception type predictions as well as predicting the exception prone ACTS tokens. While during training, we utilize existing information about the tokens inside the try block, during inference, there is no need to provide such information and the model predicts the tokens which it finds to be exception prone.

We chose BERT as the feature extractor model of D-REX since its self attention mechanism allows every element of the input sequence to directly communicate with all other elements in each layer in constant time and space complexity [133], alleviating the issue of forgetting in modeling longer sequence data compared to the recurrent neural networks [61, 117]. We found this particularly useful in learning meaningful representations from code as the lengths of methods can very both within and across the projects. We demonstrate its effectiveness in our experiments by showing that D-REX outperforms the Bidirectional LSTM [117] in exception type prediction by a considerable margin.

Overall, our contributions can be summarized as follows.

1. We present D-REX, a novel approach for the recommendation of unchecked (runtime) exceptions in Java, which outperforms multiple baselines.
2. We present an application of BERT for predicting runtime exceptions. To leverage the

sparse information of the location of try blocks during training, our main contribution, in this respect, is Location Aware BERT that produces more accurate results and is capable of identifying exception prone tokens.

3. We curate a benchmark dataset of 200K Java projects with more than ≈442K try-catch blocks handling runtime exceptions. This dataset will be made publicly available.

The remainder of this paper is structured as follows. Section 4.2 builds the motivation for our task by providing concrete examples and explains our ultimate goal in detail. In Section 4.3, we describe the dataset that we curated for training and evaluation of D-REX, and in Section 4.4, we discuss our proposed approach by explaining the architecture of D-REX, capturing ACTS and the details of LA-BERT. Section 4.5 presents the results of the experiments conducted to evaluate D-REX's effectiveness in predicting exception types and exception prone tokens. Finally, we present the related work in Section 4.6 followed by a discussion on threats to validity and future work in Section 4.7.

4.2 Motivation and Goal

Runtime exceptions frequently occur in Java programs [66]; a study [94] on 246 Android exception related bugs found the majority of them to be runtime ones. Another study on 656 Java libraries found handling of API runtime exceptions to be more frequent than the checked ones [118]. Runtime exceptions are prevalent and can cause serious issues at the program runtime (e.g., complete crash of program) if not properly tackled or handled. For example, *NullPointerException* which frequently happens in java programs, serves as a major cause of crashes in software systems [24] and is typically not caught during testing; hence, making it likely to propagate to the program runtime [44]. These findings demonstrate the importance of addressing runtime exceptions before they cause major issues.

In its simplest form, the problem of *runtime* or *unchecked* exception handling can be illustrated by the *howManyTimes()* method starting from line 2 and ending at line 8 of Listing 4.1. In this method, if the second parameter has a value of zero, a division by zero happens which leads to an *ArithmeticException.* This may not catch the developer's attention and they may miss to handle this exception. A possible fix to this issue is shown in the method starting from line 10 and ending at line 16 of Listing 4.1. In this fix, the developer is aware of the possibility of *ArithmeticException* and deals with it by implementing a pre-condition on the argument in their if statement.

Another example of a scenario susceptible to runtime exceptions is depicted in Listing 4.2 from line 2 to 10. At runtime, this method will throw a *NumberFormatException* on parsing the string values to integer values at lines 4 and 5 if a value in the input array does not represent an integer. Even if all the values in the input array are valid representations of integers, and the parsing is safely done, this method is yet susceptible to *ArrayIndexOutOfBoundsException.* The *for* loop traverses the array to its last element while inside the for loop, accessing $i + 1_{th}$ index of the array can result in *ArrayIndexOutOfBoundsException* if $i == args.length$. The method starting from line 13 of Listing 4.2 tries to tackle both problems. *NumberFormatException* is being handled by a try/catch block to alert the user about the proper input, and *ArrayIndexOutOfBoundsException* has been controlled by changing the upper bound of the loop counter from $args.length$ to $args.length - 1$.

As the two examples show, knowledge about the relevance of runtime exceptions can help the developer take the necessary actions. These actions need not necessarily be implementing an exception handler; valid actions include the addition of pre-conditions, changing the parameters types, and even rearchitecting the code. A trivial way of reminding developers about possible runtime problems would be to always remind them of all runtime exception types. However, that would have a very high rate of false positives, which would make developers ignore the reminders.

```java
//Before fix
private int howManyTimes(int a , int b){
    int result=0;
    if (a > b) {
        result = a / b;
    }
    return result;
}
//After fix
private int howManyTimes(int a , int b){
    int result=0;
    if (b!=0 && a > b) {
        result = a / b;
    }
    return result;
}
```

Listing 4.1: Example 1: Method prone to RuntimeExeption

```java
//Before fix
public static String [] findMax(String[] params){
    for (int i = 0; i < params.length; i++) {
        if (Integer.parseInt(params[i]) >
                Integer.parseInt(params[i + 1])) {
            params[i + 1] = params[i];
        }
    }
    return params;
}
//After fix
public static String [] findMax(String[] params){
    try {
        for (int i = 0; i < params.length-1; i++) {
            if (Integer.parseInt(params[i]) >
                    Integer.parseInt(params[i + 1])) {
                params[i + 1] = params[i];
            }
        }
    }
    catch (NumberFormatException e){
        System.out.println("Input array should only contain integer values.");
    }
    return params;
}
```

Listing 4.2: Example 2: Method prone to RuntimeExeption

Our goal is to design and implement an approach that can predict *relevant* runtime exception types and exception prone code elements for each method. Unlike previous approaches [92, 93, 25, 106, 76], our work does not focus on recommending exception handling code. Rather, we aim to inform the developer about the possible incidence of such exceptions and possible causes, but delegate the needed action to the developer, as wrapping runtime exceptions in a try/catch block is not always the best solution; sometimes fixing the code to prevent the exception from happening in the first place can be a better approach (as observed in the above examples).

In the absence of runtime information and user input prior to executing the program, we make use of the static signals from code to find clues pertaining to the relevance of certain runtime exceptions. For example, in Listing 4.1, using an int value as the denominator of a divide operator serves as the clues about the *ArithmeticException.* This is an example where a runtime exception is possible to happen while there is no API method call present in the code related to it. As a result, by solely relying on API method calls, we may miss recommending the relevant exception for this example. In Listing 4.2, the invocation of *parseInt()* on the method parameter value gives a clue about *NumberFormatException* and accessing the array indexes by doing an arithmetic operation $(i + 1)$ inside a for loop gives clues about *ArrayIndexOutOfBoundsException.* This demonstrates how the complex interplay of various code elements can contribute to a runtime exception. Inferring the relationships between the code elements and predicting relevant exception types accordingly using heuristics or statistical methods may not work well here, and a machine learning based method such as D-REX can help us by automatically learning the relevant patterns based on a large dataset of examples.

In addition to recommending relevant exception types, by identifying exception-prone elements of code, D-REX can help the developer find the issues in their program or implement the suitable try/catch block, as most of the time the developer is not aware of which part of their code is possibly going to throw an exception. For example, the Top 1 exception prediction of D-REX for the example in Listing 4.1 is *ArithmeticException* and the *divide* operator on line 5 is predicted as the most possible cause for it. For the example of Listing 4.2, D-REXpredicts both *NumberFormatException* and *ArrayIndexOutOfBoundsException* as its top 2 predictions, and the invocations of *parseInt()* and accessing the array inside the loop as the top exception prone parts of code. An interesting observation here is that D-REX predicted *NullPointerException* as its third exception type for this method; a closer look shows that this exception is possible to be thrown if the input array (*params*) has a Null value.

D-REX operates on Java methods and its core is a deep learning model. This model predicts the relevant exception types and exception-prone elements of code by learning from a dataset of over 350k developer-implemented try blocks containing handlers for runtime exceptions. For training, we use the information from the try/catch blocks; for evaluation, the try/catch blocks are removed from the methods to provide the model with the raw method source code that it receives from a developer.

4.3 Dataset

We used the Java dataset used in the GitHub study by Lopes et al. [77] to prepare our dataset. The benefit of this dataset is that it includes the Java projects present in GitHub until the time of paper's publication with forked projects excluded and a mapping of cloning among the projects provided. This is particularly helpful in a machine learning task like ours as we could use it to remove the repetitions among projects to reduce the duplications in training and testing datasets and train a generalized model. The whole Java dataset used in their study contains 1,481,468 projects and 72,880,615 files. From these projects, we only considered those that have more than 10 files (to narrow our focus to bigger and meaningful projects), leaving us with 670,429 projects. We then filtered out projects that are 50% or more clones of other projects; giving us 544,513 projects. We then drew a 200,000 random sample of these projects, which we name *200k* dataset. Using the file level mapping of clones in these projects [77], we also filtered out the files that are total duplicates of each other; i.e., the Type-1 clones [111].

Prior to processing the projects, we collected the list of Java SDK runtime exceptions from its documentation[2]. We collected all subclasses of the *RuntimeException* class, a set of 169 exceptions. In order to curate a labeled dataset with methods and the runtime exceptions that may arise in them, we looked for methods that have a *try-catch* block implemented in them where one of the 169 collected runtime exceptions is caught in their catch statement.

We modified Java Parser [3] to parse the projects and fetch methods and the try-catch blocks within them and the tokens needed for the Action-Context Token Sequence. After parsing the dataset, we found 577,067 examples of try-catch blocks catching a Java runtime exception. From these examples, we removed the ones that were catching the *RuntimeException* itself since it is the superclass of all of the target exceptions and thus, including it as a label can confuse the model. This left us with 470,376 examples. By analyzing the exceptions frequencies, we observed that some exceptions appear rarely in the dataset; for example, 35 exception types appear less than 10 times. We decided to exclude the examples with very rare exceptions (the ones appearing less than 100 times) to limit our attention to the mostly frequent exception types. We also removed methods with inner classes in their bodies since their structure is different from conventional methods and can introduce noise to the dataset. After these two steps, we were left with 442,446 samples and 52 exception types. Figure 4.1 shows the frequencies for the top 10 most frequent runtime exceptions in the 200k dataset. As the figure shows, the three most frequent exception types appearing more than 40,000 times are the *NumberFormatException* (142,800 times), the *IllegalArgumentException* (89,193

[2]`https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html`

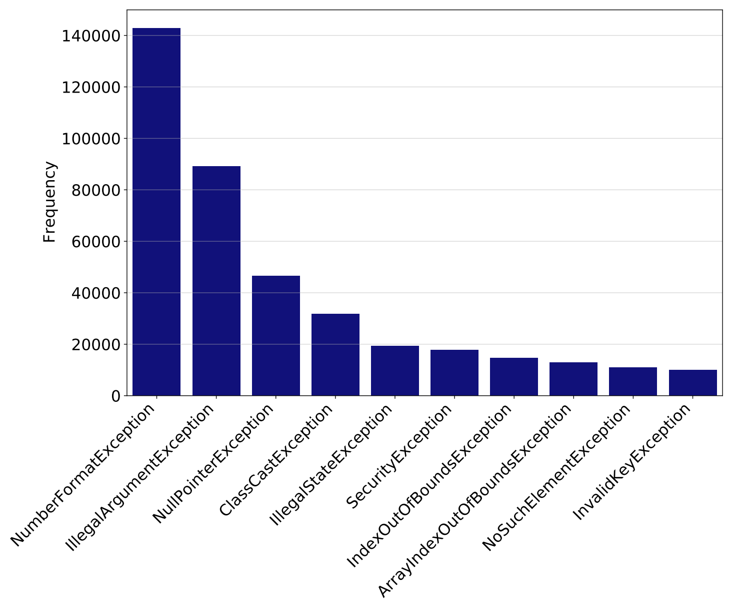


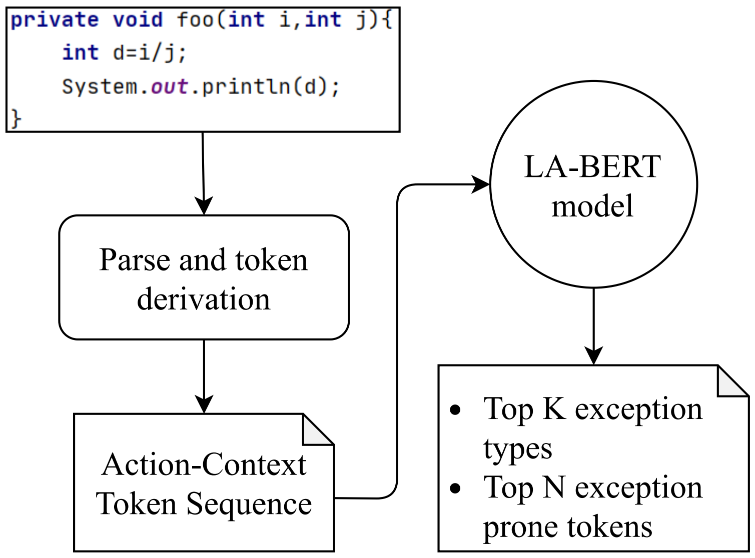
Figure 4.1: Distribution of the 10 Most Frequent Exceptions

times) and the *NullPointerException* (46,575 times). For the rest of the exception types not shown here, 12 of them appeared between 10,000 and 1,000 times, and the remaining appeared less than 1,000 times.

We split the dataset into train, validation and test sets by dedicating 80% of samples to the train set, 10% to the validation set and 10% to the test set. As a result, the train set has 353,956 samples and test and validation each have 44,245 samples. We perform the training on the train set, determine all the hyperparameters and early stopping steps using the validation set, and benchmark the accuracy of D-REX and other baselines using the hold out test set.

4.4 Proposed Approach

An overview of the D-REX prediction pipeline is shown in Figure 4.2. The prediction starts by receiving a Java method as input and parsing it to retrieve a token sequence which we


Parse and token derivation
Action-Context Token Sequence
LA-BERT model
Top K exception types
Top N exception prone tokens

```
private void foo(int i,int j){
    int d=i/j;
    System.out.println(d);
}
```

Figure 4.2: Overview of the D-REX Prediction Pipeline

call *Action-Context Token Sequence* (ACTS). The goal here is to capture the key elements of code that represent the main actions carried out in the method while decreasing the language vocabulary. This token sequence is then fed to a deep learning model, called LA-BERT, that applies transformations on its input to build a high level semantic representation, and predicts two outputs: (i) top K runtime exception types possible to happen in the method ranked by their occurrence probabilities, and (ii) top N exception prone tokens, which are tokens from ACTS that are most likely responsible for the predicted exceptions. The ACTS tokens are translated back to the code elements they are representing for better interpretation by developer. The details of the two main components of this pipeline, *Action-Context Token Sequence* and *LA-BERT*, are explained in this section.

