Bayesian Neural Networks

Supriya Jain[1] and Harrison B. Prosper[2]

[1]*State University of New York at Buffalo, USA and*

[2]*Florida State University, Tallahassee, Florida, USA*

(Dated: February 17, 2010)

Abstract

Probability theory, from a Bayesian perspective, is a powerful framework to think about, and to address, difficult problems of inference. One such problem is the training of neural networks. In this note, we give a brief overview of Bayesian neural networks (BNN) and a "how to" for the BNN part of Radford Neal's Flexible Bayesian Modeling (FBM) package, which is available at http://www.cs.toronto.edu/~radford/fbm.software.html. To illustrate the utility of the method we apply it to discriminate between two sets of Monte-Carlo generated events representative of a signal and background model.

I. INTRODUCTION

Neural networks [1] are non-linear functions that can model *any* (smooth) map of one or more real variables to the same [2]. In most applications in high energy physics, however, networks with a single output are sufficient. Accordingly, we consider only single-output networks. We begin, by considering neural networks for binary classification, which are used, typically, to separate signal from background. We also mention networks for regression, that is, for fitting functions.

A. Classification

If a network is trained with events, described by the vector of variables x, such that signal events are labeled by $t = 1$ and background events by $t = 0$, then the network output y approximates the posterior probability [3]

$$y \approx \text{Prob}(t = 1|x) = \frac{p(x|1)p(1)}{p(x|1)p(1) + p(x|0)p(0)}, \tag{1.1}$$

that is, the probability that an event defined by the variables x belongs to the signal class. $p(x|1)$ and $p(x|0)$ are the probability density functions for class 1 (the signal) and class 0 (the background), respectively, and $p(1)$ and $p(0)$ are the corresponding class prior probabilities. Usually, one trains with equal numbers of signal and background events, in which case $p(1) = p(0) = 0.5$ and the priors cancel out. The label t is referred to as the *target*.

The basic idea behind Bayesian neural networks (BNN) is to cast the task of training a network as a problem of inference, which is solved using Bayes' theorem. The latter is used to assign a probability density to each point w in the parameter space of the neural network. Each point w corresponds to a network with a specific set of parameters. In the standard methods for training neural networks, one finds a *single* point w_0 in the parameter space, that is, a single network. In the Bayesian approach, one performs a weighted average over all points, that is, all networks.

Consider a neural network with the explicit functional form

$$y(x, w) = \frac{1}{1 + \exp[-f(x, w)]}, \tag{1.2}$$

$$\text{where} \tag{1.3}$$

$$f(x, w) = b + \sum_{j=1}^{H} v_j \tanh(a_j + \sum_{i=1}^{I} u_{ij}\, x_i)\,, \tag{1.4}$$

having I inputs, H hidden units and a single output. The parameters of the network are $w = (u_{ij}, a_j, v_j, b)$; u_{ij} and v_j are called *weights* and a_j and b are called *biases*. Both are usually referred to collectively as weights.

For a correctly trained network the probability that the target is $t = 1$ is $y(x, w)$, while for $t = 0$ it is $(1 - y)$. Therefore, given the training data $\{(t_1, x_1), \cdots, (t_N, x_N)\}$ the probability of the set of targets $t = (t_1, t_2, \cdots, t_N)$, *given* the data x_i, is

$$p(t|x, w) = \prod_{i=1}^{N} y(x_i, w)^{t_i}\, [1 - y(x_i, w)]^{1-t_i}\,, \tag{1.5}$$

in which it is assumed that the events are independent. The posterior probability density for a given vector of weights w is given by Bayes' Theorem

$$\begin{aligned} p(w|t, x) &= \frac{p(t|x, w)\pi(x, w)}{p(t, x)}, \\ &= \frac{p(t|x, w)\pi(w)}{p(t, x)}\,, \end{aligned} \tag{1.6}$$

where $\pi(x, w) = \pi(w)$ is the prior over weights, which is assumed not to depend upon the data $x = (x_1, x_2, \cdots, x_N)$. A reasonable estimate of the neural network output for an event with data x' is the weighted average

$$y(x') \equiv \bar{y}(x'|t, x) = \int y(x', w)\, p(w|t, x)\, dw, \tag{1.7}$$

$$\approx \frac{1}{K} \sum_{i=1}^{K} y(x', w_i)\,, \tag{1.8}$$

where K is the number of points w (and therefore networks) drawn from the posterior density. We note again, that each point corresponds to a different network function in the class of networks with I inputs and H hidden nodes. The average is therefore an average over *networks*.

As is true of all Bayesian inferences, it is the posterior density $p(w|t, x)$ that is the complete solution to the inference problem. The average, Eq. (1.7), is but one useful summary of that solution. Another is the mode w_0 of $p(w|t, x)$, which is akin to the maximum likelihood solution. Moreover, the width of $p(w|t, x)$ gives an estimate of the accuracy of the discrimination or regression function. Since $p(w|t, x)$ is represented as a swarm of points in the network parameter space, the most direct way to propagate the uncertainty in the discriminant or regression function to any parameter of interest is to run an analysis K times, each with one of the K networks. This will yield K results whose spread will be a measure of the effect of the uncertainty in the discriminant or regression function.

