# TensorBNN: Bayesian Inference for Neural Networks using Tensorflow

B.S. Kronheim[a], M.P. Kuchera[a], H.B. Prosper[b]

[a]*Department of Physics, Davidson College, Davidson, NC 28036*
[b]*Department of Physics, Florida State University, Tallahassee, FL 32306*

## Abstract

`TensorBNN` is a new package based on `TensorFlow` that implements Bayesian inference for modern neural network models. The posterior density of neural network model parameters is represented as a point cloud sampled using Hamiltonian Monte Carlo. The `TensorBNN` package leverages `TensorFlow`'s architecture and training features as well as its ability to use modern graphics processing units (GPU) in both the training and prediction stages.

*Keywords:* Bayesian Neural Networks, Machine Learning, `TensorFlow`, Hamiltonian Monte Carlo

## 1. Introduction

`TensorBNN` is a flexible implementation of Bayesian Neural Networks (BNNs) that uses the efficient GPU algorithms and machine learning platform of `TensorFlow`, extending the package to allow for a fully Bayesian treatment of neural networks. `TensorFlow Probability` contains probabilistic architectures, but does not currently contain support for a Bayesian inference procedure.

The Flexible Bayesian Modeling (`FBM`) toolkit, developed by Radford Neal [1], provides extensive capabilities for Bayesian inference for neural networks. However, machine learning technologies have evolved significantly since the first release of `FBM`. Robust, flexible machine learning platforms such

*Email addresses:* `brkronheim@davidson.edu` (B.S. Kronheim), `mikuchera@davidson.edu` (M.P. Kuchera)

as Google's `TensorFlow` [2] and Facebook's `PyTorch` [3] contain functionality in architecture design and optimization methods that are unmatched in frameworks with smaller user-bases. In addition, these packages provide a seamless interface with Graphics Processing Units (GPUs), enabling large-scale computations with orders of magnitude speedup over CPU-only software. The `TensorBNN` package leverages these recent developments in order to provide an environment for Bayesian inference using the methods proposed by Neal [1].

Implementations of approximate BNNs, which can be run on GPUs, such as the `DenseFlipout` layers in `TensorFlow Probability` based on [4], have recently appeared. These `DenseFlipout` layers approximate the prior and posterior densities with explicit functional forms, such as a Gaussian density, and optimize the parameters using gradient descent. While such methods can be effective and are much less computationally expensive than a full Bayesian analysis, they are limited by the choice of the functional form of the posterior densities and, therefore, will be unable to model as complex a posterior density as that over the parameter space of a neural network.

The `TensorBNN` package approximates the posterior density of the parameters of a neural network as a point cloud, that is, a neural network model is "trained" not by finding a single neural network that best fits the training data, but rather by creating an *ensemble* of neural networks by sampling their parameters from the posterior density using a Markov chain Monte Carlo (MCMC) method.

The paper is organized as follows. We begin in Section 2 with a description of the salient mathematical underpinning of BNNs and `TensorBNN`, in particular. This is followed by a description of the `TensorBNN` package in Section 3. In Section 4 the performance of `TensorBNN` is illustrated with a simple example. A summary is given in Section 5.

## 2. Bayesian neural networks

### *2.1. Mathematical Details*

Neural networks are the most commonly used *supervised* machine learning models in which the training data contains both inputs and known outputs from which a regression or classification model is constructed. The standard approach to training such a model, that is, fitting the model to data, is to minimize a suitable empirical risk function, which in practice is proportional to the average of a loss function. If the average loss is a linear function of the negative log likelihood, $-\ln p(D \mid \theta)$ of the data $D$, then minimizing the average loss is identical to estimation of the neural network parameters $\theta$ via maximum likelihood. In this case, fitting a model, that is, a function $f(\theta)$, to data can be construed as a problem of *inference*. Furthermore, if a prior density $\pi(\theta)$ over the parameter space can be defined, then the inference can be performed using the Bayesian approach [1]. A Bayesian neural network can be represented as follows

$$p(x, D) = \int F(x, \theta)\, p(\theta \mid D)\, d\theta, \tag{1}$$

where

$$p(\theta \mid D) = \frac{p(D \mid \theta)\, \pi(\theta)}{p(D)}, \tag{2}$$

is the posterior density over the parameter space $\Theta$ of the neural network with parameters $\theta \in \Theta$. Since each point $\theta$ corresponds to a different neural network (from the same function class), each point in general is associated with a different output value $y = f(x, \theta)$ for the same input $x$. Therefore, the posterior density and neural network $f(x, \theta)$ together induce a predictive distribution given by

$$p(y \mid x, D) = \int \delta(y - f(x, \theta))\, p(\theta \mid D)\, d\theta. \tag{3}$$

The quantity $D = \{(t_k, x_k)\}$ denotes the training data, which consists of the targets $t_k$ associated with data $x_k$. In practice, the posterior density is represented by a point cloud $\{\theta_i\}$ sampled from the posterior density and Eq. (3) is approximated by binning the values $y = f(x, \theta)$ with $\theta \in \{\theta_i\}$.