4.4.1 Action-Context Token Sequence

In order to provide the input method to the prediction pipeline, we derive a token sequence from it that can provide the key information helpful in predicting the runtime exceptions. Previous research have explored the correlations of API method calls with the occurrence of runtime exceptions [93, 92], however, method calls alone may not be sufficient to estimate the probability of runtime exceptions. For instance, in the *Before fix* situation of the example depicted in Listing 4.2, the occurrence of *ArrayIndexOutOfBoundsException* is not recognizable if we only look at the method calls; it is accessing the indexes of the input array that causes this exception. Moreover, accessing an array for once may not introduce a high probability of exception occurrence compared to when it is accessed inside a loop. Therefore, one may find it useful to feed all method tokens to the deep learning model and let the model perform its inference and make the predictions. However, source code vocabulary has been shown to be unlimited as new identifier names get introduced constantly [8, 57]; such increase in vocabulary can make it difficult for the deep learning model to infer meaningful representations from the input and hampers the model convergence. Therefore, we need to derive a set of tokens from the methods that can decrease the vocabulary by removing unnecessary elements while capturing the key elements of code that can contribute to the occurrence of runtime exceptions. This will also help the model in learning more meaningful representations related to exception types. As the examples in Section 4.2 showed, it is often an interplay of different code elements that ultimately result in a specific runtime exception; in other words, it is a specific action (such as a method call or a mathematical operation) in a specific context (for example a for loop or a special condition in a nested if statement) that causes a specific runtime exception. The names of the variables, for example, may not introduce relevant and useful information here.

To address these needs, we build a special token sequence for each method, which we call *Action-Context Token Sequence* (ACTS), capturing two important aspects from each method: *actions* carried out in it and the *context* in which these actions are performed. Hence, there are two categories of tokens: tokens capturing actions and tokens capturing contextual information. These tokens may have identical correspondence with the tokens in code (e.g., method calls), or may be derived from code and presented by a literal value (e.g., ‘Cast’ to show type casting). Details of ACTS are as follows:

Tokens capturing method's actions

One set of captured tokens are the method calls invoked inside the method body. As the example in Listing 4.2 showed, method calls (e.g., *parseInt()*) can serve as causes for throwing a runtime exception (*NumberFormatException*). FuzzyCatch [93, 92] heavily relies on API method calls and Barbosa et. al [25] use method calls to recommend exception handling code. Saini et al. [115] refer to method calls as *'Action Tokens'*. They also introduce two other action tokens: *ArrayAccess* for accessing the index of an array and *ArrayAccessBinary* for accessing the index using an arithmetic operation (such as $i + 1$). We consider these two tokens in the ACTS as well, since as we saw in Listing 4.2, such tokens can signal for runtime exceptions.

Another set of tokens that convey actions are the Binary and Unary operations. A simple example is Listing 4.1 where *DIVIDE* operator signals for an *ArithmeticException*. Finally, we capture a set of special tokens to cover the actions that were not covered by the tokens explained so far. These tokens capture variable declarations, type-casts, object instantiations, the usages of *Null* literal, accessing objects' fields and the method's return statement.

Tokens capturing contextual information

One aspect of the contextual information are the code blocks that define the sequence of actions and hence, can contribute to the occurrence of exceptions; for example, consider invoking the *subString()* method on a string value or accessing an array indexes; both of these actions may be more exception prone if done inside a loop. To capture these blocks in a high level manner, we define a set of special tokens. For example, the beginning of a loop is captured with *BeginLoop* and its ending is captured by *EndLoop*. Similarly, the beginning and ending of if-else blocks, try-catch-finally blocks (for checked exceptions) and switch-case blocks are captured. The other set of relevant tokens are the data types. Barbosa et al. [25]

Table 4.1: Tokens in Action-Context Token Sequence

Tokens Capturing Actions	Tokens Capturing Context
1. *Method calls*	10. BeginLoop, EndLoop (any loop block)
2. ArrayAccess, ArrayAccessBinary	11. BeginIf, Else, EndIf (if block)
3. *Binary* and *unary operations*	12. BeginSwitch, EndSwitch (switch block)
4. VarDec (variable declaration)	13. BeginTry, EndTry (try block)
5. Cast (act of type casting)	14. BeginCatch (catch in a try block)
6. New (object instantiation)	15. BeginFinally (finally in a try block)
7. Null (referencing "Null" literal)	16. Array
8. FieldAccess (access an object field)	17. *Primitive data types*
9. Return (method's return)	18. *Class names*

use variable types for recommending exception handling code and discuss that methods using variables of the same types are likely to implement more similar tasks. Other than this, certain data types can correlate with certain runtime exceptions occurrence (such as the use of numerical data types correlating with mathematical runtime exceptions). Class types also give context to actions and can correlate with exceptions relevance; for example, class *Integer* can correlate with the occurrence of *NullPointerException* while the primitive type *int* will not. Hence, We capture all primitive types and class names as tokens. The *Array* token is used to capture array types.

Table 4.1 provides a summary of tokens captured in ACTS. In this table, literal tokens (e.g. Null) are expressed in regular font, while tokens that take values from the source code are expressed in *italic*.

4.4.2 LA-BERT Model

Figure 4.3 shows the high level architecture of LA-BERT, the deep learning model of D-REX. The model receives an input token sequence (tokens in ACTS) and the trained BERT layers extract a high level representation from it. This representation is then fed into two networks at the same time: a location prediction network (to predict the exception prone tokens) and an exception prediction network (to predict the possible exception types), both of which

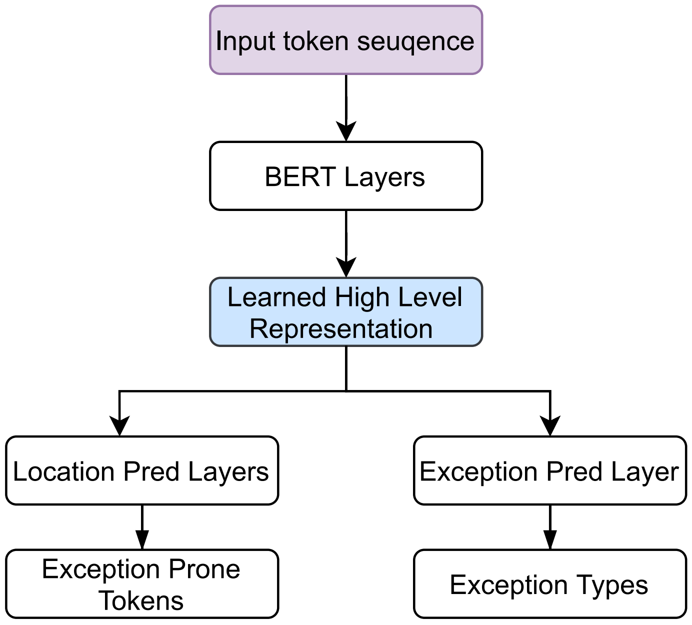


Figure 4.3: Architecture of LA-BERT Model

consist of several fully connected layers. The BERT layers and the two prediction networks are trained jointly in an end-to-end fashion.

In the rest of this section, we first give an introduction to BERT model [43], which serves as the feature extractor of D-REX and discuss its difference with other deep neural network models that can be used for this purpose. Then we explain the location awareness concept in LA-BERT and its training paradigm.

BERT vs. Recurrent Neural Network

Recent advances in natural language modeling [43, 85, 133] have seen great success in modeling sequence data. Among various deep learning based language models, BERT is the state-of-the art attention based deep neural network model, which was specially designed for extracting high level representations from sequence data. It has been proven to achieve leading performance on a variety of natural language tasks such as question answering and sentence prediction [108, 134]. Moreover, Devlin et al. [43] show that the pretrained BERT

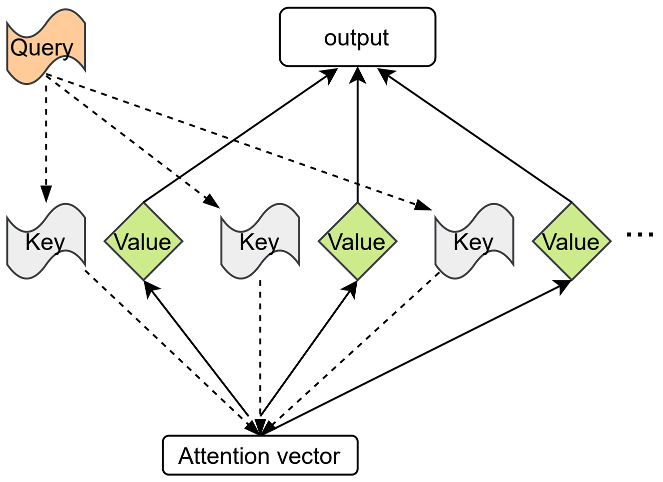


Figure 4.4: An Illustration of Self Attention Mechanism

model achieves impressive results after being fine-tuned with a few additional layers on the top. In contrast to other neural network models designed for modeling sequence data, such as recurrent neural networks (RNN) [61],

The structure of the self attention mechanism is shown in Figure 4.4. Every input element (input token in the first layer, or the output of previous layer in the other layers) possesses three different feature vectors: a key, a value and a query vector, all learned during the training. The i-th attention vector is produced by the dot product between the i-th query vector and all key vectors. This attention vector is used as a weight to get a weighted average of the value vectors of all input elements, producing the i-th output element (shown as output in the figure). Therefore the i-th output element is directly connected with every input element in the self-attention layer. However, this connection is restricted in RNN layers: the i-th output unit can only connect with the j-th input unit if $j \leq i$. Further, the connection between the i-th output unit and the j-th input unit becomes weaker and weaker as the number of units between them increases in RNN layers.

Learning Sparse Location Information

In the problem of exception type prediction, there is an extra piece of information in training data that we can exploit to provide more accurate recommendations: location information for tokens located inside the associated try block. The exception types handled in a try-catch block of a method are strongly correlated with the tokens appearing in the try block. Such information is present in our training data; hence, given a try-catch block handling a runtime exception in a method, in the training phase, we augment the input tokens sequence of the method with location indices of tokens belonging to the try block, and use it as label information for location prediction layer during joint training. Note that this information is not required as input for the location prediction layers during inference.

Denote the length of the input sequence as N, and the Bernoulli random variable for determining whether the ith token is a try block token to be z_i. Then for each input sequence, we have a vector of Bernoulli random variables

$$\boldsymbol{z} = (z_1, z_2, ..., z_N), \text{ where } z_i \sim \text{Bern}(p_i),\ p_i \in [0, 1] \tag{4.1}$$

for $i = 1, 2, ..., N$. And p_i is predicted by the model. In its essence, the model is learning the joint distribution of Bernoulli vectors $\boldsymbol{z}$, which is sparse in typical Java source code.

Since training is done with an objective function to optimize these two prediction networks as well as the BERT layers simultaneously, it forces the learned high level representation to be location aware, hence, more expressive than a model without the location prediction network, which we refer to as the *plain BERT* model. The plain BERT model can only predict exception types (and not the exception prone tokens) and we compare D-REX using LA-BERT with a version of it using plain BERT in Section 4.5.1

Training

During training, the loss function takes into account both the error from predicting the exception type and the error from predicting the tokens that belong to the try block. L, the joint objective function of exception prediction and location prediction, is formulated as follows:

$$L = L_{\text{excep}} + \lambda L_{\text{tryloc}} \tag{4.2}$$

In this equation, L_{excep} denotes the cross entropy loss resulting from the exception type prediction, and L_{tryloc} denotes the mean value of the loss of every token in the input, where each individual token uses a binary cross entropy loss [55]. λ is a tuning parameter between $[0, 1]$ balancing the magnitude of the two losses. In practice, we treated λ as a hyper parameter and conducted a grid search in a list of values: $[10^{-2}, 5 \cdot 10^{-2}, 10^{-1}, 5 \cdot 10^{-1}]$, where we found that 10^{-2} yields the best performance. Note that the weight parameters in BERT layers and in the two prediction layers are trained jointly using equation (4.2).

The number of BERT layers along with other layer specifications such as the number of units in each layer were treated as hyperparameters, which were tuned through grid search in the experiments. The best resulting architecture consists of 10 BERT layers with a hidden dimension of 512 and an embedding size of 512. Both try location prediction and exception type prediction networks are parameterized by a two-layer fully connected neural network of width 100. With the output of these two networks, equation 4.2 is used to calculate the overall loss. Adaptive moment estimation (Adam) [71] optimizer with learning rate $3 \cdot 10^{-3}$ was used to minimize equation 4.2. Both plain BERT and LA-BERT models were trained for 100 epochs and the top k accuracies converged.

4.5 Evaluation

In this section, we present the results of evaluating D-REX's exception type and exception prone token predictions. The ground truth exception types in these experiments are the runtime exceptions handled by developers in try blocks. For the exception prone token predictions, we present quantitative and qualitative analyses by looking at the top-k predictions by D-REX. In quantitative analysis, the ground truth for the exception prone tokens are the tokens inside each try block. In all experiments, for each try block of interest in each method, we removed the keyword *try* and any code provided for exception handling with the try block. This way we ensured that the input to our model does not contain any tokens related to the exception handling code.

4.5.1 Exception Type Prediction

Baselines

In order to compare D-REX with the existing methods of exception type prediction, we searched for available tools in this area. Although there have been much work in recommending exception handling strategies, the only approach we found that can predict runtime exceptions is FuzzyCatch [93, 92]. The core of this approach is a statistical model that works based on the co-occurrences of the API method calls and exception types. As mentioned by their paper, FuzzyCatch's model has been trained and tested using a set of Android projects. Hence, there are two issues that make a direct comparison between D-REX and FuzzyCatch not valid: (1) The FuzzyCatch model was trained exclusively on Android code, while the dataset used for training our D-REX model is based on randomly selected Java projects and does not focus on Android projects. This matters, because there is considerable divergence of vocabulary between Android APIs and regular Java APIs: a model trained on one vocabulary may not do well on the other. (2) FuzzyCatch is offered as IntelliJIdea and AndroidStudio plugins, and expects a human in the loop; unlike D-REX, it is not possible to run FuzzyCatch headless on a large source code dataset. Since a direct comparison with this tool is not feasible, we developed a baseline inspired by their technique that works based on the co-occurrences of all method calls and the exception types and is trained it on the 200k train set. We name this baseline *Method-Exception-Frequency (M-E-F)*. For each method, M-E-F collects all method calls in it and then queries the train dataset for the co-occurrences of each method call and all exception types. Using a fuzzy union formula (presented in [93, 92]), it then aggregates the results for all method calls and recommends the possible exceptions sorted by their predicted probability.

To compare the LA-BERT component of D-REX with other deep learning models that are potentially suitable for our purpose, we trained two other models: (1) a bidirectional LSTM model [117] (Bi-LSTM), a variant of recurrent neural network models, and (2) a plain BERT model (P-BERT) (which we described in Section 4.4.2). For each of these, LA-BERT in D-REX is replaced with the corresponding model and top k predictions are collected.

All baselines are trained on 200k train dataset and their accuracy is compared with D-REX's accuracy on 200k test dataset and a separate set of unseen java projects.

