An Introduction to Neural Networks and the Deployment of Bayesian Techniques to Address Common Pitfalls.

A Thesis

Presented to the Faculty of the

Department of Mathematics

West Chester University

West Chester, Pennsylvania

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Applied Statistics

By

Samuel Richards

May 2023

Abstract

Inspired by the mechanics of brain functionality, artificial neural networks (ANN's) are powerful statistical models that capture complex trends in data. These models incorporate a series of algorithms that compete with error functions to adjust their parameters based on some training data. Because of their immense complexity, with hidden layers operating like a black box, traditional neural networks require a lot of training data in order to produce viable results. It's not uncommon for a neural net to have several thousand or millions of parameters, each a tiny knob that needs to be fine-tuned precisely. This makes ANN's so flexible during training that they can become overfit to the data that trained them and thus less practical in application. In addition, their architecture puts a lot of trust in the algorithms that comprise them and makes it difficult to measure uncertainty in their predictions. While regularization techniques such as dropout and weight decay address these issues, constructing a network from a Bayesian approach offers another solution, particularly in short supply of data. Bayesian neural networks, which use the entire posterior distribution rather than a single parameter estimate, are models with a means of describing this uncertainty with more transparency. A parameter of a BNN has a measure of belief in its value through a posterior distribution, which allows for the same input to yield slightly varying results based on the likelihood of the output. The result is a powerful and adaptive model, with a built-in means of describing its accuracy and the capacity to learn from a small amount of data.

Contents

Notation

$\sigma(\cdot)$	Sigmoid activation function
σ^2	Variance
μ	Mean
$\mathbb{R}^n$	Set of all real numbers in n-dimensional space
$\mathcal{A}$	Network architecture
α	Regularizing constant in weight decay
β	Step size
D	Data set
D_x	Data set inputs
D_y	Data set outputs
$\mathcal{R}$	Regularizer
θ	Set of tunable parameters
λ	Position identifier in MCMC methods
$P(\cdot)$	Probability distribution
$E(\cdot)$	Expected value
y^*	Set of predicted values on new data
$\mathcal{M}^\intercal$	Transpose operation on matrix $\mathcal{M}$
$\Phi_\theta(x)$	Approximated distribution of parameters θ for input x
Θ	Set of sampled parameters θ_i from the posterior
W	Set of network weights
b	Network bias

Introduction

Deep learning is the brand of machine learning concerned with using Artificial Neural Networks (ANN's) to reveal complex abstractions within data. The evolution of the process of mimicking cognitive brain functionality within artificial machines dates back to the 1940's under the term *cybernetics*, which was originally intended to provide computational models for biological understanding [1]. Since then, the history of what we now call "deep learning" has undergone waves of renaissance. Deep learning models deviate from typical statistical models in that they do not adhere to the standard $input \rightarrow output$ relationship easily expressed by a formula. Rather, their pattern is $input \rightarrow further\ processing \rightarrow output$, which makes it difficult to describe the relationship that is actually happening between inputs and outputs and additionally troublesome to measure things like variability. Indeed, because of the blur of functionality within hidden layers, neural networks are often described as black boxes. More hidden processing layers makes for a deeper network and a darker box.

This thesis is presented to reveal some of the complexities of these high-level machines. It begins with a review of the basic structure of the Multi-Layer Perception, and describes variants associated with a change of architecture. After establishing a firm foundation for neural networks, including problems associated, this thesis will deviate topics and discuss Bayesian Statistics, based both in theory and inclusive of practical estimation techniques to apply. The fundamental idea behind Bayes is the description of the parameters sought in statistical modeling as distributional; rather than fixed, unknown constants that are estimated by the model. The reason for describing Bayesian techniques is to then return to the deep learning discussion and apply them to these intricate models. As will be discussed, neural networks have a tendency to overfit the data. This is because the models have so many estimable parameters, making them extremely flexible learners. At test time, then, they have a tendency to be brittle. Bayesian techniques to deep learning offer the necessary flexibility for a model simply by the nature of Bayes, all while being able to measure uncertainty much more explicitly than traditional deep learning models.

Artificial Neural Networks

Inspired by the interlacing of brain tissue, artificial neural networks are what makes the extension from traditional machine learning models into deep learning methods. Different types exist for different tasks, each with countless variations in architecture, optimization, and methods of performance measurement. A common theme to this thesis is the immense complexity of ANNs. Subject literature appears to have little to no generalizations for network types; each is a specialized or broadly encapsulative method for certain tasks. Often, scientists of the matter seek to compare neural network features when presented with data to determine how one fairs over another when cased with similar problems. [2] [3] [4]

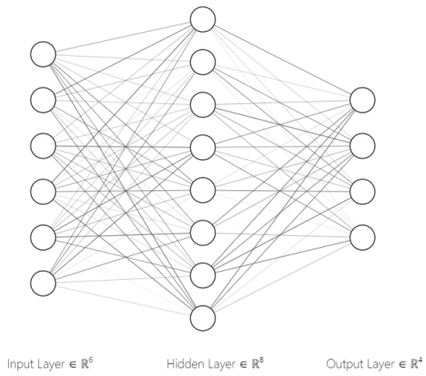


Figure 1: A two-layer neural network with the *input* as the first layer and the *hidden* layer as the second. The final layer displayed is the output result

ANNs are composed of one or more layers in between the input data and the output results. Inputs propagate forward through the layers in the network, each of which performing its respective calculations toward the overall objective. The final output is measured by a prespecified cost function (i.e. sums of squares) between the input values and desired results, and an optimization algorithm tells the network how to adjust each weight specifically in order to reduce that cost and perform better. This chapter analyzes neural network architectures, cost functions, and optimization algorithms; and outlines some common types of neural networks. It then reviews some techniques from modern literature on how to analyze and improve the performance of these advanced deep learning mechanisms.

Neural Network Architecture

Selection of ANN architecture consists of defining the number of layers, the number of neurons in each layer, the connections between them, and the activation function(s) to use. One hidden layer is sufficient to make accurate predictions from training data. More hidden layers increase the depth of the network. Deeper networks are better at generalizing test data, but are harder to optimize. [1] It is common for a L-layered network to be recognized as such by the number of L layers that have tunable parameters; although some literature describes an alternate means of describing layer sizes [5] (i.e. by counting the output layer rather than the input). This thesis will only use the tunable weights criterion so not to confuse what the l^{th} layer is referring to, as illustrated in Figure 1.

A neural network is based on a feed-forward mapping, in which connections exist only directly between any two subsequent layers, and each unit (or *neuron*) is connected to all neurons in its adjacent layer(s). Such feed-forward designs are non-linear mappings between a set of input variables and a set of output variables, in which direct connections between layers are transformed by non-linear activation functions [6]. However, linearity can be achieved through the usage of a linear activation function. In such cases, a linear neural network model is found to be more adaptive to data curvature, without the necessity for splines or higher order polynomials. Indeed, a neural network devoid of any activation function simplifies to a linear regression model [3], which will be shown later in this section.

Neurons

ThThe idea of an *artificial neuron*, represented by a circle in Figure 1, began with Frank Rosenblatt in the 1950's [7]. Rosenblatt coined the idea of a **perceptron**, which takes the value of 0 or 1. The weights, represented as connecting lines in the figure, influence the binary output of the perceptron by whether or not the weighted sum of all input input values x_i multiplied by the respective weights w_i reaches a prespecified threshold:

$$Y = \begin{cases} 0 \text{ if } \sum_i^n w_i x_i \leq M \\ 1 \text{ if } \sum_i^n w_i x_i > M \end{cases}$$

Where i can be any number of input values composed in the network, and M is the *decision boundary*, the threshold value that the weighted sum is matched with to determine the binary output Y. Because these inputs are calculated by computers, and computers like vectors, the above notation is adjusted to allow for proper computation in which $(w_i \cdot x_i)$ is the dot product of the weights and inputs. In addition, a bias term

b_i is introduced to shift the function for a clean threshold of $M = 0$.

$$Y = \begin{cases} 0 \text{ if } (w_i \cdot x_i) + b_i \leq 0 \\ 1 \text{ if } (w_i \cdot x_i) + b_i > 0 \end{cases}$$

This is represented by the *step function.*

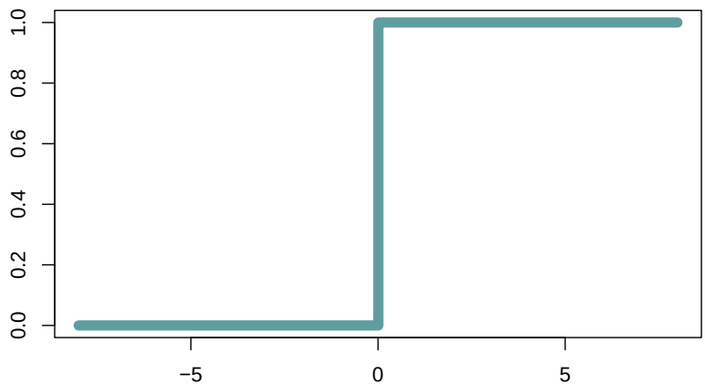


Figure 2: The step function, which takes any input and returns a binary output value 0 or 1.

The problem with the perceptron is that a small change in weight can cause the outcome to change dramatically, since the output is binary [7]. As a network adapts to one training input, it may be less accurate for others. Particularly when the data is non-linear, an inflexible decision boundary M can increase cost for several other outputs in an attempt to decrease the cost for one. By introducing a smooth activation function, the network can adapt to non-linearity in the data [6]. Typically, this is done with the *sigmoid* activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Which is represented by the *sigmoid function.*

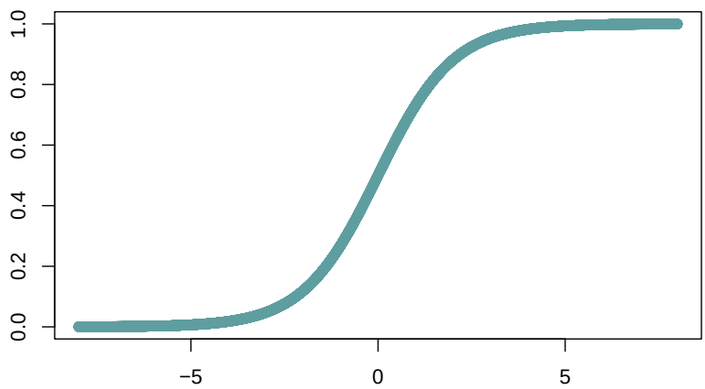


Figure 3: The sigmoid function, which allows for continuous output values from 0 to 1.

For extreme inputs, that is for combinations of the weighted inputs that sum to far more or far less than one, not much changes from the step function as the output is either very close to 0 or very close to 1. It is the smoothness in the middle that caters to modest values which allows the decision boundary to be flexible: small changes in the parameters learned by the cost of one training point will not have as much influence on the cost of other training points. The elementary topology of a neural network as featured in Figure 1 is called a **Multi-Layer Perceptron**. Despite its name, it is typically composed of sigmoid neurons. As such, the resulting calculation for the weighted sum of inputs from one layer into one sigmoid neuron in the next layer is:

$$\sigma(z) = \sigma((w_i \cdot x_i) + b_i) = \frac{1}{1 - e^{-((w_i \cdot x_i) + b_i)}}$$

And indeed $\sigma(z)$ becomes one of many inputs x_i into the next layer, which would be used to compute a weighted sum and coerced under activation again until the network has propagated to the output.

A note on linearity

A single-layer neural network would have but one step of calculations (*input* → *output*). With no hidden layer to perform subsequent calculations, the resulting interpretation would be the expected result of the given inputs.

$$y_i = (w_i \cdot x_i) + b_i$$

This is exactly what is happening with each forward pass by every connection into the next layer. Recall that the activation function of a neural network is to enhance its ability to adapt to non-linear patterns. While this is a preferable trait, linear activations exist to adapt to linear patterns [3]. If a neural network's

architecture consisted of a single hidden layer with a linear activation, it would be a **linear regression** model. In much the same menner, the network would only be able to linear changes in the input.