A practical advantage of the sampling approach is that it provides a straightforward method to estimate the uncertainty in any quantity that depends on the network parameters. This advantage, of course, exists for the maximum likelihood approach also.

However, for likelihood functions of the neural network parameters, it is unclear, given their highly non-Gaussian character, how to obtain meaningful uncertainty estimates. On the other hand, the Bayesian approach requires the specification of a prior, which introduces its own complication. But, since the network parameters are sampled from the posterior density, the sensitivity of inferences to the choice of prior can be assessed by re-weighting the sampled points by the ratio of the new prior to the one with which the sample was generated.

The the current version of `TensorBNN` is restricted to fully-connected deep neural networks (DNN),

$$y(\mathbf{x}, \theta) = f(\mathbf{b}_K + \mathbf{w}_K\, h_K(\cdots + h_1(\,\mathbf{b}_1 + \mathbf{w}_1\, h_0(\mathbf{b}_0 + \mathbf{w}_0\, \mathbf{x})\,)\cdots), \tag{4}$$

which are simply nested non-linear functions. The quantities $\mathbf{b}_j$ and $\mathbf{w}_j$—the biases and weights—are matrices of parameters and $h_j$ and $f$ are the activation and output functions, respectively; $j$ is the layer number. In Eq. (4), the functions $h_j$ and $f$ are applied element-wise to their matrix arguments. `TensorBNN` maintains the standard activation function options available in `TensorFlow`, with the addition of one custom activation function, a modified `PReLU` function, called `SquarePReLU`

$$h_j(x) = \begin{cases} x & \text{for } x \geq 0 \\ a_j^2 x & \text{otherwise,} \end{cases} \tag{5}$$

with the slope parameter $a_j^2$ for the negative $x$ region. We opted to take the parameter to be $a_j$ rather than $a_j^2$ to ensure that the slope remains positive thereby keeping the activation function one-to-one. The function $f$ is

$$f(x) = \begin{cases} 1/(1 + \exp(-x)) & \text{for a classifier and} \\ x & \text{for regression models.} \end{cases} \tag{6}$$

The weights and biases of the network along with the `PReLU` parameters are the free parameters of the neural network model.

In the following subsection, we describe the pertinent details of this implementation, while the second subsection contains relevant information on the hardware used to perform the sampling for the examples described in Section 4.

### *2.2. Likelihood and Prior*

*Likelihood.* The likelihood functions included in the package are modeled as follows:

$$p(D\,|\,\theta) = \begin{cases} \prod_k y_k^{t_k}\,(1 - y_k)^{1-t_k} & \text{for classifiers and} \\ \prod_k \mathrm{N}(t_k, y_k, \sigma) & \text{for the regression models,} \end{cases}$$
$$\text{where } y_k \equiv y(x_k, \theta) \text{ and } \mathrm{N}(x, \mu, \sigma) \text{ is a Gaussian density.} \tag{7}$$

For classifiers, the targets $t_k$ are 1 and 0 for true and false identification, while for regression models they are the desired regression values.

*Prior.* Constructing a prior for a multi-parameter likelihood function is exceedingly challenging and is especially so for functions as complicated as those in Eq. (7). For an excellent and thorough review of this problem we recommend Nalisnick [5]). We sidestep this challenge by proceeding pragmatically and relying instead on *ex post facto* justification of the choices we make: the choices are justified based on the quality of the results. Moreover, `TensorBNN` includes a re-weighting mechanism for studying the sensitivity of the results to these choices.

Each layer of the DNN contains three kinds of parameter: weights, biases, and the slope parameters of the activation functions. In the `FBM` toolkit of Neal [1], the prior for each weight and bias is a zero mean Gaussian density, with the precisions, $\sigma^{-2}$, of these densities constrained by gamma hyper-priors. In `TensorBNN`, the weights $w_i$ of a given layer are assigned the hierarchical prior

$$\begin{aligned}\pi(\mathbf{w}) = &\left(\prod_i \pi^{-1}\beta^{-1}\left[1+\left(\frac{w_i-\alpha}{\beta}\right)^2\right]^{-1}\right) && \text{(prior)}\\ &\times N(\alpha,\mu_\alpha,\sigma_\alpha)\,N(\beta,\mu_\beta,\sigma_\beta) && \text{(hyper-prior)},\end{aligned} \quad (8)$$

comprising a product of Cauchy densities with location and scale parameters, $\alpha$ and $\beta$, respectively, constrained by hyper-priors modeled as Gaussian densities. A hierarchical prior of the same form is assigned to the biases of a given layer, but with different location and scale parameters. The prior for the slope parameter of the activation function can be either an exponential density with its rate parameter constrained by an exponential hyper-prior

$$\pi(m) = \lambda\exp(-\lambda m)\,\gamma_\lambda\exp(-\gamma_\lambda\lambda), \quad (9)$$

or a Gaussian prior with location and scale parameters constrained by Gaussian density hyper-priors as follows

$$\pi(m) = N(m,\gamma,\omega)\,N(\gamma,\mu_\gamma,\sigma_\gamma)\,N(\omega,\mu_\omega,\sigma_\omega). \quad (10)$$