Accuracy Comparison on 200k Test Set

We first evaluate D-REX and other baselines on the hold out test portion of the 200k projects dataset presented in Section 4.3[3]. The results of these experiments are shown in Table 4.2. In this table, we present the accuracy of each approach on its Top 1, Top 2, Top 3, Top 5 and Top 10 recommendations. Top 1 accuracy, for example, presents the proportion of the Top 1 recommendations by each approach that matches the set of true runtime exceptions present

[3]Input file prepared for D-REX for this step is uploaded in supplementary materials.

in the dataset. More formally, if m is a method and E_m is the set of runtime exceptions caught by a try block in m, and $\hat{E}_{K,m}$ is the list of top K runtime exceptions predicted by an approach for m, sorted based on the descending order of the predicted scores, then $TopK$ accuracy over a dataset of N samples is defined as:

$$\frac{1}{N}\sum_{m=1}^{N} 1_{(E_m \cap \hat{E}_{K,m} \neq \emptyset)} \tag{4.3}$$

where $1_{(E_m \cap \hat{E}_{K,m} \neq \emptyset)}$ is an indicator function that equals 1 when $E_m \cap \hat{E}_{K,m} \neq \emptyset$ and 0 otherwise. This measurement is similar to the accuracy measurement used by FuzzyCatch and as they have also pointed out, we are not able to evaluate the results using Precision due to the lack of negative examples: given a piece of code, it is almost impossible to determine that a specific runtime exception will never happen for it. As for the recall number, the top k recall depends on the number of the true exceptions in a method, which varies method by method. However, if there is only one exception in the method, then our top 1 accuracy is equivalent to recall.

As the table shows, D-REX has been able to achieve the highest accuracy across all categories of Top K predictions. The difference, however, is mostly notable in the Top 1 results which is of utmost importance as the Top 1 prediction is the one that developers pay more attention to. In particular, D-REX has ≈81% accuracy in its Top 1 predictions, compared to Plain BERT which is the second best one with ≈79% Top 1 accuracy. This indicates that the extra information about the tokens inside the try block helps D-REX to achieve better results in the exception type prediction. It also implies that the exception prone token prediction loss, L_{tryloc} in equation 4.2, has a regularization effect for the LA-BERT model to alleviate the over-fitting during training, leading to an improvement in exception type prediction on the test set. M-E-F has the third rank in Top 1 accuracy with ≈12% of difference with D-REX. The Bi-LSTM model was not able to achieve comparable results which can attest

Table 4.2: Exception Prediction Accuracy on 200k Test Set

	Top 1	Top 2	Top 3	Top 5	Top 10
M-E-F	69.67%	81.37%	86.30%	92.38%	96.46%
Bi-LSTM	59.32%	71.71%	78.56%	85.76%	93.12%
Plain BERT	79.39%	87.49%	90.67%	93.74%	96.68%
D-REX	**81.09%**	**88.91%**	**91.70%**	**94.52%**	**97.08%**

the effectiveness of LA-BERT model over it.

Accuracy on Unseen Projects

In order to understand the effectiveness of D-REX on a variety of projects and those that were not included in the 200k dataset, we downloaded a separate set of open-source Java projects from GitHub, shown in Table 4.3[4]. For all these projects, we downloaded their default branch's version. In the table, the column"URL" shows the GitHub URL, "Last commit date" shows the date of the last commit on the repository when we downloaded it, "LOC" shows the number of source code lines calculated with IntellijIdea's Statistic plugin [5] and "#Samp" shows the number of samples (try-catch blocks with runtime exceptions) fetched from the project. To select these projects, we first downloaded and processed the set of Java projects used by Hindle et al. [60]. After parsing, we included the ones with 100 or more samples to make sure that the evaluation is done on large enough datasets. The first 4 rows of Table 4.3 show the projects selected in this step. Then, to expand the comparison to more projects, we queried the database provided by Lopes et al. [77] for projects that are not included in 200k dataset and have more than 5,000 files, and selected 8 more projects from this set. The rest of the projects listed in Table 4.3 were selected in this step. All of these projects are large projects with various developers contributing to them, hence, we expect them to have sufficiently high quality exception handling code, and therefore, evaluating our approach on them can give us more confidence about the performance of our approach.

[4]D-REX's input files for these project are uploaded in supplementary materials.
[5]https://plugins.jetbrains.com/plugin/4509-statistic

Table 4.3: Downloaded Java Projects Summary

Project	URL	Last commit	LOC	#Samp.
Batik	apache/xmlgraphics-batik	Jan 12, 2021	190,597	172
Lucene	apache/lucene-solr	Jan 29, 2021	1,350,727	411
Xalan	apache/xalan-j	May 23, 2019	170,574	218
Cassandra	apache/cassandra	Jan 29, 2021	464,594	211
SonarQube	SonarSource/sonarqube	Feb 17, 2021	516,101	203
Camel	apache/camel	Feb 17, 2021	1,539,888	213
IntelliJ IDEA CE	JetBrains/intellij-community	Feb 17, 2021	4,095,728	789
Hibernate ORM	hibernate/hibernate-orm	Feb 17, 2021	779,538	333
Object Teams	eclipse/objectteams	Jan 14, 2021	1,868,449	371
MPS	JetBrains/MPS	Feb 17, 2021	1,872,299	162
Ignite	apache/ignite	Feb 16, 2021	1,170,332	333
CloudStack	apache/cloudstack	Feb 12, 2021	661,128	283

Table 4.4 shows the the exception prediction accuracy of D-REX and other baselines on each of these projects, and overall on all of them, for the Top 1 to 3 predictions. In general, we see that D-REX is achieving the highest accuracy compared to all other baselines in Top 1 predictions, except for two of them: Hibernate ORM and MPS. For MPS, M-E-F achieves a better accuracy, with a difference of ≈3%. And for Hibernate ORM, M-E-F performs better by a margin of ≈6%. However, all models are performing bad on this project, so we looked at it more closely and found out the the exception type distribution of this project is very different from the training set; for example, *NumberFormatException*, the most frequent exception in training data, has only 33 occurrences in this project, which can explain the lower accuracy of D-REX. On other projects, D-REX has an accuracy improvement over P-BERT in a range of ≈0.2% to ≈8%, and an improvement over M-E-F in a range of ≈3% to ≈30%. We see a similar trend of D-REX performing better than other baselines in Top-2 and Top-3 predictions for the majority of projects. In the overall results aggregated over 12 projects, D-REX is performing the best in Top 1 and Top 2 predictions, and in Top 3 predictions, P-BERT has slightly better accuracy by a margin of less than 0.5%.

Table 4.4: Exception Prediction Accuracy Comparison on Unseen Java Projects

Project	Top 1				Top 2				Top 3			
	D-REX	P-BERT	Bi-LSTM	M-E-F	D-REX	P-BERT	Bi-LSTM	M-E-F	D-REX	P-BERT	Bi-LSTM	M-E-F
Batik	**81.40%**	73.26%	51.74%	51.79%	**84.88%**	82.56%	58.72%	54.17%	**87.21%**	84.88%	65.70%	57.14%
Lucene	**64.08%**	60.44%	45.50%	52.27%	**73.06%**	73.03%	59.12%	70.20%	76.70%	**77.43%**	69.83%	75.00%
Xalan	**88.53%**	88.07%	28.90%	76.70%	**92.20%**	91.74%	45.87%	80.10%	**94.04%**	93.58%	87.16%	81.55%
Cassandra	**60.38%**	54.72%	54.03%	48.82%	**71.70%**	71.23%	66.35%	59.24%	**77.36%**	75.00%	74.41%	65.88%
SonarQube	**69.46**%	62.56%	45.81%	53.20%	**76.85**%	71.43%	70.44%	70.44%	**83.74**%	79.31%	82.27%	82.76%
Camel	**61.03**%	53.05%	47.89%	49.29%	**74.18**%	73.71%	68.54%	72.99%	**84.04**%	78.87%	75.12%	77.73%
IntelliJ IDEA CE	**60.71**%	60.58%	49.68%	51.39%	71.23%	**71.48**%	60.58%	69.25%	75.41%	**76.30**%	69.33%	73.47%
Hibernate ORM	36.94%	39.34%	38.44%	**43.37**%	**68.47**%	61.86%	63.66%	59.04%	74.77%	78.38%	**81.68**%	71.69%
Object Teams	**59.30**%	58.76%	41.51%	56.76%	74.12%	**77.63**%	49.87%	72.37%	79.25%	**82.21**%	56.87%	78.08%
MPS	46.91%	43.83%	45.06%	**49.04**%	63.58%	51.85%	62.35%	**68.15**%	66.05%	70.37%	**72.22**%	71.34%
Ignite	**63.06**%	61.56%	37.54%	52.58%	**71.47**%	68.17%	52.85%	65.96%	**75.68**%	72.37%	68.47%	**75.68**%
CloudStack	**87.63**%	86.57%	65.37%	71.53%	93.29%	**94.35**%	76.68%	91.24%	95.05%	**96.47**%	81.98%	95.62%
Overall	**62.75%**	61.34%	46.09%	55.33%	**74.78%**	73.91%	60.61%	65.96%	79.29%	**79.78**%	72.48%	75.68%

4.5.2 Exception Prone Tokens Prediction

The other output of D-REX is the set of tokens that can be responsible for the predicted exceptions, in the order of their predicted probability. We evaluate the correctness of these predictions in two ways: a quantitative analysis to measure the precision of these predictions with respect to the tokens in the ground truth try blocks, and a qualitative analysis aimed at investigating the relevance of predictions with the possible exceptions through a manual analysis.

Quantitative Analysis

We define our ground truth for evaluating the correctness of the exception prone tokens predictions with this procedure: for each try-catch block present in the dataset that handles a runtime exception, we consider the method's ACTS tokens that are inside the try block as the true tokens responsible for the exception. The reason is that when developers implement try-catch blocks, they form the try block around the portion of code that they believe can cause the exception being handled. We evaluate the correctness of theses predictions with precision: for each sample, we measures the percentage of the top k predictions that are included in the ground truth try block. We do this evaluation for $k = 1, 2, 3$ and report the mean value precision ($Mean_{Precision}$) across all of the samples in the 200k test set and the

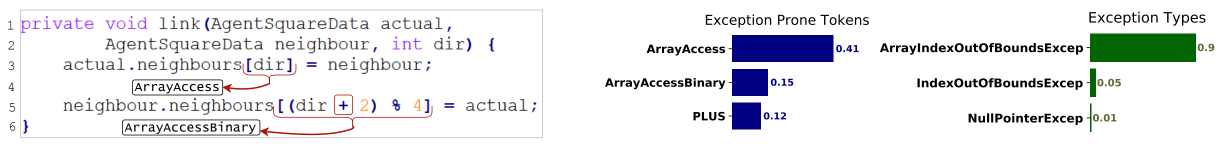
Table 4.5: Exception Prone Tokens Prediction: $Mean_{Precision}$

Dataset	$k = 1$	$k = 2$	$k = 3$
200k test set	75%	71%	69%
Unseen projects	57%	56%	56%

also, on the all samples of the unseen projects dataset explained in Section 4.5.1. For each value of k, we evaluate D-REX on the samples that have a try block consisting of more than k tokens and a method body length of more than than $2 * k$ tokens. The former is done since it is not meaningful to look at top k predictions if the true try block has less than k tokens. The latter is done to eliminate the cases where the number of tokens in method is close to k and hence, D-REX has a high a chance of having its top k predictions to be true predictions. Hence, it will help us look at more meaningful results.

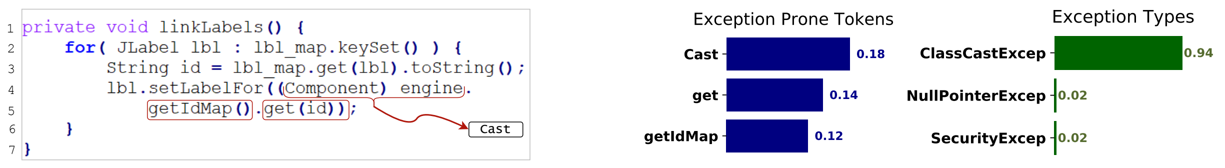
Table 4.5 shows the results of evaluating the top k predictions (where $k = 1, 2, 3$) on the two datasets. It can be seen from the table that for 200k test set, $Mean_{Precision}$ of top 1 predictions ($k = 1$) is 75%, suggesting that 75% of the times, the top 1 predicted token has actually been located in the try block. This value is 57% for the unseen projects dataset. The $Mean_{Precision}$ values for $k = 2$ and $k = 3$ are also very close to the values reported for the $Mean_{Precision}$ when $k = 1$. This is promising results as it can help developers to identify the source of exceptions by looking at the top 3 predictions.

It is worth mentioning that we did not measure recall here since while the length of try blocks can vary, D-REX always produces top k predictions. Therefore, when a method has a long try block with many tokens, recall will be dominated by the length of try block no matter how accurate the prediction is. Also, in predicting the exception prone tokens, it is the precision of top K predictions that matters most to the developer, not the fraction of all tokens in the true try block that are captured. There can be tokens in the true try block that are not directly related to the exception but has been placed in the try block because they form a code construct.



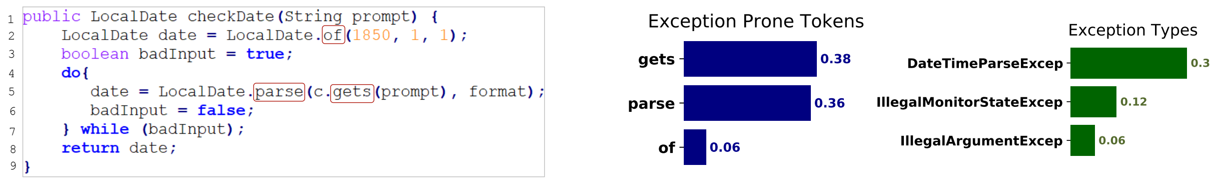
```java
1 private void link(AgentSquareData actual,
2         AgentSquareData neighbour, int dir) {
3     actual.neighbours[dir] = neighbour;
4
5     neighbour.neighbours[(dir + 2) % 4] = actual;
6 }
```

(a) Example 1



```java
1 private void linkLabels() {
2     for( JLabel lbl : lbl_map.keySet() ) {
3         String id = lbl_map.get(lbl).toString();
4         lbl.setLabelFor((Component) engine.
5             getIdMap().get(id));
6     }
7 }
```

(b) Example 2



```java
1 public LocalDate checkDate(String prompt) {
2     LocalDate date = LocalDate.of(1850, 1, 1);
3     boolean badInput = true;
4     do{
5         date = LocalDate.parse(c.gets(prompt), format);
6         badInput = false;
7     } while (badInput);
8     return date;
9 }
```

(c) Example 3

Figure 4.5: Qualitative Analysis Examples

Qualitative Analysis

To gain a deeper insight on the quality of D-REX's predicted exception prone tokens and their usability in real coding scenarios, we picked a number of methods from the 200k test dataset and manually investigated D-REX's predictions for them. Figure 4.5 shows three of these methods, where in each example, the left half shows the method with predicted exception prone tokens circled, and the right half shows the predicted exception types and exception prone tokens with their probability scores.

