2014

Bayesian Neural Networks in Data-Intensive High Energy Physics Applications

Michelle Perry

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

BAYESIAN NEURAL NETWORKS IN DATA-INTENSIVE HIGH ENERGY PHYSICS APPLICATIONS

By

MICHELLE PERRY

A Dissertation submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2014

Michelle Perry defended this dissertation on April 1, 2014.
The members of the supervisory committee were:

Anke Meyer-Baese
Professor Co-Directing Dissertation

Harrison Prosper
Professor Co-Directing Dissertation

Jorge Piekarewicz
University Representative

Sachin Shanbhag
Committee Member

Peter Beerli
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

To Anthony and all the teachers and professors that helped me fall in love with science.

ACKNOWLEDGMENTS

I have many thanks to give, and I will give them chronologically. I first have to thank my high school physics instructor, Mr. Carney. I did not have an interest in science until the first day of his physics class. It is because of Mr. Carney that I applied to colleges as a physics major. Next, I absolutely must thank Dr. Sam Tabor. He introduced me to the world of academic research and allowed me to be a major participant in many experiments at FSU and MSU, offering me so many opportunities as an undergraduate that I was unaware were even available. I fell in love with research because of Dr. Tabor and am forever grateful to him. Lastly, I want to thank my Ph.D. co-advisors, Dr. Meyer-Baese and Dr. Prosper. Through both of them, I was able to do exactly the research that I wanted for my dissertation topic. Dr. Meyer-Baese happily took the "co" title to allow me to apply the techniques she studies to high energy physics. Dr. Prosper was eager to set aside large amounts of time to work with a student who was not in his department. His enthusiasm for my project kept me going.

Most importantly, I thank Anthony with all my heart. It was amazing to go through the Ph.D. process with him; I am forever grateful for his support.

TABLE OF CONTENTS

Appendix

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

LIST OF SYMBOLS

The following short list of symbols are used throughout the document. The symbols represent quantities that I tried to use consistently.

GeV	10^6 electron-Volts
$p(A\|B)$	Probability distribution of A given B
$p(A, B)$	Joint probability distribution of A and B
$\mathcal{L}(\omega\|x, t)$	Likelihood of ω given the joint probability of x and t
GHz	10^6 Hertz

ABSTRACT

This dissertation studies a Graphics Processing Unit (GPU) construction of Bayesian neural networks (BNNs) using large training data sets. The goal is to create a program for mapping the parameters of theories of new physics to their predictions. As a target problem, we consider the phenomenological Minimal Supersymmetric Standard Model (pMSSM). The ability to create accurate and smooth mappings would allow for a more robust method of studying the Minimal Supersymmetric Standard Model, which is of great interest at the Large Hadron Collider (LHC) experiments at CERN. A systematic study of the speedup achieved in the GPU application compared to a Central Processing Unit (CPU) implementation is presented. We find a significant speedup and conclude that GPUs are a promising platform for the construction of BNNs.

CHAPTER 1

INTRODUCTION

The field of high energy physics has been pushing the bounds of computational science for decades. The work at the European Organization for Nuclear Research (CERN), which operates the world's largest scientific facility ever created, the Large Hadron Collider (LHC), has necessitated many computational advances. These range from the birth of the World Wide Web to building the world's largest computing grid. The experiments at the LHC generate 15 petabytes of data per year, and storage and analysis of these data is a huge computational endeavor. The purpose of the four experiments at the LHC is to test fundamental theories of matter and their interactions. The work in this dissertation focuses on a computational method to study one class of these theories: supersymmetry. There are many such theories; this work focuses on one, the phenomenological Minimal Supersymmetric Standard Model (pMSSM). I will first motivate the study of the theory itself, followed by a motivation for the computational methods used in this work. The computational work is a graphical processing unit (GPU) construction of Bayesian neural networks (BNNs) using large training data sets. These topics will be discussed in detail in Chapters 2 and 3 of this dissertation.

1.1 High Energy Physics Overview

By the mid-1960's, experimental observations provided evidence that matter was not solely composed of indivisible protons, neutrons, and electrons. In 1964, the theory that protons and neutrons, as well as other particles that were being observed in experiments, were themselves composed of indivisible particles, now known as "quarks", was introduced by Gell-Mann and Zweig independently [5][6]. Within a decade, evidence of the proposed particles was obtained at the SLAC National Accelerator Laboratory [7] at Stanford University. The quark model then evolved over the following ten years and became part of the Standard Model of particle physics.

The Standard Model is a theory of quantum fields and their interactions. Specifically, the electromagnetic, weak and strong nuclear fields and their interactions. In quantum field theory,

particles are excited states of fields. Therefore, the fundamental fields of the model yield particles which can, in principle, be observed to confirm or refute the theory. It was not until the mid-1970's that the physics community began accepting the existence of quarks and the Standard Model. This came about when experimental observations of a new particle, the J/Ψ, at Brookhaven National Laboratory and at SLAC could be readily explained by adding a fourth quark to the Standard Model [7]. Since the debut of the Standard Model, experiments have now confirmed all predicted particles. The last of these, the Higgs boson, discovered in 2012, was finally confirmed in 2013 at the LHC. This was a monumental discovery, not only because the existence of the Higgs boson completes the Standard Model, but also because the Higgs field was the ingredient in the Standard Model that explained why the fundamental particles have mass.

The goal of this dissertation is the development of a computational method not for the study of the Standard Model, but for a theory that goes beyond it. In the next section, I briefly explain why physicists are motivated to go beyond the Standard Model, which is, after all, the most successful theory ever created.

1.1.1 Drawbacks of the Standard Model

The Standard Model is not a fundamental theory, as it does not unify all of the fundamental forces. The list of fields incorporated in the Standard Model include all fields except for gravity. Physicists have long believed that the universe operates on a single set of rules, or a fundamental theory. The success of the Standard Model indicates that it is perhaps a subset of a larger theory, or the low-energy limit of a more inclusive theory. Moreover, there are observations that the theory does not predict. For example, astrophysical observations indicate that "visible" matter, i.e. the matter described by the Standard Model, makes up only 20% of the matter in the universe. The other 80% of matter is labelled "dark matter" by physicists and is just a term to describe the unknown. In addition, there is an extreme imbalance between matter and anti-matter in the universe, favoring matter, and there is no explanation for this. Lastly, one class of particles, neutrinos, are assumed by the Standard Model to be massless. Neutrinos do, in fact, have a small mass, as discovered by the Super-Kamiokande experiment in Japan [8]. The discovery of neutrino mass solved the issue of solar neutrinos [9]. The solar neutrino problem was a major discrepancy between the observed rate of neutrinos hitting the Earth and the rate predicted. In fact, only about one-third of the expected number appeared to be passing through the Earth.

With massive neutrinos, neutrino oscillations are possible and would account for this discrepancy. The Standard Model, while very accurate at predicting many features of the universe, does not describe all observations and leaves many questions unanswered. Therefore, physicists are searching for theories that describe the features of the Standard Model while also answering many of the questions that the Standard Model can not answer. In fact, the LHC was designed not only to search for the Higgs boson, but also to look for evidence of any signs of new physics not predicted by the Standard Model.

1.2 Supersymmetry

There exist many theories of physics beyond the Standard Model. This dissertation focuses on one class of these theories: supersymmetry [1]. Supersymmetry is by far the most studied of the theories for physics beyond the Standard Model because it arises from a basic set of principles, solving many of the questions of why certain features of the Standard Model exist. Another appealing feature of supersymmetry came about with the discovery of the Higgs boson. While the Standard Model predicts the existence of the Higgs boson, it does not predict the mass of the Higgs boson. Supersymmetry, on the other hand, favors a Higgs boson with a mass on the order of the measured mass. Another theory of physics beyond the Standard Model, string theory, actually requires the existence of supersymmetry as well as extra dimensions [10]. In supersymmetry, each fundamental particle in the Standard Model also has a corresponding supersymmetric particle. See Table 1.1 for a list of the Standard Model particles and their supersymmetric partners. The Standard Model particles are divided into two classes: fermions and bosons. The two types of particles act very differently: at most one fermion can be in a given state, while bosons can bunch together in the same state. This fundamental property is intimately connected with a property called "spin". Fermions have half unit spins and bosons have integer spins. Supersymmetry links together these two very different types of particle in that the spin of the corresponding supersymmetric particles differ by a half unit.

There has been a considerable effort to search for evidence of supersymmetry at the LHC, so far without success. To understand the recent findings, I will outline the specific supersymmetric theory of interest in this work and how it is currently being studied at the LHC. This theory is the Minimal Supersymmetric Standard Model (MSSM) and evidence is being sought in the Compact

Table 1.1: A list of Standard Model particles and their associated supersymmetric particles. This table serves as a reference for particles used in this work, please see [1] for a more in-depth look at the Standard Model and SUSY particles.

Standard Model	Supersymmetry
γ, Z^0, h^0, H^0	$\tilde{\chi}_1^0$, $\tilde{\chi}_2^0$, $\tilde{\chi}_3^0$, $\tilde{\chi}_4^0$
W^+, H^+	$\tilde{\chi}_1^+$, $\tilde{\chi}_2^+$
e^-, ν_e, μ^-, ν_μ, ν_τ	$\tilde{e}_{\bar{R}}$, $\tilde{e}_{\bar{L}}$, $\tilde{\nu}_e$, $\tilde{\mu}_{\bar{L}}$, $\tilde{\nu}_\mu$, $\tilde{\nu}_\tau$
τ^-	$\tilde{\tau}_1$, $\tilde{\tau}_2$
u, d, s, c	$\tilde{u}_R$, $\tilde{u}_L$, $\tilde{d}_R$, $\tilde{d}_L$, $\tilde{s}_R$, $\tilde{s}_L$, $\tilde{c}_R$, $\tilde{c}_L$
b	$\tilde{b}_1$, $\tilde{b}_2$
t	$\tilde{t}_1$, $\tilde{t}_2$

Muon Solenoid (CMS) and ATLAS collaborations at CERN [11][12]. The appealing feature of this theory is that it is a minimal extension of the Standard Model while still explaining some otherwise mysterious features of nature. The MSSM provides solutions to many problems, including dark matter, grand unification of three of the four fundamental forces, and the Higgs mass issue outlined above. To produce evidence of the MSSM, observations of the supersymmetric particles are sought. However, the masses of the supersymmetric particles ("sparticles") are free parameters in the theory, that is, they are unknown values. The masses of the particles dictate in which reactions they will be produced, as well as at what energies they will be produced. Not knowing the masses make the particles very difficult to search for. In fact, there are 120 free parameters in the MSSM, making the search extremely difficult, if not impossible at an all-encompassing level. The absence of sparticle discovery thus far indicates that, if they exist, they must be 100 to 1000 times heavier than the proton, and therefore a lot of energy will be needed to produce them experimentally.

In order to search for evidence of the MSSM at the LHC, physicists have looked at highly constrained versions of the MSSM. This means that most of the free parameters in the model are set to an assumed value, and the rest are allowed to vary. One such model is the Constrained Minimal Supersymmetric Standard Model (CMSSM) [13]. The CMSSM reduces the number of parameters by making assumptions about the parameters at the Grand Unification (GUT) scale (10^{16} GeV) [14]. At this scale, many masses are assumed to be the same. This reduces the 120 free parameters to just four, plus an additional sign ($+/-$) term. With these four parameters, physicists have been able to look for evidence in support of the CMSSM using direct methods.

Experiments at the LHC have almost definitively ruled out the CMSSM [11]. The question, then, is what this says about the MSSM. It turns out that refuting the CMSSM says very little about the validity of the MSSM, and that the assumptions made in the CMSSM eliminate very little of the feasible parameter space. Even supposing the MSSM to be a good approximation to the new Standard Model, the assumptions made about the particles at the GUT scale may be invalid; in reality nothing is known about the GUT scale, including the hierarchy of particle masses.

1.2.1 The Phenomenological Minimal Supersymmetric Standard Model

While studying the entire parameter space remains computationally infeasible, a less-constrained model whose assumptions are founded more upon known knowledge can be used. This is the phenomenological Minimal Supersymmetric Standard Model (pMSSM) [15]. The pMSSM defines the variables at the much lower SUSY scale (10^3 - 10^4 GeV) as opposed to the GUT scale. The major benefit of this is that assumptions are more securely based on observations. The pMSSM is a 19-dimensional proxy for the MSSM. In fact, the pMSSM encapsulates most of the physics of the MSSM. This means that a study of the pMSSM can say much more about the MSSM than any of the more highly-constrained models. The drawback is that 19 free parameters are still very difficult to navigate in searching for evidence of supersymmetry.

Computational Tasks. In order to study the pMSSM parameter space and compare predictions of the theory to experiment, the current approach is to sample points in the space using Markov Chain Monte Carlo [11]. In the CMS collaboration's most recent study of the theory, 20 million points were sampled from the parameter space, of which 7205 were randomly taken from the 20 million points for comparing the theory to predictions [11]. The theory's experimental predictions must be computed for each of the 7205 points in the subspace. This is a computational burden for each desired prediction of the theory.

A better approach would be to construct a function that maps the pMSSM parameters to the predictions. This would require a single difficult computation to construct the function, then the function could be used to represent the space in future studies. This is the goal of this dissertation. The function can be represented by a *neural network*: a function that can approximate any smooth mapping. The parameters of the neural network are found using a Markov Chain Monte Carlo method. This work explores a Bayesian implementation of neural networks, called Bayesian neural

networks. The computational challenges of constructing the neural network function with a large amount of data in a high dimensional space are discussed in Chapter 2.

1.3 Bayesian Neural Networks in High Energy Physics

Bayesian neural networks have been successfully used in the field of high energy physics. The first use was in the discovery of particles collisions in which top quarks are produced without their associated antiparticle, the so-called single top quark production, at Fermilab, which was announced in 2009 [16] [17]. Bayesian neural networks, along with other machine learning methods, such as decision trees, were instrumental in the single top quark discovery. It took a large team of physicists thirteen years to find conclusive evidence of the single top quark production at the D0 experiment at Fermilab. The group estimates that the discovery would have taken approximately twice as long using solely traditional methods of searching for a conclusive signal in the data. This means they could still be searching today!

1.3.1 Current Work

There is no doubt that Bayesian neural networks have proven to be useful in the field of high energy physics. The issue is the massive amounts of data necessary for almost all studies, causing the construction of these neural network functions to be incredibly time consuming. The construction of the functions involves many iterations of computations over all the points in the space. The use of BNNs would surely become more routine if the training process was quicker. As we shall show, the training of BNNs in data-intensive cases such as this lends itself very well to general purpose graphical processing unit (GPGPU) programming. This dissertation reports work to implement and study a GPGPU version of BNN training for use in data-intensive applications.