The overall prior $\pi(\theta)$ is a product of the priors for all weights, biases, and slope parameters, and the associated hyper-priors. For regression models, there is, in addition, a flat prior for the standard deviation $\sigma$ in Eq. (7), with a starting value set by the user.

### *2.3. Sampling the posterior density*

Since the high-dimensional integral in Eq. (1) is intractable it is typically approximated using a Monte Carlo method to sample from the posterior density $p(\theta \mid D)$. The Monte Carlo method of choice is Hamiltonian Monte Carlo (HMC) [1, 6] in which the posterior density is written as

$$p(\theta \mid D) = \exp(-V(\theta)),$$

where $V = -\ln p(\theta \mid D)$ is viewed as a "potential" to which a "kinetic" term $T = \mathbf{p}^2/2$ is added to form a "Hamiltonian" $H = T + V$. The dimensionality of the "momentum" $\mathbf{p}$ equals that of the parameter space $\Theta$. The HMC sampling algorithm alternates between deterministic traversals of the space $\Theta$ governed by a finite difference approximation of Hamilton's equations and random changes of direction. In order to achieve detailed balance and, therefore, ensure asymptotic convergence to the correct posterior density, the deterministic trajectories are computed using a reversible, leapfrog approximation, to Hamilton's equations. The HMC method has two free parameters, the step size along the trajectories and the number of steps to take before executing a random change in direction. The method used to determine these parameters is described in Section 3.5.

## 3. Implementation Details

`TensorBNN`, which is built using the `TensorFlow` [2] and `TensorFlow-Probability` [7], follows the design of BNNs described in Neal [1] with some improvements and modifications. Here we summarize the general structure of `TensorBNN` and the improvements, which include the HMC parameter adaptation scheme and the addition of pretraining.

### *3.1. Model Declaration*

`TensorBNN` provides a framework for the user-friendly construction of BNNs in a manner similar to the `Keras` [8] interface for building neural networks with `TensorFlow`. The main object in the package is the `network` object, which is the base for all operations in the package. The options that can be specified when instantiating this object are the data type, e.g. float32 or float64, the training and validation data, and the normalization scaling for the output. An example network declaration is shown below.

```
model = network(dtype,      # Data type to use
                inputDims,  # Length of 1 input
                trainX,     # Training input
                trainY,     # Training output
                validateX,  # Validation input
                validateY)  # Validation output
```

After the `network` object is instantiated, the layers and activation functions are added. Each of these is a variant of the `Layer` object. For example, a `DenseLayer` is included in the package with multiple activation functions, but users can create their own `Layer` variants and activation functions with custom priors.

The `DenseLayer` object can be initialized either randomly or with pre-trained weights and biases. When using random initialization, the Gaussian He [9] initialization is used to determine starting values of the weights, and the biases are extracted from the same random distribution. This is done to keep the starting values small, while allowing variation. As discussed in Section 2.2, the priors for the weights and biases are Cauchy densities, with the location parameter $\alpha$ and scale parameter $\beta$ constrained by a Gaussian hyper-prior (see Eq. (8)). See Table 1 for the parameter values. The $\beta$ parameters are made more flexible than the $\alpha$ values in order to keep a majority of the weights and biases close to 0, while allowing some number of weights and biases to have larger values. These values, which are summarized in Table 1, cannot be changed within a `DenseLayer` object, though it is a simple matter to create a new `Layer` variant.

| **Parameter** | **Value** |
|---|---|
| $\alpha$ | 0 |
| $\beta$ | 0.2 |
| $\mu_\alpha$ | 0 |
| $\sigma_\alpha$ | 0.2 |
| $\mu_\beta$ | 0.5 |
| $\sigma_\beta$ | 0.5 |

Table 1: `DenseLayer` initial values of prior hyper-parameters, $\alpha$ and $\beta$, and the fixed hyper-prior parameters.