The example in Figure 4.5a shows a method that can throw *ArrayIndexOutOfBoundsException*. The top 3 token predictions are *ArrayAccess*, *ArrayAccessBinary* and *PLUS* (pointing to plus operator). We see that *dir*, used as index for accessing the array, is a parameter whose value is determined at runtime, and hence, using it to access the array index at line 3 can throw the predicted exception. Even if the exception does not happen at this line, it is possible to happen at line 5 when the array index is accessed by doing a mathematical operation on *dir*. Hence, this access to array which points to an *ArrayAccessBinary* and the plus operator used in it are true exception prone tokens.

Figure 4.5b presents a method susceptible to *ClassCastException*. The top 1 predicted token is *Cast* (a derived token corresponding to the type casting implemented at line 4) which is a true prediction. The next two predicted exception prone tokens are *getIdMap()* and *get()*. At first look, they both seem to be false positive predictions, however, a close look shows that the second predicted exception type is *NullPointerException* (although with low probability) and these tokens can throw such exception. This case shows that although the focus is often on the top 1 predicted exception prone token, the next prediction gives insights to the developer as well.

We also looked at examples where the top 1 prediction of D-REX is incorrect, as shown in Figure 4.5c. This method is prone to *DateTimeParseException* with the top 2 exception prone tokens predicted as: *gets()* and *parse()*. Note that *parse()* is directly related to the predicted exception and the top 1 prediction is not a true positive. A reason can be that *DateTimeParseException* is a rare exception, with only 357 appearances in the whole 200k dataset. As a result, D-REX has not seen many examples of the tokens accompanied by this exception. Although it predicted the true token responsible for the exception, this token is in the second rank. An important note here is that the top 1 and top 2 predicted tokens have a close predicted probability showing that they were close choices for D-REX.

The above examples suggest that D-REX is able to produce meaningful predictions favoring the potentially exception prone tokens. Therefore, it can help the developer in making better judgements on the causes for exceptions in their code.

4.6 Related Work

Java exceptions and exception handling. A very recent related work is FuzzyCatch [92, 93], which focuses on recommending both exception types exception handling code in Android Ecosystem. The recommendation of exception types is based on the co-occurrences of API method calls and exception types in their dataset. D-REX is different than FuzzyCatch as D-REX works for Java (in general) and can predict exception prone tokens as well. Also, we do not focus on recommending exception handling code.

Barbosa et al. [25, 26] propose heuristic strategies based on the code context (structural information of code) to search for exception handling code. Rahman et al.'s approach [106] recommends exception handling code by doing a code search on a number of popular GitHub open source repositories. EH-Recommender [76] leverages program context in recommending exception handling code, where program context can be exceptional, architectural or functional. Maestro [84] recommends the StackOverflow post mostly related to a runtime exception using a special code representation, called Abstract Program Graph. Our work is different than these as we recommend relevant runtime exception types and exception prone code elements, not the exception handling code or post.

Nakshatri et al. [91] studied practices and patterns of handling checked exceptions revealing exception types higher in the class hierarchy being handled more than the concrete ones. Sena et al. [118] studied exception handling practices on a set of 656 Java libraries, finding a large portion of undocumented runtime exceptions in the studied libraries, and finding API runtime exception handling to be more frequent than API checked exception handling. Nguyen et al. [94] studied 246 exception related bugs from Android apps, finding 51% of them to be thrown by Android API method calls and that runtime exception are more prevalent than checked ones.

Source code representation learning. Another line of related work is source code representation learning. As discussed by Chen et al. [33], these approaches can largely be divided into learning token level and function (or method) level embeddings. In Code2Vec [12] and Code2Seq [11], authors propose to learn the 'code embedding' as continuous distributed vectors, which is similar to our goal. However, they rely on decomposing the method into a collection of paths in its abstract syntax tree (AST), and learn the representation vector for each path, as well as the the amount of attention to put in each paths. In contrast, D-REX simplifies this process and only needs to pre-process the code into a single sequence, then it leverages the representation learning ability of the state-of-art BERT model to summarize the context of the source code. Moreover, since one of our goals is to predict the exception prone likelihood of each token in the input sequence, a finer grained embedding at the token level is needed. Therefore the AST path level embedding may be less effective to infer whether each token is exception prone. Another recent work in building contextual embeddings of source code is CuBERT [69], which directly uses the architecture of BERT [43] and hence, is similar to our plain BERT model. However, CuBERT is not specifically adapted for exception type recommendation in Java. Moreover, it does not leverage the information about the tokens in the try block and does not recommend exception prone tokens.

4.7 Discussion

Runtime exceptions and try blocks used in training and evaluation of this work are extracted from a set of Java open source repositories from GitHub. We did not validate these exceptions and their associated try blocks with respect to their correctness, however, we used enough projects (200k) to address for any inaccuracy possible in the projects. We used Java Parser to parse these projects and any errors in this tool may affect the results. We aimed for comparing D-REX with the state-of-the-art tool, FuzzyCatch, however, since FuzzyCatch is exclusively trained on Android ecosystem, we were not able to perform a direct comparison.

We implemented a baseline inspired by FuzzyCatch's approach to mitigate this issue.

Runtime exception handling is an important and challenging task that requires understanding the runtime state of programs. We proposed D-REX to ease this task by simultaneously achieving two goals: 1) prediction of the relevant exception types given un-labeled code, and 2) prediction of the exception prone parts of the code. These two objectives are optimized at the same time by training D-REX from end to end. For training and evaluation of D-REX, we created a benchmark dataset which will be made publicly available. We demonstrated through experiments that D-REX significantly outperforms multiple non-BERT baselines on 200k test dataset, and on the unseen projects dataset.

Our next step is to package D-REX as an IDE plugin: developers will be able to ask for possible exceptions as they code and the corresponding exception prone code elements will be highlighted. We also aim to integrate D-REX with a component for recommending fixes to the predicted exceptions. Further, it is interesting to explore other tasks that can utilize the the location awareness of LA-BERT. One example is code refactoring where the location awareness can be used to find parts of code that need refactoring. More generally, D-REX provides a way to leverage any kind of extra information in the training data, and its application is not limited to using the try block information in the exception prediction task.

Chapter 5

Conclusion and Future Work

We have seen three successful approaches to facilitate the learning of sparse representation, where deep neural networks have shown to be useful to model various kinds of sparse distributions.

First, we demonstrated a variational inference algorithm of Bayesian variable selection with non-local prior (Chapter 2). As the variational inference is becoming increasingly popular due to its parallelizability and users can leverage off-the-shelf deep learning packages for automatic differentiation, a promising future direction is to explore different kinds of flexible variational distributions parameterized by neural networks. Further, this can be useful for a more principled approach to neural network compression [78, 135] with desirable capability to incorporate proper prior information for hardware efficiency on edge devices. Another important range of the applications for the variational variable selection is neural image compression [23, 80], where a sparse representation of the latent code is crucial in achieving the best rate-distortion performance.

Second, we showed a generative model for sparse images generation in high energy physics (Chapter 3). A stable, flexible, as well as parameter efficient model of the sparse distribution is the key to generate high quality sparse images. Therefore, future work on flexible distributions will greatly benefit this line of research. Another interesting direction for generation of larger sparse images is to extend the current model to generate image patches in an auto-regressive pattern. This will further speed up the generation and can be potentially used for generating sparse 3D objects or videos.

Lastly, we described a sparse location aware model for exception handling in software engineering (Chapter 4). There, we showed the sparse location information of the try block tokens in a JAVA code snippet is helpful to recommendation of exception and beneficial to the exception type predictions. A promising future direction is to extend the current model to predict the complete exception handling code.

Bibliography

[1] Best (and worst) java exception handling practices. https://able.bio/DavidLandup/best-and-worst-java-exception-handling-practices--18h55kh. Accessed: 2020-08-15.

[2] Beware the dangers of generic exceptions. https://www.infoworld.com/article/2073800/beware-the-dangers-of-generic-exceptions.html. Accessed: 2020-08-15.

[3] Java parser. https://javaparser.org/. Accessed: 2020-06-03.

[4] *LHCb calorimeters: Technical Design Report.* Technical Design Report LHCb. CERN, Geneva, (2000).

[5] G. Aad et al. The ATLAS Simulation Infrastructure. *Eur. Phys. J. C*, 70, 2010.

[6] S. Agostinelli et al. GEANT4: A Simulation toolkit. *Nucl. Instrum. Meth. A*, 506, 2003.

[7] Kara Agster and Rebecca Burwell. Cortical efferents of the perirhinal, postrhinal, and entorhinal cortices of the rat. *Hippocampus*, 19:1159–86, 12 2009.

[8] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013.

[9] Timothy A. Allen, Daniel M. Salz, Sam McKenzie, and Norbert J. Fortin. Nonspatial sequence coding in ca1 neurons. *Journal of Neuroscience*, 36(5):1547–1563, 2016.

[10] Leandro G. Almeida, Mihailo Backović, Mathieu Cliche, Seung J. Lee, and Maxim Perelstein. Playing Tag with ANN: Boosted Top Identification with Pattern Recognition. *JHEP*, 07:086, 2015.

[11] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *CoRR*, abs/1803.09473, 2018.

[13] Saúl Alonso Monsalve and Leigh Whitehead. Image-Based Model Parameter Optimization Using Model-Assisted Generative Adversarial Networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP:1–6, 03 2020.

[14] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H. S. Shao, T. Stelzer, P. Torrielli, and M. Zaro. The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. *JHEP*, 07:079, 2014.

[15] D. F. Andrews and C. L. Mallows. Scale mixtures of normal distributions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(1):99–102, 1974.

[16] Martín Arjovsky and Léon Bottou. Towards Principled Methods for Training Generative Adversarial Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, (2017).

[17] Artin Armagan, David B. Dunson, and Jaeyong Lee. Generalized double pareto shrinkage. *Statistica Sinica*, 23 1:119–143, 2013.

[18] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K Roy, and Kevin A Schneider. How developers use exception handling in java? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 516–519. IEEE, 2016.

[19] P. Baldi. *Deep Learning in Science: Theory, Algorithms, and Applications*. Cambridge University Press, Cambridge, UK, (2020). In press.

[20] Pierre Baldi, Kevin Bauer, Clara Eng, Peter Sadowski, and Daniel Whiteson. Jet Substructure Classification in High-Energy Physics with Deep Neural Networks. *Phys. Rev.*, D93, 2016.

[21] Pierre Baldi, Jianming Bian, Lars Hertel, and Lingge Li. Improved Energy Reconstruction in NOvA with Regression Convolutional Neural Networks. *Phys. Rev. D*, 99, 2019.

[22] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for Exotic Particles in High-Energy Physics with Deep Learning. *Nature Communications*, 5:4308, 2014.

[23] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimized image compression. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[24] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750, 2019.

[25] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. Heuristic strategies for recommendation of exception handling code. In *2012 26th Brazilian Symposium on Software Engineering*, pages 171–180. IEEE, 2012.

[26] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. A recommendation system for exception handling code. In *2012 5th International Workshop on Exception Handling (WEH)*, pages 52–54. IEEE, 2012.

[27] James Barnard, Edmund Noel Dawe, Matthew J. Dolan, and Nina Rajcic. Parton Shower Uncertainties in Jet Substructure Analyses with Deep Neural Networks. *Phys. Rev.*, D95(1):014018, 2017.

[28] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[29] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, (2010). `https://leon.bottou.org/publications/pdf/compstat-2010.pdf`.

[30] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations ICLR*, (2019).

[31] Federico Carminati, Maurizio Pierini Gulrukh Khattak, Benjamin Hooberman Amir Farbin, Wei Wei, Matt Zhang, Vitória Barin Pacela, Sofia Vallecorsafac, Maria Spiropulu, and Jean-Roch Vlimant. Calorimetry with Deep Learning: Particle Classification, Energy Regression, and Simulation for High-Energy Physics. *Deep Learning for Physical Sciences, Workshop at the 31st Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[32] Carlos M. Carvalho, Nicholas G. Polson, and James G. Scott. Handling sparsity via the horseshoe. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 73–80, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[33] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *CoRR*, abs/1904.03061, 2019.

[34] S. Chintala. How to train a GAN? In *Workshop on Generative Adversarial Networks*, 2016.

[35] Josh Cogan, Michael Kagan, Emanuel Strauss, and Ariel Schwarztman. Jet-Images: Computer Vision Inspired Techniques for Jet Tagging. *JHEP*, 02, 2015.

[36] Kyle Cranmer, Stefan Gadatsch, Aishik Ghosh, Tobias Golling, David Rousseau Gilles Louppe, Dalila Salamani, and Graeme Stewart on behalf of the ATLAS Collaboration. Deep generative models for fast shower simulation in ATLAS. *Bayesian Deep*

Learning, Workshop at the 32nd Conference on Neural Information Processing Systems (NeurIPS), 2018. `http://bayesiandeeplearning.org/2018/papers/24.pdf`.

[37] J. de Favereau et al. DELPHES 3, A modular framework for fast simulation of a generic collider experiment. *JHEP*, 1402:057, 2014.

[38] Luke de Oliveira, Michael Kagan, Lester Mackey, Benjamin Nachman, and Ariel Schwartzman. Jet-Images — Deep Learning Edition. *Journal of High Energy Physics*, 2016.

[39] Luke de Oliveira, Michela Paganini, and Benjamin Nachman. Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis. *Comput Softw Big Sci*, 2017.

[40] K. Deja, T. Trzciński, and Ł. Graczykowski. Generative Models for Fast Cluster Simulations in the TPC for the ALICE Experiment. In *Information Technology, Systems Research, and Computational Physics*, pages 267–280, Cham, (2020). Springer International Publishing.

[41] S. Delaquis, M.J. Jewell, I. Ostrovskiy, M. Weber, T. Ziegler, J. Dalmasson, L. Kaufman, T. Richards, J.B. Albert, G. Anton, I. Badhrees, P.S. Barbeau, R. Bayerlein, D. Beck, Vladimir Belov, M. Breidenbach, Thomas Brunner, G.F. Cao, W.R. Cen, and O.Ya Zeldovich. Deep Neural Networks for Energy and Position Reconstruction in EXO-200. *Journal of Instrumentation*, 13, 2018.

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Conference on Computer Vision and Pattern Recognition*, (2009). `http://www.image-net.org/papers/imagenet_cvpr09`.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[44] Benoit Gaudin, Emil Iordanov Vassev, Patrick Nixon, and Michael Hinchey. A control theory based approach for self-healing of un-handled runtime exceptions. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 217–220, 2011.