CHAPTER 2

BAYESIAN NEURAL NETWORKS

For the high energy physics applications of interest, there exist training data that provide the inputs with their target values. This indicates the use of supervised learning methods in order to generate a functional mapping from the inputs to the continuous-valued outputs (i.e. regression) and for classification. The most straightforward models for regression and classification for supervised learning utilize a linear combination of non-linear, fixed basis functions,

$$f(\boldsymbol{x}, \boldsymbol{\omega}) = h(\sum_{j=1}^{M} \omega_j \phi_j(x)), \tag{2.1}$$

where M is the total number of model parameters, w, and $\phi(\mathbf{x})$ are the basis functions. The quantity $\mathbf{h}$ is a non-linear activation function. Basis functions can take many forms. Some examples are polynomials, Gaussians and sigmoid functions. The maximum likelihood method can be applied to determine the model parameters, however, these models are limited by the dimensionality of the application's training data [18]. In our applications of interest, we typically have a large amount of training data and many inputs, preventing the use of this method. One solution to the "curse of dimensionality" is to allow the basis functions to be adaptive, altering the parameters of the basis functions during the supervised learning, or training.

The feed-forward neural network has been shown to be a successful model of supervised learning using adaptive basis functions.[18]. The result is a non-convex function for the model parameters, which is computationally more intensive during training, but results in quick evaluation for new data once training is complete. The work in this dissertation utilizes neural networks for regression due to the high dimensionality present in the high energy physics applications of interest, as discussed in Chapter 1.

2.1 Neural Networks

Extending from the linear model, the basis functions in a neural network (NN) model take the form of Eq. 2.1, i.e., each basis function is a non-linear function of a linear combination of the

inputs. The coefficients in the linear combination are adaptive. To keep track of all the parameters, I will divide them up into classes, $\omega = \{a, b, c, d\}$. Consider, first, the linear combination of inputs,

$$y(\boldsymbol{x}, d)_j = c_j \sum_{i=1}^{I} d_{ji} x_i. \tag{2.2}$$

Then, an activation function is applied to y_j:

$$z(\boldsymbol{x}, c, d)_j = h(y_j) = h(c_j \sum_{i=1}^{I} d_{ji} x_i). \tag{2.3}$$

Lastly, the linear combination of the new variable, z_j:

$$f(\boldsymbol{x}, \boldsymbol{\omega})_k = a + \sum_{j=1}^{H} b_j z_j = a + \sum_{j=1}^{H} b_j h(c_j \sum_{i=1}^{I} d_{ji} x_i). \tag{2.4}$$

Choosing tanh as the activation function gives the standard form of a neural network,

$$f(\boldsymbol{x}, \boldsymbol{\omega}) = a + \sum_{j=1}^{H} b_j \tanh(c_j + \sum_{i=1}^{I} d_{ji}\, x_i), \tag{2.5}$$

where ω are the neural network parameters $\{a, b, c, d\}$, H is the number of hidden nodes in the network, and I is the number of inputs. This is a "feed-forward" neural network, that is, the output is a deterministic function of the inputs. This is best understood visually, as in Figure 2.1.

Feed-forward neural networks have been shown to be "universal approximators" that can model *any* smooth mapping of the form $f : \mathbb{R}^n \to \mathbb{R}^m$, with $m < n$ provided there are enough hidden nodes in the network [19]. In this work, we focus on approximations to functions with $m = 1$, based on the single hidden layer NN model. The traditional method of training neural networks is to find the optimum set of parameters, ω^*, that map the input variables to the output. Test data are used to evaluate the success of a mapping. Given a training set of N input vectors $\{x_n\}$ and corresponding targets $\{t_n\}$, the goal is to find the maximum likelihood values of the parameters given the training data. The likelihood function is given by

$$\mathcal{L}(\omega|x, t) = p(x, t|\omega) = \prod_{i=1}^{N} p(x_i, t_i|\omega), \tag{2.6}$$

in which one typically assumes a Gaussian model

$$p(x_i, t_i|\omega) = e^{-\frac{1}{2\sigma^2}(t_i - f(x_i, \omega))^2}. \tag{2.7}$$

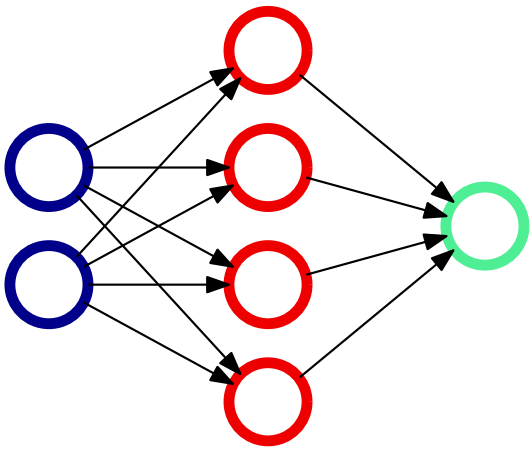


Figure 2.1: Graphical representation of a neural network with two inputs, depicted by the blue circles, four hidden nodes, which are the red circles, which map to one output, given by the green circle. Each arrow represents the connection of the output of one node to the input of another. The connections represent the neural network weights, plus a "bias" for the output.

In practice, it is common to deal with the negative log-likelihood, or

$$-\ln \mathcal{L}(\omega|x,t) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(f(x_n,\omega)-t_n)^2. \tag{2.8}$$

Finding the maximum likelihood values is equivalent to minimizing Eq. 2.8. There are many different successful methods of finding an ω^* that minimizes Eq. 2.8, the simplest being gradient descent. More efficient algorithms include conjugate gradient and quasi-Newton methods [20]. Because this work focuses on Bayesian, not maximum likelihood approaches of determining the network parameters, I refer you to other resources for the standard training methods [18]. One issue of standard methods of training neural networks is that no convenient measure of modeling accuracy is available. Another problem lies in setting up a network when the problem has unknown complexity [21]. Both issues can be addressed by a Bayesian approach for training neural networks [21].

In the Bayesian approach to neural networks, the goal is to construct a posterior probability of the neural network parameters given the training data, which assigns a probability density to each point in the neural network parameter space. The posterior probability is given by:

$$p(\omega|x,t) = \frac{p(x,t|\omega)p(\omega)}{p(x,t)}. \tag{2.9}$$

In addition to $p(x,t|\omega)$, which is the likelihood function given in Eq. 2.6, we also need the prior density $p(\omega)$. Once a posterior density is obtained, an estimate of the mapping is given by

$$\bar{f}(x'|x,t) = \int f(x',\omega)p(\omega|x,t)d\omega. \tag{2.10}$$

In practice, given the intractability of Eq. 2.10, the latter must be approximated using Monte Carlo methods to evaluate the integral. The parameter space of the neural network function is very complex and high dimensional. Because $\tanh(-x) = -\tanh(x)$, equivalent solutions can be generated by a change of sign on all weights exiting a hidden unit. For H hidden nodes, there exist H sign-flipping equivalent solutions, so for each weight vector ω, there exist 2^H other equivalent weight vectors. In addition, interchanging of the weights associated with one node to another produces an equivalent solution. In that case, there lies $H!$ equivalent solutions. Therefore, there are $H!2^H$ equivalent solutions, giving a picture of the complicated nature of the parameter space [18].

2.2 Markov Chain Monte Carlo

Using Monte Carlo integration, Eq. 2.10 is approximated by an average over a sampling of points in the parameter space according to $p(\omega|x,t)$ [22],

$$\bar{f}(x') \approx \frac{1}{K}\sum_{k=1}^{K} f(x',\omega_k). \tag{2.11}$$

The approximation converges to the exact integral as $K \to \infty$. Computationally, the task is to sample the network parameter space from $p(\omega|x,t)$ so the approximation in Eq. 2.11 holds true. While it may be infeasible to sample K independent ω_k parameters, Eq. 2.11 holds true if the ω_K are dependent, without a large covariance between parameter states, and the Markov chain has reached the equilibrium distribution, $p(\omega|x,t)$.

These dependent ω_ks are generated using a Markov Chain. Markov Chain Monte Carlo (MCMC) was first developed by Metropolis, *et al.* [23] to describe the properties of interacting molecules[1]. Owing to the high dimensionality of the problems, random sampling leads to too few meaningful samples. To obtain more meaningful points, sampling from a probability distribution, in Metropolis' case of molecules, the canonical ensemble, is used [23]. The method was extended to a general case by Hastings about 20 years later to produce the well known Metropolis-Hastings algorithm, which allows the use of non-symmetric proposal densities, and shown to be useful in higher-dimensional applications [24].

The Metropolis-Hastings algorithm proceeds as follows [20]:

1. choose starting state ω^0
2. generate a candidate state ω' from a proposal distribution $Q(\omega'|\omega^{(t-1)})$
3. calculate acceptance probability $r = \frac{p(\omega'|x)/Q(\omega'|\omega^{(t-1)}))}{p(\omega^{(t-1)}|x)/Q(\omega^{(t-1)}|\omega'))}$
4. accept ω' as new state with probability $\min(r, 1)$, otherwise keep $\omega^{(t-1)}$ as new point
5. repeat steps 2-4 T times, keeping every m^{th} ω after a burn-in period in which the chain converges to the desired distribution.

In the case of a complex, high dimensional distribution, the standard deviation of the proposal distribution, $Q(\omega'|\omega^{(t-1)})$ must be small in order to produce a reasonably high acceptance ratio of new states. Therefore, there is a high correlation between states, thus requiring a large number of steps to move through the parameter space. This problem is exacerbated by the random walk nature of this algorithm. Because of these drawbacks, the Metropolis-Hastings algorithm converges much too slowly for the Bayesian treatment of neural networks.

In order to remove the necessity to choose a proposal distribution with a small standard deviation, the Gibbs sampling method proposes a new point one parameter at a time. The Gibbs sampling method advances as follows:

[1]Metropolis demonstrated the validity of his algorithm by analyzing the ergodicity and reversibility of the algorithm. In MCMC, a chain is *ergodic*, if it will converge to the desired distribution from any initial state. The definition of an ergodic Markov chain is that there is a nonzero probability to transition from any state in the space to any other state in the space. In addition, if a transition to a new state is *reversible*, there is always an equilibrium distribution to which the chain will converge. Reversibility ensures the preservation of the desired equilibrium distribution of the system upon transitions between the current state and the proposed state. Reversibility states that, with an equilibrium distribution P and transition matrix T, a Markov chain is in detailed balance if for all pairs i and j, $P_i T_{ij} = P_j T_{ji}$. This is known as the detailed balance condition. Ergodicity holds for all MCMC methods discussed here.

1. choose starting state ω^0
2. for $j = 0 : J$ elements in ω: sample ω_j^t from conditional distribution

$$p(\omega_j|\omega_1^t, ..., \omega_{j-1}^t, \omega_{j+1}^{t-1}, ..., \omega_n^{t-1}) \tag{2.12}$$

3. repeat step 2 T times, keeping every m^{th} ω after a burn-in period in which the chain converges to the desired distribution.

Unfortunately, for a BNN, the conditional distribution given in step 2 can be very messy and difficult to construct. The computationally demanding construction of the conditional distribution combined with the penalty of random walks negates Gibbs sampling as a feasible method for BNNs [20]. A more advanced MCMC method is necessary for Bayesian Neural Networks, specifically one that avoids random walks.

2.2.1 Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo (HMC) method was first proposed by Duane, *et al.* as the Hybrid Monte Carlo method, created for their application in lattice field theory calculations [25], and extended to the training of neural networks by Radford Neal [26]. The HMC method avoids random walks by navigating a deterministic path through the parameter space given by Hamiltonian dynamics in order to propose a new point, ω', in the space. The most important result of Hamiltonian dynamics for use in MCMC is that ergodicity and reversibility hold true *exactly*, even when Hamiltonian dynamics is approximated, as will be done in our algorithm.

In this formalization of the HMC method, the space for which we are defining the equations of motion is the parameter space of the neural network. Therefore, the "position" is a point in the n-dimensional parameter space. The neural network formalization given in Eq 2.5 gives a dimensionality of $n = 1 + H(2 + I)$, where H is the number of hidden nodes and I is the number of inputs in the NN. The state of a system in Hamiltonian dynamics is given by the position, ω, and the momentum, p. The quantity that describes the state is called the Hamiltonian and given by a function, $\mathcal{H}(\omega, p)$. The Hamiltonian represents the total energy of the system. Hamilton's equations are differential equations that determine how ω and p change over time

$$\frac{d\omega_i}{dt} = \frac{\partial \mathcal{H}}{\partial p_i} \tag{2.13}$$

$$\frac{dp_i}{dt} = -\frac{\partial \mathcal{H}}{\partial \omega_i}, \tag{2.14}$$

where, for HMC, we will take the classical interpretation of $\mathcal{H}(\omega, p)$.

$$\mathcal{H}(\omega, p) = U(\omega) + K(p), \tag{2.15}$$

where $U(\omega)$ is interpreted as the potential energy of the system and $K(p)$ is the kinetic energy of the system. The kinetic energy is given by the classical definition,

$$K(p) = \frac{1}{2} p^T M^{-1} p. \tag{2.16}$$

The momentum vector p is an n-dimensional vector independent of ω, and M is a diagonal "mass" matrix, where we will use scalar values for each spatial dimension. This simplifies the kinetic energy expression in Eq. 2.16 to

$$K(p) = \sum_{i=1}^{n} \frac{p_i^2}{2m_i}. \tag{2.17}$$

The simplest case for the mass matrix is the identity matrix, $M = I$. In the metaphor of a mechanical system, the energy of the system is assumed to be distributed according to a canonical distribution, where the probability density is given by:

$$P(\omega, p) \propto e^{-\mathcal{H}(\omega,p)}, \tag{2.18}$$

where the units are chosen to make the Hamiltonian dimensionless. This is known as the joint canonical distribution. The joint canonical distribution must remain invariant throughout the traversal through the parameter space. Note that this distribution factorizes as follows

$$P(\omega, p) \propto e^{-U(\omega)} e^{-K(p)}. \tag{2.19}$$

In the case of NNs, the goal is to obtain the posterior distribution of ω. Equation 2.19 shows that both the momentum and the posterior distribution will have a canonical distribution. For the posterior distribution,

$$p(\omega|x, t) = e^{-U(\omega)}. \tag{2.20}$$

Comparing Eq. 2.20 with Eq. 2.19 leads to the potential function $U(\omega)$ given by

$$U(\omega) = -\log\left(\mathcal{L}(\omega|x, t)p(\omega)\right). \tag{2.21}$$

In order to use the HMC method, it is necessary to construct a finite difference approximation to Hamilton's equations, Eq. 2.13. In discretizing Eq. 2.13 for use in a MCMC scheme, a method

must be chosen that preserves the volume of the (ω, p) space. This is equivalent to the energy of the system, given by the Hamiltonian, $\mathcal{H}$, being conserved. The simplest discretization is Euler's method,

$$p_i(t+\epsilon) \quad = p_i(t) + \epsilon \tfrac{dp_i}{dt}, \quad = p_i(t) - \epsilon \frac{\partial U}{\partial \omega_i} \omega(t), \tag{2.22}$$

$$\omega_i(t+\epsilon) \quad = \omega_i(t) + \epsilon \tfrac{d\omega_i}{dt}, \quad = \omega_i(t) + \epsilon \frac{p_i(t)}{m_i}. \tag{2.23}$$

This method is not time reversible and diverges away from the correct solution with increasing time [20]. Therefore, it will not work for MCMC. A simple modification, however, exactly preserves the phase space volume. In the modified Euler's method, the updated momentum term is immediately used to calculate the new position term:

$$p_i(t+\epsilon) = p_i(t) - \epsilon \frac{\partial U}{\partial \omega_i} \omega(t), \tag{2.24}$$

$$\omega_i(t+\epsilon) = \omega_i(t) + \epsilon \frac{p_i(t+\epsilon)}{m_i}. \tag{2.25}$$

The modified Euler method still lacks the symmetry required for a reversible algorithm, however, this method operates much better than the Euler method because the updates are shear transformations in the (ω, p) parameter space. A shear transformation is where one set of points remains fixed while all other points shift a specified amount parallel to the unmoving set. In this case, the point $(\omega(t), p(t)) \to (\omega(t), p(t+\epsilon))$, then $(\omega(t), p(t+\epsilon)) \to (\omega(t+\epsilon), p(t+\epsilon))$. A shear transformation preserves *volume*, implying, in the case of the modified Euler's method, that there will be no divergence of the algorithm. A divergence would indicate a change in the volume of the (ω, p) space.