Similarly, it can also be noted that a neural network of similar architecture but which employs the sigmoid activation function is a generalization of **logistic regression**. [8] [9] If δ denotes the logit function expressed as the log-odds of the outcome of X (0 or 1):

$$\delta = ln\left(\frac{P(X)}{1-P(X)}\right) = (w_i \cdot x_i) + b_i$$

Reversing δ results in the relevant probability that $X = 1$ for the given vector of weights and bias.

$$P(X) = \frac{e^{\delta}}{1+e^{\delta}} = \frac{1}{1+e^{-\delta}} = \sigma(\delta)$$

In which $P(X) \in [0, 1]$. For logistic regression, or for a two-layer neural network using the sigmoid activation function, the calculation would end here and the resulting interpretation from the binary classification model a probability that the input is from category 1. In deeper neural networks, the value of $P(X)$, which is one of many, gets passed on to the next layer to join its cohorts for the next calculation.

The potential for neural networks expands well beyond linearity or the curvature matched by polynomial terms or splines. Neural nets are universal function approximators, which means that they have the capacity to learn and compute any function. [3] This is due to non-linear activation functions, for without them, transformations between successive layers would be linear, and the entire composition itself then linear, too. [5] While there is no one-size-fits all, certain activation functions have desirable properties that help give insight into the questions that neural networks answer.

Activation Functions

The sigmoid activation function gives the output of a neuron a continuous range (0,1). While use of this activation is common in practice due to its appealing differentiable properties, other activation functions can be used that coerce the output of the neuron in various ways. Many activation functions exist, with several variations of the original, each selected to improve the performance for a specific task. The sigmoid activation function is the most widely used because it is easily differentiable and overcomes the primary drawback of the step function. Hyperbolic tangent (Tanh) is another function very similar to sigmoid, yet it is steeper and symmetric about the origin with range (-1,1). This is preferable in cases where gradients are not restricted to vary in a specific direction. To mathematically compare it to sigmoid, it can be defined as $f(x) = 2 \cdot \sigma(2x) - 1$.

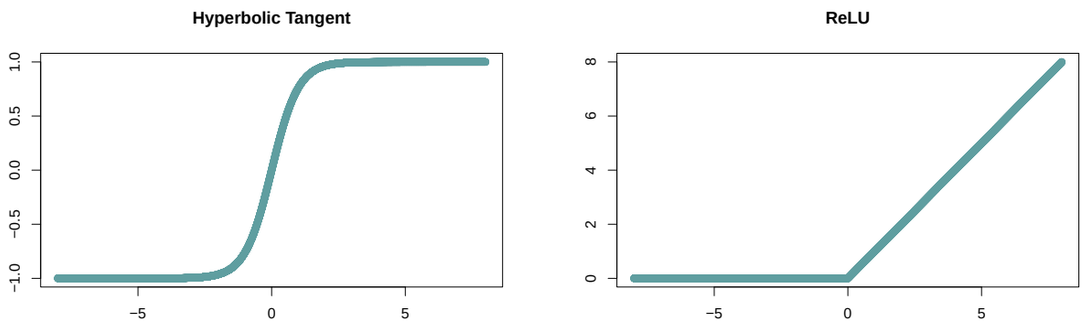


Figure 4: Hyperbolic tangent and ReLU activation functions. Notice the similarity between sigmoid and hyperbolic tangent; although the latter returnes values from -1 to 1.

Another common activation function is the *rectified linear unit*, or ReLU. It is defined mathematically as $f(x) = max(0, x)$. This means that not all neurons will be activated at the same time, which makes ReLU a very useful tool to use in hidden layers to increase efficiency and speed up the learning process. [1] [3]. Variations of ReLU include *leaky ReLU* or *exponential ReLU*, which introduce a means of reducing negative inputs to a very small but still trainable size, rather than zeroing them out entirely. Similarly, variations of sigmoid, Tanh, and many other activation functions exist and are tested in the interest of improving model efficacy [10].

Cost Functions

Cost functions, also known as error or loss functions, are identified by taking the negative logarithm of the likelihood function [6]. Because of the complexity of neural networks, they have many variations to match specific tasks, and as such many different cost functions can be applied. To give a comprehensive review of cost functions extends beyond the scope of this thesis. Instead, some recognizable cost functions that are seen in other models will be described.

For **regression** tasks, the squared error criterion is commonly used to measure cost.

$$E_D(D) = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Where $\hat{y_i}$ is the outcome of a single iteration of network forward propagation and y_i is the true observation to compare it to.

For **classification** tasks, variations of the *log-loss* function are applied depending on the task. Binary classification problems typically use the sigmoid activation function, resulting in the following loss function:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i})$$

This can be extended into multi-class classification models. For these, the generalized general form of sigmoid known as *softmax* is used as the activation function [6].

$$P(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

By the same techniques, the log-loss function generalizes to the following form:

$$MCE = -\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(\hat{y}_{ij})$$

In which the sum of all probabilities for each category equals 1. For example, if the network is designed to identify from one of three types of fish, it might output something like: 0.84 (Blue Gill), 0.07(Bass), 0.09(Barracuda).

The entropy equations above are often called the *binary cross-entropy* and *multi-class cross-entropy* in literature; or more simply the *cross-entropy* loss function, depending on the task. This can be misleading. *Any* loss measured with a negative log likelihood is a cross-entropy between the distributions defined by the training set and that of the model [1]. Therefore despite the exclusive name of the last two, all of these are information theoretic cross-entropy error functions which neural networks use optimize their parameters. This optimization procedure is typically done with **gradient descent.**

Gradient Descent

Predictions of a neural network are measured and corrected with each iteration based on reducing a cost function. This is the same as finding a local minimum of a function. Given the function $f(x) = \frac{x^3}{2} - x^2 - 12x$, Figure 5 appears to have a local minimum somewhere around approximately $x = 4$, which can be calculated with precision by differentiating the function and finding the correct turning points.

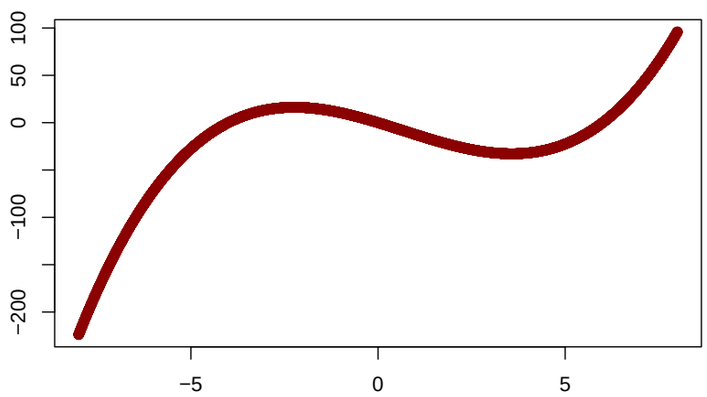


Figure 5: An arbitrary one-dimensional function that has a local minimum and a local maximum. As dimensionality increases, the ease of visualizing or finding an accurate solution disappears.

This is ideal when $x \in \mathbb{R}$, even for $x \in \mathbb{R}^2$, but gets more and more complicated as dimensionality increases to $x \in \mathbb{R}^n$. This calls for another means for finding function extrema, particularly minima. Gradient descent is an optimization process by which an extreme is found by means of iterative computation of partial derivatives for high-dimensional functions, making it intrinsically useful for neural networks to learn. To observe gradient descent in two or three dimensions is perhaps the simplest way to visualize its process. Figure 6 displays the three dimensional function $z = \frac{1}{2}x^2 + 4sin(y)$. To get to a local minimum by means of gradient descent is like rolling a ball down the side of the function with ammended physical laws of inertia, in which the ball only goes downhill and stops when it cannot move further down. The blue line is the path of the ball, each arrow a snapshot "step."

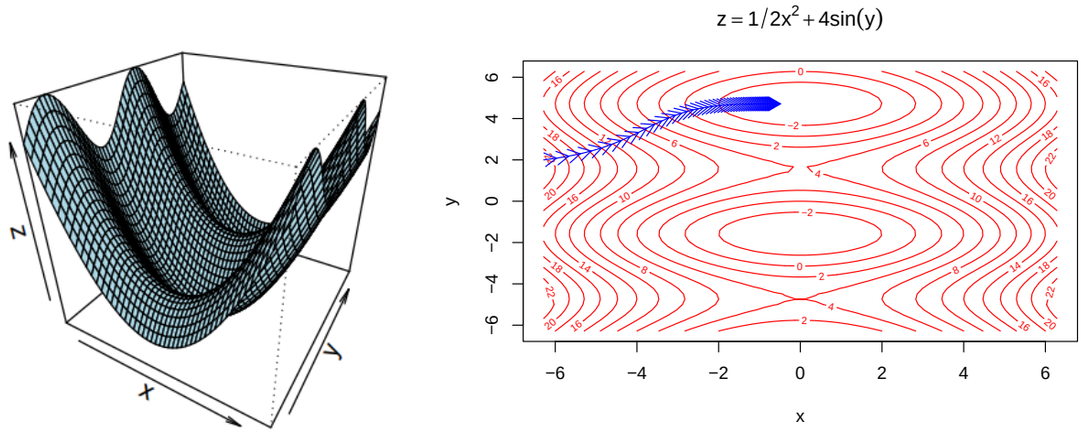


Figure 6: The function in three dimensions, with a top-down observation that shows the path of gradient descent. Each blue arrow represents a step.

Biological limitations bar the visualization of higher dimensions. For computers, this is no trouble and the process is the same, only with more components to calculate the gradient. The basic concept is to select a starting point, determine the direction of steepest descent, and take a step in that direction. When the network propagates forward for the first time, it initializes the parameters by determining the outcome error for the vector of weights w_i and biases b_i using the loss function. It then computes the gradient - the adjustment to every specific w_i and b_i parameter that gives the direction of steepest descent to the loss function - and takes a step in that direction. The process is repeated with another propagation forward to calculate the error and determine the next step.

Gradient descent is not isolated to deep learning, and in fact drives the learning of machine learning algorithms, too [1]. However, deep learning networks have multiple layers and non-linear activation, which require more careful attention to calculate the gradient. Neural networks do this by working backwards, known as *backpropagating*. The idea is to adjust the weights and biases in each network layer, starting with the desired output step, until the input layer is adjusted and the network can run again.

Backpropagation

Backpropagation is the algorithm that drives gradient descent and computes the gradient for neural networks. For a standard neural network, it is composed of the partial derivatives of the cost function with respect to each weight in each layer of the network, as well as that of each bias in each layer of the network.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_1^{(1)}} & \frac{\partial C}{\partial w_2^{(1)}} & \cdots & \frac{\partial C}{\partial w_i^{(1)}} \\ \frac{\partial C}{\partial b_1^{(1)}} & \frac{\partial C}{\partial b_2^{(1)}} & \cdots & \frac{\partial C}{\partial b_i^{(1)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_1^{(l)}} & \frac{\partial C}{\partial w_2^{(l)}} & \cdots & \frac{\partial C}{\partial w_i^{(l)}} \\ \frac{\partial C}{\partial b_1^{(l)}} & \frac{\partial C}{\partial b_2^{(l)}} & \cdots & \frac{\partial C}{\partial b_i^{(l)}} \end{bmatrix}$$

Here, the superscript $^{(l)}$ denotes the layer of the network and the subscript $_i$ denotes a specific parameter within that layer. In this section, the backpropagation calculus will be explained under the example of a *two-layer* network; although the same notation and derivation can be applied to networks of all sizes. The backpropagation algorithm is catered to a specific loss function. In this example, loss is measured by the squared differences between predicted and actual outputs.

$$C = \sum_{i=1}^{N} (\hat{y_i} - y_i)^2$$

This function takes the difference between the output from the network and the specified desired output for each observation. To understand how each parameter that makes up the output must be adjusted, the algorithm must differentiate every part of it. This is done by use of the chain rule in calculus. Figure 7 traces the relationship of the model from input to output for each i connection in one iteration of forward propagation. In other words, without the subscripts, Figure 7 would represent the forward functionality of a three-layer neural network with one neuron in each layer.