[45] Edward I. George and Robert E. McCulloch. Variable selection via Gibbs sampling. *Journal of the American Statistical Association*, 88(423):881–889, 1993.

[46] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked Autoencoder for Distribution Estimation. *CoRR*, abs/1502.03509, 2015.

[47] Soumya Ghosh, Jiayu Yao, and Finale Doshi-Velez. Structured variational learning of Bayesian neural networks with horseshoe priors. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1744–1753, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[48] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[49] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., (2014).

[50] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep AutoRegressive Networks. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1242–1250, Bejing, China, 22–24 Jun (2014). PMLR.

[51] D. Guest, J. Collado, P. Baldi, S. Hsu, G. Urban, and D. Whiteson. Jet flavor classification in high-energy physics with deep neural networks. *Physical Review D*, 94:112002, 2016.

[52] Daniel Guest, Julian Collado, Pierre Baldi, Shih-Chieh Hsu, Gregor Urban, and Daniel Whiteson. Jet Flavor Classification in High-Energy Physics with Deep Neural Networks. *Physical Review D*, 94, 2016.

[53] Song Han, Huizi Mao, and W. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2016.

[54] Bobak Hashemi, Nick Amin, Kaustuv Datta, Dominick Olivito, and Maurizio Pierini. LHC analysis-specific datasets with Generative Adversarial Networks. *CoRR*, 2019.

[55] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[56] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, (2016).

[57] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[58] Dan Hendrycks and Kevin Gimpel. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR*, abs/1606.08415, 2016.

[59] Lars Hertel, Julian Collado, Peter Sadowski, Jordan Ott, and Pierre Baldi. Sherpa: Robust Hyperparameter Optimization for Machine Learning. *SoftwareX*, 2020. In press. Software available at: https://github.com/sherpa-ai/sherpa.

[60] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[61] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[62] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2078–2087, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[63] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural Autoregressive Flows. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2078–2087, Stockholmsmässan, Stockholm Sweden, 10–15 Jul (2018). PMLR. `http://proceedings.mlr.press/v80/huang18d/huang18d.pdf`.

[64] John Ingraham and Debora Marks. Variational inference for sparse and undirected models. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1607–1616, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[65] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *CoRR*, abs/1611.01144, 2016.

[66] Shujuan Jiang, Hongchang Zhang, Qingtan Wang, and Yanmei Zhang. A debugging approach for java runtime exceptions based on program slicing and stack traces. In *2010 10th International Conference on Quality Software*, pages 393–398. IEEE, 2010.

[67] Valen E. Johnson and David Rossell. On the use of non-local prior densities in bayesian hypothesis tests. *Journal of the Royal Statistical Society Series B*, 72(2):143–170, 2010.

[68] Valen E. Johnson and David Rossell. Bayesian model selection in high-dimensional settings. *Journal of the American Statistical Association*, 107(498):649–660, 2012.

[69] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. Vienna, Austria, 2020.

[70] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[71] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[72] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR*, (2014).

[73] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10215–10224. Curran Associates, Inc., 2018.

[74] Patrick T. Komiske, Eric M. Metodiev, and Matthew D. Schwartz. Deep learning in color: towards automated quark/gluon jet discrimination. *JHEP*, 01:110, 2017.

[75] Hugo Larochelle and Iain Murray. The Neural Autoregressive Distribution Estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 29–37, Fort Lauderdale, FL, USA, 11–13 Apr (2011). PMLR. `http://proceedings.mlr.press/v15/larochelle11a/larochelle11a.pdf`.

[76] Yuhang Li, Shi Ying, Xiangyang Jia, Yisen Xu, Lily Zhao, Guoli Cheng, Bingming Wang, and Jifeng Xuan. Eh-recommender: Recommending exception handling strategies based on program context. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 104–114. IEEE, 2018.

[77] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[78] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3290–3300, Red Hook, NY, USA, 2017. Curran Associates Inc.

[79] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

[80] Y. Lu, Yinhao Zhu, Y. Yang, A. Said, and T. Cohen. Progressive neural image compression with nested quantization and latent ordering. *ArXiv*, abs/2102.02913, 2021.

[81] Yadong Lu, Julian Collado, Kevin Bauer, Daniel Whiteson, and Pierre Baldi. Sparse Image Generation with Decoupled Generative Models. In *Neural Information Processing Systems, Machine Learning and the Physical Sciences Workshop*, (2019). `https://ml4physicalsciences.github.io/2019/files/NeurIPS_ML4PS_2019_161.pdf`.

[82] Yadong Lu, Julian Collado, Daniel Whiteson, and Pierre Baldi. Sarm: Sparse autoregressive model for scalable generation of sparse images in particle physics, 2020.

[83] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *CoRR*, abs/1611.00712, 2016.

[84] Sonal Mahajan, Negarsadat Abolhassani, and Mukul R Prasad. Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1052–1064, 2020.

[85] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[86] Valen E. Johnson Minsuk Shin, Anirban Bhattacharya. Scalable bayesian variable selection using nonlocal prior densities in ultrahigh-dimensional settings. *arxiv*, 2017.

[87] T. J. Mitchell and J. J. Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83(404):1023–1032, 1988.

[88] P. Musella and F Pandolfi. Fast and Accurate Simulation of Particle Detectors Using Generative Adversarial Networks. *Computing and Software for Big Science*, 2018.

[89] Mustafa Mustafa, Deborah Bard, Wahid Bhimji, Zarija Lukić, Rami Al-Rfou, and Jan M. Kratochvil. CosmoGAN: Creating High-fidelity Weak Lensing Convergence Maps using Generative Adversarial Networks. *Computational Astrophysics and Cosmology*, 6, 2019.

[90] Vaishnavh Nagarajan and J. Zico Kolter. Gradient descent GAN optimization is locally stable. In *Advances in Neural Information Processing Systems 30*, pages 5585–5595. Curran Associates, Inc., (2017).

[91] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in java projects: An empirical study. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 500–503. IEEE, 2016.

[92] Tam Nguyen, Phong Vu, and Tung Nguyen. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 390–393. IEEE, 2019.

[93] Tam Nguyen, Phong Vu, and Tung Nguyen. Code recommendation for exception handling. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1027–1038, 2020.

[94] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. An empirical study of exception handling bugs and fixes. In *Proceedings of the 2019 ACM Southeast Conference*, pages 257–260, 2019.

[95] Nikiforos Nikiforou. Performance of the ATLAS Liquid Argon Calorimeter After Three Years of LHC Operation and Plans For a Future Upgrade. In *3rd International*

Conference on Advancements in Nuclear Instrumentation Measurement Methods and their Applications, (2013).

[96] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and Checkerboard Artifacts. *Distill*, 1(10), October 2016. `http://distill.pub/2016/deconv-checkerboard`.

[97] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel Recurrent Neural Networks. In *Proceedings of The 33rd International Conference on Machine Learning ICML*, volume 48 of *Proceedings of Machine Learning Research*, pages 1747–1756, New York, New York, USA, (2016). PMLR.

[98] Sydney Otten, Sascha Caron, Wieske de Swart, Melissa van Beekveld, Luc Hendriks, Caspar van Leeuwen, Damian Podareanu, Roberto Ruiz de Austri, and Rob Verheyen. Event Generation and Statistical Sampling for Physics with Deep Generative Models and a Density Information Buffer. *CoRR*, 2019.

[99] Michela Paganini, Luke de Oliveira, and Benjamin Nachman. CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks. *Phys. Rev. D*, 97, 2018.

[100] Trevor Park and George Casella. The bayesian lasso. Technical report, 2005.

[101] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017.

[102] Juho Piironen and Aki Vehtari. On the Hyperprior Choice for the Global Shrinkage Parameter in the Horseshoe Prior. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 905–913. PMLR, April 2017.

[103] Nicholas G. Polson, James G. Scott, Bertrand Clarke, and C. Severinski. *Shrink Globally, Act Locally: Sparse Bayesian Regularization and Prediction*, volume 9780199694587. Oxford University Press, January 2012.

[104] G. r. Khattak, S. Vallecorsa, and F. Carminati. Three Dimensional Energy Parametrized Generative Adversarial Networks for Electromagnetic Shower Simulation. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3913–3917, 2018. `https://ieeexplore.ieee.org/document/8451587`.

[105] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR*, 2015.

[106] Mohammad Masudur Rahman and Chanchal K Roy. On the use of context in recommending exception handling code examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 285–294. IEEE, 2014.

[107] Rahmat Rahmat, Rob Kroeger, and Andrea Giammanco. The Fast Simulation of The CMS Experiment. *Journal of Physics: Conference Series*, 396, 2012.

[108] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[109] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1530–1538, Lille, France, 07–09 Jul 2015. PMLR.

[110] David Rossell and Donatello Telesca. Nonlocal priors for high-dimensional estimation. *Journal of the American Statistical Association*, 112(517):254–265, 2017.

[111] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[112] Y. Rubner, C. Tomasi, and L. J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40, 2000. `https://link.springer.com/article/10.1023/A:1026543900054`.

[113] Peter Sadowski, Julian Collado, Daniel Whiteson, and Pierre Baldi. Deep learning, dark knowledge, and dark matter. In *Proceedings of the 2014 International Conference on High-Energy Physics and Machine Learning - Volume 42*, HEPML'14, page 81–97. JMLR.org, 2014.

[114] Peter Sadowski, Julian Collado, Daniel Whiteson, and Pierre Baldi. Deep Learning, Dark Knowledge, and Dark Matter. In *Proceedings of the NIPS 2014 Workshop on High-energy Physics and Machine Learning*, volume 42 of *Proceedings of Machine Learning Research*, pages 81–87, Montreal, Canada, (2015). PMLR. `http://proceedings.mlr.press/v42/sado14.html`.

[115] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[116] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications. *CoRR*, 2017.

[117] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11):2673–2681, November 1997.

[118] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the exception handling strategies of java libraries: An empirical study. In *Proceedings*

of the 13th International Conference on Mining Software Repositories, pages 212–222, 2016.

[119] Ilsoo Seong, Lars Hertel, Julian Collado, Lingge Li, Nitish Nayak, Jianming Bian, and Pierre Baldi. Convolutional Neural Networks for Energy and Vertex Reconstruction in DUNE. In *33rd Conference on Neural Information Processing Systems (NeurIPS), Machine Learning and the Physical Sciences Workshop*, (2019). `https://ml4physicalsciences.github.io/2019/files/NeurIPS_ML4PS_2019_77.pdf`.

[120] Viraj Shah, Ameya Joshi, Sambuddha Ghosal, Balaji Sesha Sarath Pokuri, Soumik Sarkar, Baskar Ganapathysubramanian, and Chinmay Hegde. Encoding Invariances in Deep Generative Models. *CoRR*, 2019.

[121] Babak Shahbaba, Lingge Li, Forest Agostinelli, Mansi Saraf, Gabriel A. Elias, Pierre Baldi, and Norbert J. Fortin. Hippocampal ensembles represent sequential relationships among discrete nonspatial events. *bioRxiv*, 2019.

[122] Chase Shimmin, Peter Sadowski, Pierre Baldi, Edison Weik, Daniel Whiteson, Edward Goul, and Andreas Søgaard. Decorrelated Jet Substructure Tagging using Adversarial Neural Networks. *Physical Review D*, 96, 03 2017.

[123] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164, 2009.

[124] Torbjorn Sjostrand, Stephen Mrenna, and Peter Z. Skands. PYTHIA 6.4 Physics and Manual. *JHEP*, 0605:026, 2006.

[125] Anant Subramanian, Danish Pruthi, Harsh Jhamtani, Taylor Berg-Kirkpatrick, and Eduard H. Hovy. SPINE: sparse interpretable neural embeddings. *CoRR*, abs/1711.08792, 2017.

[126] Fei Sun, J. Guo, Yanyan Lan, Jun Xu, and X. Cheng. Sparse word embeddings using l1 regularized online learning. In *IJCAI*, 2016.

[127] Robert Tibshirani. Regression shrinkage and selection via the lasso. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 58:267–288, 1994.

[128] Michalis K. Titsias and Miguel Lázaro-Gredilla. Spike and slab variational inference for multi-task and multiple kernel learning. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2339–2347. Curran Associates, Inc., 2011.

[129] Jakub M. Tomczak and Max Welling. Improving variational auto-encoders using householder flow. *CoRR*, abs/1611.09630, 2016.

[130] Francesco Tonolini, Bjorn Sand Jensen, and Roderick Murray-Smith. Variational sparse coding, 2019.

[131] Benigno Uria, Iain Murray, and Hugo Larochelle. RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems 26*, pages 2175–2183. Curran Associates, Inc., (2013).

[132] Rianne van den Berg, Leonard Hasenclever, Jakub M. Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, pages 393–402, 2018.

[133] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[134] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[135] Ying Wang, Y. Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. *ArXiv*, abs/2007.10463, 2020.

[136] Ying Wang, Yadong. Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. *European Conference on Computer Vision*, abs/2007.10463, 2020.

[137] Menno P. Witter, Thanh P. Doan, Bente Jacobsen, Eirik S. Nilssen, and Shinya Ohara. Architecture of the entorhinal cortex a review of entorhinal anatomy in rodents with some comparative notes. *Frontiers in Systems Neuroscience*, 11:46, 2017.

[138] Li Xu, Shicheng Zheng, and Jiaya Jia. Unnatural l0 sparse representation for natural image deblurring. In *CVPR*, pages 1107–1114. IEEE Computer Society, 2013.

[139] Kai Zhou, Gergely Endrődi, Long-Gang Pang, and Horst Stöcker. Regressive and Generative Neural Networks for Dcalar Field Theory. *Phys. Rev. D*, 100:011501, 2019.

[140] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *CoRR*, 2017.

