Bayesian Neural Networks

Marco Gallo

November 2025

Abstract

My goal in this work is to introduce the *Bayesian Neural Networks*. It is divided in three parts: the first one will introduce the Classical Neural Networks, the second part explores the Bayesian Neural Network, the third part is on python were I will apply this 2 tools.

Contents

1 Neural Networks

1.1 Introduction

Neural networks are now among the most widely used methods in the field of artificial intelligence—and beyond. They are extremely powerful and have an iconic name inspired by the human brain, where neurons activate and communicate with each other through synapses during the thought process. This biological analogy also underpins the functioning of artificial neural networks.

These models are inherently non-linear and, given appropriate inputs and sufficient complexity, they can approximate virtually any non-linear function. Let's take a closer look at how they work internally.

1.2 Structure of a Neural Network (1)

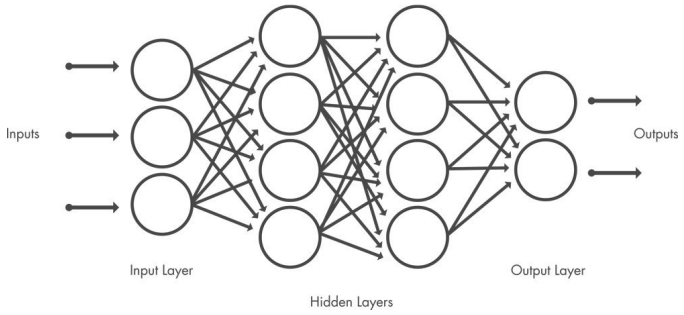

Figure 1: Structure of a two layer NN

As we can see, a neural network typically consists of three types of layers: an input layer, one or more hidden layers, and an output layer. Each layer contains multiple neurons that process the received inputs. We will analyze the behavior of the neurons later; for now, let us focus on the overall structure:

- **Input Layer:** This is the layer that receives the predictors, which can be expressed as a vector of dimension $1 \times P$, where P represents the number of variables under consideration.
- **Hidden Layers:** These are the layers where most of the computation takes place. Both the width and the depth of the network are arbitrary, and different choices lead to different architectures.
- **Output Layer:** As the name suggests, this is the layer where the final outputs are produced. It has dimension $1 \times O$, where O denotes the size of the output.

1.3 Structure of a Neural Network (2)

Let us consider a single-layer neural network (with one hidden layer) and a single output unit. Its output can be written as:

$$f(x) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right).$$

The vector $\mathbf{w}$ contains the weights of the hidden layer, while $\boldsymbol{\beta}$ contains the weights of the output layer. In neural networks, each neuron is often

called a *perceptron*. In the expression above, the computation performed by a single neuron in the hidden layer corresponds to the argument inside the activation function $g(\cdot)$.

More specifically, each neuron has:

- weights w_{kj} connecting it to the previous layer (in this case, the input layer);
- a bias term w_{k0}, which shifts the weighted sum.

The value inside $g(\cdot)$ is then transformed by the **activation function**. In principle, any nonlinear function could be used. The most common examples include:

- The sigmoid function:
$$g(x) = \frac{1}{1 + e^{-x}}.$$
- The ReLU function:
$$g(x) = \max(0, x).$$

To conclude this section, note that this model is structurally similar to a linear regression model. The key difference lies in the activation function: its nonlinearity enables the network to capture nonlinear relationships that would otherwise be missed.

1.4 Estimation Process

In this section, we describe how the estimation process works. For simplicity, we still consider a single-layer neural network. The parameters of the model can be estimated using nonlinear least squares. In other words, we aim to minimize the following objective:

$$\arg \min_{\{w_k\}_{k=1}^K, \boldsymbol{\beta}} \frac{1}{2} \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2,$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right).$$

This optimization problem is non-convex, meaning that multiple local minima may exist. A common approach to finding a solution is the **gradient descent algorithm**, which updates the parameters through:

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla B(\theta^m),$$

where ρ is a learning rate parameter and θ denotes the vector of parameters. If ρ is too small, convergence is very slow; if it is too large, the algorithm may fail to converge.