It may happen that some of the weight vectors correspond to networks that are tightly fitted to the training data. Such networks will typically perform poorly on events not in the training sample. However, if one averages, with appropriate Bayesian weighting, as in Eq. (1.7), over many networks one expects to produce a network that is less likely to be overtrained. Moreover, there is less need to severely limit the number of hidden nodes because the posterior density will automatically prune away weight vectors that correspond to unnecessarily large networks. Indeed, networks have been trained [4] successfully that contain more weights than the number of training data! The lesson to be drawn is this: We should be wary of our usual intuition about the relationship between the amount of data required to train a network and the number of parameters. In highly non-linear functions, such as that in Eq. (1.2), our intuition borne of experience with linear systems does not apply. From a Bayesian perspective, the network should be as large as is computationally feasible so that the space of functions defined by the network parameter space includes some that are good approximations to the true mapping.

In order to compute the average in Eq. (1.7), it is necessary to generate a sample of points w from the posterior density, Eq. (1.6). Unfortunately, sampling from the posterior density, Eq. (1.6), is not feasible using simple methods. Instead, a sample is created using Markov Chain Monte Carlo methods [4]. The idea is to step through the network parameter space in such a way that points are visited with a probability given by the posterior density $p(w|t, x)$. Points where $p(w|t, x)$ is large will be visited more often than points where $p(w|t, x)$ is small. The methods of choice for sampling complex densities originate from the field of computational statistical physics. Briefly, this is what is done. The problem of moving through the network parameter space is regarded as the problem of following a system consisting of a *single* particle moving through a potential. The posterior density is written as

$$p(w|t, x) = \exp[-V(q)], \tag{1.9}$$

where $V(q) = -\ln p(w|t, x)$, with $q \equiv w$, is intepreted as a spatially varying "potential" through which the "particle" moves. One adds a "kinetic energy" term $T(p) = \frac{1}{2}p^2$, where p is a vector with one component for each dimension of the network parameter space. The "mass" of the "particle" can be taken to be unity without loss of generality. The motion of the particle is governed by its "Hamiltonian" $H = T + V$. It can be shown that the particle will *eventually* visit every point q (that is, w) arbitrarily closely in such a way that the density of points is proportional to $\exp(-H)$. By randomly (and appropriately) injecting or removing "energy" from the system different constant energy regions of the phase space (p, q) can be explored. A Markov chain $q_1, q_2, \ldots q_N$ is thereby created, which converges (eventually) to a sequence of points that constitute a sample from the density $p(w|t, x)$. Since the correlation coefficient between adjacent points is very high, typically 0.9 or higher, one usually saves a point, that is, a network, after every M iterations, with $M \sim 20$. The trajectory between the M points consists of a deterministic set of L steps, typically with $L \sim 100$, governed by Hamilton's equation. Details of the hybrid Monte Carlo algorithm is described are given in the next section.

B. Hybrid Monte Carlo Method

The hybrid Monte Carlo (HMC) method is a generalization of the Metropolis-Hastings algorithm [5] that is more efficient at sampling very complicated parameter spaces, such as those that occur in lattice gauge theory and large neural networks. The basic idea is to reduce the degree of random walking through the parameter space by alternating between deterministic and non-deterministic steps.

Given the Hamiltonian

$$H(p, q) = \frac{1}{2}\sum_{i=1}^{D} p_i^2 + V(q), \tag{1.10}$$

the goal is to generate a sample of points $\{(p, q)\}$ from the canonical density

$$f(p, q) = \frac{1}{Z}\exp(-H(p, q)), \tag{1.11}$$

where Z is a normalization factor called the partition function. The variables p and q are independent so marginalization over p trivially recovers the required density $f(q) = p(w|t, x)$ over the network parameter space.

As a particle moves according to Hamilton's equations (Newton's equations in disguise),

$$\frac{\partial H}{\partial p} = p_i = \frac{dq_i}{dt}, \tag{1.12}$$

$$-\frac{\partial H}{\partial q} = -\frac{\partial V}{\partial q} = \frac{dp_i}{dt}, \tag{1.13}$$

the volume of phase space occupied by the system is preserved. Moreover, the equations are time reversible. Consequently, these equations can be used to transition from one state, defined by p and q, in a Markov chain. However, the Hamiltonian is conserved. Therefore, in order for the system to become ergodic, that is, to visit all states (p, q) such that the probability density is given by Eq. (1.11) it is necessary to introduce random changes in energy.

Hamilton's equation must be solved numerically by discretizing the equations in time. This introduces errors that render phase space conservation and time reversibility approximate, both properties of which are required to achieve accurate sampling of $f(p, q)$. One way to preserve these properties, while approximating Hamilton's equations is the *leap-frog algorithm*, which proceeds in three steps:

$$p_i(t + \epsilon/2) = p_i(t) - \frac{\epsilon}{2}\frac{\partial V(q(t))}{\partial q_i}, \tag{1.14}$$

$$q_i(t + \epsilon) = q_i(t) + \epsilon p_i(t + \epsilon/2), \tag{1.15}$$

$$p_i(t + \epsilon) = p_i(t + \epsilon/2) - \frac{\epsilon}{2}\frac{\partial V(q(t + \epsilon))}{\partial q_i}. \tag{1.16}$$

The momentum is computed at time $t + \epsilon/2$ and it is used to compute the position and momentum at time $t + \epsilon$.