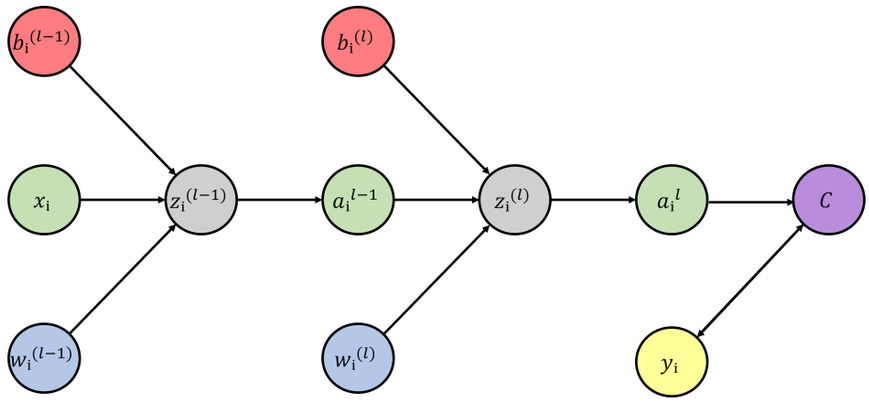


Figure 7: The function $z_i^{(l-1)}$ in the input layer, which is composed of the effects of the input, weight and bias, is processed by an activation function $a_i^{(l-1)}$. The output is sent as the input into the next layer, along with the relevant weight and bias parameters, which compose $z_i^{(l)}$. This is processed again by an activation function, generating the output $a_i^{(l)} = \hat{y}_i$. $\hat{y}_i$ is matched with y_i to calculate the error of this iteration.

By tracing the lineage of effects on the network's final output backwards ("back-propagating"), the output is found to be influenced by:

1. an *activation* function in between the hidden layer and the output, which itself is influenced by the resulting computation of the hidden layer *weight*, *bias*, and
2. *activation* function from the previous layer, which is itself influenced by the resulting computation of the previous layer *weight*, *bias*, and input layer.

In the two-layer network example, identifying the lineage stops here. For deeper networks, item (2) is repeated with the term "input layer" replaced by "activation of earlier layers," and Figure 7 would be longer.

Partial derivatives with respect to weight

Table 7 serves as a visual aid to chain rule calculations. For example, the vector of weights in the *hidden* layer $w_i^{(l)}$ influences on C through three steps. This requires computing the effect $w_i^{(l)}$ has on $z_i^{(l)}$, that $z_i^{(l)}$ has on $a_i^{(l)}$, that $a_i^{(l)}$ has on C. Therefore, the amount to adjust each weight in the hidden layer is represented through use of the chain rule by:

$$\begin{aligned}\frac{\partial C}{\partial w_i^{(2)}} &= \frac{\partial z_i^{(2)}}{\partial w_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial C}{\partial a_i^{(2)}} \\ &= a_i^{(1)} \sigma'(z_i^{(2)}) 2(a_i^{(2)} - y_i)\end{aligned}$$

where $a_i^{(1)}$ is the activation for each observation in the first (input) layer, $\sigma'(z_i^{(2)})$ is the derivative of the sigmoid activation function with respect to each observation in the second (hidden) layer, $a_i^{(2)}$ is the activation for each observation in the hidden layer (the predicted output) and y_i is the vector of actual training values this network hopes to achieve.

Propagating a step further back, the amount to adjust the weight in the *input* layer is represented by:

$$\begin{aligned}\frac{\partial C}{\partial w_i^{(1)}} &= \frac{\partial z_i^{(1)}}{\partial w_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial C}{\partial a_i^{(2)}} \\ &= x_i \sigma'(z_i^{(1)}) w_i^{(2)} \sigma'(z_i^{(2)}) 2(a_i^{(2)} - y_i)\end{aligned}$$

where x_i is the input from our training data, $\sigma'(z_i^{(1)})$ is the derivative of the sigmoid activation function with respect to each observation in the input layer, $w_i^{(2)}$ is the vector of weights in the hidden layer, and all else is the same as is in the previous equation.

Partial derivatives with respect to bias

Using the same logic, the amount to adjust the bias in the *hidden* layer is represented by:

$$\frac{\partial C}{\partial b_i^{(2)}} = \frac{\partial z_i^{(2)}}{\partial b_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial C}{\partial a_i^{(2)}}$$

Because the bias is a constant, that is $\dfrac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = 1$, the chain-rule formula simplifies to the following:

$$\frac{\partial C}{\partial b_i^{(2)}} = 1 \cdot \sigma'(z_i^{(2)}) 2(a_i^{(2)} - y_i)$$

Moving further up the chain, the amount to adjust the bias in the *input* layer is represented by:

$$\begin{aligned}\frac{\partial C}{\partial b_i^{(1)}} &= \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}}\frac{\partial a_i^{(1)}}{\partial z_i^{(1)}}\frac{\partial z_i^{(2)}}{\partial a_i^{(1)}}\frac{\partial a_i^{(2)}}{\partial z_i^{(2)}}\frac{\partial C}{\partial a_i^{(2)}} \\ &= 1 \cdot \sigma'(z_i^{(1)}) w_i^{(2)} \sigma'(z_i^{(2)}) 2(a_i^{(2)} - y_i)\end{aligned}$$

The calculations above determine the direction of steepest descent toward lowering the cost function. They describe how every i weight or bias in either layer should be adjusted in order to most effectively lower the cost for the given moment. A step is taken in that direction, and the network iterates over again. In a network with more layers L , the number of equations needed for the full back-propagation increases as well. As shown, the chain rule increases more and more as lineage tracing gets closer to the initial input layer, resulting in more and more complexity in the calculations. And because backpropagation is catered to the cost function and the activation function, the specific derivatives calculated will be different as these are changed as well.

More Optimization Algorithms

With each iteration, after the gradient is computed, a step is taken in that direction. Yet, just how far a step is to be determined. If the step is too large, a local minimum may not be found and the algorithm never converge. If the step is too sall, it may be extremely computationally intensive to reach convergence. Sometimes, it can be shown mathematically how to determine an optimal step size for special cases [11], or several step sizes can be tested [1] and an appropriate one determined. Other algorithms exist for training neural networks, either as variants to gradient descent or a completely new means; some of which aim to save computation or adjust the step size according to the model. For example, *stochastic gradient descent* saves computation by randomly sampling observations one at a time to calculate the gradient and estimate network parameters. With each iteration, another observation is sampled and the parameters updated. Other methods like Adagrad (Adaptive Gradient Algorithm), RMSProp (Root Mean Square Propagation), and RProp (Resilient Backpropagation)[12] aim to adapt the *step size* rather than maintaining a fixed value for each iteration.

Types of Neural Networks

The previous section described a Multi-Layer Perceptron network in which connections exist between all neurons of all layers in a dense fashion. This is pretty typical as an introduction into ANN's. This section is devoted to other network types and their most practical uses. While not exhaustive, they are worth noting. Any of them can be specialized with different features like layer sizes, optimizers, and more to match a specific task. Listed are a few primary network types and their general purposes.

Convolutional Neural Networks (CNN)

A Convolutional Neural Network (CNN) is best suited for data that has a grid-like topology [1]. Image data is a premier use for a CNN. Image data is enormous; one pixel is a parameter value that needs to be estimated. CNN's prevail over the MLP for computer vision tasks because the image can be assessed in small pieces at a time, rather than all at once in a densely-connected network. What defines a CNN is a neural network that uses the **convolution** operation in at least one of its layers. The convolution operation is as follows:

Convolution operation continuous

$$C(t) = \int x(\tau) w(t-\tau) d\tau$$

Convolution operation discrete

$$C(t) = \sum_{\tau=-\infty}^{\infty} x(\tau) w(t-\tau)$$

$x(t)$ is the **input** and $w(t-\tau)$ is known as the **kernel**. The shorthand operation for the convolution operation is denoted with an asterisk:

$$(x * w)(t)$$

In image data, the kernel is a discrete matrix that slides across the image and computes the scalar product for each cell value. This allows it to flag visual patterns by expressing numerically where they see certain features of an image and about what they are [13]. This makes for a few notable assumptions CNN's have [1]: for one, the output can be seen as a weighted average at every moment the convolution operation is performed. Thus, w must be a valid probability distribution or this ceases to be the case. Additionally, values of the kernel and input tensors are assumed to be zero in infinite space except for the finite set in which the values are stored (i.e. the dimension of the image). With all of these in play, the convolution produces an infinite summation over the finite array of elements, as represented below.

Two-dimensional image X with a two-dimensional kernel W

$$C(i,j) = (X * W)(i,j) = \sum_m \sum_n X(m,n) W(i-m, j-n)$$

m and n are the representative pixel locations in the relevant dimension, while i and j are the kernel locations, where the convolution is taking place. The infinite summation is made possible by the fact that values are zero wherever the image is not present.

Convolutional neural networks are not limited to two-dimensional image data. The convolution operation can be performed on time series data [1], in which the kernel runs across a one-dimensional array at each timepoint. It can also be escalated to three or more dimensions. For color images, the third dimension defines the RGB color scale [14]. The appendix for this thesis contains R code for a CNN that classifies small images with the `keras` package.

Recurrent Neural Networks (RNN)

Recurrent Neural Networks do not act in the same $forward \leftrightarrow backward$ manner of a typical multi-layer perceptron. Rather, they are set up to process information in two directions at any time. Connections between nodes of a RNN can upcycle, allowing for the output of one neuron to be input into a neuron before it [15]. These feedback loops make RNN's especially useful for sequential data, such as language/text classification and time series forecasting. As information is added simply by using the model, the model learns new patterns.

Generative Adversarial Networks (GAN)

A unique element of these networks is that they are actually comprised of two networks - a *generator* and a *discriminator* - to compete against one another and generate new data. This is typically applied in image generation applications. The generator creates random data to send to the discriminator, which is designed to differentiate between real training values (i.e. labeled images) and the fake generated ones. These two networks interact until the generator starts to produce realistic data that the discriminator cannot easily differentiate from the training data. This network type deviates from the other common types above in that it has a source of random variation in its training.

At the cutting edge, neural networks can be built with the `keras` package in R or Python. `keras` is a high-level API centered on the construction of deep learning models in a fast, user-friendly environment [16]. It uses the TensorFlow engine built by Google and is written in Python by François Chollet, one of the main contributors of TensorFlow. `keras` is fully integrated with R, though an installation of Python is required to run the package. Features of the `keras` package are detailed in the appendix of this thesis, but more extensive tutorials on how to build neural networks this way can be found in (Rai, 2019 [14])

Techniques to Improve Model Performance

The primary objective in machine learning (and therefore deep learning) is to perform well on new, unseen data. The central challenge is finding the ideal balance between **overfitting** and **underfitting**. In order to accomplish this balancing act, model performance is rated by its prediction on a test set and regulated to preserve the flexibility of a high-parameter network.

Rating Model Performance

During the build process, a model is faced with a training set and measured against a test set. With the training set, the model engages its optimization techniques (i.e. gradient decent) in order to minimize a cost function (i.e. least squares). The measure of error that is used to optimize parameters based on reducing its theoretic value is known as the *training error*. When model performance on new predictions is measured against the test set, the expected value of the error between predictions and actual test data is the *test error* (commonly referred to as the *generalization error*). [1]

When the model cannot obtain a sufficiently low training error, it is *underfit*. This happens when a linear model is applied to a non-linear set of observations, as shown in Figure 8. Additional processing is enrolled to accommodate the model to the data contours, creating a model with a lower error; but attempting to fit the model too closely to the data creates a model with additional bias. Often times, when a model is optimized toward a minimum training error, test error passes through a minimum rather than decreasing monotonically [17]. When the gap between the training error and test error is too great, the model is *overfit*.

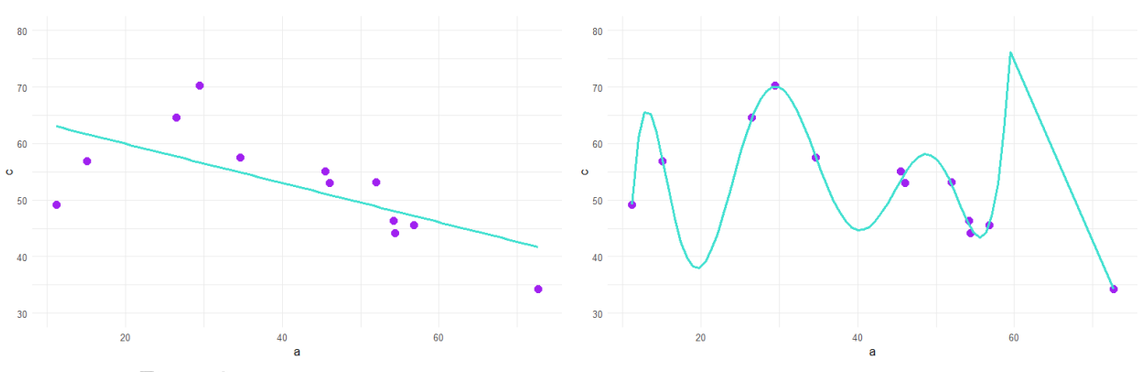


Figure 8: The regression model to the left has minimal capacity. It underfits the data. The model to the right depicts an overfitting model.