Appendix A

Supplementary Material for Chapter 2

A.1 Derivation of the Evidence Lower Bound for non-local priors

Suppose $\boldsymbol{y} \in R^D$ is our observed data, the model defines the likelihood function $f(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{z}))$, where $\boldsymbol{\beta} \in R^M$ is the model parameter, and $\boldsymbol{z} \in R^M$ is the Bernoulli mask over $\boldsymbol{\beta}$. It is equivalent to a spike and slab prior on $\tilde{\boldsymbol{\beta}} = \boldsymbol{\beta} \odot \boldsymbol{z}$, where $\odot$ is the element-wise product and $\boldsymbol{\beta}$ and $\boldsymbol{z}$ are independently distributed. The general form of Evidence Lower Bound (ELBO) follows the following form:

$$\begin{aligned}
\text{ELBO} =& E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log P(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{z}, \boldsymbol{x})) - E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log \frac{q_{\pi,\eta}(\boldsymbol{\beta}, \boldsymbol{z})}{p(\boldsymbol{\beta}, \boldsymbol{z})} && \text{(A.1)} \\
=& E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log P(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{z}, \boldsymbol{x})) - [q_\pi(\boldsymbol{z}=0) \log \frac{q_\pi(\boldsymbol{z}=0)}{p_\pi(\boldsymbol{z}=0)} + && \text{(A.2)} \\
& q_\pi(\boldsymbol{z}=1) \log \frac{q_\pi(\boldsymbol{z}=1)q_\eta(\boldsymbol{\beta}|\boldsymbol{z}=1)}{p_\pi(\boldsymbol{z}=1)p_\eta(\boldsymbol{\beta}|\boldsymbol{z}=1)}] \\
=& E_{q_\pi(\boldsymbol{z})q_\eta(\boldsymbol{\beta}|\boldsymbol{z})} \log P(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{z}, \boldsymbol{x})) - [\sum_i^M \text{KL}(q_{\pi_i}(z_i)||p(z_i)) + && \text{(A.3)} \\
& \sum_i^M q_{\pi_i}(z_i=1) \cdot \text{KL}(q_{\eta_i}(\beta_i|z_i=1)||p(\beta_i|z_i=1))]
\end{aligned}$$

From the second equation to the third, we adopt Mean Field assumption which allows the factorization of $q_\eta(\boldsymbol{\beta}|\boldsymbol{z}) = \prod_i q_{\eta_i}(\beta_i)q_{\pi_i}(z_i)$.

In practice we use the hard concrete relaxation [], $\bar{s}$ to approximate the Bernoulli random variable z, to allow for gradient based optimization:

$$\begin{aligned}
& s = \text{Sigmoid}((\log u - \log(1-u) + \alpha)/\lambda), \quad u \sim \text{Uniform}(0,1) \\
& \bar{s} = s(\zeta - \delta) + \delta
\end{aligned} \tag{A.4}$$

Hence we have the approximation: $\gamma_j = p(z_j = 1) \approx q(\bar{s} \neq 0) = \text{Sigmoid}(\alpha_j - \lambda \log \frac{-\delta}{\zeta})$, where we set $\delta = -0.1, \zeta = 1.1, \lambda = \frac{2}{3}$ as hyperparameters. Under this relaxation:

$$\tilde{\beta}_j = \beta_j \cdot z_j \approx \beta_j \cdot \bar{s}_j \tag{A.5}$$

Since for concrete random variable s_j, we have

$$\lim_{\lambda \to 0} E s_j = \text{Sigmoid}(\alpha_j) \tag{A.6}$$

in [?]. Thus:

$$\lim_{\lambda \to 0} E\bar{s}_j = (\zeta - \delta)\text{Sigmoid}(\alpha_j) + \delta \tag{A.7}$$

Therefore at test time, we use the expectation $\lim_{\lambda \to 0} E\bar{s}_j$ as our estimated approximate value for Ez_j.

Next we prove **when the expected value of hard concrete variable** $\lim_{\lambda \to 0} E\bar{s}_j > 0$**, the gradient always points to the direction such that it pushes** $\lim_{\lambda \to 0} E\bar{s}_j$ **to zero**.

A.2 Proof of Theorem 1

Lemma 1. The absolute value of the derivative of $\text{KL}(p_{z_j}||p_{z_j})$ with respect to α_j: $|\frac{\partial \text{KL}(p_{z_j}||p_{z_j})}{\partial \alpha_j}|$ is bounded by a constant $C > 0$.

Proof.

$$\begin{aligned}
\frac{\partial \text{KL}(p_{z_j}||p_{z_j})}{\partial \alpha_j} =& \frac{\partial \gamma_j}{\partial \alpha_j} \frac{\partial \text{KL}(p_{z_j}||p_{z_j})}{\partial \gamma_j} && \text{(A.8)}\\
=& \frac{\partial \gamma_j}{\partial \alpha_j} \frac{\partial \gamma_j \log \frac{\gamma_j}{\gamma_{0j}} + (1-\gamma_j)\log\frac{(1-\gamma_j)}{(1-\gamma_{0j})}}{\partial \gamma_j} && \text{(A.9)}\\
=& \frac{\partial \gamma_j}{\partial \alpha_j}(\log \frac{\gamma_j}{1-\gamma_j} + \log\frac{(1-\gamma_0)}{\gamma_0}) && \text{(A.10)}\\
=& (1-\gamma_j)\gamma_j \log\frac{\gamma_j}{1-\gamma_j} + (1-\gamma_j)\gamma_j \log\frac{(1-\gamma_0)}{\gamma_0} && \text{(A.11)}
\end{aligned}$$

Since $0 \le (1-\gamma_j)\gamma_j \le 1/4$, the second term is bounded. And since $\lim_{\gamma_j \to 0}(1-\gamma_j)\gamma_j \log\frac{\gamma_j}{1-\gamma_j} = \lim_{\gamma_j \to 1}(1-\gamma_j)\gamma_j \log\frac{\gamma_j}{1-\gamma_j} = 0$, and $(1-\gamma_j)\gamma_j \log\frac{\gamma_j}{1-\gamma_j}$ is continuous. when $\gamma_j \in (0,1)$. Therefore by *Extreme Value Theorem*, $|(1-\gamma_j)\gamma_j \log\frac{\gamma_j}{1-\gamma_j}| < C$ for some constant C. □

Theorem 2. Consider a linear regression model, where the data is generated by: $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{y} \in R^N$, $\boldsymbol{\beta} \in R^p$, $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \phi^2 \boldsymbol{I})$, $\boldsymbol{X} \sim N(\mathbf{0}, \Sigma)$, where $\max(\Sigma_{ij}) < C_0$. If the ground truth value of $\beta_{j,true} = 0$, denote $\max_{j \neq j'} |\beta_{j'} \tilde{\beta}_j| = C_1$, we have

$$E\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} < -2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2 \beta_{j'}^2 \bar{s}_{j'} - C_0 C_1 (M-1)] + \frac{C}{N}$$

where $C > 0$ is a positive constant defined in Lemma 1.

Remark: when $N > \frac{C}{2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2 \beta_{j'}^2 \bar{s}_{j'} - C_0 C_1 (M-1)]}$, we have $E\frac{\partial \mathcal{L}}{\partial \alpha_j} < 0$.

Proof. In linear regression setting, the ELBO in (A.1) can be expressed as:

$$\begin{aligned}\mathcal{L} &= \frac{1}{N}\text{ELBO} \\ &= -\frac{1}{N}\sum_{i=1}^{N}(y_i - \sum_{j=1}^{M} x_{ij}\tilde{\beta}_j)^2 - \frac{1}{N}\sum_{j=1}^{M}[\gamma_j \log\frac{\gamma_j}{\gamma_{0j}} + (1-\gamma_j)\log\frac{(1-\gamma_j)}{(1-\gamma_{0j})} + \\ &\gamma_j \text{KL}((q_{\eta_j}(\beta_j|z_j=1)||p(\beta_j|z_j=1))]\end{aligned} \tag{A.12}$$

where $\gamma_j = q(z_j = 1)$ is the variational approximation probability, and $\gamma_{0j} = p(z_j = 1)$ is the prior probability of $z_j = 1$. From (4) it is easy to get $\frac{\partial \gamma_j}{\partial \alpha_j} = \gamma_j(1-\gamma_j) \geq 0$.

$$\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} = \frac{1}{N}\sum_{i=1}^{N} 2x_{ij'}\beta_{j'}\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}(y_i - \sum_{j=1}^{M} x_{ij}\tilde{\beta}_j) - \frac{1}{N}[\frac{\partial \text{KL}(p_{z_{j'}}||p_{z_{j'}})}{\partial \alpha_{j'}} - \text{KL}(p_{\beta_{j'}}||p_{\beta_{j'}})] \tag{A.13}$$

where the KL divergence denotes $\text{KL}(p_{z_{j'}}||p_{z_{j'}}) = \gamma_{j'}\log\frac{\gamma_{j'}}{\gamma_{0j'}} + (1-\gamma_{j'})\log\frac{(1-\gamma_{j'})}{(1-\gamma_{0j'})}$. And we have:

$$\begin{aligned}\sum_{i=1}^{N} 2x_{ij'}\beta_{j'}\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}(y_i - \sum_{j=1}^{M} x_{ij}\tilde{\beta}_j) =& \sum_{i=1}^{N} 2x_{ij'}\beta_{j'}\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}(\sum_{j=1}^{M} x_{ij}\beta_{j,true} - \sum_{j=1}^{M} x_{ij}\tilde{\beta}_j + \epsilon_i) \\ =& \sum_{i=1}^{N} 2x_{ij'}\beta_{j'}\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}(\sum_{j\neq j'}^{M} x_{ij}(\beta_{j,true} - \tilde{\beta}_j) + x_{ij'}(\beta_{j',true} - \tilde{\beta}_{j'}) + \epsilon_i)\end{aligned} \tag{A.14}$$

Since $Ex_{ij} = 0$ for all i, j, $Ex_{ij}x_{ij'} = |\Sigma_{jj'}| < C_0$. Now let's consider the expectation of each term in equation (A.14), when $\beta_{j',true} = 0$. For $j \neq j'$:

$$E\sum_{j\neq j'}^{M} 2x_{ij'}x_{ij}\beta_{j'}(\beta_{j,true} - \tilde{\beta}_j)\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}} \leq 2C_0\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}|\sum_{j\neq j'}^{M} \beta_{j'}\tilde{\beta}_j|$$

And for $j = j'$:

$$E2x_{ij'}^2\beta_{j'}(\beta_{j',true} - \tilde{\beta}_{j'})\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}} = -2\sigma^2\tilde{\beta}_{j'}^2\bar{s}_{j'}\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}$$

For the remaining term related to ϵ_i, the expectation is equal to zero.

As a summary, using the above derivations, equation (A.13) can be re-written as follows:

$$E\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} \leq \frac{1}{N}\sum_{i=1}^{N}[2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}(C_0|\sum_{j\neq j'}^{M}\beta_{j'}\tilde{\beta}_j| - \sigma^2\bar{s}_{j'}\tilde{\beta}_{j'}^2)] - \frac{1}{N}\frac{\partial \gamma_{j'}}{\partial \alpha_{j'}}[\frac{\partial \mathrm{KL}(p_{z_{j'}}||p_{z_{j'}})}{\partial \gamma_{j'}} - \mathrm{KL}(p_{\beta_{j'}}||p_{\beta_{j'}})] \tag{A.15}$$

We then look at the second term of equation (A.15). Since KL divergence is always greater or equal to zero, we have $-\mathrm{KL}(p_{\beta_{j'}}||p_{\beta_{j'}}) < 0$. Further, we have:

$$\frac{\partial \gamma_{j'}}{\partial \alpha_{j'}} \frac{\partial \mathrm{KL}(p_{z_{j'}}||p_{z_{j'}})}{\partial \gamma_{j'}} = \frac{\partial \gamma_{j'}}{\partial \alpha_{j'}} \log \frac{\gamma_{j'}}{1-\gamma_{j'}} \tag{A.16}$$

$$= (1-\gamma_{j'})\gamma_{j'} \log \frac{\gamma_{j'}}{1-\gamma_{j'}} \tag{A.17}$$

is bounded above by a constant according to Lemma 1.

Combining the above computation, and assuming $\max_{j \neq j'} |\beta_{j'}\tilde{\beta}_j| = C_1$, equation (A.15) can be re-written as:

$$\begin{aligned}
E\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} &\leq -2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2\beta_{j'}^2\bar{s}_{j'} - C_0C_1(M-1)] - \frac{1}{N}\frac{\partial \gamma_{j'}}{\partial \alpha_{j'}}[\log \frac{\gamma_{j'}}{1-\gamma_{j'}} + \mathrm{KL}(p_{\beta_{j'}}||p_{\beta_{j'}})] \\
&\leq -2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2\beta_{j'}^2\bar{s}_{j'} - C_0C_1(M-1)] - \frac{1}{N}\frac{\partial \gamma_{j'}}{\partial \alpha_{j'}}\log \frac{\gamma_{j'}}{1-\gamma_{j'}} && \text{(A.18)} \\
&< -2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2\beta_{j'}^2\bar{s}_{j'} - C_0C_1(M-1)] + \frac{C}{N} && \text{(A.19)}
\end{aligned}$$

Therefore, when

$$N > \frac{C}{2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}[\sigma^2\beta_{j'}^2\bar{s}_{j'} - C_0C_1(M-1)]} \tag{A.20}$$

we have $E\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} < 0$. This completes the proof. □

Remark: As a special case when $C_0 = 0$, and

$$N > \frac{C}{2\frac{\partial \bar{s}_{j'}}{\partial \alpha_{j'}}\sigma^2\beta_{j'}^2\bar{s}_{j'}} \tag{A.21}$$

we have: $E\frac{\partial \mathcal{L}}{\partial \alpha_{j'}} < 0$

Appendix B

Supplementary Material for Chapter 3

B.1 2D Toy Example

We simulate a dataset containing pairs of two variables x_0 and x_1, such that $x_0 \sim p(x_0|x_1)$ and $x_1 \sim p(x_1)$. In this toy example we show that the autoregressive model is still able to learn to generate the joint distribution of x_0 and x_1, even though during training it is forced to learn $x_0 \sim p(x_0)$ first, and then to learn the dependency $p(x_1|x_0)$. The simulated training data contains 1000 pairs of $\{x_0, x_1\}$ according to $x_1 \sim N(0, 1)$ and $x_0 = x_1 + \epsilon$, where $\epsilon \sim N(0, 1)$, a standard normal distribution independent of x_1. The joint distribution of x_0, x_1 is shown in figure B.1. The toy autoregressive model learns to generate x_0 using two learnable parameters, μ_0 and $\log(\sigma_0)$, corresponding to the mean and log standard deviation of x_0. It has a single linear layer for predicting μ_0 and $\log(\sigma_0)$, which corresponds to the mean and log standard deviation of x_1. The model is trained for 5000 iterations, by maximizing the likelihood $p(x_0, x_1)$. During the generation stage, the model generates x_0 without knowing x_1. Since the goal of the model is to generate the joint distribution of $(x_0, x_1) \sim P(x_0, x_1)$, to do this it only needs to learn the marginal distribution, which is $x_0 \sim N(0, 2)$ and the

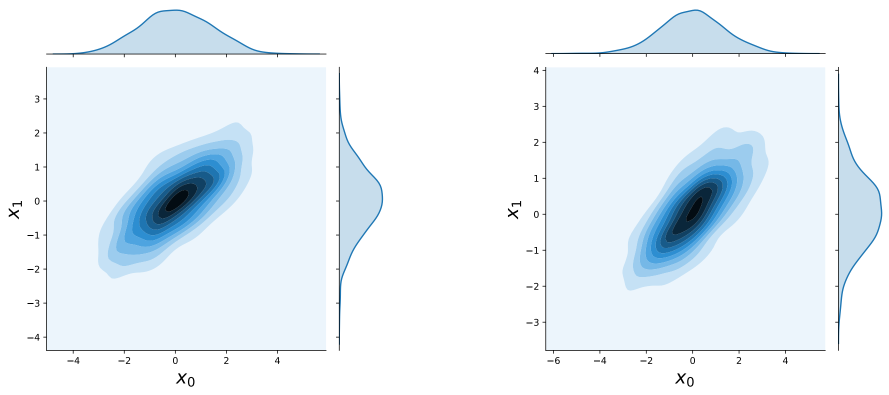


Figure B.1: **Left**: Density plot of training data. **Right**: Density plot of generated data. The two distributions are very close, showing the ARM is able to learn the joint distribution of x_0 and x_1 well.

relationship $x_1 = x_0 - \epsilon$. figure B.1 shows the result of training this model and we can see it correctly learns the means and variances of $\{x_0, x_1\}$ along with the data distribution despite the fact that it has to generate x_0 before generating x_1.