Any numerical method to solve Hamilton's equations will introduce artifacts that will cause the Hamiltonian at the start and end of a trajectory to differ slightly, so the reversibility will not be exact. However, these effects cancel *exactly* by accepting with probability

$$\alpha = \min\left[1, \frac{\exp(-H(p', q'))}{\exp(-H(p, q))}\right], \tag{1.17}$$

$$= \min[\exp(-\Delta H)], \quad \text{where } \Delta H \equiv H(p', q') - H(p, q), \tag{1.18}$$

the state (p', q') determined from the state (p, q) after L leap-frog steps. The HMC algorithm is shown in pseudo-code (1).

```
initial position $q = q_0$
for $t = 1 \cdots M$
    sample $p \sim f(p)$
    perform $L$ leap-frog steps from $(p, q)$ to $(p', q')$
    calculate the acceptance probability $\alpha = \min[1, \exp(-\Delta H)]$
    sample $u \sim \texttt{uniform}(0, 1)$
    if $u \leq \alpha$
        accept $(p', q')$
    else
        keep $(p, q)$
```

Algorithm 1: Hybrid Monte Carlo Method

C. Prior

Every Bayesian inference requires the specification of a prior. For such a highly non-linear problem, however, the choice of prior is not obvious. A reasonable first guess is to use Gaussians, centered at zero, and designed to favor smaller rather than larger weights. Smaller weights yield smoother fits to data. In Radford Neal's package (described in the next section), independent Gaussian priors are specified for each weight. However, the variance for weights belonging to a given group (either *input-to-hidden weights* (u_{ij}), *hidden-biases* (a_j), *hidden-to-output weights* (v_j) and *output-bias* (b)) is chosen to be the same: σ_u^2, σ_a^2, σ_v^2, or σ_b^2, respectively. However, since we do not know, *a priori*, what these variances should be, their values are allowed to vary over a large range, while favoring small variances. This is done by assigning each variance a gamma prior

$$\pi(p = 1/\sigma^2) = \left(\frac{\alpha/2}{\mu}\right)^{\alpha/2} p^{(\alpha/2)-1} \exp\left(-p\frac{\alpha/2}{\mu}\right) / \Gamma(\alpha/2)\,, \tag{1.19}$$

with the mean μ and shape parameter α set to some fixed plausible values. Here p, the inverse of the variance, is sometimes referred to as the *precision*. The gamma prior is referred to as a *hyperprior* and the parameter (here the precision) for which it is a prior is called a *hyperparameter*.

D. Regression

If we wish to fit a function to data, the form given in Eq. (1.5) for the probability of the given targets is inappropriate. Assuming that the noise in the targets is Gaussian, we may use instead

$$\begin{aligned} p(t|x, w) &= \prod_{i=1}^{N} \exp[-(t_i - f(x_i, w))^2/2\sigma^2], \\ &= \exp[-\sum_{i=1}^{N}(t_i - f(x_i, w))^2/2\sigma^2], \end{aligned} \tag{1.20}$$

in which $f(x_i, w)$ is given in Eq. (1.2). Even if the noise in the target is not Gaussian, Eq. (1.20) may still yield reasonable results, provided that the value of σ is chosen to match the noise level in the targets.

II. GETTING STARTED

The example is taken from Neal's documentation. In the following [..] denotes something optional and {..} denotes something of which one can have zero, one or more.

1. Specify a network with P inputs, H hidden units and 1 output, and its associated priors. (The "-" are significant and denote options not used. The "/" is part of the syntax. The backslash is a continuation mark.) This command creates a *binary* file that logs all transactions pertaining to the network.

   ```
   net-spec binary-log-file P-inputs H-hidden 1 / - ih bh - ho - bo
   ```

   Priors:

   - ih prior for input to hidden weights
   - bh prior for biases for hidden layer
   - ho prior for hidden to output weights
   - bo prior for bias for output

   Prior specification:

```
[x]Width[:Alpha]
```

where `Width` and `Alpha` are the parameters of the gamma prior for the given group.

- If `x` is omitted, $\mu = 1/\texttt{Width}^2$.
- If `x` is present *and* $\alpha < 2$, $\mu = N^{2/\alpha}/\texttt{Width}^2$.

Typically, one chooses `Width = 0.05` — a small mean sigma, but $\alpha = 0.5$, which allows for much larger sigmas.

Example

```
                                     ih        bh           ho          bo
net-spec runtrain.bin 4 8 1 / -  0.05:0.5 0.05:0.5 -  x0.05:0.5 -  100
```

In this example, the width of the prior for the output bias `bo` is set to a large value (100) so that the value of the bias is only minimally constrained. This seems reasonable for binary classification. However, such a large value may not be appropriate for regression problems. If the output bias is unconstrained, it may simply converge to the average of the targets, while the other parameters converge to zero. In this case, the input to the network will be ignored and the network will always return the average value of the targets!