Somewhere between these extremes is the model's optimal *capacity*. Capacity is a model's ability to fit a wide variety of functions, and machine learning models perform best when their capacity is appropriate for the complexity of the task [1]. In linear regression, increasing the model's capacity would be to include polynomial terms or spline regression to shape the model beyond a straight line, as shown in Figure 8. For neural networks, adding additional neurons or hidden layers increases model capacity. Because a neural network model has many more parameters to be estimated than its linear regression counterpart, it is much more susceptible to overfitting. As capacity increases, so too does the amount of data needed for the network to be able to generalize [18].

Addressing Model Performance

There is no one way to set the dials of a neural network. Generally, they are established by trial and error, train/test split, or more sophisticated techniques as cross-validation to compare networks trained with different parameter values [19]. The class of techniques used to calibrate the model toward training data with a focus on generalizability for new data is known as **regularization**. This is really any modification made to the learning process that aims to reduce test error (but not training error) [1]. This thesis covers three common techniques to regularize neural networks: *Early stopping*, *dropout*, and *weight decay*.

Early stopping is when the optimization algorithm is halted before the test error increases too much. It is hoped that the algorithm stops at an optimal model capacity. To do this, as the model is being fit to the training data, its test error is simultaneously calculated. Often, the algorithm stops when the test error reaches a minimum. [20]

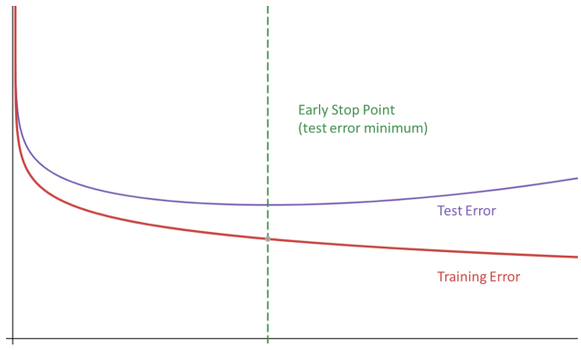


Figure 9: Often, as training error is reduced, test error passes through a minimum. early stopping generally tells the optimization algorithm to stop when this is reached.

Dropout refers to the technique of temporarily removing neurons (and all involved connections) at random at each iteration of training.[21] For example, a dropout with control parameter $p_{drop} = .25$ means that at each cycle of training, a random 75% of the network is used and a new 75% is sampled for the next step. This simultaneously speeds up computation as well as prevents overfitting because it reduces the number of calculations at each step of the model building process.

Weight decay is the neural network equivalent to shrinkage in machine learning. For more information on shrinkage and regularization techniques in traditional machine learning, see (Goodfellow, 2016) [1]. Weight decay is when a term is added to the optimizer to penalize weights in hopes to achieve a smoother fit [19]. The penalty term forces the weight coefficients to be small because the optimizer seeks to reduce the overall function. To illustrate, suppose a network architecture $\mathcal{A}$ is defined for a model. This architecture contains the specifications of the number of layers, neurons in each layer, activation functions, and available connections between layers. The function mapping is defined as $\hat{y}(x_i; W, \mathcal{A})$, which outputs an estimated value for each input x_i from the given architecture $\mathcal{A}$ and set of weights W. If the gradient descent algorithm is used to find optimal parameter values based on minimizing the following loss function E_D:

$$E_D(D|W, \mathcal{A}) = \sum_i [\hat{y}(x_i; W, \mathcal{A}) - y_i]^2$$

the learning algorithm sets to reduce E_D with the implementation of a *regularizer* term for the weights E_W. This regularizer can be defined as the following:

$$E_W(W|\mathcal{A}) = \sum_j w_j^2$$

The full optimization algorithm that uses weight decay then seeks to minimize:

$$F = \alpha E_W(W|\mathcal{A}) + \beta E_D(D|W, \mathcal{A})$$

where α and β are "black box" parameters [19]. α is a control parameter known as the *regularizing constant* (commonly referred to as the decay rate) and β is the learning rate (step size). It can be seen that as $\alpha \to \infty$, $W \to 0$ since the goal is to minimize F. Rules of thumb exist for setting these control parameters, and cross-validation can be applied to determine their optimal values as well. In a later section, a Bayesian method will be applied to offer another method to determining α and β.

Quantification of Uncertainty

Determining uncertainty in neural networks is not as simple as applying error bounds on the estimation. Further construction is necessary to determine the level of confidence in a deep network's abilities, which stem from several methods. The methods discussed here come from (H. M. Dipu Kabir, et.al) [4]. Thorough notation and detail will be spared as this thesis is not a comprehensive examination of uncertainty tactics. Rather, methods will be discussed in their elementary form to later illustrate the ease of using Bayes as an alternative.

The *Delta method* uses the same principles as finding the slope of a curve from on $\frac{\Delta y}{\Delta x}$. It turns a nonlinear pattern into a linear approximation. More information can be found in [22] and [23]. Delta method for deep learning is performed in two phases: the first using first-order Taylor expansion to approximate the covariance matrix of the outputs and the second making predictions from this approximation alongside the network predictions to measure uncertainty. It is found that the covariance matrix grows quadratically with the number of parameters, making it extremely computationally costly to calculate, store, and use. Further, when early stopping is used to train a network to prevent overfitting, the delta method results in wider prediction intervals [4].

The *Mean Variance Estimation Method (MVEM)* uses a dedicated neural network to obtain the mean in tandem with the network geared toward finding the variance. The network to find the mean ($NN_{\hat{y}}$) is trained, followed by the network to estimate the variance (NN_{σ}) based on the parameter values of the first network. This method allows extreme flexibility to account for heteroscedastic variance of target predictions. There are no restrictions on the architectures of the two networks; they do not even have to be the same. Error bounds can be calculated based on the estimated variance of the secondary neural net. The main drawback to MVEM is that it assumes ($NN_{\hat{y}}$) accurately estimates the target values y_i because (NN_{σ}) uses it to calculate its parameter values. This can make for over- or underestimated error bounds.

The *Bootstrap Method* is the most popular among traditional prediction interval uncertainty quantification. It is a method of treating the training data as the population and sampling from it with replacement to generate a distribution for the statistic of interest. In neural networks, a common technique is to generate B training data sets by bootstrap resampling and build B neural network models from them. The mean and variance of all point estimates from each neural network is calculated and used to create the model uncertainty. This can be computationally costly, especially for data-hungry deep networks with many neurons in each layer.

Each of these methods has specialized variations to generate the most useful measurements of uncertainty; even to begin with it is only a start. Each has been shown to have its own preference over others and pitfalls. Indeed, as if building a neural network was not complex enough, the additional computations to deliver insight to the confidence of a network's predictions can be quite cumbersome. At the apotheoses of this thesis, it will be shown that applying Bayesian techniques to building a deep learning model yields uncertainty measures built right into the model itself.

Bayesian Statistical Methods

Often called inverse probability in its budding stages, [24] Bayesian Statistics is an approach to analyzing data by use a theorem developed by the Reverend Thomas Bayes; although the theorem was not published until after his death. Bayes' Theorem (or Bayes' Rule) describes the conditional probability of an event. The frequentist approach describes probability as the relative frequency of a favorable outcome as the number of samples increases to infinity. In Bayes. probability is representative of the current state of knowledge on the potential outcomes based on prior knowledge.

Consider a tiny example where a coin was determined to have three potential levels of fairness assigned: $P(tails) = .75, P(tails) = .5, P(tails) = .25$. The frequentist would flip as many times as necessary to test a hypothesis with high enough power to determine the fairness of the coin beyond reasonable doubt. The Bayesian would assess the distributional relationship for the fairness of the coin based on prior knowledge and the likelihood of its current outcome. Suppose after two flips the choice came up (heads,heads). Since the set of possible outcomes is (H,H),(H,T),(T,H),(T,T), there is only one way out of four to produce (H,H). This number is multiplied by each value of fairness to determine the potential for the true fairness value.

$$\begin{aligned} P(tails) &= (.75, .5, .25) \\ P(tails|H,H) &= P(H,H) \cdot [P(heads) = 1 - P(tails)] \\ &= \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16} \\ &= \frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8} \\ &= \frac{1}{4} \cdot \frac{3}{4} = \frac{3}{16} \end{aligned}$$

Therefore, based on the prior knowledge of potential fairness levels, the most likely fairness is that the coin has a probability of showing tails of .25. However, the distribution of likely fairness levels is very wide so the other options should not be ruled out. What is important is that the parameter assignment has a distribution that still gives usable information even with very little data. As more flips are conducted, more probability density will accumulate toward the true $P(tails)$ identified by the frequentist's estimation; although under Bayes it will always be interpreted as a random variable.

Fundamentals of Bayesian Inference

It is important to note that to apply statistical methods by use of Bayes does not violate any accords of frequentism and that it does not stand as an opposition to the frequentist analogue. Bayesian methods simply offer another perspective, with some useful features in specialized cases - one of which is the ability to generate mathematically savvy results with very little data. The goal is not to construct a model to estimate parameters as fixed values. Rather, parameters are treated as random variables themselves, with the same properties as any other random variable - each could have a mean, a variance, minimum maximum, etc., and the distribution of that parameter reveals that information. Whereas frequentism often employes maximum likelihood modeling $p(D|\theta)$ to find the parameters θ which give the most likely estimation of the data, a Bayesian probabilistic approach finds the most likely distribution of parameters from the data $p(\theta|D)$[25]. Bayesian inference (by Bayes' Theorem, given data set D) is as follows:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{\int_{\theta'} p(D|\theta')p(\theta')d\theta'}$$

This says that for each value ofind tf the parameter, count all the ways in which the data can be produced, and multiply that sum by the plausibility that the parameter is that value. Do this for every parameter value, and divide by the sum of the product. This gives the updated distribution of parameter values.

From Prior to Posterior

To determine the posterior $p(\theta|D)$ requires selection of a prior $p(\theta)$ and determination of the likelihood of the data under supposition of θ, $p(D|\theta)$. The denominator is often regarded as the *normalizing constant*, which ensures the outcome is a probability between 0 and 1. Without it, the posterior is represented as being proportional to the prior and the likelihood $p(\theta|D) \propto p(D|\theta)p(\theta)$, but not interpretable as a posterior probability. The issue tends to be in that the normalizing constant is a hardship to calculate exactly and often impossible at all. This is where estimation techniques come in, but more on that later. Typically, priors

within the same family as the likelihood (i.e. a Bernoulli likelihood and a Beta prior) have analytically helpful properties [25]. These *conjugate priors*, while mathematically useful, often place undesirable assumptions on the model. Therefore it is usually preferred not to rely on conjugate priors in order to sacrifice mathematical precision for model accuracy by approximation.

Bayesian Updating is a concept in which the posterior distribution can be reused as the prior distribution when more data arrives [26]. In the coin flip example, this would be the addition of more flips, generating a more narrow distribution centered on the most likely value of coin fairness based on the flip results. Each time the coin is flipped, new prior probabilities for coin fairness would be generated based on the outcome from the previous flips. This is a useful attribute because the original prior outlines some assumptions of a model, which are then used to compute the posterior, the conditional probability of the parameters in light of those assumptions and the data likelihood. By holding on to the posterior, the model can predict more realistic expectations based on the previous assumptions and the pattern of data that the model has already seen before.