In the code snippet below, a `DenseLayer` and an activation function are added to the network. This process would be the same for any layer or activation function with a different object added.

```
model.add(             # Add layer command
        DenseLayer(    # Dense layer object
        inputDims,     # Size of layer input vector
        width,         # Size of layer output vector
        seed=seed,     # Random number seed
        dtype=dtype))  # Layer datatype
model.add(Tanh())      # Hyperbolic tangent activation
```

| **Name** | **Description** |
|---|---|
| `Sigmoid` | $1/(1+e^{-x})$ |
| `Tanh` | $\tanh(x)$ |
| `Softmax` | $e^{x_i}/\sum_n e^{x_n}$ |
| `ReLU` | $\begin{cases} 0 & x \le 0 \\ x & x > 0 \end{cases}$ |
| `leakyReLU` | $\begin{cases} \alpha x & x \le 0 \\ x & x > 0 \end{cases}$ |
| `Elu` | $\begin{cases} e^x - 1 & x \le 0 \\ x & x > 0 \end{cases}$ |
| `PReLU` | leakyReLU with trainable $\alpha$ |
| `SquarePReLU` | $\begin{cases} \alpha^2 x & x \le 0 \\ x & x > 0 \end{cases}$ $\alpha$ is trainable |

Table 2: The built-in activation functions of `TensorBNN`.

Within the package there are eight options for activation functions, which are listed in Table 2. All the activation functions except `SquarePReLU` are standard and present in `TensorFlow`. The `SquarePReLU` was developed specifically for `TensorBNN`. Both `PReLU` and `SquarePReLU` have trainable slope parameters $\alpha$. They have, however, different priors and hyper-priors, given in Eqs. (9) and (10), with the fixed and initial values of their parameters listed in Table 3. For `PReLU`, the exponential distribution was chosen to model the prior belief that the slopes should be close to zero. The rates were picked to allow for larger slopes, while still enforcing the belief that smaller slopes are preferred. For `SquarePReLU`, as we are considering the square root of the

| **Name** | **Parameter** | **Description** |
|---|---|---|
| `leakyReLU` | $m = 0.3$ | fixed slope parameter |
| `PReLU` | $m = 0.2$ | initial slope parameter |
|  | $\lambda = 0.3$ | initial rate parameter for the $m$ prior |
|  | $\gamma_\lambda = 0.3$ | rate parameter for the $\lambda$ hyper-prior |
| `SquarePReLU` | $m = 0.2$ | initial slope parameter |
|  | $\gamma = 0.3$ | initial mean for the $m$ prior |
|  | $\omega = 0.3$ | initial standard deviation for the $m$ prior |
|  | $\mu_\gamma = 0.0$ | mean for the hyper-prior of $\gamma$ |
|  | $\sigma_\gamma = 0.1$ | standard deviation for the $\gamma$ hyper-prior |
|  | $\mu_\omega = 0.3$ | mean for the $\omega$ hyper-prior |
|  | $\sigma_\omega = 0.3$ | standard deviation for the $\omega$ hyper-prior |

Table 3: The initial and fixed parameters of the built-in activation functions of `TensorBNN`.

slope, which can be positive or negative, the Gaussian prior was chosen because it is continuous as 0 and enforces the prior preference for small slopes. Once again, the priors for these activation functions cannot be changed, but a custom activation function with different priors can be created.

When the `add` method of the `network` class is called, the layer or activation function is added to a list of layers. Additionally, the weights, biases, and trainable activation functions from the layer along with the hyper-parameters are stored.

### *3.2. Hamiltonian Monte Carlo Initialization*

After building the network architecture, the Hamiltonian Monte Carlo (HMC) sampler must be initialized. This is done through a method of the `network` class. An example usage is presented below. As described in Section 2.3, HMC is a Markov chain Monte Carlo method where sampling is performed by moving through the parameter space in a manner governed by a fictitious potential energy function determined by the posterior density of the neural network parameters. The numerical approximation used is the leapfrog method, in which the number of leapfrog steps together with the step size determine the distance traveled to the next proposed point. Naturally, larger step sizes yield longer deterministic trajectories, but they also increase the accumulated error due to the numerical approximation and so lower the acceptance rate. Unfortunately, choosing good values for the number of steps and the step size can be challenging. Therefore, `TensorBNN` contains an algorithm, called the parameter adapter, to find these parameters automatically.

The adapter searches a discrete space of step sizes and number of leapfrog steps using the algorithm of [10]. This discrete space is determined by a minimum and maximum step size and a minimum and maximum number of steps. In addition, the adapter accepts the number of iterations before a reset is performed. A reset is performed if after this number of iterations no point has been accepted. It is also given the number random pairs of step size and leapfrog steps to try before using the search algorithm from [10]. Finally, it also accepts two constants $a$ and $\delta$, which assume the values 4 and 0.1, respectively as suggested in [10].