The function B is the objective we want to minimize, and we compute its gradient with respect to the parameters. But how can we compute the gradient of such a complex function?

Fortunately, this is simpler than it appears: the objective function is composed of nested functions, so we can apply the **chain rule**. If we have:

$$f(g(h(x))),$$

its derivative is:

$$\frac{d}{dx} f(g(h(x))) = f'(g(h(x)))\, g'(h(x))\, h'(x).$$

Since neural networks are made of composite functions, this approach applies naturally. This procedure is known as **backpropagation**.

Pros and Cons of Neural Networks

Pros:

- Ability to model highly complex and nonlinear relationships that cannot be captured by traditional linear models.
- Universal function approximators: with sufficient depth and data, they can approximate virtually any measurable function.
- High flexibility: neural networks can be applied to numerous tasks such as regression, classification, clustering, and generation.
- Excellent performance on unstructured data such as images, audio, and text.
- Scalable: increasing the number of layers or neurons often improves performance (given proper regularization).
- Compatible with advanced techniques such as transfer learning, reducing training time and computational cost.

Cons:

- Require large amounts of data to achieve good performance, especially deep architectures.
- Computationally expensive to train
- High risk of overfitting, particularly when data are limited or the model is overly complex.

- Difficult to interpret: parameters do not have intuitive meaning as in classical statistical models.
- Optimization is non-convex, so convergence to a global minimum is not guaranteed.
- Sensitive to hyperparameter choices (learning rate, number of layers, regularization, etc.).

2 Bayesian Neural Networks

2.1 Introduction

Bayesian Neural Networks (BNNs) are a probabilistic extension of classical neural networks. The key idea is to treat the model parameters θ as random variables drawn from prior probability distributions, which allows us to encode prior knowledge directly into the model. Bayesian inference is then used to update these priors using the data, yielding a posterior distribution over the parameters.

Unlike standard neural networks, which provide point estimates, BNNs produce *predictive distributions*, enabling us to quantify uncertainty in the predictions. Before discussing inference techniques, we first introduce the likelihood formulation.

2.2 Likelihood

To compute the likelihood of the parameters, we rely on Bayes' theorem:

$$p(A \mid B) = \frac{p(A, B)}{p(B)} = \frac{p(A)\, p(B \mid A)}{p(B)}.$$

In our setting, $A = \theta$ represents the neural network parameters we aim to estimate, while $B = \mathcal{D} = (\mathcal{D}_x, \mathcal{D}_y)$ represents the dataset, where $\mathcal{D}_x$ denotes the input features and $\mathcal{D}_y$ the observed outputs.

Thus, the posterior becomes

$$p(\theta \mid \mathcal{D}_x, \mathcal{D}_y) = \frac{p(\mathcal{D}_y \mid \mathcal{D}_x, \theta)\, p(\theta)}{\int_\Theta p(\mathcal{D}_y \mid \mathcal{D}_x, \theta')\, p(\theta')\, d\theta'}.$$

Our goal is to perform inference on this posterior distribution. This can be achieved through approaches such as Monte Carlo sampling or Variational Inference.

The predictive distribution for a new pair (x_i, y_i) is given by

$$p(y_i \mid x_i, \mathcal{D}) = \int_\Theta p(y_i \mid x_i, \theta)\, p(\theta \mid \mathcal{D})\, d\theta.$$

Given an input vector $\mathbf{x}$, the neural network outputs a vector $\mathbf{y} \in \mathbb{R}^C$, where C denotes the dimension of the output space. Using Monte Carlo samples $\theta_1, \ldots, \theta_{N_s}$ drawn from a posterior approximation, the predictive mean can be estimated as:

$$\hat{y}_i = \frac{1}{N_s} \sum_{j=1}^{N_s} \mathrm{NN}_{\theta_j}(x_i).$$

Similarly, the predictive covariance can be estimated as:

$$\hat{\Sigma}_{y_i} = \frac{1}{N_s - 1} \sum_{j=1}^{N_s} \varepsilon_{j,i}\, \varepsilon_{j,i}^{\top}, \qquad \varepsilon_{j,i} = \mathrm{NN}_{\theta_j}(x_i) - \hat{y}_i.$$

