# GPU Implementation of Bayesian Neural Networks in Data-Intensive Applications

Michelle E. Perry, Dr. Anke Meyer-Baese, Dr. Harrison Prosper*
Florida State University, Department of Scientific Computing, *Department of Physics

## Introduction

The search for new physics has entered an era in which more general models, such as the 19-parameter phenomenological minimal supersymmetric standard model (pMSSM), have been used to interpret data at the Large Hadron Collider. Unfortunately, due to the complexity of the calculations, the predictions of these models are available only at a discrete set of parameter points. It would be useful, however, to have a computationally routine way to construct smooth mappings of the model parameters to the model predictions. We propose to construct the mappings using Bayesian neural networks (BNN). The main limitation to the widespread use of BNNs is the time required to construct them. We describe ongoing work to implement the construction of BNNs using the Hybrid Markov Chain Monte Carlo (MCMC) method using Graphical Processing Units (GPUs).

## Bayesian Neural Networks

Neural networks (NNs) are non-linear "universal approximators" that can model *any* smooth mapping of the form $f : \mathbb{R}^n \to \mathbb{R}^m$, with $m < n$ [1]. In this work, we focus on approximations to functions with $m = \mathbf{1}$, based on the single hidden layer NN model,

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

where $\omega$ 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.
In a traditional NN, the goal is to find a single set of parameters, $\omega^*$, such that $f(x, \omega^*)$ approximates the desired mapping. For BNN, the training of the network is treated as an inference problem that yields a probability density $P(\omega|T)$ over the network parameter space, where $T$ denotes the training data, $\{(t, x)_k\}$, $k = \mathbf{1}, \cdots, N$ and $t_k$ is the known "target" value associated with the input vector $x_k = (x_1, \cdots, x_I)_k$. The desired mapping is approximated by the average

$$BNN(x) = \int f(x, \omega)\, P(\omega|T)\, d\omega, \tag{2}$$

$$\approx \frac{1}{K} \sum_{j=1}^{K} f(x, \omega_j), \tag{3}$$

where the set $\{\omega_j\}$ is sampled from the posterior density

$$P(\omega|T) \propto \exp\left[-\frac{1}{2\sigma^2} \sum_{k=1}^{N} (t_k - f(x_k, \omega))^2\right] \cdot \mathtt{prior}(\omega). \tag{4}$$

Figure 1 shows a simple 2-D example of such a calculation.

## Hybrid Monte Carlo

Due to the complicated multidimensional domain of the NN, we use the Hybrid Monte Carlo (HMC) algorithm to sample the posterior density $P(\omega|T)$ [2]. This Markov Chain Monte Carlo (MCMC) algorithm treats the parameter space as a Hamiltonian system, with "potential energy"

$$U(\omega) = -\ln P(\omega|T) \tag{5}$$

and "kinetic energy"

$$K = \frac{1}{2} \sum_{i=1}^{N} p_i^2. \tag{6}$$

```
HMC_iteration(q0) {
  q = q0;
  p = norm_dist();
  p = p - 0.5*eps*delU(q); // half step in momentum
  for(int i=0; i<L; i++){  // do L steps
    q = q + eps * p; //step in q
    //step in p, except at last step
    if(i != L-1)
      p = p - eps * delU(q);
  }
  p = p - 0.5*eps*delU(q); // half step in momentum
  // calculate Hamiltonian for old and new points
  H0 = U(q0) + K(p0);
  H = U(q) + K(q);
  //decide which point to keep
  if(rand < exp(H - H0)){
    return q; //accept
  } else {
    return q0; //reject
  }
}
```