2. Specify the kind of model, either *binary* (for classification) or *real* (for regression).

   Example

   ```
   model-spec runtrain.bin binary
   ```

   If regression is required, `binary` must be replaced by `real Width[:Alpha]`, where `Width` is the standard deviation σ appearing in Eq. (1.20).

3. Specify data for training and testing. (The "." are significant and denote options not used.) The "2" indicates that the targets are the integers 0 and 1. It is omitted for regression problems.

   ```
   data-spec binary-log-file N-inputs N-targets [2] /
   trainingData-Filename@first-row[:last-row]{,index} . /
   testingData-Filename@first-row[:last-row]{,index} .
   ```

Example

```
data-spec runtrain.bin 4 1 2 / runtrain.dat@2:2001 . runtrain.dat@2002:4001 .
```

In the example described in Section III, we consider the same set of events for both training and tesing in the above step, since we repeat the testing externally through a method described in Section III B. For this external testing we use an independent set of events compared to those used for the training.

4. Store a network with index 0, in which the hyperparameters have values 0.5 and the parameters are zero. Each network is identified by an index number, which is the iteration number. An iteration typically consists of many steps through the parameter space. In the example, each iteration consists of 20 steps.

   Example

   ```
   net-gen runtrain.bin fix 0.5
   ```

5. Specify the Markov chain operations to be performed in the *initial* phase.

   Steps per *iteration*: 20 repetitions of

   (a) Gibbs sampling for noise level

   (b) A heatbath replacement of the momentum variables

   (c) Hybrid Monte Carlo update with a trajectory 100 leap-frog steps long, using a window of 10 and a stepsize adjustment factor of 0.2. (See Ref. [4] for details.)

   Example

   ```
   mc-spec runtrain.bin repeat 20 sample-noise heatbath \
   hybrid 100:10 0.2
   ```

6. Do one Markov chain iteration.

   Example

   ```
   net-mc runtrain.bin 1
   ```

7. Specify the Markov chain operations for *simulation* phase: Hybrid Monte Carlo with "persistence"

   Example

   ```
   mc-spec runtrain.bin repeat 20 sample-sigmas heatbath 0.95 \
   hybrid 100:10 0.3 negate
   ```

8. Run 200 Markov chain iterations, that is 200×20 steps in this example.

   Example

   ```
   net-mc runtrain.bin 200
   ```

9. Use `net-plt t t binary-log-file` to display the iteration numbers. This is a quick way to check how many iterations have been performed.

10. Make some predictions.

    ```
    net-pred options binary-log-file range

    options:

        i   Display the input values for each case
        t   Display the target values for each case
        r   Use the raw form of the target values, before transformation

        p   Display the log probability of the true targets (to base e)

        m   Display the guess based on the mode, and whether it is in error
        n   Display the guess based on the mean, and its squared error
        d   Display the guess based on the median, and its absolute error
        D   Display the guess based on the mean of the median for each iteration.
            This is mostly useful to get an accurate median for one network.

    range:
    ```

```
Networks to use to form average.
```

Example Average over the last 50 networks.

```
net-pred itmp runtrain.bin 151:200
```

11. Display parameters of network specified by index `nn-index`.

    Example

    ```
    net-display -p runtrain.bin <nn-index> > runtrain.out
    ```

12. Use `netwrite.py` to create a C++ network function that performs the averaging, but also allows access to individual networks.

    Example

    ```
    netwrite.py -r101:200 runtrain.bin
    ```

    In the example, the file runtrain.cpp is created using the last 100 networks stored in the binary log file runtrain.bin.

III. AN EXAMPLE: DISCRIMINATE SIGNAL FROM BACKGROUND

To illustrate the utility of the method we apply it to discriminate between two sets of Monte-Carlo generated events representative of a signal and background model. The signal and background consist of events from $p\bar{p} \to t\bar{t} \to \ell +$ jets and $p\bar{p} \to W +$ jets respectively, where $\ell = e$ or μ and the jets are mainly light-quark jets. The final state in these events contains one high transverse momentum lepton (e or μ), at least two high transverse energy jets, and significant missing transverse energy due to the neutrinos from the W-boson decays. The measured energies and transverse momenta of the final state objects are smeared to account for realistic detector effects.

A. Training

For the training of the BNN networks, we consider four input variables, eight hidden nodes and a single output. The input variables involve the transverse energies, spatial distribution, and invariant masses of the measured final state objects. The variables chosen provide some level of discrimination already between the signal and background as shown in Fig. 1. The goal is to enhance this discrimination through an application of the BNN networks. For this we generate a Markov chain of networks with a training sample, runtrain.dat, consisting of an admixture of 5000 signal events and 5000 background events. This is used to construct the posterior density, $p(w|t, x)$ over the network parameter space. A random sample of 50 networks (iterations) is drawn from the posterior density using the Markov Chain Monte Carlo technique. An average of networks over the last 10 iterations ($N_{\text{net}} = 10$) is used to define the Bayesian neural network output defined by Eq. 1.7. This approximate the discriminant in Eq. 1.1. The BNN outputs for the signal and background normalized to unity, as well as some representative figures of merit are shown in Fig. 2.