If we assume that the predictive outputs $\mathbf{y}$ follow a Gaussian distribution, then the standardized sample mean

$$\frac{\hat{y}_i - \mu_i}{\sqrt{\hat{\Sigma}_{y_i}/N_s}}$$

follows a Student's t-distribution with $N_s - 1$ degrees of freedom. Even if Gaussianity does not strictly hold, the Central Limit Theorem ensures that, for sufficiently large N_s, $\hat{y}_i$ converges in distribution to a Gaussian random variable. This allows us to construct confidence intervals for the predictions.

2.3 Posterior Inference

Parameter estimation in the Bayesian framework requires computing the posterior:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{p(\mathcal{D})}.$$

The term $p(\mathcal{D})$ is the evidence or normalization constant:

$$p(\mathcal{D}) = \int_{\Theta} p(\mathcal{D} \mid \theta)\, p(\theta)\, d\theta.$$

This integral is typically intractable due to the high dimensionality of the parameter space Θ. To address this issue, we introduce two common approaches:

- Monte Carlo Markov Chain (MCMC) sampling,
- Variational Inference (VI).

In the following sections we describe these methods in detail and explain how they can be applied to Bayesian Neural Networks.

2.4 MCMC: Hastings–Metropolis

Monte Carlo Markov Chain (MCMC) methods form a family of algorithms that allow us to sample from a target distribution by using a secondary proposal distribution. The goal is to generate samples of the parameters and use them to approximate their posterior distribution. Here, I introduce a common variant: the Metropolis–Hastings algorithm.

- Draw $\theta_0 \sim$ initial probability distribution.
- For $n = 0$ to N:
  - Draw a candidate point $\theta' \sim Q(\theta' \mid \theta_n)$.
  - Compute the acceptance probability

$$p = \min\left(1,\ \frac{f(\theta')\,Q(\theta_n \mid \theta')}{f(\theta_n)\,Q(\theta' \mid \theta_n)}\right).$$

  - Draw $k \sim \text{Bernoulli}(p)$.
  - If $k = 1$, accept the proposal by setting $\theta_{n+1} = \theta'$; otherwise keep the current value: $\theta_{n+1} = \theta_n$.

Using this algorithm requires choosing a proposal distribution Q, which, given the current state θ_n, generates a new candidate θ'. The acceptance probability p compares how well the new parameter values explains the data (through the target density f) while correcting for any asymmetry in the proposal distribution Q. Note: if Q is symmetrical, the Q terms elide, and it remains just $f(\theta')/f(\theta_n)$.

The new value θ' is accepted with probability p (at most 1). If accepted, it becomes θ_{n+1}; otherwise, the chain stays at θ_n. After running the chain for a sufficient number of iterations (often discarding an initial burn-in period), the resulting samples provide an approximation of the posterior distribution of the parameters.

Metropolis–Hastings is a powerful and general method, though its performance may degrade as the dimensionality of the parameter space increases or if the proposal distribution is not well tuned. In our case $f(\theta) = \frac{p(D|\theta)p(\theta)}{\int_\Theta p(D|\theta)p(\theta)d\theta}$, and denominator constant elides in the fraction.

2.5 Variational Inference

Variational Inference (VI) is a method for approximating the posterior distribution by means of an alternative distribution $q_\zeta(\theta) \in \mathcal{Q}$, parameterized by ζ. The goal is to find, within the family $\mathcal{Q}$, the distribution that is closest to the true posterior $p(\theta \mid \mathcal{D})$.