One of the changes from Neal's procedure is the use of HMC to sample the hyper-parameter space instead of Gibbs sampling. This was done because Gibbs sampling requires knowledge of the conditional distribution for each hyper-parameter given that the rest are fixed. While this is possible to calculate for some hyper-priors the package allows these to be changed to custom priors, for which it may not be possible to compute the conditional densities. It was, therefore, simpler to use a second HMC sampler to sample the hyper-parameters since this works for any hyper-prior. The number of steps for the HMC hyper-parameter sampler is kept constant, but the step size is modified depending on the acceptance rate of the sample for 80% of the burn-in period. The `TensorFlow Probability` HMC implementation is used for the sampling.

```
model.setupMCMC(
        0.005,  # Starting stepsize
        0.0025, # Minimum stepsize
        0.01,   # Maximum stepsize
        40,     # Number of adapter stepsize options
        2,      # Initial number of leapfrog steps
        2,      # Minimum number of leapfrog steps
        50,     # Maximum number of leapfrog steps
        1,      # Increment between possible leapfrog step
                #   values in step adapter
        0.01,   # Hyper-parameter stepsize
        5,      # Number of hyper-parameter leapfrog steps
        20,     # Number of burn-in epochs
        20,     # Number of cores
        2)      # Number of averaging steps for adapters)
```

## *3.3. Model Training*

The model is trained using the `train` method of the `network` class. The parameters of the `train` method determine 1) the likelihood function, 2) the metrics to be calculated during training, 3) the number of burn-in epochs, 4) how often to save networks, and 5) the directory in which to store the models. An example is provided below. The prior is determined by the `Likelihood` object. The priors included in the package are those described in Section 2.2, though any desired likelihood function can be used through a custom `Likelihood` object. The built-in Gaussian prior for the standard deviation of the likelihood function used for regression allows specification of the initial value of its standard deviation.

The metrics to be computed are specified by including a list of all desired `Metric` objects. The built-in metrics are `PercentError, SquaredError`, and `Accuracy`. All three accept the normalization scalars of mean and standard deviation to calculate the metrics for unnormalized data, as well as an option to take the exponential of output values before computing for the metrics, for the case of log-scaled outputs. The `PercentError` metric is defined as

$$PE = 100\,\mathbb{E}\left[\frac{|y_{pred} - y_{true}|}{y_{real}}\right].$$

`Squared Error` is defined as

$$SE = \mathbb{E}\left[(y_{pred} - y_{true})^2\right].$$

In both cases, the expectation is with respect to the input data, $y_{pred}$ is the predicted value for a given input, and $y_{true}$ is the target. Finally, `Accuracy` is simply the number of correct predictions divided by the total number of predictions for a classification task.

```
 # Declare Gaussian Likelihood with sd of 0.1
likelihood =  GaussianLikelihood(sd = 0.1)
metricList = [ # Declare metrics
    SquaredError(mean = 0, sd = 1, scaleExp = False),
    PercentError(mean = 10, sd = 2, scaleExp = True)]
network.train(
        320,   # Nmber of training epochs
        2,     # Increment between network saves
        likelihood,
        metricList = metricList,
        folderName = "Regression",
               # Name of folder for saved networks
        networksPerFile=50)
               # Number of networks saved per file
```

In training, the HMC samplers run for the specified number of epochs. An epoch corresponds to iterating the differential equations inside the main HMC sampler for the specified number of leapfrog steps, updating the weights, biases, and activation functions, and then repeating this process with the hyper-parameters.

## *3.4. Predictions and Post Processing*

In order to prevent the output files in which the networks are saved from becoming too large and to allow predictions midway through training, the number of networks written to each file can be specified by the user, as noted above. One file is saved with the shapes of each matrix that defines the network architecture so that they can be properly extracted. Another file is saved that contains the layer names so the network can be reconstructed. The `predictor` object, which is instantiated as follows,

```
# instantiate a predictor object
network = predictor(
          "modelDir/", # Directory where model is located
          dtype=dtype) # Data type of model

```

uses these files to make predictions. Once the object is instantiated, predictions can be made by calling its `predict` method with an input data matrix and specifying that every $n$ saved networks are to be used.

```

predictions = network.predict(
              inputData,# Input data for model
              n=10)     # Predict using every 10 networks

```

Beyond making predictions from saved networks, the `predictor` class is also capable of re-weighting networks given a new set of priors. The ability to re-weight the point cloud of networks makes it possible to study the sensitivity of results to the choice of prior. We can compute the posterior density $p_1(\theta|D)$ with the prior used in the generation of the point cloud and we can calculate the posterior density $p_2(\theta|D)$ using a different prior. By weighting each network $j$ by the ratio $p_2(\theta_j|D)/p_1(\theta_j|D)$, given the network parameters $\theta_j$, we can approximate the point cloud that would have been generated had we used the alternative prior. In general, however, the only components of

$p_1(\theta|D)$ and $p_2(\theta|D)$ that differ are their priors. Therefore, we can substitute $\pi_2(\theta_j)/\pi_1(\theta_j)$ for $p_2(\theta_j|D)/p_1(\theta_j|D)$. But since it may be useful to study the effect of different likelihoods on the results, the code includes the option to supply new likelihoods. The option to re-weight allows fine tuning of the networks after training and makes it possible to explore the impact of different priors on the results.