Practical Computing

As mentioned, the premier disadvantage of Bayesian inference is that is requires computationally intractable integrations. In response, a world of practical approximation techniques exists, some of which will be discussed in this section. These techniques use a valid distribution to approximate the posterior and can be shown to still have pristine results while dodging tricky or impossible integrals [27]. Simpler models can be estimated by means of grid approximation [26], in which the domain of the posterior is split up into many finite intervals and values calculated accordingly. Quadratic Approximation is another method for approximating the posterior. It assumes the area by the peak of the posterior distribution is approximately normally distributed, which is common [26], and approximates that distribution because the logarithm of a normal distribution forms a quadratic function. Both of these techniques pail in comparison to use of **Markov Chain Monte Carlo** Methods. These methods sample the normalizing constant denominator to adequately represent the original posterior distribution to a certain degree. In essence, a Monte Carlo method is one which uses the effect of repeated random sampling to generate results, and a Markov Chain is an iteration where the t^{th} outcome is dependent upon the $(t-1)^{th}$ outcome. MCMC is a class of algorithms based on this feature. Some algorithms below are described for their techniques to estimating very closely a complex posterior distribution. Unless otherwise cited, descriptions come from (McElreath, 2016 [26]) and (Gelman et. al, 2021 [28])

Metropolis Algorithm

The Metropolis Algorithm is named after the first author of the published article that introduced the world to the research group's "Fast Computing Machines" [29]. It is the foundation from which different strategies for sampling from posterior distributions arise. The algorithm is as follows:

1. Set current point λ_0 with probability $p(\lambda_0|y) > 0$
2. Sample a proposal λ^* from a proposal distribution $J_t(\lambda^*|\lambda^{t-1})$ at time t.
3. The acceptence of the proposal is $A = min\left(\frac{p(\lambda^*|y)}{p(\lambda^{t-1})|y}, 1\right)$

This means that the proposed move will always be accepted if it is toward a higher probability position that current. Otherwise, it will be accepted with probability $\frac{p(\lambda*|y)}{p(\lambda^{t-1})|y}$

Works when the proposal distribution is symmetric, that is, when $J_t(\lambda^*|\lambda^{t-1}) = J_t(\lambda^{t-1}|\lambda^*)$

Metropolis-Hastings Algorithm

To allow for non-symmetric distributions, the Metropolis-Hastings algorithm generalizes the Metropolis algorithm. The ratio of densities becomes:

$$\frac{p(\lambda^*|y)/J_t(\lambda^*|\lambda^{t-1})}{p(\lambda^{t-1}|y)/J_t(\lambda^{t-1}|\lambda^*)}$$

and a minimum between that ratio and 1 is selected.

Hamiltonian Monte Carlo

Rather than a random walk through the posterior, HMC moves the exploration problem into a space containing a position parameter λ and a momentum parameter r. HMC has the ability to make confident, far stride and as a result converge faster than other algorithms [25]. The parameters of HMS are step size s and number of steps L. Sampling from a standard multivariate normal distribution, Leapfrog updates are made to determine the proposed position $\tilde{\lambda}$ and momentum $\tilde{r}$. The proposed values are determined, through use of Laplace transform $\mathcal{L}$, by

$$min\left(1, \frac{exp[\mathcal{L}(\tilde{\lambda} - \frac{1}{2}\tilde{r} \cdot \tilde{r}]}{exp[\mathcal{L}(\lambda^{m-1}) - \frac{1}{2}r^0 \cdot r^0]}\right)$$

and, if accepted, the new position is $\lambda^m \leftarrow \tilde{\lambda}$ and momentum $r^m \leftarrow -\tilde{r}$

Gibbs Sampling

Gibbs Sampling is another variant of M-H that uses clever proposals based on the criterea of alternating conditional sampling

Two-variable example: [30]

$p(x, y)$ is a probability density that can be sampled more easily as conditional distributions $p(x|y)$ and $p(y|x)$. Gibbs Samping is as follows:

1. initialize (x_0, y_0)
2. sample $x_1 \sim p(x|y_0)$, then $y_1 \sim p(y|x_1)$, then $x_2 \sim p(x|y_1)$, then $y_2 \sim p(y|x_2)$
3. ...
4. sample $x_n \sim p(x|y_{n-1})$, then $y_n \sim p(y|x_n)$

In higher dimensions:

- λ_j represents the subvector that makes up the components of λ. For example, a parameter representing categories "blue", "red", "green", and "yellow" has four components.
- At each iteration t, each level of λ_j^t is sampled (λ_{blue}^t, λ_{red}^t, λ_{green}^t, λ_{yellow}^t)
- $p(\lambda_j|\lambda_{-j}^{t-1}, y)$, where λ_{-j}^{t-1} represents all components of λ except for λ_j.

Variational Inference

MCMC methods are the bets methods for approximating the posterior with samples from the exact posterior, but their Achilles' heel lies in their lack of scalability [31]. Variational inference methods convert an integration problem into an optimization problem [25]. They introduce a distribution $q(\theta; H)$ called the *variational distribution* which is parameterized by H. $q(\theta; H)$ must be flexible enough to provide a wide variety of posterior distributions in order to capture the posterior $p(\theta|D)$. Closeness between the variational distribution and the true posterior is measured by Kullback-Leibler divergence [32]:

$$KL(q||p) = E_q\left[log\frac{q(\theta; H)}{p(\theta|D)}\right]$$

The family of parameters H is optimized to minimize $KL(q||p)$, in a similar manner that gradient descent seeks to optimize network parameters. Much of variational inference has been omitted for this thesis, but extensive detail on these techniques can obtained from (Blai, 2017 [32]).

Bayesian Data Analysis

A nice feature of Bayesian statistics is the ability to incorporate prior knowledge into the model, systematically updating the relative beliefs in parameters as more data becomes available. Even with few data points, a model can be built (although exhibit a high level of uncertainty), and from it added upon to build a better model. Another nice property of Bayes is the ability to extract the desired distribution of interest independent of the particular parameter settings [6]. Between the two of these, meaningful predictions can be extrapolated without having to employ cross-validation or bootstrapping techniques.

Marginalization

Bayesian inference makes use of *marginalization*, a method by which nuisance parameters are integrated out rather than estimating all model parameters. Marginalization reduces the dimensionality by considering the sum of the estimated probabilities under each condition of the nuisance parameter. That is, for two random variables Y_1 and Y_2 within the probability functiuon $p(y_1, y_2)$, the marginal probability of Y_1 would be $p(y_1) = \sum_{y_2} p(y_1, y_2)$ for discrete variables and $p(y_1) = \int_{y_2} p(y_1, y_2)$ for continuous. [33]

The same principle is applied here. Suppose a model is determined by use of Bayes. Its frequentist counterpart may use Ordinary Least Squares to find a set of parameters θ which minimizes its error. In Bayes, given a data set D, the parameters θ can be expressed by the posterior distribution $p(\theta|D, \alpha, \sigma^2)$. Here, α is the decay rate seen above derived from the choice of prior, which assumes residual normailty with variation σ^2 (see Tipping, 2004 [27] for derivation). To determine the distribution of new predictions y^* requires integrating the product of the posterior and the likelihood of new predictions with respect to the parameters of the model:

$$p(y^*|D, \alpha, \sigma^2) = \int p(y^*|\theta, \sigma^2)p(\theta|D, \alpha, \sigma^2)d\theta$$

Therefore, $p(y^*|, \alpha, \sigma^2)$ is the *posterior predictive distribution* of predicted outputs y^* under the circumstances set forth in the model. If θ and σ^2 were known to be true, then the likelihood would determine the prediction for future data. Since these are unknown, and because under Bayes parameters are random variables, the predictive distribution is a weighted average over the posterior. Thus, the predictive distribution can be interpreted as the expectation of the single network likelihood under the posterior [34] [31]. Setting aside the complex notation, this is simply a means of generating predictions from the parameter distribution gained by use of Bayes' Rule. It describes the uncertainty the model has and allows for better flexibility without supplemental engineering. With few data points to train from, the predictive distribution will be very sparse. [27]

Ideally, to be fully Bayesian and practice the mastery of marginalization is to integrate out *all* variables that are not directly related to the task at hand [6]. If the goal is to generate a distribution for the prediction of y^* given the training labels D, the desired distribution is $p(y^*|D)$. For all other parameters, *hyperpriors* $p(\alpha)$ and $p(\sigma^2)$ are defined. The general Bayesian predictive framework employs the full posterior [27]:

$$p(\theta, \alpha, \sigma^2|D) = \frac{p(D|\theta, \sigma^2)p(\theta|\alpha)p(\alpha)p(\sigma^2)}{\iiint p(D|\theta, \sigma^2)p(\theta|\alpha)p(\alpha)p(\sigma^2)\ d\theta\ d\alpha\ d\sigma^2}$$

Bayesian inference would proceed with integrating the product of the likelihood of predictions y^* by the full posterior to get $p(y^*|D)$:

$$p(y^*|D) = \iiint p(y^*|\theta, \sigma^2)p(\theta, \alpha, \sigma^2|y)\ d\theta\ d\alpha\ d\sigma^2$$

The equation above represents the distribution of model predictions based solely on the data. It is independent of model parameters, control parameters, or noise assumptions. However, this moves beyond the scope of this thesis. What will be sought in future sections is the predictive distribution given the certain criteria set forth in the model; that is, for a model with specified assumptions to learning, the predictive distribution comparable to $p(y^*|D, \alpha, \sigma^2)$ marginalized over its parameters θ.

Bayesian Regularization

The first chapter of this thesis described weight decay as a regularization technique. Recall it left with the following full-optimization formula:

$$F = \alpha E_W(W|\mathcal{A}) + \beta E_D(D|W, \mathcal{A})$$

This section will apply a Bayesian Approach to setting α and β parameters, as told by (Mackay, 1992) [19] and (Forsee and Hagan, 1997) [35]. To begin, a probabilistic interpretation of network learning [19] is introduced. Under this framework, the weights of the network are represented as random variables, rather than fixed, unknown, and estimable. By Bayes' rule:

$$P(W|D, \alpha, \beta, \mathcal{A}) = \frac{P(D|W, \beta, \mathcal{A})P(W|\alpha, \mathcal{A})}{P(D|\alpha, \beta, \mathcal{A})}$$

where D is the observed data and $\mathcal{A}$ is the network architecture.[1] Taking into assumption that the model residuals follow a normal distribution, the likelihood function [35], representing the probability of the data given weights W, is

$$P(D|W, \beta, \mathcal{A}) = \frac{1}{Z_D(\beta)} e^{-\beta E_D}$$

and by establishing the prior distribution as normal, its density is represented as:

$$P(W|\alpha, \mathcal{A}) = \frac{1}{Z_W(\alpha)} e^{-\alpha E_W}$$

for which

$$Z_D(\beta) = \left(\frac{\pi}{\beta}\right)^{n/2} \text{ and } Z_W(\alpha) = \left(\frac{\pi}{\alpha}\right)^{N/2}$$

Putting this all together, the posterior distribution can be represented as:

$$P(W|D, \alpha, \beta, \mathcal{A}) = \frac{\frac{1}{Z_W(\alpha)} \frac{1}{Z_D(\beta)} e^{-(\beta E_D + \alpha E_W)}}{P(D|\alpha, \beta, \mathcal{A})} = \frac{e^{-(\beta E_D + \alpha E_W)}}{Z_F(\alpha, \beta)}$$

The exponent in the numerator is the negative of F. Thus it can be noted by the formula above that under Bayes, the optimal weights should maximize the posterior probability, which is equivalent to minimizing the full optimization formula $F = \alpha E_w(W|\mathcal{A}) + \beta E_D(D|W, \mathcal{A})$ as seen in chapter 1.

Optimizing the Regularization Parameters

By use of Bayes' rule, the posterior probability for the parameters α and β is:

$$P(\alpha, \beta|D, \mathcal{A}) = \frac{P(D|\alpha, \beta, \mathcal{A})P(\alpha, \beta)}{P(D|\mathcal{A})}$$

Suppose a uniform prior is chosen for (α, β). By this, the posterior distribution would be maximized by maximizing the *likelihood* function. What's more is that the likelihood function is the normalizing constant

[1] Note that (Mackay, 1992) defines a term $\mathcal{R}$ as the chosen or "prior" regularizer, which represents the potential for selecting alternative control parameters for for E_W. If included, Bayes' rule would then be represented as:

$$P(w|D, \alpha, \beta, \mathcal{A}, \mathcal{R}) = \frac{P(D|w, \beta, \mathcal{A}, \mathcal{R})P(w|\alpha, \mathcal{A}, \mathcal{R})}{P(D|\alpha, \beta, \mathcal{A}, \mathcal{R})}$$

and subsequent formulation would be modified to represent $\mathcal{R}$. However, this example only considers the sums of squares regularizer E_W from chapter 1 and so the extended notation has been be omitted for this thesis.

above. With algebra, and by maintaining the previous assumptions, the likelihood function would take the form:

$$P(D|\alpha,\beta,\mathcal{A}) = \frac{P(D|W,\beta,\mathcal{A})P(W|\alpha,\mathcal{A})}{P(W|D,\alpha,\beta,\mathcal{A})}$$

and since the heavy integration of the normalizing constant is already simplified, this equation can be computed.

$$\frac{\frac{1}{Z_W(\alpha)}\frac{1}{Z_D(\beta)}e^{-(\beta E_D+\alpha E_W)}}{\frac{1}{Z_F(\alpha,\beta)}e^{-F(W)}} = \frac{Z_F(\alpha,\beta)}{Z_D(\beta)Z_W(\alpha)}\cdot\frac{e^{-(\beta E_D+\alpha E_W)}}{e^{-F(W)}} = \frac{Z_F(\alpha,\beta)}{Z_D(\beta)Z_W(\alpha)}$$

The denominator is comprised of known constants $Z_D(\beta)$ and $Z_W(\alpha)$. What is left is to determine $Z_F(\alpha,\beta)$. Having outlined the Bayesian techniques, this thesis will not cover this decomposition. In practice, it requires estimation by Taylor series expansion and computation of the Hessian matrix of the full optimization formula $H = \beta\nabla_D^E + \alpha\nabla^2 E_W$ to determine the number of effective parameters in the neural network that are reducing the error function. This decomposition can be found in (Mackay, 1992) and (Forsee and Hagan, 1997). This technique to setting parameters α and β sets the stage for the final chapter of this thesis. As will be shown, combining the complexity of neural networks with the probabilistic framework of network learning provides transparent interpretations to complex abstractions within data.