B. Verification

Determining when a Markov chain has converged is difficult, in general, though many useful convergence diagnostics exist [5]. However, for the BNN we know what *form* the answer should be. The Bayesian neural network $y(x)$ should approximate Eq. 1.1. That is,

$$y(x) = \frac{p(x|S)}{p(x|S) + p(x|B)}, \tag{3.1}$$

where $p(x|S)$ and $p(x|B)$ are the probability density functions for the signal (class 1) and background (class 0), respectively, and we have assumed equal numbers of signal and background events used for the training. This then suggests the following diagnostic algorithm.

1. Weight N signal events by the BNN output.

2. Weight N background events by the BNN output.

3. Add the 1-dimensional densities of the weighted datasets.

4. Verify, on an independent set of unweighted signal events, that the 1-dimensional *signal* densities are recovered.

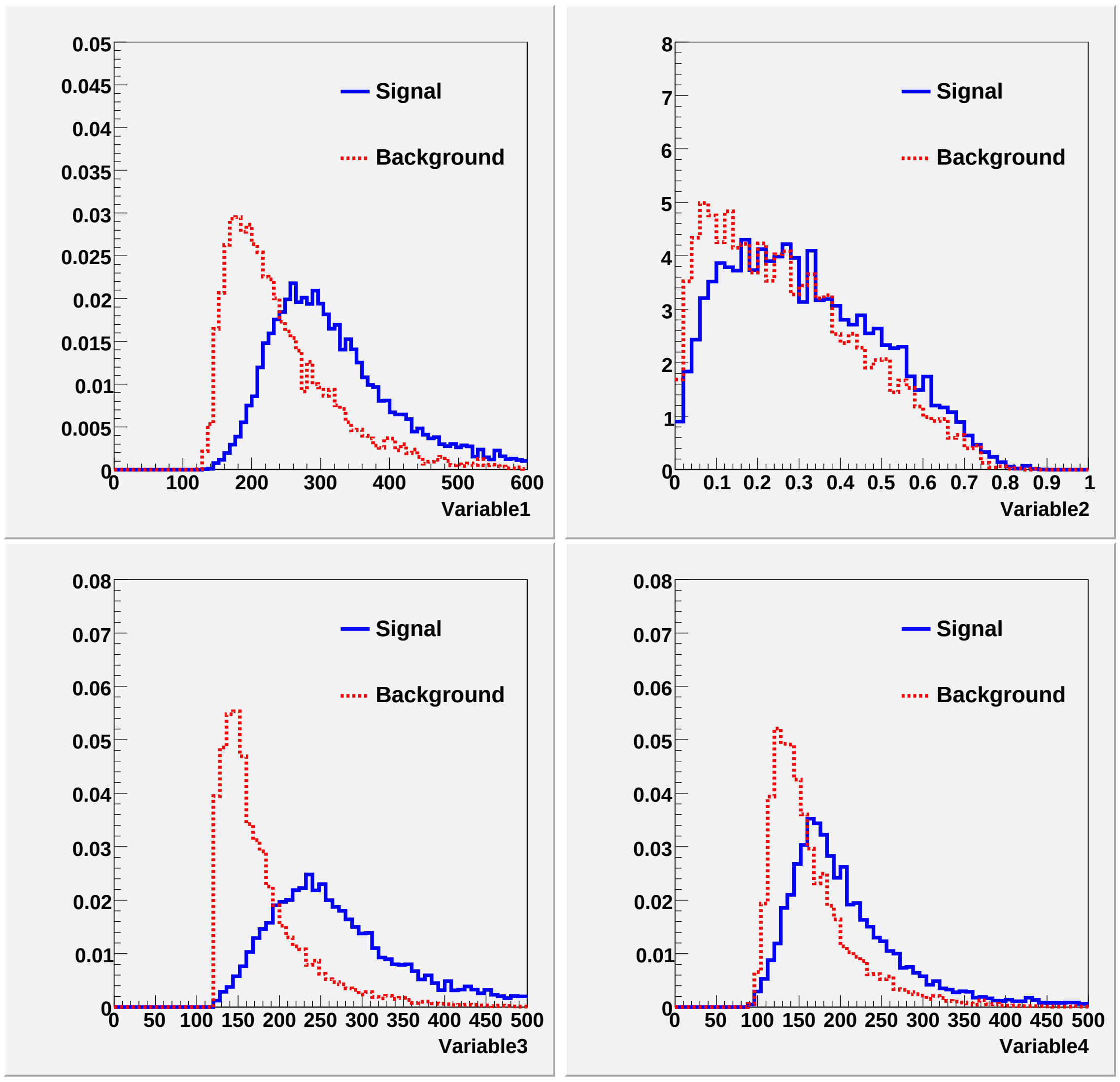

FIG. 1: Input variables from signal and background for the BNN training. The distributions are normalized to unit area for a comparison of their shapes.