Re-weighting is implemented in the `reweight` method of the `predictor`, which requires an architecture file containing the architecture of the network with different priors. The method can also accept the training data and a `likelihood` object for use in calculating the network probabilities should the impact of the `likelihood` need to be studied. The user can choose to use only every $n$ networks when making predictions as illustrated in the code snippet below. In order to use these features a separate `Layer` object must be created for each new prior. These objects are then passed as a dictionary to the `predictor` object when it is instantiated. Optionally, a modified version of the `likelihood` object used while training can be included to study the impact of the modifications. The code below shows the instantiation of a `predictor` object with a custom `Layer` added and a call of the `reweight` method which returns a weight for each network.

```
 # declare predictor object
network = predictor("/path/to/saved/network",
                    dtype = dtype,
                    # data type used by network
                    customLayerDict={"dense2": Dense2}
                    # A dense layer with a different
                    # prior
                    likelihood=modifiedLikelihood)
                    # An optional likelihood function

weights = network.reweight(
                    n = 10, # Use every 10 saved networks
                    architecture = "architecture2.txt")
                    # New architecture file
```

The `predictor` can also calculate the autocorrelation function of the networks, which is useful for choosing a suitable value for $n$ in the `predictor`. Given the sequence of networks $f_1, f_2, \cdots$, the autocorrelation and normalized autocorrelation are defined by

$$C(n) = \frac{1}{N-n}\sum_{i=1}^{N-n}(f_i - \overline{f})(f_{i+n} - \overline{f}), \quad \text{and}$$
$$\rho(n) = C(n)/C(0), \tag{11}$$

respectively, where $\bar{f}$ is the average prediction over the networks and $n$ is the autocorrelation lag. One expects an approximately exponential decrease of $\rho(n)$ with $n$. The larger the value of $n$ the smaller the correlation between the networks separated by a lag of $n$ and, therefore, the more independent the resulting ensemble of networks. The autocorrelation is computed with the `autocorrelation` method to which an input data matrix `inputData` is passed along with the maximum lag $n_{\text{max}}$. The output of the method is a list containing the average autocorrelation for each value of $n$ from 1 to $n_{\text{max}}$, where $n$ enters the calculation as in Eq. (11). The average is taken with respect to the input data matrix.

The user can also opt to calculate the autocorrelation length, which may be taken to be the smallest recommended lag $n$. The `autoCorrelationLength` method accepts the same inputs as `autocorrelation` and returns a single float. Both of these methods make use of the `emcee` package [11]. Here is an example of the usage of both methods.

```
autocorrelations = network.autocorrelation(inputData,
                                           75) #nmax
corrLength = network.autoCorrelationLength(inputData,
                                           75) #nmax
```

## *3.5. Algorithmic Adjustments to Prior Algorithms*

The HMC parameter adaptation scheme briefly mentioned in Section 3.2 is a modified version of the method described in Ref. [10]. Our method adapts the number of steps and stepsize in a leapfrog trajectory in attempt to maximize the length of the trajectories in the network parameter space in each iteration of the HMC sampler.

In `TensorBNN`, two variations are introduced. First, in the case that the HMC sampler iterates through 10 leapfrog trajectories without having accepted a point, the algorithm resets and begins with the stepsize maximum and minimum values decreased by a factor of two. This was empirically observed to prevent the HMC sampler from remaining stagnant with an extremely low acceptance rate. Additionally, after each reset of the parameter adapter, the value of the step size and number of leapfrog steps were randomized for the first 20 iterations in order to prevent the algorithm from converging to a boundary point of the interval that had a high acceptance, but was not optimal, as it was observed to do without this randomization. This algorithm was implemented using a combination of `TensorFlow` and `NumPy` [12].

In order to begin the Markov chain sampling of network parameters at a position that reduces the number of needed burn-in steps, we trained a fully connected neural network using `AMSGrad` [13] with the version of `Keras` in `TensorFlow`. We observed empirically that choosing a starting point for the Markov chain based on pre-training done using `Keras` led to a faster burn-in time. The pre-training is done through three training cycles, each containing a fixed number of epochs. After each cycle the learning rate was decreased by a factor of ten, and the best network, judged by the loss computed using the validation data, is selected as the starting point for the next cycle.

## 4. Use Cases

Here we present a simple application of `TensorBNN`, which highlights the important properties of a BNN. The goal is to train a BNN to learn the function

$$y = x\sin(2\pi x) - \cos(\pi x). \tag{12}$$

In this example, the caveat is the network will be trained on only 11 evenly spaced examples for $x \in [-2, 2]$, but we will then ask the network to make predictions for $x \in [-4, 4]$. The training data and real curve can be seen in Figure 1. We also present training on 31 examples to show that the networks behave as expected: improved performance with more data.