Bayesian Neural Networks

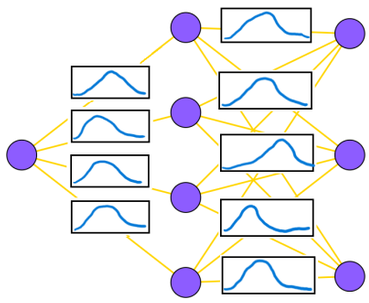


Figure 10: A two-layer Bayesian Neural Network with parameters as distributions rather than strict point estimates.

In the "traditional" non-Bayesian neural network framework, to train a neural network to minimize its error based on a single point estimate for each w_i, waged against a cost function, is to maximize the likelihood of the training data (i.e. finding weights w_i that maximize $P(D_y|D_x, \theta)$ [5] [36]. The drawback to an ANN is that the distribution of network parameters $\theta = (W, b)$ across the network is unknown. [25] Without supplemental engineering, measurements of the model's uncertainty cannot be quantified.

In Bayes, an interval of potential parameters based on probabilities $p(\theta|D)$ is computed by use of Bayes' Rule. As such, the network can express uncertainty in its weights and biases through this posterior distribution. What's more is by the behooving feature of marginalization, uncertainty can be quantified for the predicted outputs $p(y|D)$ (for either y as a regression prediction or classification label); even the very architecture of the model $p(\mathcal{A}|D)$ can be expressed as a distribution [5]. Bayesian Neural Networks (BNN) are able to learn from few data points and tell more of the story than their ANN counterparts. Usually, fewer data points results in a network with higher prediction variability. Even a network with very low variability, say one trained with many data points, comes readymade with a regularizer to ward overfitting. [37]

Architecture

Conceptually, the overall architecture remains when working with probabilistic (Bayesian) networks. The selection of layers, neurons, activation and so on remain merely the same. Indeed, more complex network types such as Convolutional Nets [38] can be implemented and trained by a Bayesian approach. The overall model does not have to be restricted by Bayes alone; it can be one layer or more in a network of otherwise point-estimate layers [31]. The fundamental aspect of BNNs is their ability to be trained with a probabilistic approach; which means that the same network trained more than once will (very likely) never be tuned exactly the same way.

Stochastic Modeling

A researcher unsatisfied with a single point estimate for network output may consider simulating multiple networks, introducing a probabilistic effect to parameter tuning so as to not generate the same results. Stochastic neural networks, which use either stochastic activations [39] or stochastic weights, simulate a set of possible networks $\mathcal{N}$ with associated probability $p(\mathcal{N})$. Aggregating independent predictors can lead to better predictions than single point estimates [31]. In comparing the predictions of multiple samples of $\hat{y}_i$, stochastic models better measure uncertainty. A simple example is the regularization technique dropout [40] described earlier. A neural network trained with dropout has a component of randomness introduced by the selection of neurons removed from training. If the dropout model was trained over and over again, it would contain slightly different parameter values reflective of this random noise. There are other ways to introduce stochasticity into the model. The Generative Adversarial Network [41] introduced in Chapter 1 implements a probabilistic component for its discriminator to identify. Simply put, Bayesian neural networks are a special case of stochastic neural networks in which the ensemble of possible models is obtained using Bayesian inference. [19]

Selection of Priors

There is no one-size-fits-all for machine learning models, and deep learning models only inflate this fact due to their enormous complexity [1]. Therefore, it can be interpreted [31] that prior assumptions are in place for all machine learning models, be it the optimization algorithm, regularizer, architecture, etc. These are implicit for non-Bayesian networks. However, with Bayes, the prior assumptions are made explicit. Yet, given the complexity of the network, it is not always intuitive how to select priors. A beginner's start for a linear regression task is to select a conjugate prior $p(\theta) \sim N(\mu, \sigma^2)$ (which is normal linear regression). This is analogous to the point-estimate weight decay regularization described in earlier chapters, but will be further discussed later in this chapter. Despite this, there is no theoretical argument that makes a normal prior better than any other [42]; it simply has nice mathematical properties.

Development

Recall that θ represents the model weights and biases (W, b). D is the training data from which the model learns, its inputs and outputs denoted with subscripts $_x$ and $_y$. Applying Bayes' Theorem, to determine the

posterior distribution of θ requires selection of a prior and determination of the likelihood of the data:

$$p(\theta|D) = \frac{p(D_y|D_x, \theta)p(\theta)}{\int_\theta p(D_y|D_x, \theta')p(\theta')d\theta'}$$

The normalizing constant in the denominator is what has been seen before: the difficult (profoundly intractable for any informative BNN's) integral that requires estimation by Markov Chain Monte Carlo or variational inference. If a normal prior is selected, the same calculation as that presented at the end of chapter 2 would apply to solve for $p(\theta|D)$; this just presents some notational simplicity. Having displayed mathematical formulation previously, the majority of this chapter maintains probability notation to exemplify the concepts of probabilistic modeling.

Inference

Marginalization takes reign when determining the distribution of network predictions. This means integrating out θ from the final model. Given the posterior distribution of network parameters $p(\theta|D)$, the distribution of network predictions $p(y|x, D)$ is calculated as:

$$p(y^*|x, D) = \int_\theta p(y^*|x, \theta)p(\theta|D)d\theta$$

Usually in practice, due to the nature of stochastic networks, a collection of samples Θ is taken from $p(\theta|D)$ to generate an approximate distribution of parameter estimates $\Phi_\theta(x)$. From this, multiple y_i are selected, which is the same as gathering the set of samples Y from $p(y^*|x, D)$ [31]. These samples are aggregated to measure uncertainty and generate an estimate $\hat{y}$.

Description of Uncertainty

A quick recap is in order for uncertainty definitions: *Aleatoric* uncertainty is the level of uncertainty due to the noise or random variation of the data (i.e. error). *Epistemic* uncertainty is the measure of uncertainty a model has (i.e. variance). BNN's allow for distinguishability between these types of uncertainty [31]. In other words, they can tell the difference between measurable and immeasurable uncertainty. $p(\theta|D)$ measures the epistemic uncertainty in the model. With few data points, the model will express a high level variation in its choice of parameters rather than blindly returning a seemingly confident answer. With more data points, this uncertainty reduces. Aleatoric uncertainty is measured by $p(y|x, \theta)$, which is the conditional probability of the predicted output given the input predictors and model parameters. This conditional probability is not isolated to Bayesian models; however, use of Bayes provides a means of inverting conditional probabilities when necessary [31].

Let $y = \Phi_\theta(x) + \epsilon$ represent the value of y from the approximated distribution of parameter estimates (with error ϵ) given input x. For regression tasks, usually models are averaged to summarize BNN predictions:

$$\hat{y} = \frac{1}{|\Theta|}\sum_{\theta_i} \Phi_{\theta_i}(x)$$

Uncertainty is computed by the *covariance matrix*:

$$\Sigma_{y|x,D} = \frac{1}{|\Theta| - 1}\sum_{\theta_i} (\Phi_{\theta_i}(x) - \hat{y})(\Phi_{\theta_i}(x) - \hat{y})^\intercal$$

For classification tasks, the estimator is the most likely class, that is $\hat{p} = max(p_i)$. Uncertainty is measured by the relative probability of each class, summarized by the average.

$$\hat{p} = \frac{1}{|\Theta|}\sum_{\theta_i} \Phi_{\theta_i}(x)$$

It will be demonstrated in a later example that, using the full strength of the stochastic network, the underlying distribution of prediction values can be approximated and visualized. [41]

Regularization with Priors

ANN's built from a non-Bayesian approach aim to minimize a loss function. As mentioned earlier, this is what maximizes the likelihood of the data. Mathematically:

$$min[E_D(D|\theta)] \equiv max[p(D_y|D_x, \theta)]$$

(For notational symmetry, $E_D(D|W, \mathcal{A})$ as was described in previous sections is now represented as $E_D(D|\theta)$)

Introducing prior assumptions, the Bayesian approach to finding the most likely point estimate from the posterior would look like:

$$max[p(D_y|D_x, \theta)p(\theta)]$$

Non-Bayesian networks introduce a regularizing term to prevent issues of overfitting. Assuming loss is calculated as the negative log-likelihood, the non-Bayesian network with a term comparable to a prior would be represented by an additive term [31]. That is:

$$max[p(D_y|D_x, \theta)p(\theta)] \equiv min[E_D(D|\theta) + E_\Theta(\theta)]$$

Where $E_\Theta(\theta)$ is the new representation of $E_W(W|\mathcal{A})$ from earlier sections. Therefore, it can be concluded that the prior distribution acts as a regularizing term for a BNN. What's more is that the distributional assumptions of a prior overtly impact the regularization technique a BNN utilizes in comparison to its ANN counterpart [37]. That is how the selection of a Gaussian prior is the BNN parallel to a non-Bayesian ANN with weight decay. A perfect starting point for future research, a simulation study may show comparisons between the priors of Bayesian machine learning models and the regularizers of their frequentist counterparts.

Survey of Models

On March 11, 2011 a magnitude 9.1 earthquake struck Japan, the epicenter just 76km from the eastern coast of the Tohoku region. This caused a tsunami that collapsed a nearby nuclear reactor resulting in the Fukushima Daiichi Nuclear Disaster, a meltdown that resulted in 19,500 total deaths. The reactor was built to withstand earthquakes up to 8.6 in magnitude. The following example, inspired by (Silver, 2015)[43] illustrates a statistical model of annual earthquake frequencies of the greater Tohoku region. Data was queried using the United States Geological Survey's ANSS Comprehensive Earthquake Catalog (ComCat). The full analysis was conducted in R 4.2.2 and is detailed in the appendix of this thesis. The purpose of this survey is to demonstrate the predictive capabilities of neural networks compared to a traditional model for this specific circumatance. Three models are tested: A Poisson regression model, a multi-layer perceptron network (MLP) with no regularization, and a multi-layer perceptron that employs Bayesian regularization in the manner described earlier (Bayesian Reglarized Neural Network, or BRNN). While no model surveyed will be used to extrapolate cases such as a magnitude 9.1, models could be apprehended for a future study and modified to accommodate such rare event predictions.

After pre-processing, the data was coerced into a table of earthquake sizes (measured by Richter magnitude rounded to one decimal place) and the average annual frequency *at or above* each size in the greater Tohoku area over the 46-year span from 1965 up to the Great Quake of March 11, 2011. The following plot was generated to display the data. There are 31 data points within the magnitude interval $[4.5, 7.7]$. Data was split 80/20 for training and testing respectively. The surveyed models will be used to generate predictions of the *expected annual frequency of each size earthquake*. These models are evaluated based on the predictive accuracy on the 20% held-out test set. A mode with a smaller test error (calculated as the mean of the square differences between predictions and test data points) supposes a preferable model.