Mathematically, the above procedure amounts to multiplying the densities $Np(x|S)$ and $Np(x|B)$ by the BNN function $y(x)$

$$p_y(x|S) = N\,p(x|S)\,y(x), \tag{3.2}$$

$$p_y(x|B) = N\,p(x|B)\,y(x), \tag{3.3}$$

and forming the sum

$$g(x) = p_y(x|S) + p_y(x|B), \tag{3.4}$$

$$= N[p(x|S) + p(x|B)]\,y(x). \tag{3.5}$$

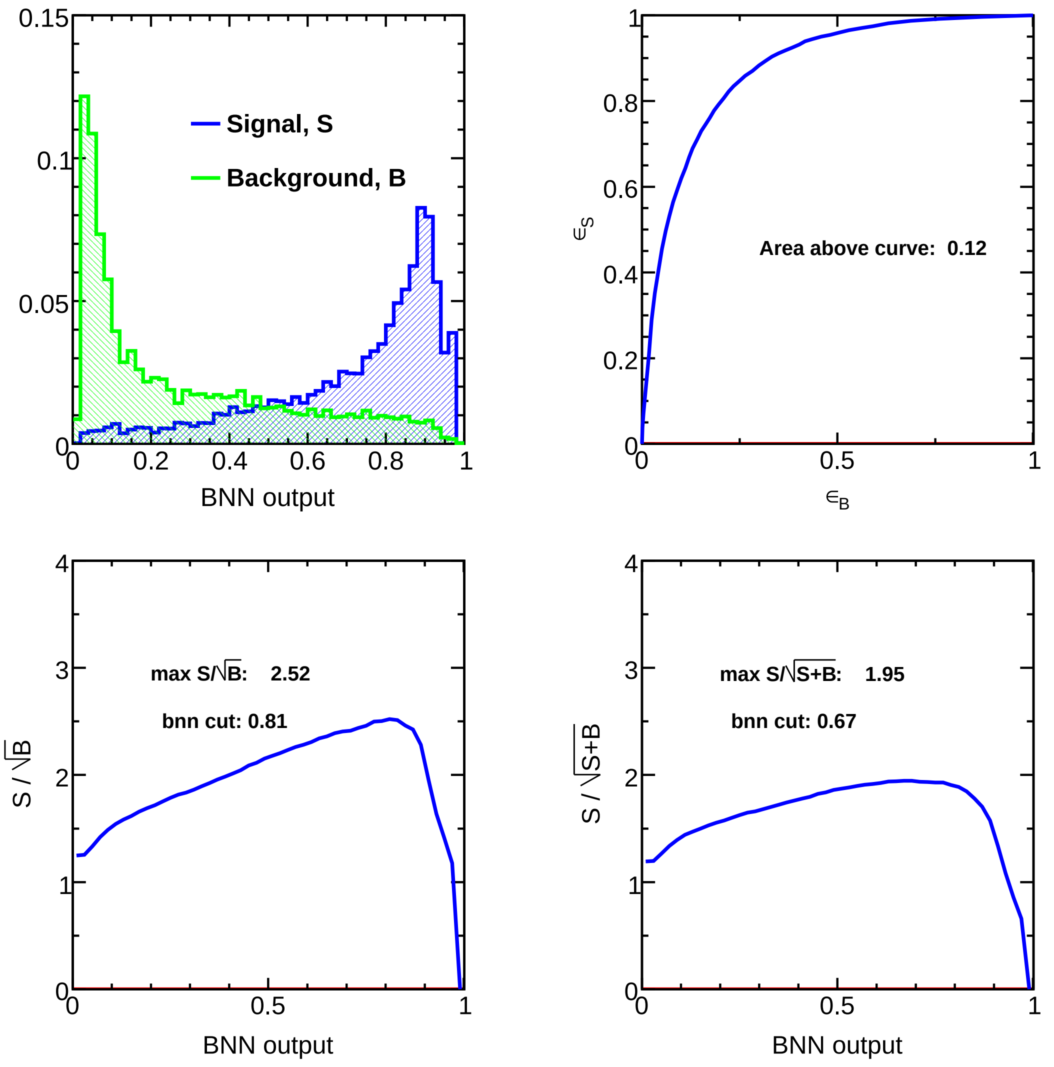

FIG. 2: BNN output and representative figures of merit. In the top-right plot, ϵ_S and ϵ_B are the efficiencies for the signal and background, respectively, for different values of cuts on the BNN output. In the bottom two plots, S and B are the numbers of selected events from the signal and background samples, respectively, after applying cuts on the BNN output.

If, as the Markov chain proceeds, $y(x)$ approximates Eq. 3.1, which is the desired outcome, then we should find that $g(x) \to Np(x|S)$, that is, we recover the signal density. In particular, we should recover all possible marginal densities, that is, projections to 1-dimension,

$$m(x|S) = N \int_{x \notin z} g(z)\, dz. \tag{3.6}$$

We illustrate in Fig. 3 the above diagnostic for each of the input variables. Note that each of the input distributions has been transformed to be in the range [-1, 1] instead of their original range for plotting convenience (in order to allow a convenient automatic binning of all distributions when making such plots through a common macro). The BNN-weighted signal distribution is shown by the blue histogram, the BNN-weighted background distribution is shown by the green histogram, while their sum, $g(x) = p_y(x|S) + p_y(x|B)$, is shown by the red histogram. The signal distribution, $m(x|S)$, is shown by the black dots. If the black dots agree with the red histogram in all the input variables, then this provides confidence that the Markov chain has converged and the output discriminant approximates the functional form in Eq. 3.1.