Another simple modification introduces reversibility to the modified Euler's method through a symmetric update. The method is called the Leapfrog method [18]. The approach is to take a half step in momentum, followed by a full step in position, followed again by a half step in momentum. The symmetry is evident, leading to a reversible scheme:

$$p_i(t + \frac{\epsilon}{2}) = p_i(t) - \frac{\epsilon}{2} \frac{\partial U}{\partial \omega_i} \omega(t), \tag{2.26}$$

$$\omega_i(t+\epsilon) = \omega_i(t) + \epsilon \frac{p_i(t+\frac{\epsilon}{2})}{m_i}, \tag{2.27}$$

$$p_i(t+\epsilon) = p_i(t + \frac{\epsilon}{2}) - \frac{\epsilon}{2} \frac{\partial U}{\partial \omega_i} \omega(t+\epsilon). \tag{2.28}$$

In addition, it is evident that each update is also a shear transformation, so volume is preserved here as well. Maintaining the reversibility and volume preservation, any number of full steps can be taken in an iteration of this method, while updating both the position and momentum on all intermediate steps. The leapfrog method is the discretization method used in this work due to the reversibility and volume preservation of the scheme. The reversibility of the method guarantees conservation of energy, or the Hamiltonian. This also guarantees zero error in the calculation of the leapfrog step. If a method is reversible, meaning taking a step backwards will place ω exactly in its previous position, the error must be zero. Though the leapfrog method introduces no error, approximations in the scheme will introduce error. With the discretization of Hamilton's equations, we can now describe the HMC algorithm.

The following outlines one iteration of the HMC algorithm.

1. randomly generate new momentum variables from their canonical distribution, which is a Gaussian distribution.
2. do a half step in momentum $p_i(t + \frac{\epsilon}{2}) = p_i(t) - \frac{\epsilon}{2}\frac{\partial U}{\partial \omega_i}\omega(t)$.
3. do a full step in position, then a full step in momentum (except for the last step), L times
$$\omega_i(t + \epsilon) = \omega_i(t) + \epsilon\frac{p_i(t + \frac{\epsilon}{2})}{m_i},$$
$$p_i(t + \epsilon) = p_i(t + \epsilon) - \epsilon\frac{\partial U}{\partial \omega_i}\omega(t + \epsilon).$$
4. do a half step in momentum $p_i(t + \epsilon) = p_i(t + \frac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial U}{\partial \omega_i}\omega(t + \epsilon)$.
5. negate the momentum, $p_i = -p_i$ in order to preserve symmetry.
6. accept the new point (ω^*, p^*) with a probability $\min[1, e^{-\mathcal{H}(\omega^*,p^*)+\mathcal{H}(\omega,p)}]$.

Perhaps the easiest way to visualize the algorithm is through pseudo code. See Listing 2.1 for the pseudo code and Table 2.1 for a description of the variables.

Listing 2.1: Pseudo code of the Hamiltonian Monte Carlo method. Looping over vector elements is implied in mathematical operations. The `HMC_iteration()` function is one HMC iteration. Its argument is the current point in the parameter space and it returns the new point in the parameter space, which can be the same as the input point. See Table 2.1 for a description of the variables.

```
HMC_iteration(w0) {
  w = w0;
  p = norm_dist();
```

```
  p = p - 0.5*eps*delU(w); // half step in momentum
  for(int i=0; i<L; i++){  // do L steps
    w = w + eps * p; //step in q
    //step in p, except at last step
    if(i != L-1)
      p = p - eps * delU(w);
  }
  p = p - 0.5*eps*delU(w); // half step in momentum
  // calculate Hamiltonian for old and new points
  H0 = U(w0) + K(p0);
  H = U(w) + K(w);
  //decide which point to keep
  if(rand < exp(H - H0)){
    return w; //accept
  } else {
    return w0; //reject
  }
}
```

Table 2.1: Definition of variables used in the HMC algorithm pseudocode in Listing 2.1

`w`	"position" in the parameter space, given by ω.
`p`	momentum vector, one element for each dimension of ω.
`eps`	step size. This can be a scalar or a vector.
`U(w)`	the potential energy function, $U(\omega)$. See Eq. 2.8 for BNN $U(\omega)$.
`delU(w)`	gradient of the potential energy function.
`H`	Hamiltonian given by $\mathcal{H} = U(\omega) + K(p)$. See Eq. 2.13.

The HMC model parameters are the number of leapfrog steps, L, and the step size, ϵ. Tuning these parameters is important in ensuring that the algorithmtraverses the space efficiently. The "acceptance rate", the percentage of new accepted points, is the typical indicator of optimal tuning. For the HMC algorithm, an acceptance rate of 65% for $L > 1$ has been shown to be optimal [26]. The tuning of L and ϵ is typically the most challenging portion of the algorithm. One approach is to dynamically change the variables each iteration depending on the current acceptance rate.

2.3 Bayesian Neural Network Algorithm

The HMC pseudo code in Listing 2.1 outlines the algorithm to construct a BNN where ω are the neural network parameters and the potential energy, $U(\omega)$, is defined as the negative log likelihood

of the neural network model, $-\log\left(\mathcal{L}(\omega|x,t)p(\omega)\right)$, given in Eq. 2.8.

To understand the computational complexity of the BNN training algorithm, the evaluation of $U(\omega)$ must be examined. The negative log likelihood function depends on the evaluation of the network for the entire set of training data, as seen in Listing 2.2. In the case that the number of training events, N is large, the evaluation of $U(\omega)$ becomes the most computationally intensive portion of the algorithm. The computation time of $U(\omega)$ is so significant that the computational complexity of the algorithm can be analyzed in terms of the number of $U(\omega)$ evaluations.

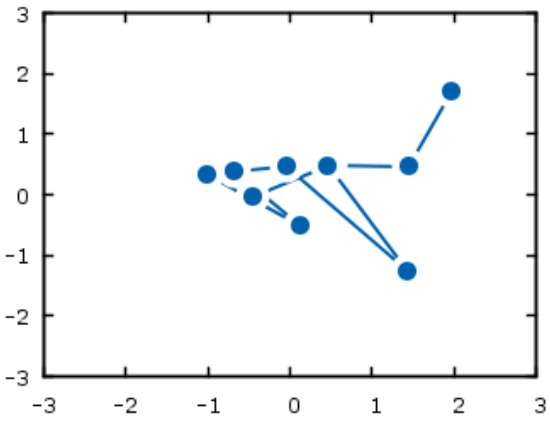
Listing 2.2: Evaluation of $U(\omega)$ for a BNN, where $U(\omega) = -\log\left(\mathcal{L}(\omega|x,t)p(\omega)\right) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(f(x_n,w)-t_n)^2$. In the case where N is large, this computation becomes the most time consuming portion of the algorithm.

```
inline HMC_type BNN_regression::U(std::vector<HMC_type> &w) {
  HMC_type d = 0;
  /* Loop over all N training events */
  for(int th=0;th<N;th++){
    HMC_type f = q[0];
    for(int j=0;j<H;j++){
      HMC_type inSum = 0.0;
      for(int i=0;i<I;i++){
        inSum += w[2*H+1+I*j+i]*x[th*I+i];
      }
      f+=w[j+1]*tanh(w[H+1+j]+inSum);
    }
    d += (t[th]-f)*(t[th]-f);
  }
  return sum/(2*sig*sig) + LnPrior(w);
}
```

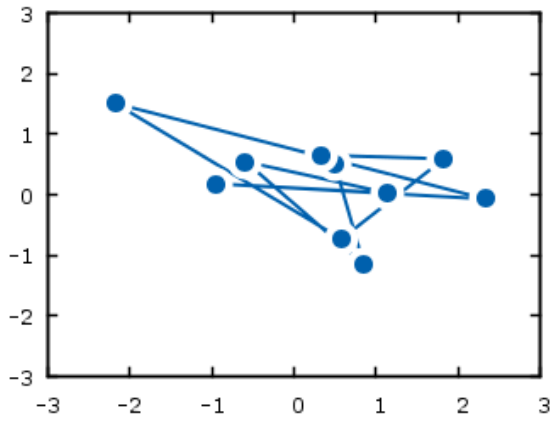
2.4 Computational Advantages and Limitations

While it may seem clear that the HMC method will converge in much fewer steps than the more basic random walk MCMC methods, it may not be clear how this translates to computation time. In fact, the HMC will converge must faster than the random walk methods. Since the Gibbs method is not feasible in the BNN calculation, I will compare computation time between the Metropolis and HMC methods. The Gibbs method, however, scales similarly as the Metropolis algorithm.

The increased computation time for the HMC method is evident due to the necessary calculation of the gradient of the "potential energy" in the momentum and position updates. If we use a finite difference approximation for the gradient, that is an additional $\mathcal{O}(n_p)$ evaluations of $U(q)$, where



(a) 10 steps of the Metropolis-Hastings Algorithm. The proposal distribution is a two dimensional Gaussian, $Q(\omega'|\omega^{t-1}) = e^{-\frac{1}{2}(q[0]^2+q[1]^2)}$.



(b) 10 steps of the HMC Algorithm with $L = 40$, $\epsilon = 0.05$, where L is the number of leapfrog steps and ϵ is the step size.

Figure 2.2: A comparison of the traversal the parameter space for both the Metropolis-Hastings algorithm and the HMC algorithm. The target distribution in both cases is a two-dimensional Gaussian, $e^{-\frac{1}{2}(x^2+y^2)}$. The algorithms both start at $(x, y) = (0.5, 0.5)$. For the HMC algorithm, there were $L = 40$ leapfrog steps. To compensate for this in terms of computation time, $L = 40$ Metropolis iterations were completed in between each point on the plot. It is evident that, even after 40 iterations, the Metropolis-Hastings points are more correlated than the HMC points.

n_p is the number of neural network parameters. Let us look at the distance traversed from the starting point in terms of the number of $U(q)$ calculations of each algorithm. The Metropolis-Hastings algorithm computes an $\mathcal{O}(np)$ $U(q)$ calculations per iteration. Owing to the random-walk nature of the Metropolis-Hastings algorithm, the distance traveled in L iterations is $\mathcal{O}(\sqrt{L})$ [27], where L it the number of Metropolis-Hastings iterations. In the HMC algorithm, the path is deterministic and the distance traveled in L leapfrog iterations is $\mathcal{O}(L)$. The difference between a random-walk traversal of the parameter space and a deterministic traversal is shown in Fig. 2.2. The deterministic nature of an HMC iteration allows the algorithm to traverse the space more efficiently.

CHAPTER 3

GRAPHICS PROCESSING UNITS

The video game industry has pushed the limits of graphics processing, developing processors that have an enormous amount of parallel computational power. Originally developed for the rapid creation and manipulation of images displayed on a screen, Graphics Processing Units (GPUs) are now used as highly parallel processors for general computation. General purpose computing on the GPU is generally referred to as GPGPU. GPGPU is a constantly evolving field due to its relative newness and therefore there is constant improvement of languages and processors. GPGPU computing gained a significant following with the release of NVIDIA's GPU parallel computing model and platform, named CUDA (Compute Unified Device Architecture) in 1996 [28]. The popularity of GPGPU increased with the development of more user-friendly GPU code with more built-in features. In addition, GPUs themselves are becoming more accommodating for GPGPU computing. In fact, there now exist GPUs that have no associated screen output, like a CPU super computer. We can expect coding for the GPU to become much more user-friendly as the use of GPUs in general computing becomes even more mainstream. In fact, a "production release" of a new CUDA programming model, CUDA 6, was released on March 5, 2014, during the writing of this dissertation. CUDA 6 boasts easier memory management, more accelerated libraries, and multi-GPU scaling.

3.1 About GPUs

A GPU is a specialized processor developed for extremely efficient calculations of graphics. Graphics are represented on a computer by a matrix of values. Therefore, graphics computations are large vector and matrix transformations.

GPUs have been developed to perform matrix calculations rapidly. The GPU is not designed to replace the Central Processing Unit (CPU), which is the main processor of a computer. The designers of GPUs take advantage of the fact that a GPU does not need to be a general purpose processor; it can be highly specialized. In vector and matrix computations, each element's computations are

computed independently from one another. Therefore, fast vector and matrix transformations are achieved through "highly parallel" computation. Parallelism refers to the number of computations that can be executed simultaneously. The GPU permits massively parallel computation of similar calculations, and will not perform well for single threaded or few-threaded applications, as described in Section 3.1.2.