## BNN Algorithm

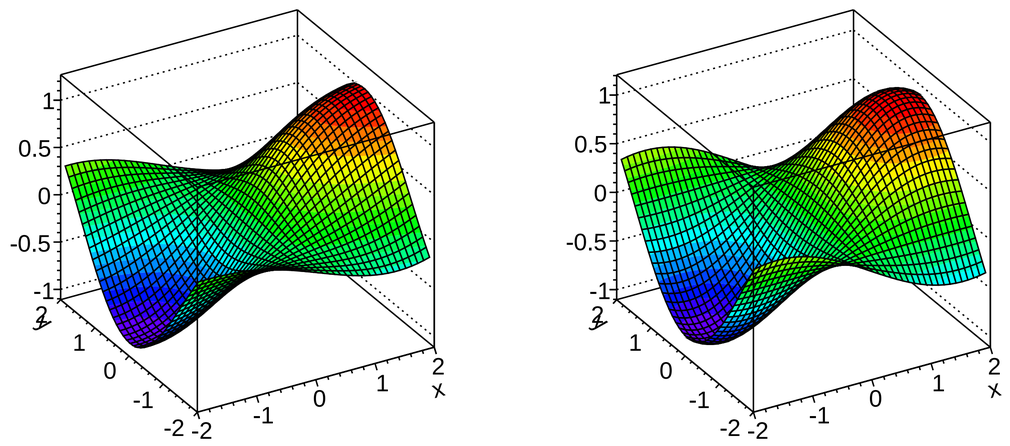


Figure 1 : The left is a BNN function trained with targets of the form z = sin(x)cos(y) and a discrete set of points (x,y) sampled from a uniform distribution. $N = \mathbf{1000}$, $H = \mathbf{5}$, and $I = \mathbf{2}$, and 400 HMC iterations. The right is a plot of $sin(x)cos(y)$.

In the HMC algorithm, the calculation of the potential energy $U$ is the most time consuming. As the amount of training data, number of inputs, and number of hidden nodes increase, the evaluation time of $U$ increases. For applications in high energy physics, such as those testing multi-parameter physics models, a faster implementation of BNNs is desirable. Each term in the sum in Eq. (4) is independent and data parallel, and with training sets of over $\mathbf{10^6}$ events, these applications are ideal for GPUs. For these applications, our approach is to run the HMC algorithm on the CPU but evaluate the sum on the GPU.

## GPU Implementation

The GPU is a processor developed for graphics rendering, which devotes more processors to data processing than does a CPU, but fewer to data flow and caching, as displayed in Fig. 1. Therefore, GPUs are ideal for compute intensive, highly parallel, applications such as the construction of BNNs.

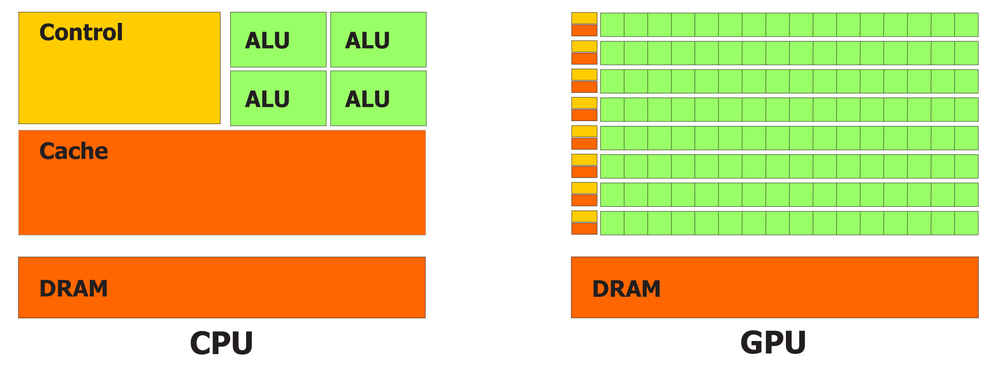


Figure 2 : Comparison of CPU to GPU. Figure courtesy of NVIDIA.

In this work, we use the CUDA C Programming language for NVIDIA GPUs. Each term in the sum in Eq. (4) is computed in one thread. The dynamic nature of CUDA allows the threads to be distributed to available CUDA cores as efficiently as possible. The more cores that are available on the GPU — a typical GPU has 256, the greater the degree of parallelization. A parallel reduction of the results from each thread provides additional speedup. Our goal is to reduce the time required to train multivariate BNNs by two orders of magnitude for applications with $> \mathbf{20}$ inputs and $\mathbf{10^6}$ training events. Preliminary results indicate speedup of 10–50$\times$ the serial implementation. We plan to implement further optimizations. The calculation of smooth prediction functions for the phenomenological Minimal Supersymmetric Standard Model is an application currently being explored using a GPU BNN application.

## References

[1] Hornik, K. (1991). *Approximation Capabilities of Multilayer Feedforward Networks*.
[2] Neal, R. M. (2004). *MCMC using Hamiltonian dynamics*. Chapman Hall CRC