The BNN trained on this data had three hidden layers with 10 perceptrons in each and the hyperbolic tangent activation function was used. The values for each of remaining training parameters can be found in Table 4. As the dataset used for this task was so small and the network was not especially large training can be completed using either a GPU or CPU in a number of minutes.

A graphical representation of the training results is shown in Figure 2. These plots demonstrate three important properties of BNNs. Firstly, in the upper graph, we see that as the predictions reach farther from the training samples, the uncertainty increases. This is expected behavior; near the training data, the network can be reasonably certain of the prediction due to the

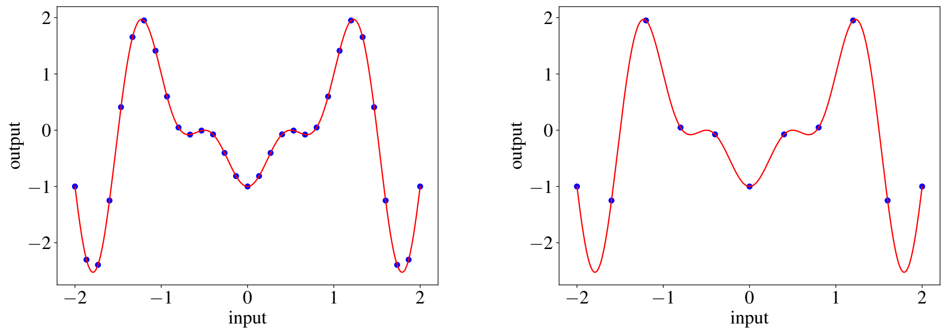

Figure 1: The training data (blue dots) and actual curve (red line) for the training examples.

information provided to the model during training. However, farther from the training examples, there are many possible predictions that are consistent with the training data. Looking at the plots for training with 31 points, we also see that the overall predicted curve is much closer to the true curve and that the uncertainty is lower, as expected.

Secondly, the credible intervals of the BNN do not always contain the true value. In the upper graph in Figure 2, it is clear that the BNN is quite wrong about the behavior of the curve between the first two and last two training points. This, however, is not unreasonable, as the value of the curve decreases dramatically between those values, without any training data to indicate this behavior. The BNN clearly predicts that the curve should follow a roughly straight path between the points, with some possible variance. This is a plausible prediction given the training data, and demonstrates how, despite the uncertainty estimates inherent in the BNN, it is still susceptible to sudden jumps between training data points. This behavior highlights the importance of providing enough training data to accurately represent the mapping. While the BNN will predict greater uncertainty in regions without data, there is no guarantee that the credible intervals encompass the true value, especially if the true values vary vastly from the training data values. Note, however, that these conclusions are contingent on the form of the prior. It is therefore good practice to study the sensitivity of conclusions to modifications of the prior in any real-world analysis.

The need for caution when extrapolating too far from where the data lie is demonstrated clearly in the bottom graph of Figure 2. Extrapolating

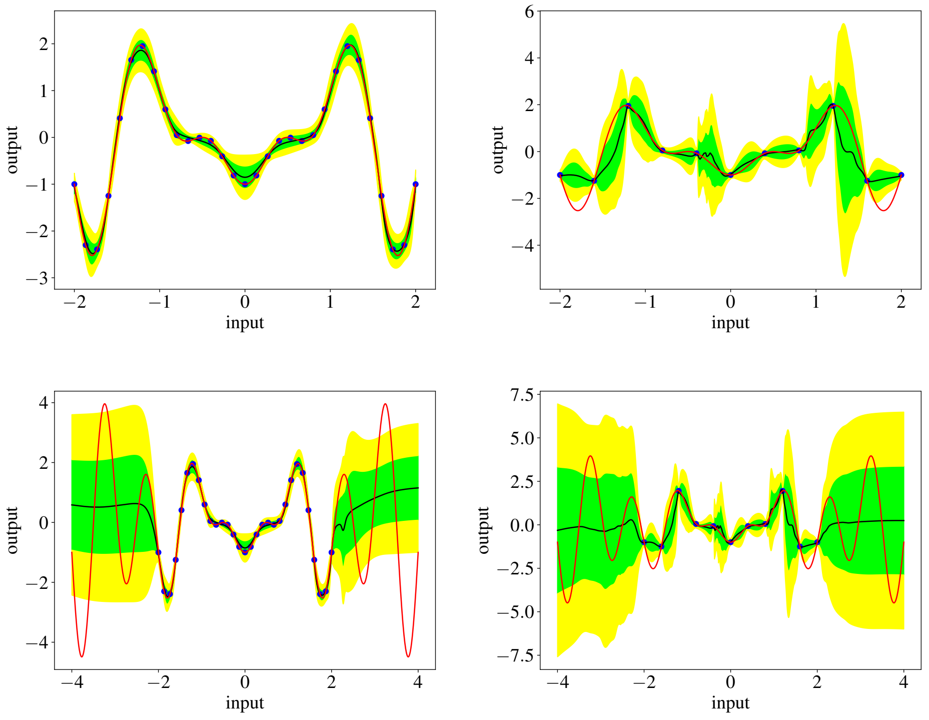

Figure 2: The training data (blue dots) and actual curve (red line) for the training example, along with the mean of the predictions (black line), and the mean plus and minus 1 and 2 standard deviations (green and yellow shading). The top plots are over the training range, and bottom over an extended range. The left plots are with 11 training points, the right plots with 31.