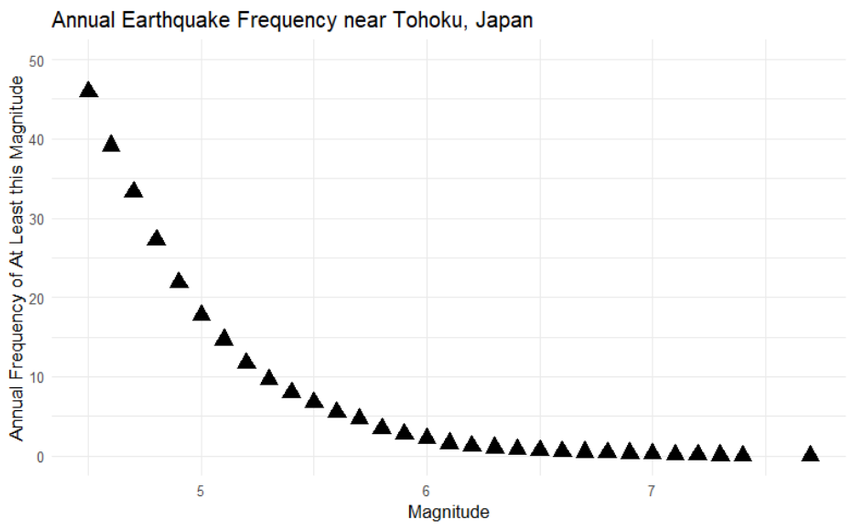


Figure 11: Annual Tohoku earthquake frequencies of magnitude 4.5 and above.

Poisson Regression

As a standard model for count data, a Poisson regression model is fit to establish a baseline to compare against neural networks. The below output is the result from the general linear model function in base R, used on the training data, with the Poisson regression model specified by `family = "poisson"`.

```
##
## Call:
## glm(formula = freqc ~ mag, family = "poisson", data = train)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -0.39403  -0.20754  -0.02925   0.08617   0.22166
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
```

```
## (Intercept)  13.0940     0.7242   18.08   <2e-16 ***
## mag          -2.0467     0.1474  -13.88   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 373.0667  on 23  degrees of freedom
## Residual deviance:   0.7758  on 22  degrees of freedom
## AIC: Inf
##
## Number of Fisher Scoring iterations: 3
```

This shows that for every unit increase in magnitude, the expected difference in the logs of annual frequency is -2.0467. To obtain a more appropriate test error, cross-validation was applied to aggregate the test errors of 200 Poisson regression models. Because the data was so sparse, the distribution of calculated test errors was very wide and contained a few outliers that skewed the resulting average. Thus, the *median* test error was taken to better measure prediction accuracy and is shown in Table 1. A plot of the data with the regression line is shown below:

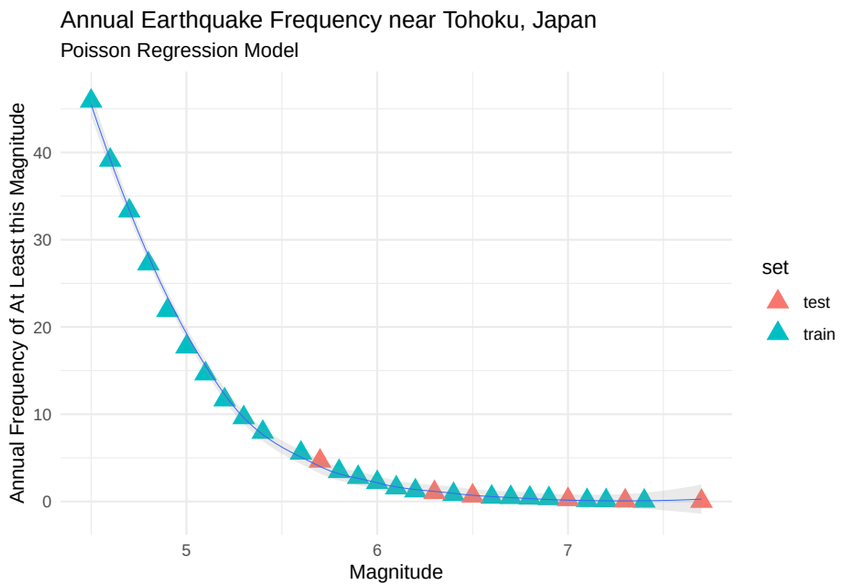


Figure 12: A Poisson regression line fit to the data.

Multi-Layer Perceptron

Next, the `neuralnet` package [44] was used to fit a multi-layer perceptron network to the training data. Several networks were tested, containing one hidden layer of differing sizes. For computational efficiency and for simplicity, deeper networks were not considered. The function uses resilient backpropagation by default and sigmoid activation between layers. A diagram of the network is displayed below:

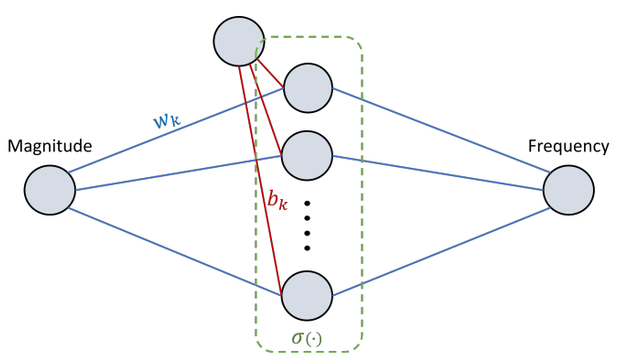


Figure 13: Two-layer MLP network(s) of various sizes, with weight and bias connections displayed. Magnitude is the explanatory variable and Frequency is the response. Sigmoid is used as the hidden layer activation.

To determine the optimal number of neurons in the hidden layer, 50 of each network was generated and the median test error calculated. It was hoped to compare as many of each neural network as the Poisson model, but this number had to be reduced to spare computational cost. The results of each aggregate model are displayed in Table 1 below.

Baseline	Test Error
Poisson Regression Model	0.1862851
Hidden Units	Test Error
3	0.2441865
6	0.2818492
9	0.1785652
10	0.2633863
20	0.1706793

Table 1: Test accuracy for a two-layer multi-layer perceptron network of different hidden layer sizes compared to the baseline Poisson model.

Based on the results under the conditions of the task, it is recommended to choose a simpler model: the Poisson regression has a comparable test error to the MLP for any size surveyed. It even out performs the MLP in most cases. By choosing a simpler model, the Poisson model would be able to extrapolate results better than any MLP surveyed here devoid of regularization.

Additionally, the cost function used in the MLP model is not appropriate for the task. By default, the `neuralnet` package uses the least squares criterion; essentially treating the model like a linear regression task. In fact, the data had to be transformed to the logarithmic scale in order for the resilient backpropagation algorithm to converge for most cases. Several attempts were made to replicate the networks used in a simulation study by (Fallah,et.al, 2009 [45]) to compare the performance of a neural network Poisson regression model with its traditional counterpart. Such a model would have to be amended to compete with a cost function for count data. Particularly, based on negative log likelihood for Poisson regression, the cost function would be:

$$E_D = -\sum_{i=1}^{N} [-\hat{y_i} + y_i log(\hat{y_i})]$$

The final result would be exponentiated to generate predictions. However no successful attempt was made using the `neuralnet` package, and instead a linear regression MLP was used. (See Appendix for more details)

Bayesian Regularized Neural Network

Recall the regularized network weights from chapter 2. Given the posterior distribution of network parameters $p(\theta|D, \alpha, \beta, \mathcal{A})$, the likelihood of new predictions $p(y^*|\theta, \mathcal{A})$, the posterior predictive distribution would be:

$$p(y^*|x, D, \alpha, \beta, \mathcal{A}) = \int p(y^*|\theta, \mathcal{A}) p(\theta|D, \alpha, \beta, \mathcal{A}) \; d\theta$$

which is the product the posterior distribution and the likelihood of the prediction given network weights and architecture, integrated over the weights. This gives an output predicted distribution of y^*, given the same prior assumptions as went into building the model before.

The final set of neural network models to be tested offers another potential solution than the MLP. It uses the `brnn` package [46] to fit a neural network with Bayesian regularization by means as described at the end of chapter 2 and in further detail by (Mackay, 1992). The models surveyed were two-layer neural networks with the same number of hidden neurons as the multi-layer perceptron.

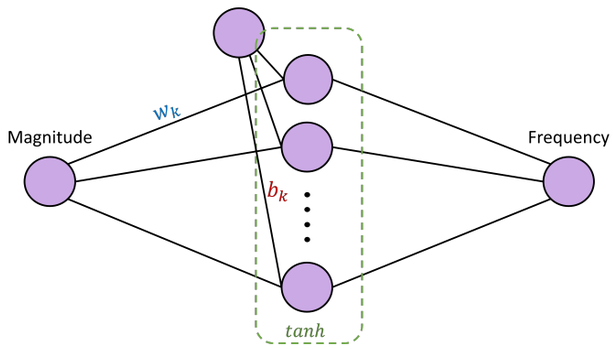


Figure 14: Two-layer BRNN network(s) of various sizes, with weight and bias connections displayed. Magnitude is the explanatory variable and Frequency is the response. Hyperbolic tangent is used as the hidden layer activation.

By default, the `brnn` package uses hyperbolic tangent activation betwen layers and the Gauss-Newton optimization algorithm (Forsee and Hagan, 1997). This means that is suffers the same drawbacks as the MLP model - it treats the task as a linear regression. Only this time, it employs a regularization technique to ease the brittle nature of a high-parameter model. Although the Gauss-Newton algorithm had little trouble converging on the standard scale, for continuity the models were built in the same manner as the multi-layer perceptron: data was transformed onto the log scale and test accuracy was measured by aggregating 50 models for the median test error.

Baseline		Test Error
Poisson Regression Model		0.1862851
Hidden Units	Test Error (MLP)	Test Error (BRNN)
3	0.2441865	0.1888387
6	0.2818492	0.1949408
9	0.1785652	0.1440102
10	0.2633863	0.1851772
20	0.1706793	0.1182969
50	-	0.4055160

Table 2: Test accuracy for all MLP and BRNN networks of different sizes compared to baseline Poisson model.

Out of all network sizes tested, the best has 20 neurons in the hidden layer. This was surprising at first, so an additional network with 50 hidden neurons was tested. This network was shown to be less accurate for predictions.

The MLP network seemed to perform as well as the baseline Poisson regression model when there were 9 or 20 hidden units. Measures of test error are comparably equal and could be a product of random variation. The MLP with 3, 6, or 10 neurons in the hidden layer is shown to have less predictive accuracy than baseline. It was expected that test error would initially decrease with the addition of hidden units and eventually increase, indicating to high a complexity for the amount of data and resulting in an overfit model. This did not follow the expected pattern, perhaps because so few networks were tested or that the model made inappropriate assumptions for count data. While the predictive accuracy of the Bayesian model was moderately stable and very close to baseline, a higher complexity model is shown to outperform the baseline Poisson regression model in light of few data points. The BRNN model with 20 neurons in the hidden layer performed the best overall. Thus it is shown that a model with more parameters and a regularizing term can outperform a lower capacity model with no added regularizer. This is paramount for neural networks because of their tendency to overfit the training data. Ultimately, the MLP and BRNN are both unfit for this particular type of data, but the BRNN with 20 hidden neurons is less inappropriate than the MLP because it has the lowest test error.

Using this neural network model, the expected frequency of any magnitude prediction has no single expected output. Rather, because the parameters are trained using Bayes, the output estimate $\hat{y}$ comes from the posterior predictive distribution $p(y^*|x = 6.1, D, \alpha, \beta, \mathcal{A})$. Because this is a stochastic model, it can be generated over and over again and different predictions simulated from the same prerequisites. Figure 15 shows this distribution based on 1000 network outputs [2] from the two-layer BRNN with 20 hidden neurons.