B.2 MADE Structure

The MADE structure enforces the auto-regressive property on fully connected layers by using a carefully selected binary mask on the weights of the layer. The joint likelihood of the MADE structure can be evaluated in one forward pass of the network during training, which is not possible in other models like Pixel-RNN [97] and Pixel CNN++ [116]. This allows MADE to take advantage of the GPU acceleration. In our SARM implementation, we consider a simple MADE structure with input x and a stack of multiple hidden layers $\mathbf{h}(\mathbf{x})$, where each $\mathbf{h}(\mathbf{x})$

follows:

$$
\begin{aligned}
\mathbf{h}(\mathbf{x}) &= \mathbf{f}\left(\mathbf{b} + \left(\mathbf{W} \odot \mathbf{M}^{\mathbf{W}}\right)\mathbf{x}\right) \\
\boldsymbol{\theta} &= \mathbf{f}\left(\mathbf{c} + \left(\mathbf{V} \odot \mathbf{M}^{\mathbf{V}}\right)\mathbf{h}(\mathbf{x})\right)
\end{aligned}
\tag{B.1}
$$

Here $\boldsymbol{\theta}$ is the output, and $\mathbf{f}$ is the activation function of the hidden layer. In practice, we found Gaussian Error Linear Units (GeLU) [58] works better in our experiments than other activations such as *Sigmoid* and *tanh*. Both $\mathbf{W}$ and $\mathbf{V}$ are weight matrices, with corresponding masks: the hidden mask $\mathbf{M}^{\mathbf{W}}$, and the output mask $\mathbf{M}^{\mathbf{V}}$. Each matrix is multiplied element-wise with each mask.

Suppose $\mathbf{x} \in R^D$, it can be shown that for the input mask:

$$
\mathbf{M}^{\mathbf{W}}_{k,d} = 1_{k \bmod D \leq d} = \begin{cases} 1 & \text{if } k \bmod D \leq d \\ 0 & \text{otherwise} \end{cases} \tag{B.2}
$$

Likewise, suppose $\mathbf{h}(\mathbf{x}) \in R^H$, then for the output mask:

$$
\mathbf{M}^{\mathbf{V}}_{k,d} = 1_{k \bmod D < d} = \begin{cases} 1 & \text{if } k \bmod D < d \\ 0 & \text{otherwise} \end{cases} \tag{B.3}
$$

Then the output $\boldsymbol{\theta}$ satisfies autoregressive structure: for any i, θ_i only depends on $x_{j<i}$. As shown in figure 3.3, the parameter θ_i is used to generate the ith pixel during generation. For example, if the likelihood is a logistic distribution, then $\theta_i = [\mu_i, s_i]$, where μ_i, s_i corresponds to the mean and scale of a logistic distribution.

During generation, at step i we take the previously generated $x_0, x_1, \ldots, x_{i-1}$ and pad the remaining $x_i, \ldots, x_{D-1}$ with zeros. Then we input this vector in the MADE structure so that

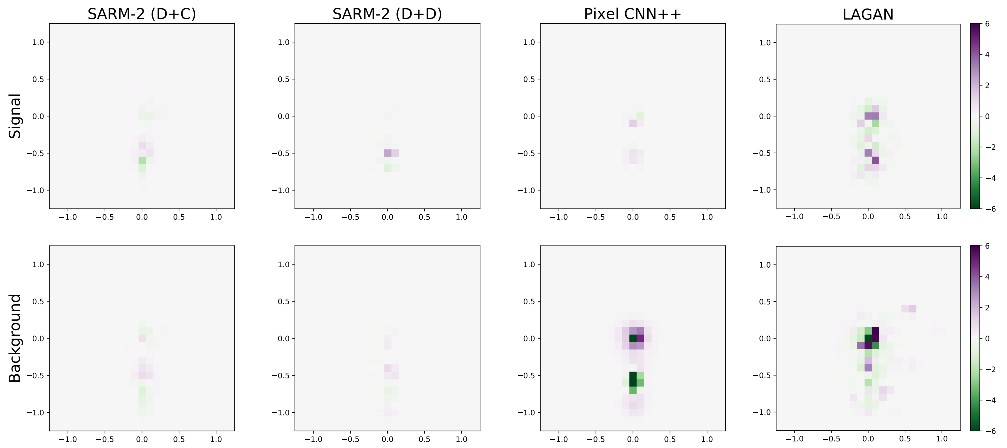


Figure B.2: Error measured by subtracting of the pixel-wise average of the images created by each generative model and the pixel-wise average of the images generated with Pythia. The SARM models have lower error than both Pixel CNN++ and LAGAN with most of the errors are concentrated in the center of the image.

the output θ_i depends only on $x_0, \ldots, x_{i-1}$. Finally, we sample the pixel x_i conditioned on θ_i and repeat this process until every pixel is generated.

B.3 Further Analysis of the Jet Structure Study

Figure B.2 shows the subtraction between the pixel-wise average of the images from each generative model and the pixel-wise average from Pythia. Notice the differences are concentrated in the middle of the images where there are higher value pixels. The images generated by both SARM models have small differences compared to the ones generated by LAGAN for both signal and background and by Pixel CNN++ for background. Also, Pixel CNN++ has higher errors in background images compared to signal images.

Figure B.3 shows the distribution of pixel values across all the generated images. For the signal images, all the models match the Pythia distribution for pixel values below 200 but the models have difficulties at higher values. SARM-2 (D+D) and LAGAN have the closest match at high pixel values while SARM-2 (D+C) and Pixel CNN++ overestimate them. For

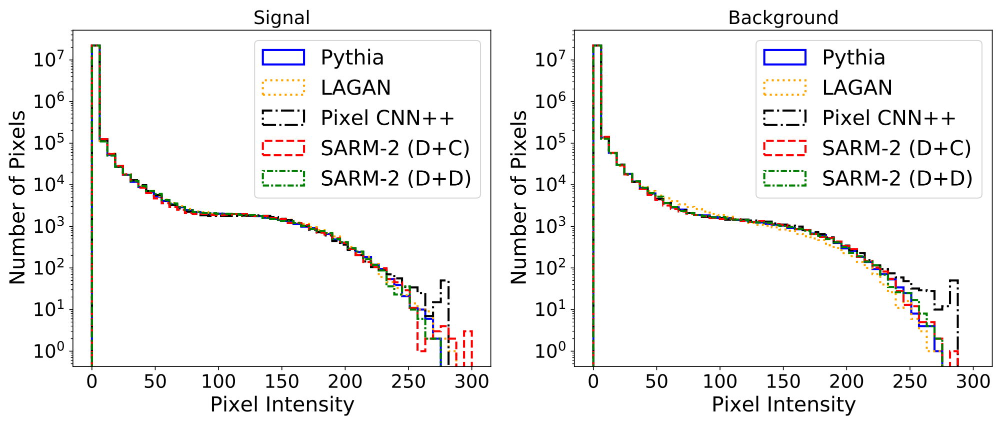


Figure B.3: Distribution of aggregated pixel intensity in the generated images for jet substructure study. Notice most of the differences happen at high pixel values where there are fewer events. LAGAN also has a harder time replicating the distribution of background images across all pixel values compared to the other models.

the background images, most of the models accurately predict low value pixels, but LAGAN slightly overestimates pixels in the range 50 to 100 and underestimates them afterwards. For high pixel values, Pixel CNN++ strongly over-estimates pixels in the range 250-300 while the other models remain reasonably close to Pythia. In both cases the models have difficulties learning the high value pixels, which is expected since there are very few pixels in this range in the Pythia distribution.

B.4 Further Analysis of the Muon Isolation Study

B.4.1 LAGAN

Despite our best efforts, the LAGAN model performed poorly every time it was trained on the muon isolation dataset. As seen in Figures B.4 and B.5 the pixel-wise average image doesn't capture the radial structure present in the dataset and some of the pixels with high values seem to be present in many of the images. This seems to be due to a low amount of variability in the generated images, typical of mode collapse in GANs. This performance is

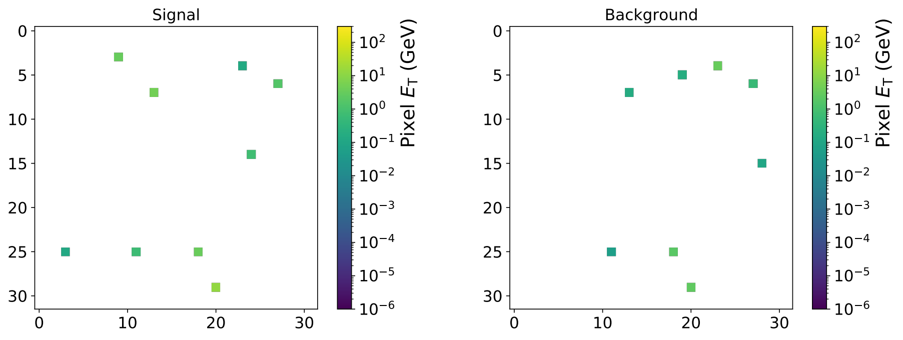


Figure B.4: Typical muon images generated using LAGAN. The figures are plotted in log scale, where the white space represents pixels with value zero.

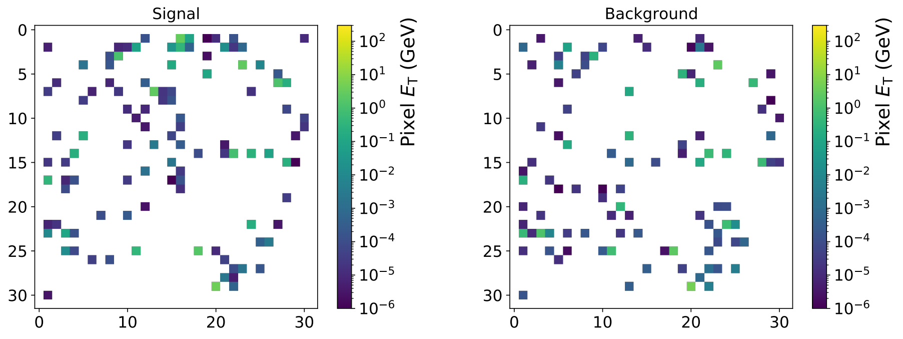


Figure B.5: Pixel-wise average of muon images from LAGAN for signal and background. The average images generated by LAGAN fail to reproduce the radial structure present in the average Monte Carlo images (figure 3.11).

also reflected in the distributions of P_T and mass (figure B.6) and the respective Wasserstein distances which are one order of magnitude worse than the values for the other models (table B.1).

SARM vs Pixel CNN++

Figure B.7 shows the subtraction between the pixel-wise average of the images from each generative model and the pixel-wise average from Pythia in the muon isolation dataset. For the signal data, all models show very small differences, evenly distributed across the radial structure of the images. In particular, Pixel CNN++ is over-representing most of the pixels

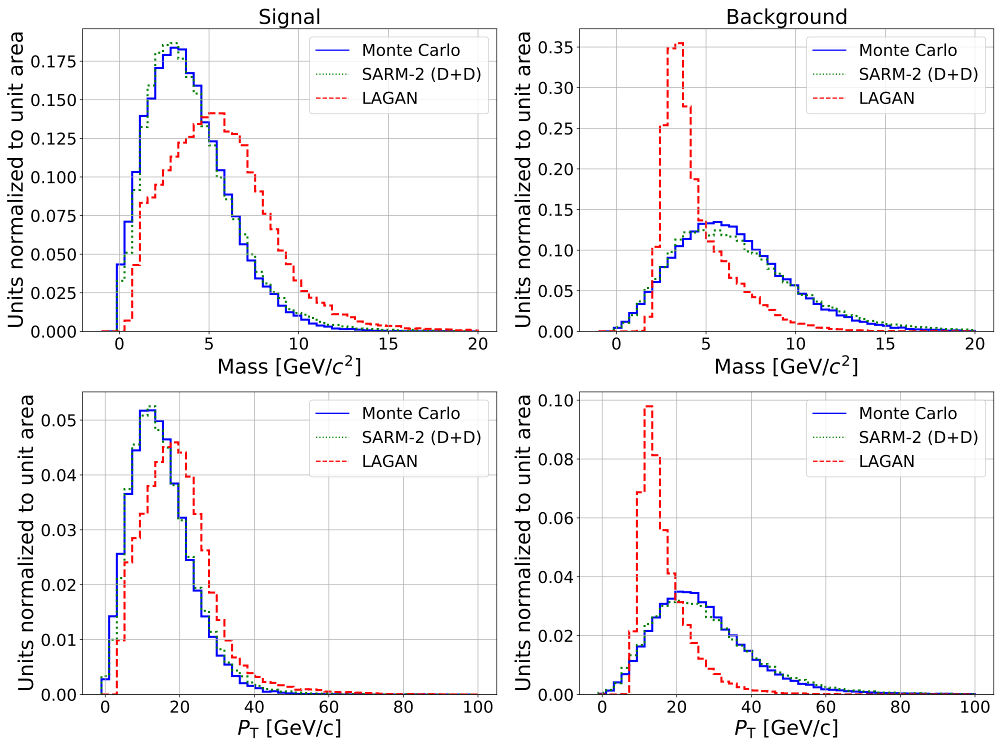


Figure B.6: Comparison of the mass and P_{T} distributions of the images generated by LAGAN, SARM-2 (D+D), and the Monte Carlo simulations for both signal and background muons.

in the artificial checkerboard pattern noted before. For the background data the errors are slightly higher for all models. The SARM models have more difficulties with the pixels in the center and tend to over-represent them while Pixel CNN++ under-represents the center and over-represents the periphery.

Figure B.8 shows the distribution of pixel values across all the generated images. For both signal and background the Pixel CNN++ model is under-representing pixels with high intensity, while the SARM models match the distribution quite well. Like in the jet substructure study, most of the errors correspond to pixels with high intensity values, which is expected since these values are rare in the training data, making it difficult to correctly learn their distribution.