3.1.1 GPU Architecture

The GPU devotes more of its resources to computation because the capabilities of the processor are much more limited than the CPU. GPU calculations comprise significant arithmetic computation on a large amount of data. Conversely, the CPU must accommodate many more types of computations and instructions. On the GPU chip itself, compute-intensive, highly parallel computations are accomplished by devoting more transistors to data processing and fewer to data flow and cacheing [4], as illustrated in Fig. 3.1. Conversely, the CPU must have a significant number

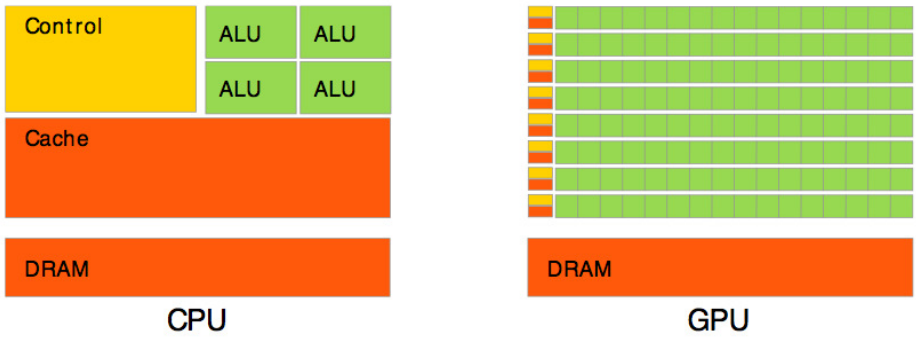


Figure 3.1: An illustration depicting the differences between CPU and GPU architectures. The GPU architecture devotes more transistors to data processing. Image courtesy of NVIDIA [4]. The size of each processor component indicates the number of processors dedicated to that component. The control of the CPU is much larger because the CPU must handle many more types of instructions. The GPU has a control per "multiprocessor", and each multiprocessor is devoted primarily to parallel arithmetic computations. Also, because of the data-parallel nature of graphics processing, the GPU cache is per multiprocessor, where the CPU shares one large cache. The Dynamic Random Access Memory (DRAM) is treated similarly in both the GPU and CPU. In fact, in some integrated chips, they both share DRAM.

of transistors dedicated to the Control Unit (CU) for processing many types of instructions. In

order to understand the GPU in more detail, a few terms relating to computer architecture must be defined.

core a physical computational unit, multiple of which can make up a single processor.

thread a programmed set of instructions that can be executed independently of other threads.

mutithreading the ability for multiple "threads" to execute on a single core at the same time, utilizing unused computation time, such as when one thread is idle.

latency the time required to perform a single action, such as execute a thread.

throughput the number of actions performed at a time, for example, how many threads can be executed simultaneously.

clock rate the frequency at which a processor runs. An instruction is measured in clock cycles, therefore an instruction on a processor with a higher clock rate will be executed faster than one with a lower clock rate.

In CPUs, latency is extremely important. Generally there are many different tasks that need to be executed as quickly as possible. To accomplish this, CPUs are designed to operate at a fast clock speed. However, processors with a higher clock speed run hotter than those with a lower clock speed. Until the early 2000s, CPUs were made faster by adding more transistors in accordance with Moore's law [29]. Then, it seemed that this procedure would no longer work. The CPUs were becoming far too hot to run efficiently. The solution was to introduce processors with more than one core. This is called a multicore processor. The multicore processor succeeded in using less energy because the cores shared resources and the cores are not always all running simultaneously [30]. In graphics computations, throughput is much more important than latency. If the computations can be done with a high level of parallelism, the latency of a single thread is much less important. Therefore, GPUs are many-core processors, with many more cores than the typical CPU. However, the penalty for this is a significantly slower clock rate. Multithreading is a feature utilized in both the CPU and GPU, although implemented in different ways.

The GPU consists of multiple "streaming multiprocessors". Each multiprocessor contains multiple "stream processors", which are Arithmetic Logic Units (ALUs) [31]. A "stream processor" is a processor that can execute limited functionality in parallel, for example, multiple identical kernels. In NVIDIA GPUs, each stream processor executes a "warp". A warp executes 32 identical threads

simultaneously[4]. Each line of the kernel is executed on all threads at the same time. Therefore, logic constructs in which all threads may not have the same operations will become serialized for each case. For this reason, logic algorithms should be avoided or rewritten. This computational approach is called Single Thread Multiple Data (STMD) [4]. Each multiprocessor shares fewer special function units, which calculate mathematical functions that are not provided by the ALU. In the GPU illustration in Figure 3.1, each row is a single multiprocessor. Therefore, the figure represents a GPU with 8 multiprocessors, each with 16 stream processors. A core on a GPU corresponds to one stream processor. In the case of Figure 3.1, the GPU contains 128 CUDA cores. This allows for significant parallel arithmetic computations on the GPU. It should now be evident that a core on the CPU is not equivalent to a core on the GPU. To summarize the main differences between the CPU and GPU architectures, see Table 3.1.

Table 3.1: Highlights of CPU architecture versus GPU architecture.

CPU	**GPU**
general purpose	specialized
multicore	many core
low latency	high throughput

3.1.2 GPU Programming Model

The GPU architecture described in section 3.1.1 is best suited to applications with certain characteristics. There are:

1. a large computational demand,
2. a large amount of intrinsic parallelism,
3. a need for throughput over latency.

These three characteristics are realized in data-parallel calculations that require significant computation per data element. A data-parallel computation is one in which the data are distributed amongst different compute nodes and the same computations are performed on the data independently. Therefore, the computations must depend on one portion of the data only, and the

computations must be capable of being performed completely independently from one another. This is the case for our BNN application, which is discussed in detail in Section 3.3.

The GPU programming model is a Single Program Multiple Data (SPMD) model [32]. The program is the function that is operated on the data independently. Therefore, the program is more like a subprogram and is generally referred to as a *kernel*. The kernel is executed N times by N different threads in parallel [4]. In order to distribute the work on the GPU cores, the threads are grouped into blocks. An entire block is transferred to a core and the threads in that block all share the same local memory. The blocks of threads are known as "thread blocks". The number of threads per block is usually set to 256, but a block can contain up to 1024 threads. The thread blocks must be able to be executed independently. This allows the thread blocks to be grouped together and distributed to all of the available multiprocessors. A thread block can not span more than one streaming multiprocessor, but the threads within a block are then distributed amongst the "streams", or cores, in the streaming multiprocessor. There can be one or more threads per stream. The blocks themselves are grouped into grids. A grid is transferred to a GPU. If the application uses multiple GPUs (such as in a high performance GPU cluster), the grids get distributed amongst the GPUs. For this application, we are assuming the use of a single GPU.

Thread blocks allow the code to automatically scale with the number of cores. Moreover, the number of cores on the GPU does not have to be known by the programmer and the same code will run on any CUDA-capable GPU, regardless of the number of cores. Therefore, the programming model is device independent. To understand the distribution of threads on the GPU, as well as the memory that is accessible by each thread, see Fig. 3.2. In fact, the memory hierarchy of a GPU is not straightforward. The "per-thread local memory" referenced in Fig. 3.2 physically resides in the device memory, which is the same location as the global memory, but is accessible only by a single thread. In addition, there are a certain number of registers per thread. These are generally used for any memory allocated in the device code. The shared memory is physically located in each streaming multiprocessor and the memory is accessible to all threads in the same thread block. The global memory is accessible by all threads, and therefore all blocks, in a grid. If using more than one GPU, grids are distributed amongst GPUs, and a grid can not span more than one GPU. There is no synchronization or memory sharing between GPUs. Accessing device memory from the device code is very fast. However, the time required to transfer data between thee host and

the device is relatively long. Therefore, the frequency of host and device data transfer should be minimized as much as possible for the application.

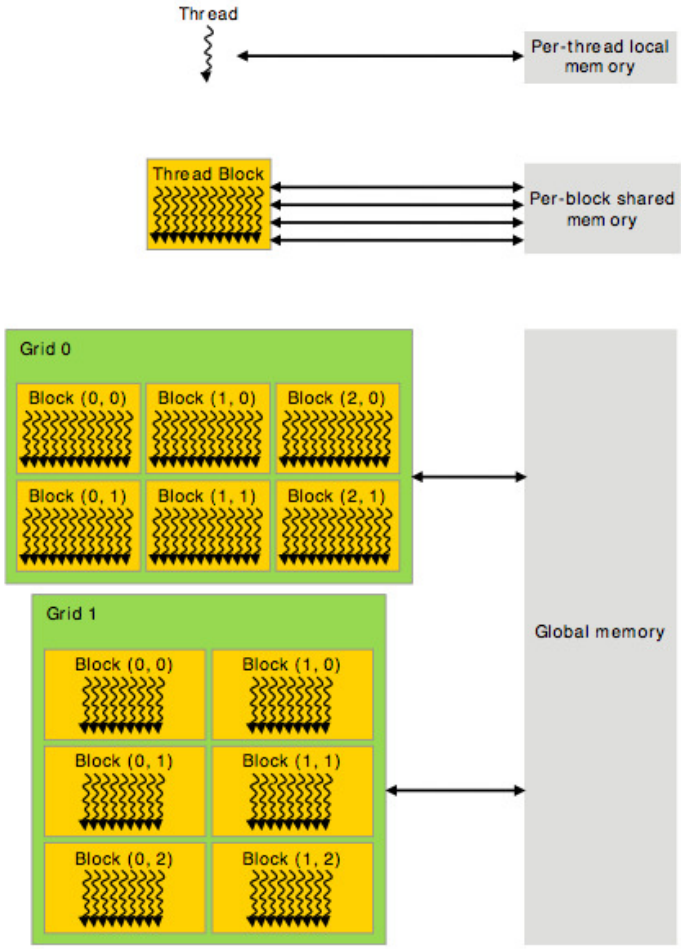


Figure 3.2: Schematic of the memory hierarchy of threads on a GPU. Image courtesy of NVIDIA [4]. There are three different types of read-write memory on a GPU, per-thread memory, per-block shared memory, and global memory.

Each thread operates the same kernel on a specific set of data. The blocks and threads are indexed with identifiers from 0 to the number of threads and blocks. Referencing the thread and block identifier is what allows each thread to operate on different data.

3.2 CUDA

While there are other languages, both open source and proprietary, that interface with the GPU, this work uses NVIDIA's CUDA framework. CUDA was chosen because, at the time of this dissertation work, it was the best documented, supported, and tested interface available. Therefore, I will discuss details of programming on the GPU in terms of the CUDA framework. The drawback of the CUDA framework is that it requires an NVIDIA graphics card. The general concepts are similar for other popular interfaces, such as OpenCL (an acronym for Open Computing Language), but not equivalent, and I want to be clear and concrete in my explanation in order to avoid vagueness. OpenCL is a framework for parallel processing on any external device, not only the GPU. The programming model for OpenCL is more abstract than CUDA, because CUDA takes advantage of the fact that it is always applied to GPU hardware.

With CUDA, the GPU is considered an external device. The CPU is called the host. Therefore, the code is divided into device code, for the GPU, and host code, for the CPU. This work uses the CUDA C extension in the CUDA 5 programming model. CUDA supports the full C/C++ languages on the host with additions that allow for interfacing with the GPU. The GPU supports C with some restrictions. The files utilizing the CUDA C extension libraries must be compiled with the NVIDIA CUDA compiler, `nvcc` and have the extension `.cu`. Object files compiled with `nvcc` are compatible with C++ and C compilers. In the CUDA 5 programming model, GPU memory must be explicitly allocated and freed on the CPU and GPU, and then the contents of this memory must be transferred between the host and the device. As noted above, while the memory bandwidth within the GPU is high, the memory bandwidth between the host memory and the device memory is low. Therefore, reducing the number of times a memory transfer occurs leads to increased speedup.

The setup of CUDA source code containing both host code and device code is accomplished by the addition of CUDA directives for device code indicating if the function is a kernel that may be launched from the host code, or a device function known only to the GPU.

Listing 3.1: The syntax for indicating device code. The `__global__`kernel is launched from the host while the `__device__` function can only be called from the device.

```
/* a GPU kernel that can be launched by CPU */
__global__ void diff(float* q, int H, int I) {
  /** Device kernel code **/
```

```
  float result = gpu_func(q);
};

/* a GPU function that can be called only from the GPU */
__device__ float gpu_func(float* q) {
  /** Device function**/
};
```

In order to launch the kernel execution on the GPU from the CPU, first the number of blocks per grid and the number of threads per block must be defined. Generally 256 threads per block are used. This value allows each thread to have access to a sufficient number of registers and most GPUs do not have a maximum allowed number of threads per block less than 256. This value should always be a factor of 32, because instructions are issued simultaneously in groups of 32 threads. These are the warps discussed in Section 3.1.1. The number of blocks is then determined by the total number of threads necessary for the computation. The number of blocks does not have to be associated with the number of cores available. The GPU dynamically handles the scheduling of allocation and execution of blocks on available cores. In the example below, the name of the kernel function on the GPU is `diff`. The brackets contain the number of blocks and threads per block to be generated. The parentheses following the brackets contain the kernel function arguments, in terms of the variables existing in device, that is GPU, memory. The pointers for any device memory that will be used in the kernel must be an argument to the function. The function is launched from the host code as follows.

Listing 3.2: The syntax for launching device (GPU) kernels from the host (CPU).

```
/* execute __global__ void diff(float* dev, int H, int I) */
 diff<<<blocksPerGrid,threadsPerBlock>>>(dev_q, H, I);
```

Since the kernel launch from the CPU is actually launching N kernels on the GPU, where N is the number of threads, all `__global__` GPU kernels must have a void return. All memory transfers between the CPU and GPU must be done using the `cudaMemcpy` function shown in Listing 3.3. In the thread example 3.4, each thread computes an element of a desired resultant array, `d[N]`. The resultant array must then be transferred back to the host after the calculations are completed. Proper handling of CPU and GPU memory is extremely important. See below for an example of how memory is allocated and transferred from the CPU to the GPU in CUDA. The following is host code.

Listing 3.3: Memory allocation and transfer from host to device.

```
/** allocate NN parameters q **/
dev_q = new float[np]; // device vector q
cudaMalloc(&dev_q,size_np); // allocate on GPU
float* qq = &in_q[0]; // pointer to q vector on CPU
/* copy from CPU to GPU */
cudaMemcpy(dev_q, qq, size_np, cudaMemcpyHostToDevice);
/* Call GPU kernel or do other CPU work
.
.
*/
cudaFree(dev_q); // Free GPU memory
```

In order to have each kernel operate on different data, each block and thread within each block is indexed. These indices allow the different threads to access different data. The blocks and grids can be indexed in one, two or three dimensions. The following is a one dimensional block and grid example where a row of I data elements are summed on the device.

Listing 3.4: Example of using thread indices to evaluate the kernel on a portion of the data. `threadIdx`, `blockIdx`, and `blockDim` are CUDA language extensions that retrieve information about the thread currently being executed. The result of the calculation for each thread is stored in an element of the array `d[]` that is then brought back to the CPU in the host code after all threads finish executing. See Listing 3.3 for memory transfer syntax.

```
int thread = threadIdx.x + blockIdx.x*blockDim.x;
float d[thread] = 0;
for(int i = 0; i<I; i++) {
  d[thread] += x[thread*I+i];
 }
```

The device and host code can be placed in the same file. See Listing 3.5 for an example of how the components are put together in a fully functional CUDA program. See Appendix A, for a simplified rendering of the CUDA code necessary for constructing a BNN. For more a in-depth look at CUDA programming for the GPU, refer to the CUDA Programming Guide for the most current version of CUDA [4].