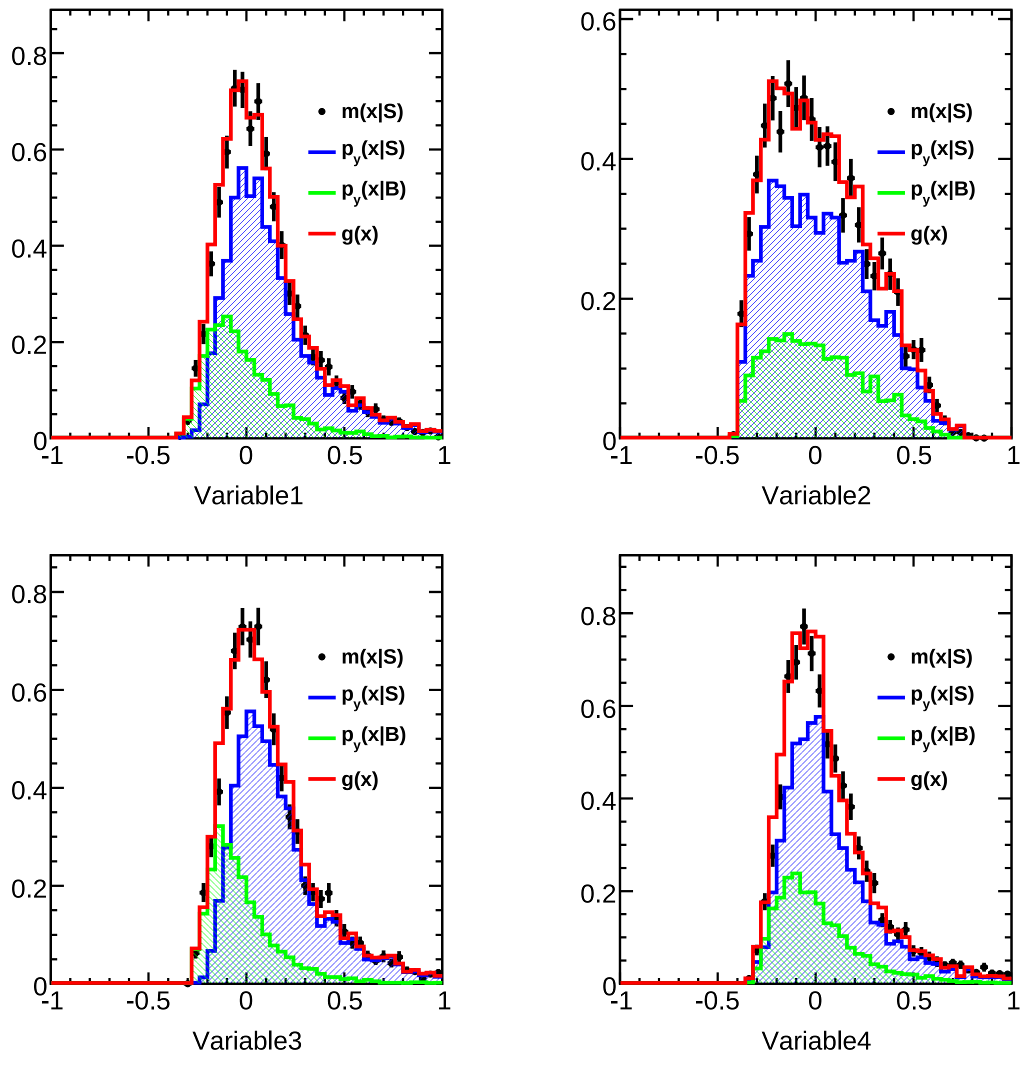

FIG. 3: Verification plots showing the convergence of the Markov chain, and hence, of the BNN output discriminant.

C. Conclusions

Bayesian learning of neural networks could take us another step closer to realizing optimal and robust results in classification problems. It also allows a fully probabilistic approach with proper treatment of uncertainties. But, of course, the key question is: does the averaging help? The answer, in principle, in yes. More to the point, we have found the answer to be yes, in practice. Figure 4 is, in effect, comparing the normalized BNN outputs for each of the last 10 networks to that obtained from averaging over these (shown by the thicker, red curve). As one might have expected, the distributions of the 10 networks show some scatter. One expects, however, the Bayesian average to be a more robust estimate of the true signal class probability. Figure 5 compares the signal efficiency versus the background efficiency for each of the individual networks to that of the average. We see a slight improvement in the preformance upon using the average over networks than using any one individually. This is an indication, therefore, that the averaging helps.

The BNN method, however, could be computationally demanding. A large number of points is needed so that one can abstract a subset of (several hundred) networks that are approximately statistically independent. But this method has been successfully used in D0's search for single top quarks [6], and has shown a sensitivity comparable to that obtained from other state-of-the-art multivariate analyses like the Decision Trees [7], and Matrix Elements [8] approach.

[1] C. M. Bishop, *Neural Networks for Pattern Recognition,* (Clarendon Press, Oxford, 1998); R. Beale and T. Jackson, *Neural Computing: An Introduction,* (Adam Hilger, New York, 1991).

[2] E.K. Blum and L.K. Li, "Approximation theory and feedforward networks," Neural Networks, **4**, 511-515 (1991).