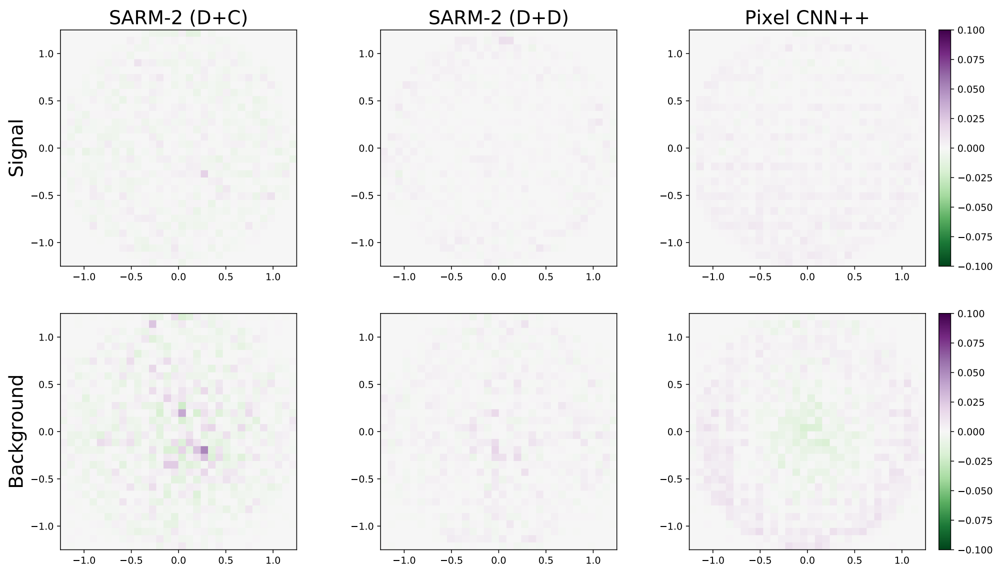


Figure B.7: Subtraction between the pixel-wise average of generated images vs Monte Carlo images. The errors are evenly distributed in the signal images, while they are concentrated in the center for the background images. In the center there is larger number of high intensity pixels.

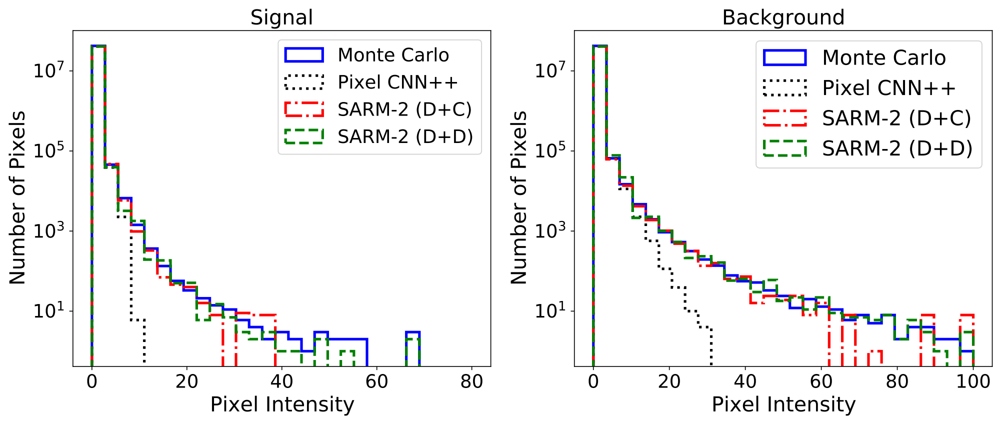


Figure B.8: Distribution of pixel intensity for muon isolation study. Pixel CNN++ under-represents the distribution while the SARM models miss the high pixel values where there are fewer events.

Table B.1: Wasserstein distance of the physical constituents jet P_{T} and mass distributions between the original muon images from the Monte Carlo generator and the images created by the generative models. A small distance signifies a good agreement. SARM-2 (D+D) is the two-stage SARM model with a discrete mixture.

	P_{T}		Mass	
	Signal	Background	Signal	Background
LAGAN	4.81	10.88	1.81	2.17
SARM-2 (D+D)	**0.56**	**0.93**	**0.17**	**0.31**

B.5 Architecture and Hyperparameter Optimization

We performed a search over the architectures of the SARMs including the number of hidden layers structure, the size of the central area for the two-stage approach and the size of the intermediate upsampling layer using SHERPA [59]. We also conducted search of the transformation parameter p with values $[1, 1.1, 1.2, 1.3, 1.5, 2]$ for the D+D models. All models were implemented in Pytorch [101], and were trained for 300 epochs with outward spiral (CCW) order using the Adam optimizer [70] with learning rate 3e-4, decreased by half every 100 epochs and mini-batch size 128.

For the jet substructure study, the best SARM-2 configuration had a center area of side length 3. For the D+D models, we used 5 hidden layers with an upsampling layer of size 10 and found that a power transformation with $p = 1.0$ yields slightly better results. For the D+C models, we found that the model with 3 hidden layers and a mixture of 5 truncated logistic for the C component works well for both signal and background images. In the generation order experiments, similarly we used SARM-1 (D+D) models with 5 hidden layers, an upsampling layer of size 10 and a power transformation with $p = 1.0$, effectively no transformation. And all models are trained with identical settings: learning rate of 3e-4, decreased by half every 100 epochs and mini-batch size 128. For the LAGAN model we used the publicly available version of LAGAN optimized by the original authors.

For the muon isolation study, the best model we found had 5 hidden layers, and a center area of side length 7 for both D+D and D+C models. For the SARM-2 (D+D), we used an upsampling layer of size 10 and found that a power transformation with $p = 1.2$ for signal and $p = 1.3$ for background provided the best results. And for the D+C models, we found again that a mixture of 5 truncated logistic for the C component works well for both signal and background images.

For the classification tasks, we trained five convolutional neural networks with the same structure on each of the datasets. We randomly split the data into a 90% subset for training and a 10% subset for validation. The validation set is used for early-stopping during training to avoid over-fitting. The convolutional neural network model has 2 convolutional blocks, 2 fully connected layers with 100 rectified linear units, and a sigmoid unit at the end to predict the probability of the image being signal. Each convolutional block contains two convolutional layers with 3x3 kernels and 30 filters with rectified linear units followed by a maxpooling layer with 2x2 kernel. All models were trained in PyTorch using the Adam optimizer, with a learning rate of 0.001 and a batch size of 128.

B.6 Complexity Analysis

Next we compare the number of parameters for the different models in table B.2. Note that the original Pixel CNN++ model [116] uses 160 convolutional filters. With all these filters, each forward pass takes more than 1 second on 4 NVIDIA TITANX GPU cards, resulting in a generation speed that is one order of magnitude slower than the traditional Monte Carlo methods, thus defeating the original purpose. Therefore, in our implementation of the Pixel CNN++ model, we limit the number of its filters to 20 to speed up the generation process and reduce the memory requirements.

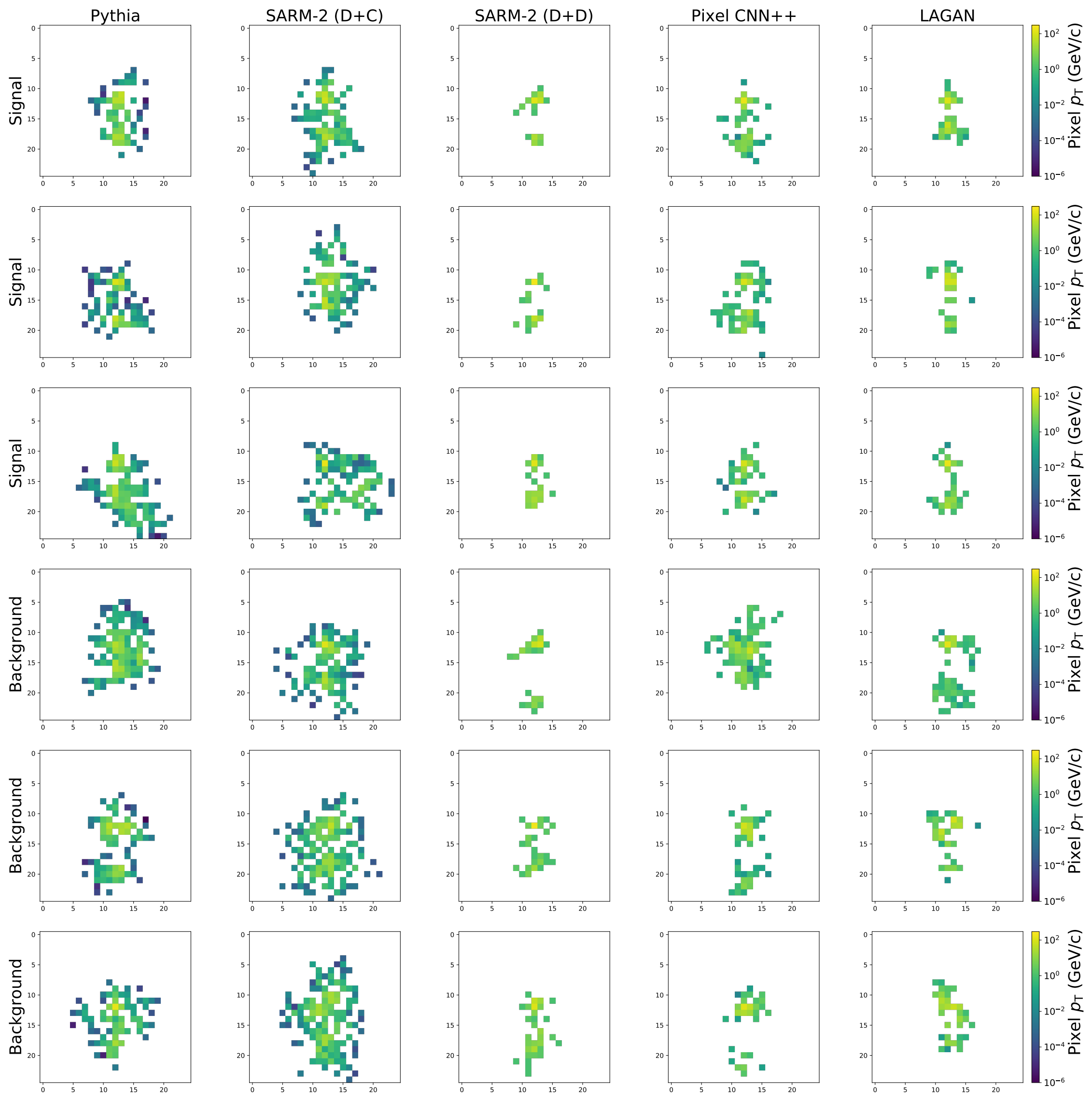


Figure B.9: Additional typical images from the jet substructure study

B.7 Sample Images

In this section, we show more generated images from both the jet substructure study and the muon isolation study in figure B.9 and figure B.10.

Table B.2: Model complexity comparison in terms of the number of parameters.

Model	Num. of Parameters
Pythia [39]	-
Pixel CNN++	0.7M
SARM-2 (D+D)	21M
SARM-2 (D+C)	7M
LAGAN	5M

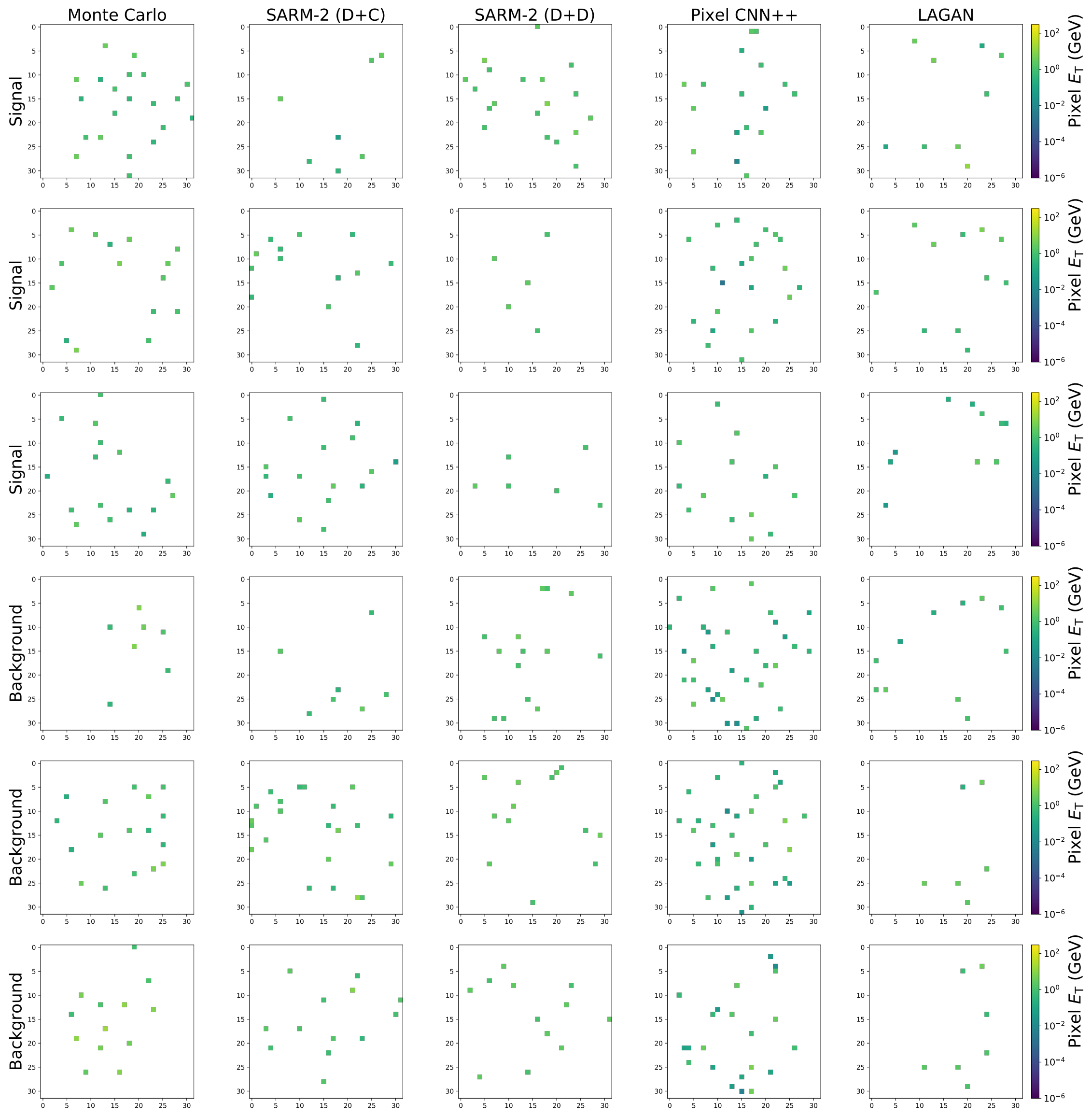


Figure B.10: Additional typical images from the muon isolation study