Listing 3.5: A fully working CUDA source code example. This code is compiled by `nvcc -o vec_difference vec_diff.cu` where `vec_diff.cu` is the name of this CUDA file.

```
#include<iostream>
using namespace std;
/* device code */
__global__ void difference(float* A, float* B, float* C, int N) {
  int thread = blockDim.x*blockIdx.x + threadIdx.x;
```

```
  if(thread < N){
    C[thread] = A[thread] - B[thread];
  }
};

/* host code */
int main() {
  int N = 256;
  size_t size = N*sizeof(float);
  float* A = (float*)malloc(size);
  float* B = (float*)malloc(size);
  float* C = (float*)malloc(size);

  for(int i=0;i<N;i++){
    A[i] = i;
    B[i] = i;
  }
  // Allocate vectors in device memory
  float* device_A;
  cudaMalloc(&device_A, size);
  float* device_B;
  cudaMalloc(&device_B, size);
  float* device_C;
  cudaMalloc(&device_C, size);
  // Copy vectors from host memory to device memory
  cudaMemcpy(device_A, A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(device_B, B, size, cudaMemcpyHostToDevice);

  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
  difference<<<blocksPerGrid, threadsPerBlock>>>(device_A, device_B, device_C,
  cudaMemcpy(C, device_C, size, cudaMemcpyDeviceToHost);
  for(int i=0;i<N;i++){
    cout << h_C[i] << endl;
  }
  // Free device memory
  cudaFree(device_A);
  cudaFree(device_B);
  cudaFree(device_C);
  // Free host memory
  free(A);
  free(B);
  free(C);
  return 0;
}
```

3.2.1 CUDA Thrust Library

This work also makes use of the CUDA-based template library, Thrust [33]. The Thrust library includes a collection of basic data-parallel functions and algorithms. For those whose goal is the use of one of the functions or algorithms to speed up data-parallel code, Thrust can be used without much knowledge of the GPU. The task is to

- include the necessary thrust libraries
- create any necessary host and device vectors using Thrust syntax
- call Thrust library function
- compile using `nvcc`.

See Listing 3.6 for a basic example. The Thrust library allows the simple utilization of the GPU's resources without dealing with the specifics of memory allocation and algorithms. The drawback is that the applications restricted to the built in parallel functions available in Thrust.

Listing 3.6: A simple Thrust implementation. This program uses the parallel reduce function available in the Thrust library. A reduce function is a sum of all elements in a vector.

```
#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/reduce.h>

int main() {
  int N = 1000;
  thrust::device_vector<float> d_vec(N,1);
  /* Initialize to 1, 2, 3, etc*/
  thrust::sequence(d_dev.begin(),d_dev.end());
  /* sum all elements of the vector */
  float sum = thrust::reduce(d_dev.begin(),d_dev.end());
  return 0;
}
```

The thrust libraries can also be used in conjunction with programmer-supplied CUDA code. Thrust has a pointer to device memory that has been allocated using CUDA C. In this work, a parallel reduction is applied to the vector calculated in the `diff()` device function outlined in Listings 3.8. See Listing 3.7 to see how Thrust works with memory already allocated on the GPU.

Listing 3.7: An outline of how the Thrust device pointer is used to apply a thrust library function to an array already allocated on the device in CUDA.

```
dev_d = new float[N]; //device vector
cudaMalloc(&dev_d,size_n); // allocate dev_d memory on GPU
thrust::device_ptr d_dev_ptr(dev_d);
diff<<<blocksPerGrid,threadsPerBlock>>>(dev_q,dev_d); //call GPU function
/* Call Thrust function to sum results from diff() */
HMC_type sum = thrust::reduce(d_dev_ptr,d_dev_ptr+N);
```

3.3 BNN Application in Data-Intensive Cases

The application of this dissertation is the BNN algorithm described in section 2.3, applied to the pMSSM, which is described in Section 1.2.1. The computational hurdle in using the BNN algorithm to map pMSSM parameters to pMSSM predictions is that the construction of the BNN using HMC is extremely time consuming. For example, with $I = 19$ parameters, and $H = 10$ hidden nodes, the number of neural network parameters is $1 + H(I + 2)$, which yields 211 parameters for the pMSSM map.

Figures 3.3 to 3.5 show a few results using BNNs constructed with the flexible Bayesian modeling packag (`fbm`) of Neal [26]. This highly optimized code, which was executed on 3.6 GHz CPUs, took 25 s / saved point, that is, 1.25 s per iteration, where each iteration comprises $L = 100$ deterministic steps of the HMC algorithm. As is clear from the figures, the predictions for the supersymmetric top masses are very well modeled using 10,000 pMSSM points, but the modeling of the prediction for the neutral Higgs boson mass, while reasonable, would benefit from the use of a larger training sample. Our goal is to achieve a substantial reduction in these training times by using GPUs.

The evaluation of the neural network function for each training event is independent of all other training events. Therefore, these calculations can be done in parallel. A parallel algorithm for the sum of the square of differences between the target and the computed neural network value in Eq. 2.8 can also be implemented. This is known as a parallel reduction algorithm. The high level of parallelism, due to the independence of the training events, paired with a significant amount of computation per node in the evaluation of a NN with many parameters, fulfills the requirements for a good GPU application outlined in section 3.1.2.

The device kernel is defined as follows. Each kernel evaluates one term of the log likelihood terms in Eq. 2.8.

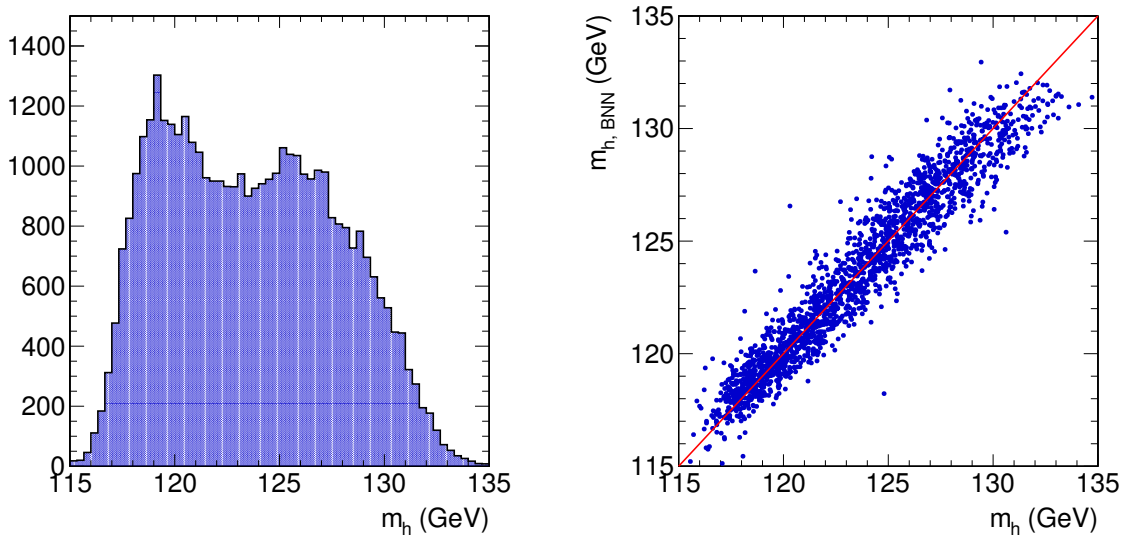


Figure 3.3: (left) Distribution of the predicted mass, m_h, of the light neutral Higgs boson in the pMSSM. (right) The value of a BNN model of the function $m_h = f(\theta)$ compared with the actual prediction of m_h, where θ denotes the 19 pMSSM parameters. The BNN function was modeled with a $(I, H, 1) = (19, 10, 1)$ neural network, 10,000 iterations — each comprising $L = 100$ deterministic steps, and 10,000 pMSSM training points. Of the 10,000 iterations and therefore 10,000 NN points sampled, every 20 were were saved yielding 500 saved points, the last 250 of which were used to approximate the integration over the neural network parameter space.

Listing 3.8: The variable q contains the neural network parameters, x is the array of training data, w contains the event weights, t are the target values, d contains the resultant array of differences, and H and I are the number of hidden nodes and inputs, respectively.

```
__global__ void diff(HMC_type* q, HMC_type* x, HMC_type* w,
                     HMC_type* t, HMC_type* d, int H, int I) {
  int th = threadIdx.x + blockIdx.x*blockDim.x;
  HMC_type f = q[0];
  for(int j=0;j<H;j++){
    HMC_type inSum = 0.0;
    for(int i=0;i<I;i++){
      inSum += q[2*H+1+I*j+i]*x[th*I+i];
    }
    f+=q[j+1]*tanh(q[H+1+j]+inSum);
  }
  d[th] = w[th]*(t[th]-f)*(t[th]-f);
};
```

Therefore, the BNN algorithm is parallelized through the evaluations of $U(\omega)$. To be clear, in the Listing examples, ω is given by `q`. This is the most computationally demanding portion of the

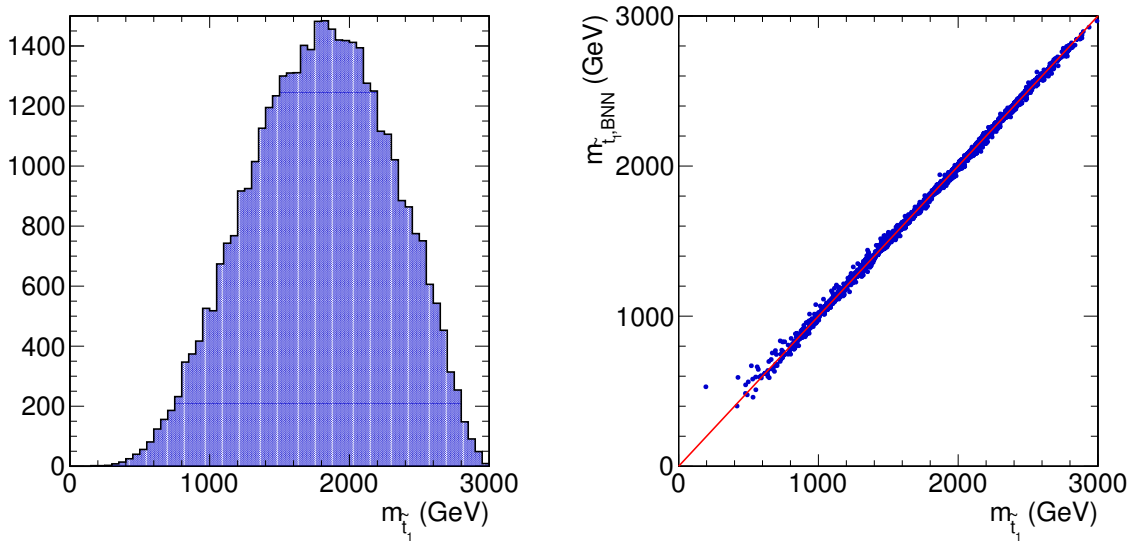


Figure 3.4: (left) Distribution of the predicted mass, $m_{\tilde{t}_1}$, of the supersymmetric top ("stop") $\tilde{t}_1$ in the pMSSM. (right) The value of a BNN model of the function $m_{\tilde{t}_1} = f(\theta)$ compared with the actual prediction. See Fig. 3.3 for details of the BNN.

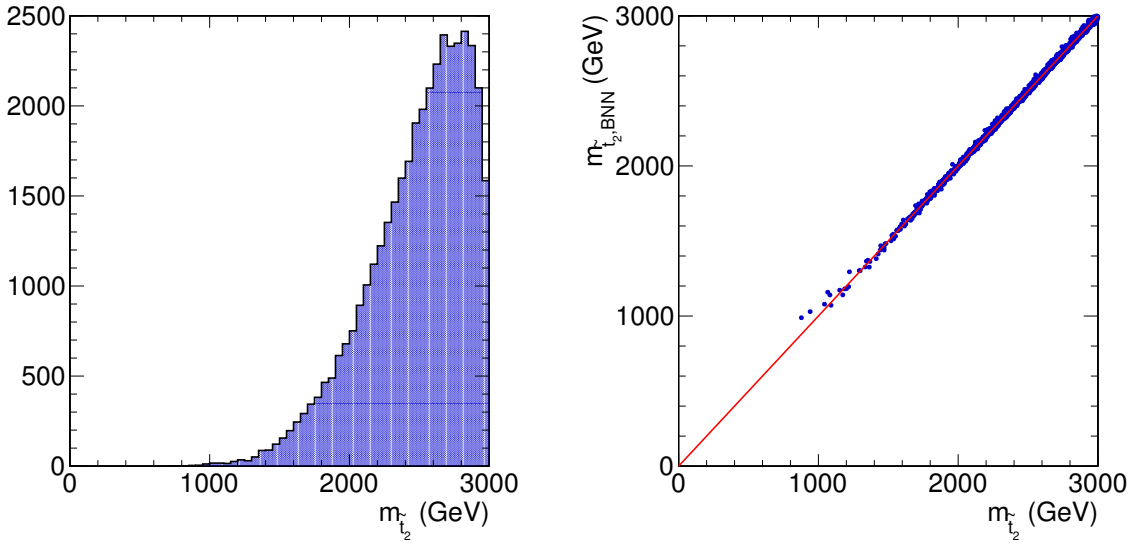


Figure 3.5: (left) Distribution of the predicted mass, $m_{\tilde{t}_2}$, of the supersymmetric top $\tilde{t}_2$ in the pMSSM. (right) The value of a BNN model of the function $m_{\tilde{t}_2} = f(\theta)$ compared with the actual prediction. See Fig. 3.3 for details of the BNN.

algorithm. The training data, x, and the targets, t, are loaded onto the device in the constructor of the BNN class. The neural network parameters, q, must be updated every BNN iteration because their values are dependent on the previous iteration.

CHAPTER 4

RESULTS

The goal of this dissertation is to decrease the runtime of the BNN algorithm in the case of large training sets. In order to gain the runtime improvement, a GPU implementation was introduced, as described in Section 3.3. Because of the difference in architecture between the CPU and GPU, an exact comparison is not possible. To determine the method to measure the speedup, the application and target audience for this code was considered. Speedup in this work is defined as $S = \frac{t_{CPU}}{t_{GPU}}$, where t_{CPU} is the time to run a completely linear implementation on the CPU and the t_{GPU} is the GPU implementation in Appendix A.

The use of BNNs in the field of high energy physics has been proven useful but has been limited due to the computation time necessary for construction of the BNN, as discussed in Section 1.3. Because this application is well suited for the GPU, as described in Section 3.1.2, it was appealing that the physicists interested in using BNNs to study the pMSSM could use their personal computers. The many-core nature of the GPU allows for high performance computing cheaply for suitable applications. If the algorithm is fast enough to run on a personal computer, then there appears to be enough interest that the use of BNNs in high energy physics applications would become more routine. There are many more applications outside of the pMSSM that have similar computational challenges. The event-based data in high energy physics indicates that many more of these applications would benefit from a GPU BNN implementation.

Because the objective of this study is to create an application for use on a personal computer, I used the resources available on my personal laptop for all of the speedup testing.

4.1 Systematic Study of BNN in Data-Intensive Cases

The following results were all obtained on my MacBook. GPUs are generally categorized by the graphics card on which they reside. The MacBook contains a GeForce 320M graphics card. This is a low-end graphics card. The ability to achieve the speedup reported here on a low-end GPU model is a very good indicator that this application is well suited for anyone with a programmable

GPU. See Table 4.1 for a comparison between the graphics card used for these studies compared to the graphics card currently shipped with a MacBook Pro today. The comparison emphasizes the rapidly increasing development of GPU technology, and highlights the fact that the speedups reported in this dissertation could be considered at the lower end of the possible speedup that could be achieved in this application. The CPU used in these results is the Intel Core 2 Duo.