[3] D.W. Ruck et. al., "The multilayer perceptron as an approximation to a Bayes optimal discriminant function," IEEE Trans. Neural Networks **1 (4)**, 296-298 (1990); E.A. Wan, "Neural network classification: a Bayesian interpretation," IEEE Trans. Neural Networks **1 (4)**, 303-305 (1990); H. B. Prosper, "Some Mathematical Comments on Feed-Forward Neural Networks," DØNote 1606 (1993); H. B. Prosper, "More Mathematical Comments on Feed-Forward Neural

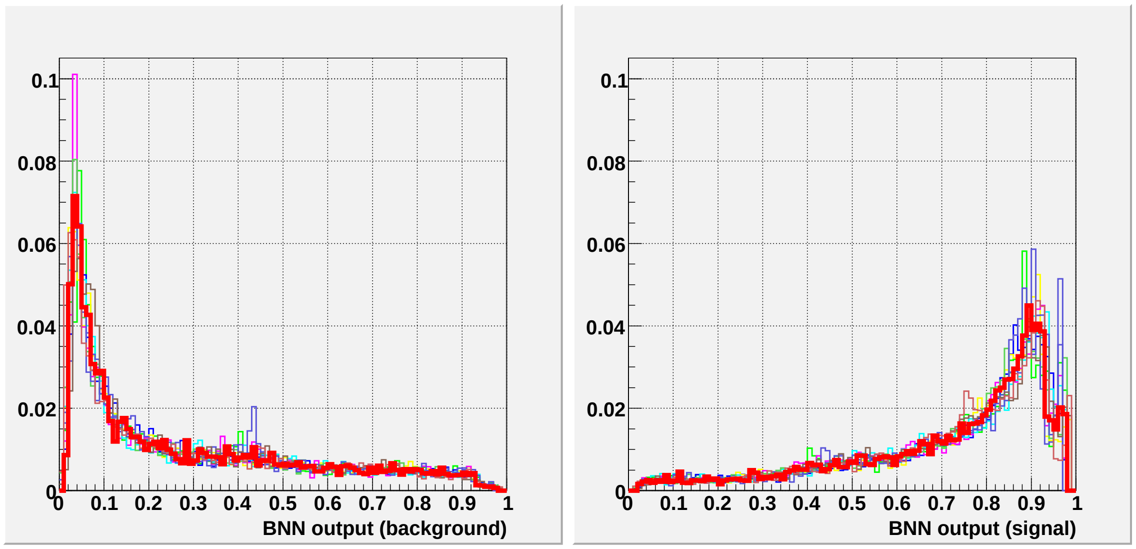

FIG. 4: BNN outputs normalized to unity for each of the last 10 networks, superposed by that obtained from an average over them (thicker, red curve).

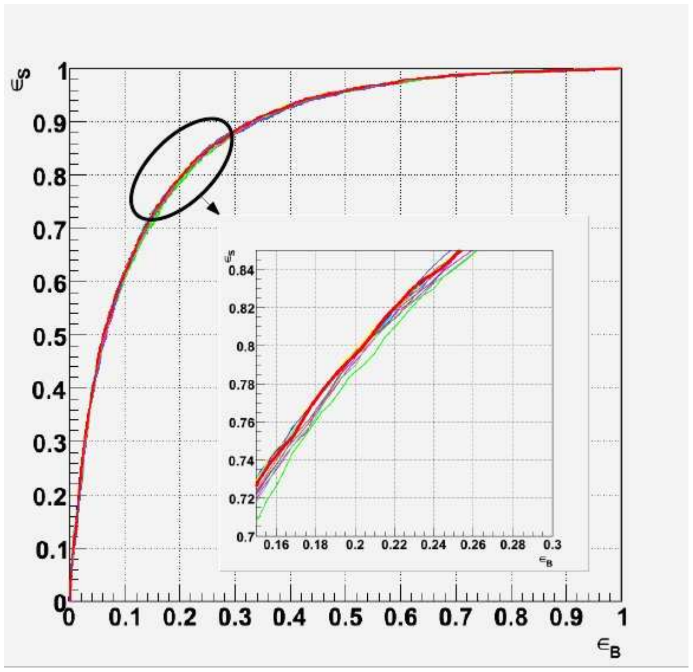

FIG. 5: Signal efficiency versus background efficiency for each of the last 10 networks, superposed by that obtained from an average over them (thicker, red curve).

Networks," DØNote 4493 (2004).

[4] R. M. Neal, *Bayesian Learning of Neural Networks,* (Springer-Verlag, New York, 1996).

[5] B. A. Berg, *Markov Chain Monte Carlo Simulations and Their Statistical Analysis*, (World

Scientificm Singapore, 2004).

[6] V.M. Abazov *et al.* (D0 Collaboration) Phys. Rev. D **78**, 012005 (2008); V.M. Abazov *et al.* (D0 Collaboration) Phys. Rev. Lett. **103**, 092001 (2009).

[7] L. Breiman *et al.*, *Classification and Regression Trees* (Wadsworth, Stamford, 1984).

[8] V.M. Abazov *et al.* (D0 Collaboration), Nature **429**, 638 (2004).