| Parameter | Value |
|---|---|
| leapfrog start | 2 |
| leapfrog min | 2 |
| leapfrog max | 50 |
| step size start | 5e-3 |
| step size min | 2.5e-3 |
| step size max | 1e-2 |
| leapfrog grid step | 1 |
| hyper step size | 1e-2 |
| burnin | 20 |
| epochs | 320 |

Table 4: network parameters

beyond the region containing the training data, the BNN predictions become highly uncertain as expected, but the credible interval may not necessarily contain the true curve. This is, of course, a general observation about any method of extrapolation; the latter depends on assumptions about how the function ought to behave in the region devoid of data.

Beyond these simple examples, a more complex application of `TensorBNN` can be found in [14]. In that paper, the package is used to predict the cross sections for supersymmetric particle creation at the Large Hadron Collider, predict which supersymmetric model points are theoretically viable, and predict the mass of the lightest neural Higgs boson, which is generally identified with the Higgs boson discovered at CERN [15, 16].

## 5. Summary

`TensorBNN` is a framework that allows for the full Bayesian treatment of neural networks while leveraging GPU computing through the `TensorFlow` platform. Through an automation of the search for the parameters needed for an effective Hamiltonian Monte Carlo sampler and the ease of using a pre-trained network, `TensorBNN` is able to decrease the computation needed to converge to a sample from the posterior density of the neural network parameters. The algorithm automatically adapts hyper-parameters to reduce the amount of fine-tuning required by the user. As shown through the simple examples, the distribution of the network predictions behave according to expectations and can be yield good results even on difficult learning problems. The package provides a flexible means to perform regression and binary classification and is designed with the potential for expansion in terms of allowed network architectures and output possibilities.

## 6. Acknowledgements

This work was supported in part by the Davidson Research Initiative and the U.S. Department of Energy Award No. DE-SC0010102.

## References

[1] R. M. Neal, Bayesian Learning for Neural Networks, Lecture Notes in Statistics doi:10.1007/978-1-4612-0745-0.

[2] M. Abadi, *et al.*, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, URL `https://www.tensorflow.org/`, software available from tensorflow.org, 2015.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library, in: Advances in Neural Information Processing Systems 32, 8024–8035, 2019.

[4] Y. Wen, P. Vicol, J. Ba, D. Tran, R. Grosse, Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches, 2018.

[5] N. T. Eric, On Priors for Bayesian Neural Networks, Ph.D. thesis, University of California, Irvine, URL `https://escholarship.org/uc/item/1jq6z904`, 2018.

[6] M. Betancourt, A Conceptual Introduction to Hamiltonian Monte Carlo, arXiv: Methodology .

[7] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, R. A. Saurous, TensorFlow Distributions, 2017.

[8] F. Chollet, et al., Keras, `https://keras.io`, 2015.

[9] K. He, X. Zhang, S. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, in: The IEEE International Conference on Computer Vision (ICCV), 2015.

[10] Z. Wang, S. Mohamed, N. De Freitas, Adaptive Hamiltonian and Riemann Manifold Monte Carlo Samplers, in: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13, JMLR.org, III–1462–III–1470, URL `http://dl.acm.org/citation.cfm?id=3042817.3043100`, 2013.

[11] D. Foreman-Mackey, D. W. Hogg, D. Lang, J. Goodman, emcee: The MCMC Hammer, PASP 125 (2013) 306–312, doi:10.1086/670067.

[12] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, URL `http://www.scipy.org/`, 2001–.

[13] S. J. Reddi, S. Kale, S. Kumar, On the Convergence of Adam and Beyond, in: International Conference on Learning Representations, URL `https://openreview.net/forum?id=ryQu7f-RZ`, 2018.

[14] B. Kronheim, M. Kuchera, H. Prosper, A. Karbo, Bayesian Neural Networks for Fast SUSY Predictions, 2020.

[15] G. Aad, et al., Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC, Phys. Lett. B 716 (2012) 1–29, doi:10.1016/j.physletb.2012.08.020.

[16] S. Chatrchyan, et al., Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC, Phys. Lett. B 716 (2012) 30–61, doi:10.1016/j.physletb.2012.08.021.