Table 4.1: Comparison of the GPU used in this study (purchased in 2010) and a current laptop GPU model (2014). The values in all fields indicate an incredible improvement in GPU performance could be achieved with newer GPU models. Content from NVIDIA [2]

	NVIDIA GeForce 320M	NVIDIA GeForce 750M
CUDA cores	48	384
clock speed	450 MHz	967 MHz
memory	512 MB	2048 MB

Comparatively, the CPU in the current MacBook Pro release is the Intel Core i7. As discussed in Section 3.1, the clock speed of CPUs have not been rising, but they are becoming more parallel, though not highly parallel like the GPUs. Therefore, if the linear version of the BNN took advantage of multithreading and four CPU cores, then a speedup is expected for the CPU version as well.

Table 4.2: Comparison of the CPU used in this study (purchased in 2010) and a current laptop CPU model (2014). Content from Apple [3].

	Intel Core 2 Duo	Intel Core i7
cores	2	4
clock speed	2.4 GHz	2.3 GHz
number of threads	1 MB	8 MB

The test problem used for the systematic timing study used simulated proton-proton collision data at 8 TeV using Pythia8 [34]. Specifically, we look at the collision of two protons, which produces two Z bosons, which each decay into two leptons. This reaction is written as the $pp \rightarrow ZZ \rightarrow 4\ell$ decay channel. The task is to take the information for the two leptons of one of the Z bosons and map them to the mass of the Z boson from which they decayed. See Fig. 4.1 for the mass distribution of the Z boson in which we are interested. The inputs are the momentum variables

for the two leptons. These are p_t, η, and ϕ for each lepton. Therefore, there are 6 inputs for this application. The training data and targets were normalized to have a mean of zero and standard deviation on one. This makes the tuning of L ϵ much easier. In order to study the speedup of the BNN construction for the GPU relative to the CPU, the times required to run the code over 100 HMC iterations were compared. Training a network with only 100 HMC iterations will not converge to the posterior distribution, but is sufficient for timing studies.

First, the effect of the number of training events was analyzed. Fig. 4.2 shows the time taken to run the algorithms. The results indicate that as the amount of data increases, the computation time

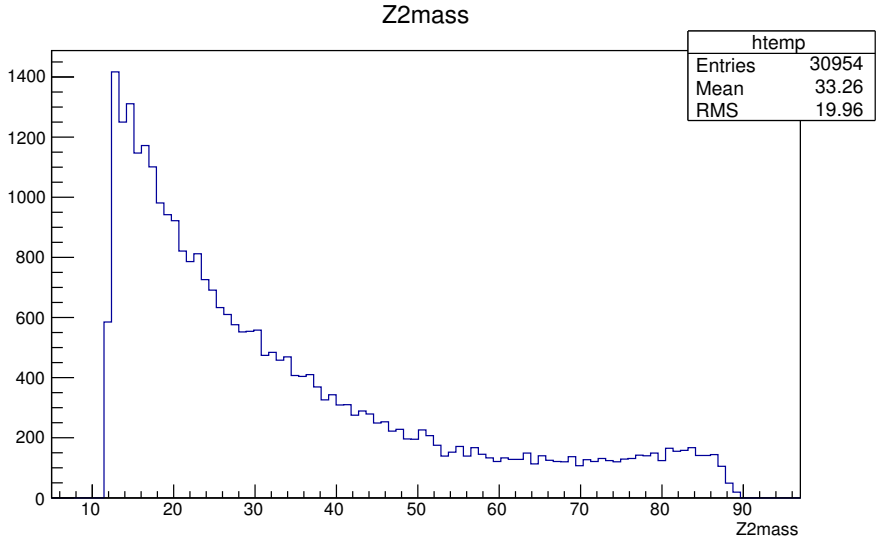


Figure 4.1: Mass of second Z boson from $pp \to ZZ$ reaction from . This is the target data for the neural network. (Plot to be improved)

of the CPU rises faster than the GPU. This result is expected since the GPU version parallelizes the data by event. Therefore, there are more threads with an increasing number of training events, but these are grouped into blocks which are then executed in parallel on the GPU when they receive a free streaming multiprocessor. There would be an increased time, then, only when the number of blocks becomes another factor greater than the number of streaming multiprocessors. Next, we investigated the effect of increasing the number of hidden nodes in the network. The optimal number of hidden nodes for a given application is unknown, but more complicated mappings typically benefit from more hidden nodes. Increasing the number of hidden nodes increases the computation time per

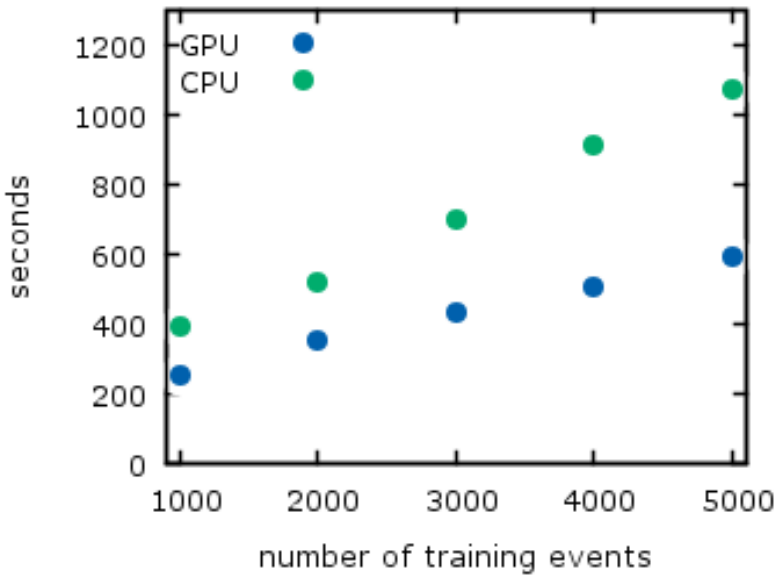


Figure 4.2: Times required to run 100 HMC iterations on the CPU and GPU. The parameters used for the BNN were $I = 6$ inputs, $H = 8$ hidden nodes, $nOut = 100$ HMC iterations, while testing the number of training events ranging from 1000 to 5000.

thread in the GPU version. Figure 4.3 shows the increase in computation time with an increasing number of hidden nodes. The computation time for the CPU increases much faster than for the GPU. In the GPU implementation, the increased time is in the computation in each thread, so the effect is mitigated by the number of threads running in parallel. This result is very promising for more complicated applications. Our target application, the pMSSM, requires at least 19 hidden nodes.

Overall, the speedup achieved through the GPU is significant. In Figures 4.2 and 4.3, there is a minimum of 2x speedup for the GPU for the parameters scanned, and the speedup increases as the computation becomes more costly.

4.2 Application to the Phenomenological Minimal Supersymmetric Standard Model

In Section 4.1, the training of a BNN on a GPU was shown to produce significant speedup. Therefore, results for the training of a BNN for pMSSM predictions are given solely on the GPU

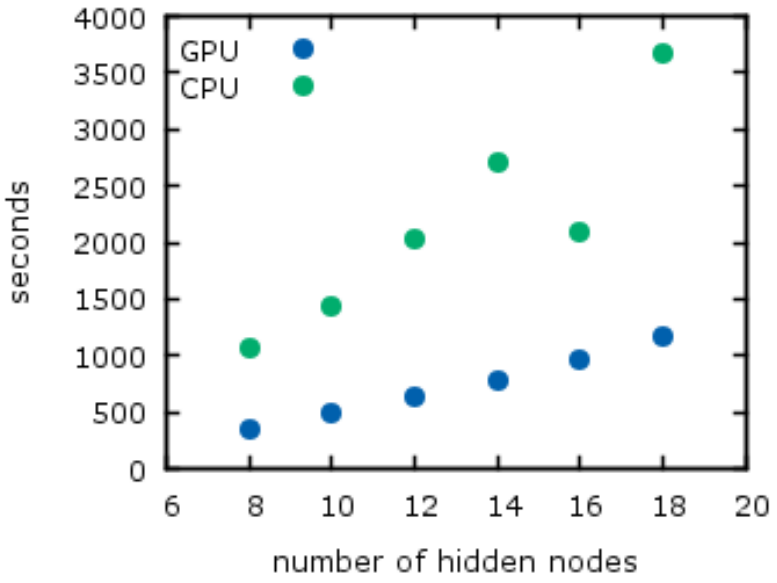


Figure 4.3: Times required to train a BNN on the CPU and GPU. The parameters used for the BNN were $I = 6$ inputs, $nTrain = 2000$ training events, $nOut = 100$ HMC iterations, while testing the number of neural network hidden nodes ranging from 6 to 20. The outlier in this plot needs to be further tested and analyzed.

due to the increased execution time of the problem. For this study, I report times on the same GPU defined in the first column of Table 4.1, along with results using the GPUs available on the Spear cluster at the Research Computing Center at Florida State University. The Spear cluster gives users access to Tesla M2050 GPUs. Table 4.3 highlights some features of the two GPUs used in this study.

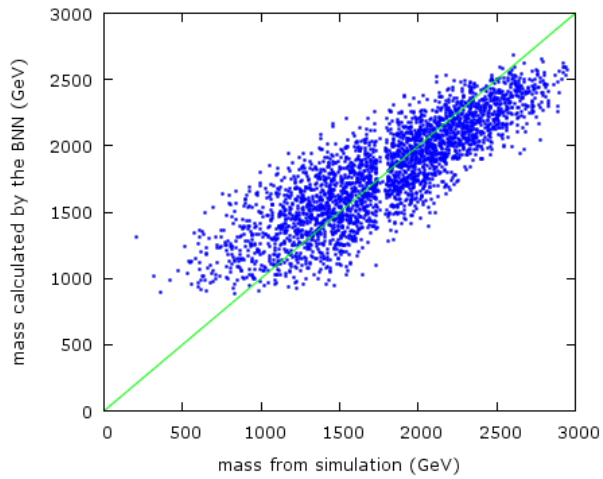
Table 4.3: Architecture specifications for the two GPUs used to train a BNN for a mapping of the pMSSM parameters to a prediction, the mass of the $\tilde{t_1}$ particle. The GeForce 320M is an older basic laptop GPU, while the Tesla M2050 is a new higher-end GPU available at FSU's RCC.

	GeForce 320M	Tesla M2050
CUDA cores	48	448
clock speed	450 MHz	1546 MHz
memory	512 MB	2687 MB

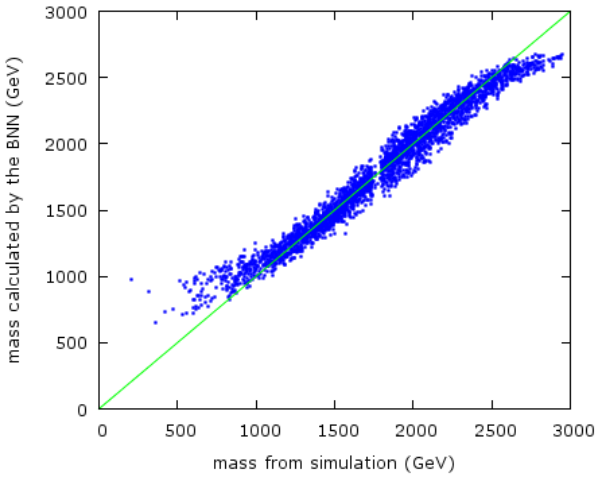
To test the time required to construct a BNN for the mapping of the pMSSM to its predictions, one of its predictions, the mass of the $\tilde{t_1}$ particle, is used as the target. As before, the speedup in the case of increasing number of training events and number of hidden nodes is studied.

The network used for testing has $I = 19$ inputs, which are the 19 parameters of the pMSSM model. The right column of Table 1.1 in the Introduction lists these parameters. For the HMC tuning parameters, $L = 100$ deterministic leapfrog steps, and a step size of $\epsilon = 0.0003$ was used. Fig. 4.4 shows the results from the BNN training using data from the same set as the training data. The first figure of the plot is the BNN using the first 100 networks produced in the training. The second also uses 100 networks, but these were sampled after a 50 iteration "burn-in" phase of training. After the burn-in phase, every tenth network was saved for use in the BNN equation. This plot indicates that the network is being appropriately trained.

Figure 4.5 shows the time taken for 100 HMC iterations on the GeForce 320M and the Tesla M2050. The increased number of nodes and the faster clock speed have a significant effect on the speedup of the algorithm. Figure 4.6 shows the computation time on both GPUs as a function of the number of hidden nodes. This figure highlights the effect of the increased clock speed of the Tesla M2050.



(a) Results from the mapping of the 19 pMSSM parameters to a prediction, the mass of the $\tilde{t}_1$ after 100 HMC iterations. For the network, $I = 19$, $H = 10$, $L = 100$, $\epsilon = 0.0003$ and 5000 training events were used.



(b) Results from the mapping of the 19 pMSSM parameters to a prediction, the mass of the $\tilde{t}_1$ after 1050 HMC iterations. For the results, the first 50 HMC iterations were not used, and every 10 networks were used after this "burn-in" phase. For the network, $I = 19$, $H = 10$, $L = 100$, $\epsilon = 0.0003$ and 5000 training events were used.

Figure 4.4: A comparison of the results of the BNN training after 100 and 1050 iterations. The data is normalized to have a mean of zero and standard deviation of one, and then renormalized after the training. The choice of HMC tuning parameters is simpler to determine with normalized data. This mapping was done without much effort in finding optimum tuning parameters.

The results of the pMSSM training indicate that this problem is very well suited for the GPU and the use of BNNs should be considered for the study of higher-dimensional supersymmetry models.