This is achieved by minimizing the Kullback–Leibler divergence:

$$q^\star = \arg\min_{q\in\mathcal{Q}} \mathrm{KL}(q(\theta) \,\|\, p(\theta \mid \mathcal{D})) = \arg\min_{q\in\mathcal{Q}} \int q(\theta)\, \log\left(\frac{q(\theta)}{p(\theta \mid \mathcal{D})}\right) d\theta.$$

The KL divergence quantify how much two probability distributions differ; in this case, we measure how far the approximation $q(\theta)$ is from the posterior $p(\theta \mid \mathcal{D})$.

After some algebraic manipulation, we obtain:

$$\mathrm{KL}(q(\theta) \,\|\, p(\theta \mid \mathcal{D})) = -\int q(\theta)\, \log \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{q(\theta)}\, d\theta + \log p(\mathcal{D}).$$

Since the term $\log p(\mathcal{D})$ does not depend on q, we may ignore it. By changing the sign, we obtain a maximization problem and define the Evidence Lower Bound (ELBO):

$$\mathcal{L}(q) = \int q(\theta)\, \log \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{q(\theta)}\, d\theta = \mathbb{E}_q[\log p(\mathcal{D} \mid \theta) + \log p(\theta) + -\log q(\theta)]\,.$$

When q is parameterized by ζ, the ELBO becomes:

$$\mathcal{L}(\zeta) = \mathbb{E}_{q_\zeta}[\log p(\mathcal{D} \mid \theta) + \log p(\theta) - \log q_\zeta(\theta)]\,.$$

Choosing the family $\mathcal{Q}$ is crucial: exponential-family distributions, Gaussian distributions, and mean-field (factorized) families are common choices due to their flexibility and analytical tractability.

2.6 Stochastic Gradient Descent

To maximize the ELBO with respect to the parameters ζ, we need to compute its gradient. However, for large datasets or complex models, computing the exact gradient is prohibitively expensive.

For this reason, *Stochastic Gradient Descent* (SGD) is employed. The update rule is:

$$\zeta_{t+1} = \zeta_t + \beta_t \left[\widehat{\nabla}_\zeta \mathcal{L}(\zeta)\right]_{\zeta=\zeta_t},$$

where β_t is the learning rate and $\widehat{\nabla}_\zeta \mathcal{L}(\zeta)$ is a stochastic estimate of the gradient computed from a mini-batch.

The procedure works as follows:

- Shuffle the dataset.
- Split it into mini-batches (e.g., 32, 64, or 128 samples).
- For each mini-batch:

– Compute the stochastic gradient estimate:

$$\widehat{\nabla}_\zeta \mathcal{L}(\zeta).$$

– Update the parameters:

$$\zeta_{t+1} = \zeta_t + \beta_t \, \widehat{\nabla}_\zeta \mathcal{L}(\zeta_t).$$

- After all mini-batches are processed, one epoch is completed.
- Repeat for multiple epochs.

The space of probability distributions is not Euclidean, meaning that the usual quadratic metric $\|\zeta_t - \zeta\|^2$ is not appropriate. To account for the curvature of the space, the *Fisher Information Matrix* (FIM) is used, leading to the *natural gradient*:

$$\zeta_{t+1} = \zeta_t + \beta_t \, \widetilde{\nabla}_\zeta \mathcal{L}(\zeta), \qquad \widetilde{\nabla}_\zeta \mathcal{L}(\zeta) = \mathcal{I}_\zeta^{-1} \, \nabla_\zeta \mathcal{L}(\zeta).$$

The Fisher Information Matrix is defined as:

$$\mathcal{I}_\zeta = \int \left[\nabla_\zeta \ln q_\zeta(x)\right] \left[\nabla_\zeta \ln q_\zeta(x)\right]^T q_\zeta(x) \, dx.$$

Bibliography

[1] M. Magris and A. Iosifidis, *Bayesian learning for neural networks: an algorithmic survey*, Artificial Intelligence Review, vol. 56, pp. 11773–11823, 2023.

[2] L. V. Jospin, H. Laga, F. Boussaid, W. Buntine, and M. Bennamoun, *Hands-on Bayesian Neural Networks – A Tutorial for Deep Learning Users*, arXiv:2007.06823, 2022.