4.3 Discussion

4.3.1 Z2 Mass Study

The timing results in Figs. 4.2 and 4.3 follow the scaling expected in comparing the CPU to the GPU. To obtain a rough estimate of how the timing is expected to scale in the case of increasing the number of training events and the number of neural network hidden nodes, I will simplify the architectures. In the simplified picture, the goal is an idealized equation of the computation time of the two architectures in the case of BNN training. For this toy problem, I consider the

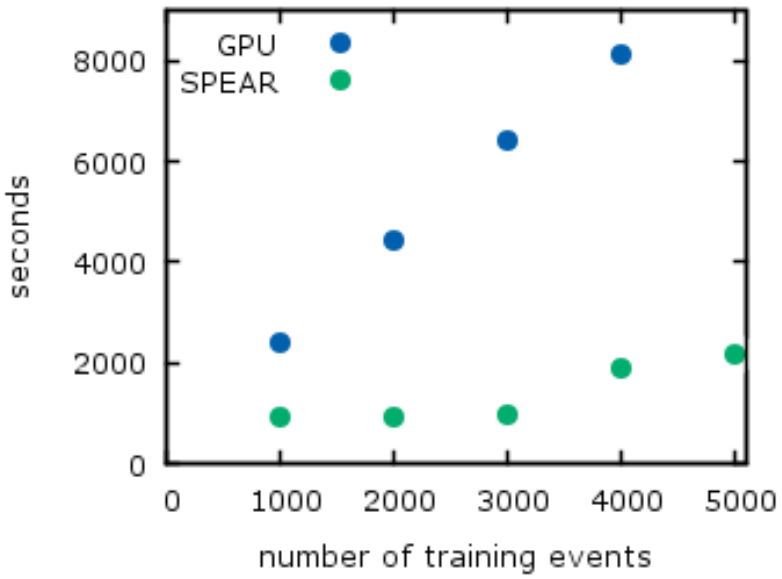


Figure 4.5: Times required to train 100 HMC iterations on a GeForce 320M and TESLA M2050 GPU. The parameters used for the BNN were $I = 19$ inputs, $H = 10$ hidden nodes, $nOut = 100$ HMC iterations, while testing the number of training events ranging from 1000 to 5000. Note that the fastest portion of the algorithm, the evaluation of $U(\omega)$ was evaluated on the described GPUs. The rest of the algorithm was computed on the CPU native to the computer that was used.

time required to calculate $U(\omega)$, the most computationally intensive portion of the BNN algorithm. First, I consider the scaling in terms of the variable number of training events, N_{train}. For the CPU, the total time to compute $U(\omega)$, given by T_{CPU} is the number of training events used, N_{train} times the amount of time taken to evaluate the neural network on the CPU, τ_{CPU}. This gives

$$T_{CPU} = N_{train} \times \tau_{CPU}. \tag{4.1}$$

The total GPU time must be in terms of the architecture. The GPU has M multiprocessors, each with C cores, each of which run a warp, $w = 32$. Naively, this gives $M \times C \times w$ simultaneously executing threads[1]. Some GPUs allow more than one block per multiprocessor, if the GPU supports enough threads. The total number of threads is the number of training events, N_{train}. For this

[1]This is, however, an extremely naive approximation. The expression technically refers to the "concurrency" of the system, or how many threads can exist on the GPU simultaneously. The GPU can not actually evaluate an arithmetic instruction on every thread in every clock cycle. The throughput, measured in Gflops/s must be studied to truly look at the parallelism of the system. Our discussion makes use of the concurrency in order to discuss the scaling in terms of the parameters in the BNN. The scaling discussion should be considered more than ideal.

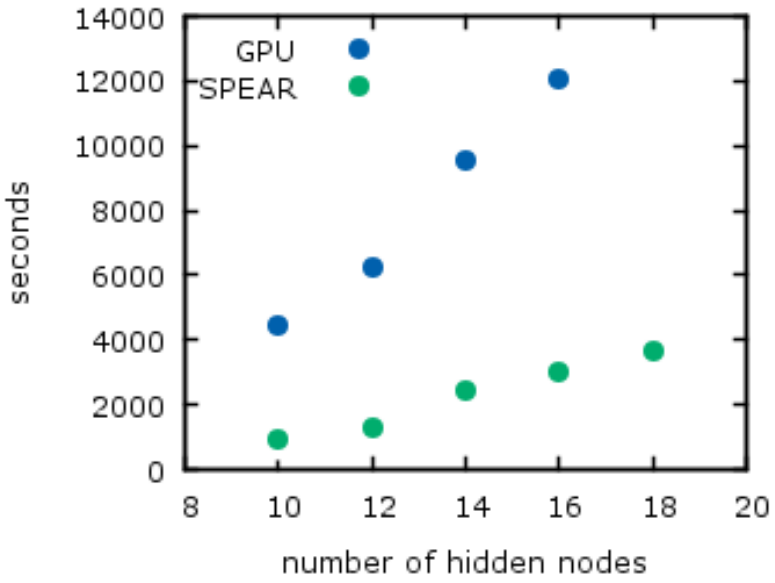


Figure 4.6: Times required to train a BNN on a GeForce 320M and TESLA M2050 GPU. The parameters used for the BNN were $I = 19$ inputs, $nTrain = 5000$ training events, $nOut = 100$ HMC iterations, while testing the number of neural network hidden nodes ranging from 10 to 18.

application, the standard 256 threads per block are used. The number of blocks, n_B, is then $n_B = \texttt{ceil}(\frac{N_{train}}{256})$. If the number of blocks that can execute in parallel is M, then the GPU will have to do $\texttt{ceil}(\frac{n_B}{M})$ iterations of the computation. In addition to the computation time, the time required to transfer the neural network weights, ω for the current HMC iteration must be taken into account. Memory transfer between the host and device is slow, as discussed in Section 3.1.2. If τ_{GPU} is the amount of time the GPU uses to evaluate the neural network, then the total time for the evaluation of $U(\omega)$ is given by

$$T_{GPU} = \frac{n_B \times \tau_{GPU}}{M} + t_{mem} = \frac{N_{train} \times \tau_{CPU}}{256 \times M} + t_{mem}. \tag{4.2}$$

In Eq. 4.2, the 256 threads per block and the number of multiprocessors, M are constants dependent on the GPU architecture, and can be absorbed into a constant variable, $\gamma_{GPU} = 256 \times M$,

$$T_{GPU} = \frac{N_{train}\tau_{GPU}}{\gamma_{GPU}} + t_{mem}. \tag{4.3}$$

For this exercise, we will assume that the time of computation is much larger than the memory transfer time $t_{mem} << \frac{N_{train}\tau_{GPU}}{\gamma_{GPU}}$. The speedup $S_{N_{train}}$, then, is

$$S_{N_{train}} = \frac{T_{CPU}}{T_{GPU}} = \frac{\gamma_{GPU} N_{train} \tau_{CPU}}{N_{train}\tau_{GPU}} = \frac{\tau_{CPU}}{\gamma_{GPU} . \tau_{GPU}} \tag{4.4}$$

As discussed in Section 3.1.1, the clock speed of the GPU is significantly slower than the clock speed of the CPU. Therefore τ_{GPU} and τ_{CPU} are not equivalent. In the testing reported in Section 4.1, the CPU clock speed is approximately six times faster than the GPU clock speed. If we assume that the evaluation time of the neural network between the CPU and GPU scales similarly to the clock speed, we can approximate $\tau_{CPU} = c\tau_{GPU}$. Plugging this into Eq. 4.4 and grouping together the constants yields

$$S_{N_{train}} = \frac{T_{CPU}}{T_{GPU}} = \alpha_N. \tag{4.5}$$

The scaling between the execution times on the GPU and CPU is expected to be constant. The results in Fig. 4.2 follow the expected trend.

The scaling between the CPU and GPU with an increasing number of hidden nodes, H, is simpler to study. In the case of the GPU, the additional computation time, c_H is a factor of the computational time of the neural network, τ_{GPU}. For the total computation time on the GPU, this gives

$$T_{GPU} = \frac{c_{h_{GPU}} N_{train} \tau_{GPU}}{\gamma_{GPU}} + t_{mem}. \tag{4.6}$$

In the case of the CPU, the total time increase is a factor, c_{h_CPU}, times the computation time of the neural network, τ_{CPU} times the number of training events, N_{train}, giving

$$T_{CPU} = c_{h_{CPU}} N_{train} \tau_{CPU}. \tag{4.7}$$

The speedup, then, is

$$S_H = \frac{T_{CPU}}{T_{GPU}} = \frac{\gamma_{GPU} c_{h_C PU} N_{train} \tau_{CPU}}{c_{h_{GPU}} N_{train} \tau_{GPU}}. \tag{4.8}$$

Assuming again that τ_{GPU} and τ_{CPU} scale by a constant, and grouping together the constants into α_H, the speedup becomes

$$S_H = \alpha_H. \tag{4.9}$$

Therefore, we expect the CPU to increase proportional to N_{train}, which we see in the results in Fig. 4.3.

In conclusion, the results give the speedup expected from the approximate calculations above. The factors α_N and α_H are hardware dependent and their ranges can be large due to the vast range of CPUs and GPUs available.

4.3.2 PMSSM Study

The timing results in the pMSSM study show the effect of the GPU architecture on the time of execution. As seen in Table 4.3, there is a significant gain in the number of nodes and clock speed. In the case where the number of training events is increased, the effect of the number of hidden nodes is evident. Figure 4.5 shows that for the Tesla GPU, the training time does not noticeably increase until 4000 iterations. This is because the Tesla GPU has 14 multiprocessors, and the GPU can hold 14 blocks at a time. The point with 3000 training events is divided into 12 blocks. Therefore, the points from 1000, 2000, and 3000 training events have not utilized the entire occupancy of the GPU. At 4000 training events, the data are divided into 16 blocks. At this point, two blocks must wait until there is available space on the GPU to execute. This is seen in the figure by the jump in execution time at 4000 iterations.

In the case of increasing number of hidden nodes, the effect of the increased clock speed of the Tesla GPU can be seen. There is a large difference between the execution time of the GPUs in the case of increasing the computation time and complexity per thread.

The results of the pMSSM study indicate that that the GPU is very well suited to the training of BNNs in cases of large amounts of training data and complex neural networks. This is a good method to better understand the higher dimensional models within the MSSM.

4.4 Using the GPU BNN Program

Since this program was written with the intention of use by many people, I would like to comment on using the program from a user's prospective. I wrote the GPU implementation for constructing a BNN keeping in mind the target group of users: physicists. This program is broadly applicable to those outside of the high energy physics community, but the setup and use of the program should be very comfortable for physicists. The only GPU knowledge that the user must obtain is 1) that they have a CUDA-enabled GPU, and 2) that they have the NVIDIA Toolkit and `nvcc` compiler installed. To use the BNN libraries, the user must do the following.

1. Include the BNN library of interest. In this work, we considered regression. This is given by `BNN_Regression`.
2. Create a `main()` function.
3. Load all of the information necessary for the BNN. These include
   (a) the number of inputs, I,
   (b) the number of training events to use, $nTrain$,
   (c) the number of burn-in iterations, the number of output networks, and how often to write out a network, $nBurn$, $nOut$, $nEvery$,
   (d) the training data, x,
   (e) the training data targets, t,
   (f) the number of hidden nodes in the network, H,
   (g) the tuning parameters L and ϵ,
   (h) the per-event weight, if necessary, w,
   (i) the output file name for the resultant networks, `file`.
4. Instantiate the class `BNN_Regresssion BNN(I,nTrain,nBurn,nOut,nEvery,x,t,H,L,`ϵ`,w,file)`
5. Train the BNN, `BNN.run()`

Through testing, it is clear that the algorithm converges much faster if the input data and targets are normalized to have a mean of zero and standard deviation of one. This is most likely due to the BNN tuning parameters, L and ϵ being much easier to optimize with normalized values.

There are two features that cater to the high energy physicists. The first is that the program works with ROOT, CERN's data analysis framework. Also, ROOT is written in C++, so the ROOT users are already familiar with the language environment necessary for this program. There is an optional function to load in data event by event directly from the ROOT file, saving the user time[2]. The second is the inclusion of event weights for the training data. Often, in high energy physics, each simulated event is weighted and this information should be taken into account when calculating the negative log likelihood,

$$-\ln \mathcal{L}(\omega|x,t) = \frac{1}{2\sigma^2}\sum_{n=1}^{N} w_n (f(x_n,\omega) - t_n)^2, \tag{4.10}$$

[2]This capability currently exists in the code, but is not yet thoroughly tested.

where the w_n represents the weight for each event. The BNN software options available now do not currently have support for weighted training data. Weighted training data are useful beyond the scope of physics as well.

This program was designed for ease of use for those familiar with the C++ programming environment. See Appendix B for an example of a user-defined main function using the BNN software.

CHAPTER 5

SUMMARY

A computer program was created that utilizes the GPU to construct a BNN for regression and classification in cases where a large amount of training data are necessary to generate the mapping from input variables to targets. The intent of this program is to encourage the use of BNNs in cases where computation time was a hinderance for routine use. High energy physicists have an immediate use for this program in their ongoing study of the pMSSM. The pMSSM is a supersymmetric theory that encapsulates most of the physics of the MSSM, which is of great interest to the physics community. The MSSM is a minimal extension to the Standard Model through the addition of supersymmetry which associates a supersymmetric particle with every particle of the Standard Model. The Standard Model and supersymmetric particles are listed in Table 1.1. However, many variables associated with superparticles are unknown, making the search for them extremely difficult. Scientists are not currently able to search the vast MSSM parameter space in order to compare predictions to theory. Instead, highly constrained versions of the MSSM are studied experimentally. Recently, various results which all but ruled out a highly-constrained version of the MSSM, the CMSSM [11]. It was subsequently shown that the elimination of the CMSSM does not refute the MSSM [11]. The pMSSM is a much better proxy for the MSSM that encapsulates much of the physics of the MSSM. The pMSSM has many more parameters than the constrained models in use, but far fewer than the large MSSM. The pMSSM is still computationally intensive, and better ways of studying the parameter space are sought.

This work proposes the use of BNNs to construct a mapping from the pMSSM parameters to their predictions using a Bayesian approach. In this approach one constructs a posterior probability, $p(\omega|x,t)$ of the neural network parameters given the training data. Then, the mapping is estimated by

$$\bar{f}(x'|x,t) = \int f(x',\omega)p(\omega|x,t)d\omega. \tag{5.1}$$

Equation 5.1 is approximated using the HMC method.

The HMC method moves through the neural network parameter space in a deterministic fashion before proposing a new point. The deterministic path is calculated by treating the traversal as the path of a particle moving through space. The path is determined by Hamilton's equations, with the position as the point in the parameter space and the momentum as an independent variable [26]. The leapfrog method is used to discretize Hamilton's equations. There is zero error leapfrog method. This is because the method is reversible and therefore conserves energy. Error is accumulated due to the approximation of the gradient. HMC requires the calculation of the derivative of the log likelihood of the posterior distribution, which increases the computation time per iteration substantially. However, with the avoidance of random walks, the chain converges to the posterior distribution much faster than other MCMC methods.

In order to reduce the computation time for cases, such as the pMSSM, that require many training events, a GPU implementation for the construction of BNNs is studied. The GPU is well suited for computationally intensive, data parallel applications. The computational bottleneck in the BNN algorithm with a large number of training points is in the calculation of the potential energy in the HMC algorithm. The potential energy $U(\omega)$ is given by the negative log likelihood

$$-\log\left(\mathcal{L}(\omega|x,t)p(\omega)\right) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(f(x_n,w)-t_n)^2. \tag{5.2}$$

This requires a sum over all of the training data, which requires most of the computation in the algorithm. This sum is parallelized on the GPU, with each thread evaluating of the network for one training point. NVIDIA's CUDA C extension is used for the GPU parallelization.

The implementation was tested systematically using a test problem namely, modeling the invariant mass of one of the Z bosons in the $pp \to ZZ \to 4\ell$ decay channel. The time of the BNN training was studied using a varying number of hidden nodes and number of training points. The results displayed in Figs. 4.2 and 4.3 show a minimum speedup of the GPU over the CPU by a factor of two. The figures also show that the largest speedup is achieved when the number of hidden nodes increases. This is due to the increased computation time per node.

The results indicate that the GPU is a good resource for the construction of BNNs with many hidden nodes and a large number of training events. The results indicate that a GPU implementation of BNNs is a good method to study the pMSSM. Once the BNN is constructed, the evaluation is quick, and therefore the study of the parameter space becomes much easier.

In this dissertation, I fully implemented the GPU and CPU BNN algorithms from scratch. I developed them with the intention that they be used by users interested in a wide range of BNN applications. Features of my implementation are discussed in Section 4.4.

5.1 Future Work

There are many improvements to this algorithm that I would like to implement. The primary improvement is an optimization of the HMC algorithm. The linear version of this code does not currently run as fast as the highly-optimized (`fbm`) program. The relative speedups between the CPU and GPU are expected to scale similarly to the results given in Section 4.1 because the speedup is attained through the calculation of $U(\omega)$. Reducing the number of times $U(\omega)$ is called each HMC iteration would reduce the execution time considerably. In addition, I would like to implement an adaptive tuning of the HMC parameters L and ϵ to eliminate the need to search for optimal values by running the algorithm many times.

There are further optimizations to improve the performance of the GPU implementation. Two methods that I would like to study in detail are 1) the use of shared GPU memory for accessing the training data on the GPU and 2) the use of pitched memory, or texture memory to store the data on the GPU.

In addition, it would be extremely interesting to compare GPU performance using the current CUDA programming standard, CUDA 5.5 against the new CUDA programming model, CUDA 6. There seems to be a lot of promising improvements in CUDA 6. Without doubt, the new method of memory management will provide much cleaner code and fewer bugs.

APPENDIX A

GPU IMPLEMENTATION OF BNN

Listing A.1: A simple CUDA implementation of the BNN algorithm.

```
/** device code **/
__global__ void diff(HMC_type* q, HMC_type* x, HMC_type* w, HMC_type* t,
                     HMC_type* d, int H, int I) {
  int th = threadIdx.x + blockIdx.x*blockDim.x;
  HMC_type f = q[0];
  for(int j=0;j<H;j++){
    HMC_type inSum = 0.0;
    for(int i=0;i<I;i++){
      inSum += q[2*H+1+I*j+i]*x[th*I+i];
    }
    f+=q[j+1]*tanh(q[H+1+j]+inSum);
  }
  d[th] = w[th]*(t[th]-f)*(t[th]-f);
};

/** host code **/
/* allocate device memory in constructor and transfer
   training data to device */
BNN_regression::BNN_regression(int l, int nOut_, int nRep_, int nBurn_,
                               int h, int inp, std::vector<HMC_type> &data,
                               std::vector<HMC_type> &weights,
                               std::vector<HMC_type> &targets, std::string s)
  : HMC_base(l, 1+h*(2+inp), nOut_, nRep_, nBurn_),
    v_x(data),
    H(h),
    I(inp),
    N(targets.size()),
    sig(1),
    v_w(weights),
    v_t(targets),
    oFile(s),
    sigb(100),
    sigv(1.65),
    siga(0.86),
    sigu(0.54)
{
```

```
  np = getNP();
  of.open(oFile.c_str(),std::ofstream::out);
  of << I << "\t" << H << std::endl;

  /** memory allocation on GPU **/
  //size in bytes of the NN parameter vector
  size_np = np*sizeof(HMC_type);
  // size in bytes of the number of training event
  size_n = N*sizeof(HMC_type);
  /* point vectors to arrays */
  x = &v_x[0];
  t = &v_t[0];
  w = &v_w[0];

  /** allocate memory on device **/
  /** allocate data x **/
  dev_x = new HMC_type[N*I];
  cudaMalloc(&dev_x,size_n*I);
  cudaCheckError("cudaMalloc dev_x");
  /** allocate dev targets t **/
  dev_t = new HMC_type[N];
  cudaMalloc(&dev_t,size_n);
  /** allocate dev weights w **/
  dev_w = new HMC_type[N];
  cudaMalloc(&dev_w,size_n);
  /** allocate NN parameters q **/
  dev_q = new HMC_type[np];
  cudaMalloc(&dev_q,size_np);
  /** allocate diff array **/
  dev_d = new HMC_type[N];
  cudaMalloc(&dev_d,size_n);

  /** copying memory from CPU to GPU **/
  cudaMemcpy(dev_x, x, size_n*I, cudaMemcpyHostToDevice);
  cudaCheckError("cudaMemcpy dev_x");
  cudaMemcpy(dev_t, t, size_n, cudaMemcpyHostToDevice);
  cudaCheckError("cudaMemcpy dev_t");
  cudaMemcpy(dev_w, w, size_n, cudaMemcpyHostToDevice);
  cudaCheckError("cudaMemcpy dev_w");
};
/* Destructor */
BNN_regression::~BNN_regression() {
  /*  Free device memory */
  cudaFree(dev_x);
  cudaFree(dev_t);
```

```
  cudaFree(dev_w);
  cudaFree(dev_q);
  cudaFree(dev_d);
};

inline HMC_type BNN_regression::U(std::vector<HMC_type> &in_q) {
  HMC_type* qq = &in_q[0];
  /* define threads per block and calculate blocks per grid */
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

  cudaMemcpy(dev_q, qq, size_np, cudaMemcpyHostToDevice);
  cudaCheckError("cudaMemcpy dev_q");
  /* point thrust variable to vector on device memory */
  thrust::device_ptr<HMC_type> d_dev_ptr(dev_d);
  /* Call device kernals */
  diff<<<blocksPerGrid,threadsPerBlock>>>(dev_q, dev_x, dev_w,
                                          dev_t, dev_d, H, I);
  /* call CUDA thrust library for a parallel
     reduction of diff results */
  HMC_type sum = thrust::reduce(d_dev_ptr,d_dev_ptr+N);
  return sum/(2*sig*sig) + LnPrior(in_q);

};
/* Definition of prior for NN parameters */
inline HMC_type BNN_regression::LnPrior(std::vector<HMC_type> &q) {
  HMC_type prior = 0.0;
  prior += 0.5*q[0]*q[0]/(sigb*sigb);
  for (int v=1; v<H+1;v++){
    prior +=  0.5*q[v]*q[v]/(sigv*sigv);
  }
  for(int a=H+1;a<2*H+1;a++){
    prior +=  0.5*q[a]*q[a]/(siga*siga);
  }
  for(int u=2*H+1; u<H*(2+I)+1; u++){
    prior +=  0.5*q[u]*q[u]/(sigu*sigu);
  }
  return prior;
};
/* HMC parent class */
inline std::vector<HMC_type> HMC_base::it(std::vector<HMC_type> &q0) {
  std::vector<HMC_type> eps(NP);
  Uq0 = U(q0);
   for(int i=0;i<NP;i++){
     eps[i] = 0.02;
```

```
 }
std::vector<HMC_type> dU(NP);
q = q0;

assert((int)p.size() == NP);

for(int i=0;i<NP;i++) {
    p[i] = rand_p.Gaus();
}
std::vector<HMC_type> p0 = p;
// 1/2 step in momentum
dU = delU(q);
for(int i=0; i<NP;i++){
  p[i] = p[i] - 0.5*eps[i]*dU[i];
  if(p[i] != p[i]){
    assert(0);
  }
}

for(int i=0;i<L;i++){
  // 1 step in position
  for(int j=0;j<NP;j++) {
    q[j] = q[j] + eps[j]*p[j];
  }
  dU = delU(q);
  // 1 step in momentum, except at the end
  if(i != (L-1)){
    for(int j=0;j<NP;j++) {
      p[j] = p[j] - eps[j]*dU[j];
      if(dU[j] != dU[j]){
         std::cout << "dU[" << j << "] = " << dU[j] << std::endl;
        assert(0);
      }
    }
  }
}
// 1/2 step in momentum
dU = delU(q);
for(int i=0; i<NP;i++){
  p[i] = p[i] - 0.5*eps[i]*dU[i];
  if(p[i] != p[i]){
    assert(0);
  }
}
// to symmetrize momentum
```

```
  for(int i=0; i<NP;i++){
    p[i] = -p[i];
    if(p[i] != p[i]){
      assert(0);
    }
  }
  // Calculate original and new U & K
  HMC_type Uc = U(q0);
  HMC_type Un = U(q);
  HMC_type Kc = 0;
  for(int i=0;i<NP;i++){
    Kc += 0.5*p0[i]*p0[i];
  }
  HMC_type Kn = 0;
  for(int i=0;i<NP;i++){
    Kn += 0.5*p[i]*p[i];
  }
  // accept or reject?
  HMC_type f = exp(Uc-Un+Kc-Kn);
  HMC_type rndm =  (HMC_type)rand()/(HMC_type)RAND_MAX;
  if(rndm < f) {
    return q;   //accept
  } else {
    return q0; //reject
  }
};
int HMC_base::getNP() {
  return NP;
};
void HMC_base::run(){
  int nIter = nBurn + nRep*nOut;
  for(int i=0;i<NP; i++){
    q[i] = rand_p.Gaus();
  }
  int accept = 0;
  float acc_ratio = 0.0;
  for(int i=0; i<nIter; i++){
    q0 = q;
    q = it(q0);
    if(q != q0) {
      accept++;
    }
  }
};
```

APPENDIX B

EXAMPLE OF USER-DEFINED MAIN FUNCTION USING BNN SOFTWARE

Listing B.1: An example of how a user uses the BNN training software developed in this dissertation. Note that the user does not have to do any GPU coding.

```
#include <fstream>
#include <sstream>
#include <string>

#include "BNN_regression.cuh"

using namespace std;


int main() {
  // define HMC variables
  int nBurn = 50;
  int nOut = 100;
  int nRep = 10;
  int L = 100;
  // define BNN paramters
  int dim = 19;
  int H = 10;

  // store training data into vectors or arrays
  string filename = "data/pMSSM_train_t1.dat";

  string line;
  ifstream datafile(filename.c_str());
  if(!datafile){
    cout << "ERROR: data file not found" << endl;
  }

  // number of training events used
  int nTrain = 2000;
  vector<float> x(dim*nTrain);

  int row = 0;
  int ind = -10;
```

```
  string dummy;
  getline(datafile, dummy);

  // targets and weights
  vector<float> t(nTrain);
  vector<float> w(nTrain);

  while(datafile.good() && row < nTrain){
    getline(datafile,line);
    stringstream ss(stringstream::in | stringstream::out);
    ss << line;
    for(int j=0; j<dim;j++) {
      ind = (dim)*row+j;
      ss >> x[ind];
    }
    ss >> t[row];
    w[row] = 1;
    row++;
  }
  datafile.close();

  // network results file
  string outFile = "results/pMSSM_3_23_H.txt";

  // instantiate a BNN regression class
  BNN_regression BNN(L, nOut, nRep, nBurn, h, dim, x, w, t, outFile);
  // run the BNN training
  BNN.run();

  return 0;
}
```

REFERENCES

[1] S. P. Martin, *A Supersymmetry Primer*. 1997.

[2] NVIDIA, "Notebook GPUs," 2014.

[3] Apple, Inc., "Macbook pro," March 2014.

[4] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[5] M. Gell-Mann, "A schematic model of baryons and mesons," *Physics Letters*, vol. 8, no. 3, pp. 214 – 215, 1964.

[6] G. Zweig, "An SU(3) model for strong interaction symmetry and its breaking. Version 2," 1964.

[7] E. Riordan, "The discovery of quarks," *Science*, vol. 256, pp. 1287–1293, 1992.

[8] Super-Kamiokande Collaboration, "Evidence for oscillation of atmospheric neutrinos," *Phys. Rev. Lett.*, vol. 81, pp. 1562–1567, Aug 1998.

[9] SNO Collaboration, "Measurement of the rate of v+d→p+p+e interactions produced by b8 solar neutrinos at the sudbury neutrino observatory," *Phys. Rev. Lett.*, vol. 87, p. 071301, Jul 2001.

[10] J. Ellis, "Outstanding questions: physics beyond the standard model," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 370, no. 1961, pp. 818–830, 2012.

[11] CMS Collaboration, "Phenomenological MSSM interpretation of the CMS 2011 $5fb^{-1}$ results," CMS Physics Analysis Summary CMS-PAS-SUS-12-030, 2013.

[12] A. Strubig, S. Caron, and M. Rammensee, "Constraints on the pMSSM from searches for squarks and gluinos by ATLAS," *JHEP*, vol. 1205, p. 150, 2012.

[13] J. Ellis, T. Falk, G. Ganis, K. A. Olive, and M. Srednicki, "The CMSSM parameter space at large $\tan\beta$," *Physics Letters B*, vol. 510, no. 1-4, pp. 236 – 246, 2001.

[14] E. Dudas, Y. Mambrini, A. Mustafayev, and K. Olive, "Relating the CMSSM and SUGRA models with GUT-scale and super-GUT-scale supersymmetry breaking," *The European Physical Journal C*, vol. 72, no. 9, pp. 1–17, 2012.

[15] C. F. Berger, J. S. Gainer, J. L. Hewett, and T. G. Rizzo, "Supersymmetry Without Prejudice," *JHEP*, vol. 0902, p. 023, 2009.

[16] D0 Collaboration, "Observation of single top-quark production," *Phys. Rev. Lett.*, vol. 103, p. 092001, Aug 2009.

[17] T. Aaltonen *et al.*, "First Observation of Electroweak Single Top Quark Production," *Phys.Rev.Lett.*, vol. 103, p. 092002, 2009.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 ed., 2007.

[19] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 3 ed., 2007.

[21] J. Lampinen and A. Vehtari, "Bayesian approach for neural networks – review and case studies," *Neural networks*, vol. 14, no. 3, pp. 257–274, 2001.

[22] P. C. Bhat and H. B. Prosper, "Bayesian neural networks," 2006.

[23] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[24] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[25] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid monte carlo," *Physics Letters B*, vol. 195, no. 2, pp. 216 – 222, 1987.

[26] R. M. Neal, *MCMC using Hamiltonian dynamics*. Chapman Hall CRC Press, 2004.

[27] M. Creutz, "Global Monte Carlo algorithms for many-fermion systems," *Phys. Rev. D*, vol. 38, pp. 1228–1238, Aug 1988.

[28] NVIDIA, "NVIDIA unveils CUDA-the GPU computing revolution begins," press release, 2006.

[29] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[30] R. Basmadjian and H. De Meer, "Evaluating and modeling power consumption of multi-core

processors," in *Proc. of the 3rd Int'l Conf. on Future Energy Systems (e-Energy 2012)*, ACM, 2012.

[31] M. Arora, "The architecture and evolution of cpu-gpu systems for general purpose computing," 2012.

[32] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[33] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," *GPU Computing Gems*, vol. Jade Edition, 2011.

[34] T. Sjostrand, S. Mrenna, and P. Z. Skands, "A Brief Introduction to PYTHIA 8.1," *Comput.Phys.Commun.*, vol. 178, pp. 852–867, 2008.

BIOGRAPHICAL SKETCH

Michelle was born in Winter Park, FL and grew up in Pittsburgh, Pennsylvania. She returned to Florida to attend Florida State University for her Bachelor's degree, where she continued her graduate work.