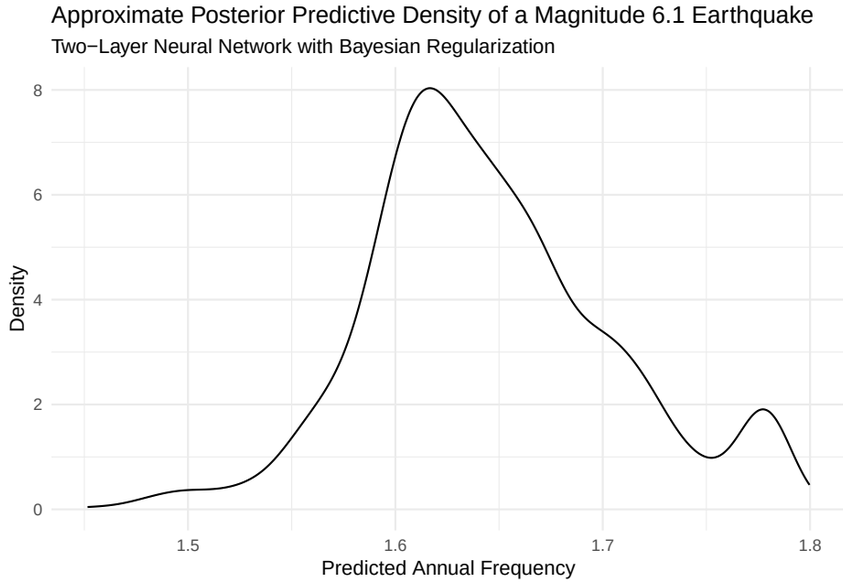


Figure 15: Simulated predictive outputs for a magnitude 6.1 earthquake from the BRNN model

The model describes the expected annual frequency of a magnitude 6.1 earthquake in the Tohoku region to be somewhere between 1.4 and 1.8 per year. Most of the density is around 1.6-1.7 per year. By use of Bayes, the entire distribution is used to tell the story.

[2]Note that there are *only* 1000 predictions, so the distribution is a rough approximations. With more attempts, or by means of other approximation techniques like those described earlier, a more accurate distributional estimate could be attained.

Conclusion

Closing Discussion

The example above surveyed several statistical models to give an applied illustration to just some of the concepts this thesis discussed. It was not intended to be a thorough simulation study on a wide variety of network types for the defined case. However, it did show some enticing results while raising a few questions: how would the MLP and BRNN have performed if they were matched to the count data? Or perhaps more interestingly, if the models were built to be able to extrapolate results, how might they compare to predict events so rare as a magnitude 9.1? The `brnn` package cannot be customized far, and the `neuralnet` package claims to be ammendable toward count data, but any attempt to do so resulted in an error. With a more appropriate set of data, large or small, it would be interesting to compare these models with a simple linear regression.

Further Research

This thesis was intended to describe from initial foundations the elementary concepts of "black-box" neural networks primarily from a theoretical perspective. It is the responsibility of the practitioner to understand the mathematical functionality of statistical models so to properly evaluate and interpret results; and better appreciate the computers that do the calculations. Neural networks are complex models with features too easily overlooked by placing a high level of trust in computers to spare the learner the mechanical details. That is why not much time is spent on the ending example. Rather, just enough time was expended on writing an example in R to discover the many packages and techniques available to build these high-level machines. If this thesis is a reader's start to deep learning, many other avenues for future study exist to build from the ideas initiated here. Being the writer's start to deep learning, this paper will close with specific objectives for future research.

Model Comparison

As promised at the beginning of this chapter, Bayes can be applied to select a model architecture [36] with appropriate capacity for a specific task. Consider three networks of different architectures, say with a different number of hidden layers. Suppose they are Bayesian networks represented as $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ with the subscript representing the number of hidden layers. Bayes can be applied to determine the distribution of network architectures:

$$p(\mathcal{A}|D) = \frac{p(D|\mathcal{A})p(\mathcal{A})}{p(D)}$$

Without justification to prefer one model over the other, the prior $p(\mathcal{A})$ would be the same. Therefore, the complexity of the model is contingent only upon the data likelihood under each model $p(D|\mathcal{A})$. Different models can be compared; the model with the highest likelihood of the data has better evidence for its predictions. Below is an arbitrary representation of these networks. More complex models fit a wider range of data [36], indicated by the wider curves. However, these have less likelihood than simpler models for certain data sets; one of which is represented by the red line.

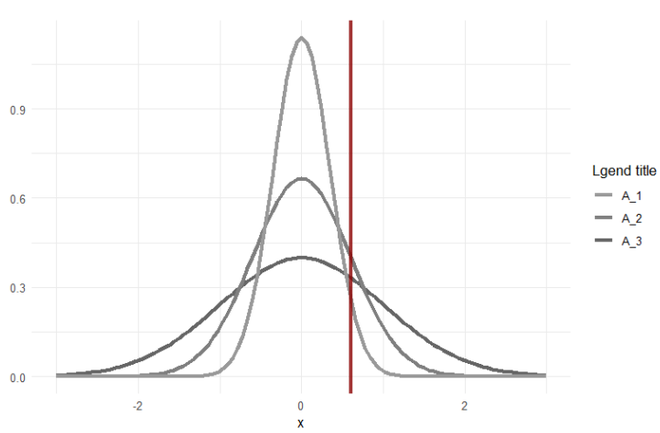


Figure 16: Arbitrary representation of network generalizability, based on model complexity controlled by the number of hidden layers. Data within the central tendency of the red line may find the network with two hidden layers to be the most preferable

It would be interesting to compare specific architectures for the Tohoku Earthquake example, or any other appropriate example too. A future simulation study may consider various data sets for the application of neural networks, trained by Bayesian or non-Bayesian methods, with Bayes to determine the optimal architecture for each specified task. Moving beyond, the application of Bayes to convolutional layers, specifically a comparison of probabilistic and deterministic models for high-level computer vision tasks, would be worth studying as well.

Poisson Regression

It was noted that the neural networks in the provided example were not appropriate for the task of identifying earthquake frequencies because they did not have the same properties as a Poisson regression does for count data. Starting from the work of (Rodrigo, Tsokos, 2020) [47] which introduces the need for nonlinearity in Poisson regression tasks, more research could be applied in cases where higher levels of abstractions can be revealed within certain count data. It would be of particular interest to study in what cases a traditional Poisson regression may prevail and at what level of complexity a Bayesian neural network (utilizing the proper cost function) may outperform.

Online Learning

Additionally, this thesis has inspired some practical examples which could be tested for very current data. One can imagine scenarios in which the data is becoming available in real time, or even situations where a model is trained incrementally on data, as though it was fed through a hopper. This makes especially interesting Bayesian techniques for online learning [48], in which the BNN is trained in this way. The Bayesian Updating rule would apply to cyclically recycle posteriors into priors in the presence of new data. Further studies on this concept would be interesting to apply to practical application, such as in stock price fluctuation or weather prediction.

BNN/ANN Regularizers

It was noted several times that the assumptions of the prior distribution directly impact the BNN, specifically the recognizable regularization technique that its point-estimate correspondent would employ. Literature [37] [49] makes note of alternative priors and their coincidence with other regularization techniques (i.e. LASSO). However, no publication exists based solely on the resemblance of probabilistic models with specific priors and their non-Bayesian counterparts. It could be noteworthy to compare models built under each paradigm and demonstrate the level of engineering each would require as well as their predictive capabilities.

Appendix

R Code for Keras Examples

Multi-Layer Perceptron

7 A standard MLP is called with the `keras_model_sequential()` function. With the below syntax, a network architecture can be defined, model compiled and optimization schemes defined, and fit accordingly to the data.

```
library(keras)
neuralnetwork <- keras_model_sequential()           # Define architecture
      neuralnetwork %>%
      layer_dense(units = 6, activation = 'relu', input_shape = c(21)) %>%
      layer_dense(units = 4, activation = 'softmax') %>%
          compile(loss = 'mean_squared_error',      # Compile model
          optimizer = 'sgd',
          metrics = 'accuracy') %>%
              fit(training,                         # Train model
              trainLabels,
              epochs = 200,
              batch_size = 32,
              validation_split = 0.2)
```

When defining network architecture, each layer can be added and their size and activation function specified. `layer_dense` indicates a fully-connected (dense) layer, meaning connections exist between all neurons of all subsequent layers. In the compilation stage, a loss function and optimization algorithm can be defined. The `metrics` argument rates the model performance. Lastly fitting a model involves commands that tell `keras` how to train and aggregate simultaneous models to generate a useful network.

Convolutional Neural Networks

In `keras`, the following code defines a two-dimensional network for computer vision tasks in a CNN. Because it is two-dimensional, it is restricted to grayscale images. For color images, a three-dimesional network is needed.

```
# Model architecture
CNN <- keras_model_sequential()
       CNN %>%
       layer_conv_2d(filters = 32,
                     kernel_size = c(3,3),
                     activation = 'relu',
                     input_shape = c(28,28,1)) %>%
       layer_conv_2d(filters = 64,
                     kernel_size = c(3,3),
                     activation = 'relu') %>%
       layer_max_pooling_2d(pool_size = c(2,2)) %>%
       layer_dropout(rate = 0.25) %>%
       layer_flatten() %>%
       layer_dense(units = 64, activation = 'relu') %>%
       layer_dropout(rate = 0.25) %>%
       layer_dense(units = 10, activation = 'softmax')
```

This network analyses grayscale images of 28x28 pixels. The first output convolutional layer size is calculated by subtracting the kernel size from the pixel size and adding 1: it is a [28-3+1,28-3+1] = 26x26 pixel image with 32 filters (defined above). Notice it is of reduced height and width, but with increased depth. The second convolutional layer size is calculated by a re-iteration of the previous formula with the updated dimension: [26-3+1,26-3+1] = 24x24. Note that CNN layers are not densely connected as they are for MLP's. Because the data is so fine-grained (i.e. one data point with a relatively low pixel value has $28 \cdot 28 = 784$ parameters), loosening the layer connection makes for more efficient processing. All-in-all, this network has 609,354 estimable parameters among its layers. Had there existed a dense connection in convolutional layers, the number of parameters would have exceeded 40 million.

Recurrent Neural Networks (RNN)

The `keras` model for RNN architecture is defined below:

```
# Model architecture
RNN <- keras_model_sequential()
        RNN %>%
        layer_embedding(input_dim = 1000, output_dim = 32) %>%
        layer_simple_rnn(units = 8) %>%
        layer_dense(units = 1, activation = "sigmoid")
```

The embedding layer of the RNN allows for ease of computation by translating words into vectorized densities of words in vocabulary. `input_dim` specifies the vector of values able to be put into the network (1000 words) and `output_dim` defines the vector length. The RNN layer `layer_simple_rnn` defines a fully-connected RNN where the output is intended to be returned to the input. Lastly, the density layer delivers the final result. Above, it is set to one unit (binary outcome). Increasing the number of units in this layer would result in more classifiers for the output based on a one-hot encoded result from the network.

Full Earthquake Workthrough

Data was queried using USGS Earthquake Catalog: https://earthquake.usgs.gov/earthquakes/search/ to select all recorded earthquakes of magnitude 4.5 and above with the following query parameters:

- latitude 35.4 - 41.2
- longitude 137.5 - 145.2
- Timeframe(UTC): 2011-03-11 00:00:00 - 1965-01-01 00:00:00

The data was stored locally in a .csv file named `earthquakes`.

The organization also has an package called `rcomcat` to query data directly into `R`, but its version was not compatible at the time of this thesis.

Data Preprocessing

```
# load the data
earthquakes_full <- read.csv("data/earthquakes.csv")

# subset data to include observations from January 1, 1965 to
#  just before the Greak Quake of March 11, 2011
earthquakes_subset <- earthquakes_full[which(earthquakes_full$time >= "1965-01-26T23:47:37.120Z" &

# fine-tune the geographic area of the model
earthquakes_subset <- earthquakes_subset[which(earthquakes_subset$latitude > 35.72 &
                                               earthquakes_subset$latitude < 40.82 &
                                               earthquakes_subset$longitude > 139.37 &
                                               earthquakes_subset$longitude < 143.37),]

# data frame of magnitudes (rounded to .1) and relative frequencies
eq <- data.frame(table(round(earthquakes_subset$mag, 1)) ) %>%
                rename(mag = Var1,
                       freq = Freq) %>%
                mutate(mag = as.numeric(as.character(mag))) %>% #change to numeric rather than factors
                mutate(freq = freq/(2011-1965))      #AVERAGE annual frequencies over the 46-year span

# create a new variable representing the frequency of earthquakes of AT LEAST that magnitude
for(i in 1:as.numeric(count(eq))){
  eq$freqc[i] <- sum( eq$freq[c(i:as.numeric(count(eq)))] )
}
```

Train/Test Split

```
# shuffle the data
df <- eq[sample(nrow(eq)), ]

# Extract 80% of data into train set and the remaining 30% in test set
train_test_split <- 0.8 * nrow(df)
train <- df[1:train_test_split,]
test <- df[(train_test_split+1): nrow(df),]

test$set <- "test"
train$set <- "train"
```

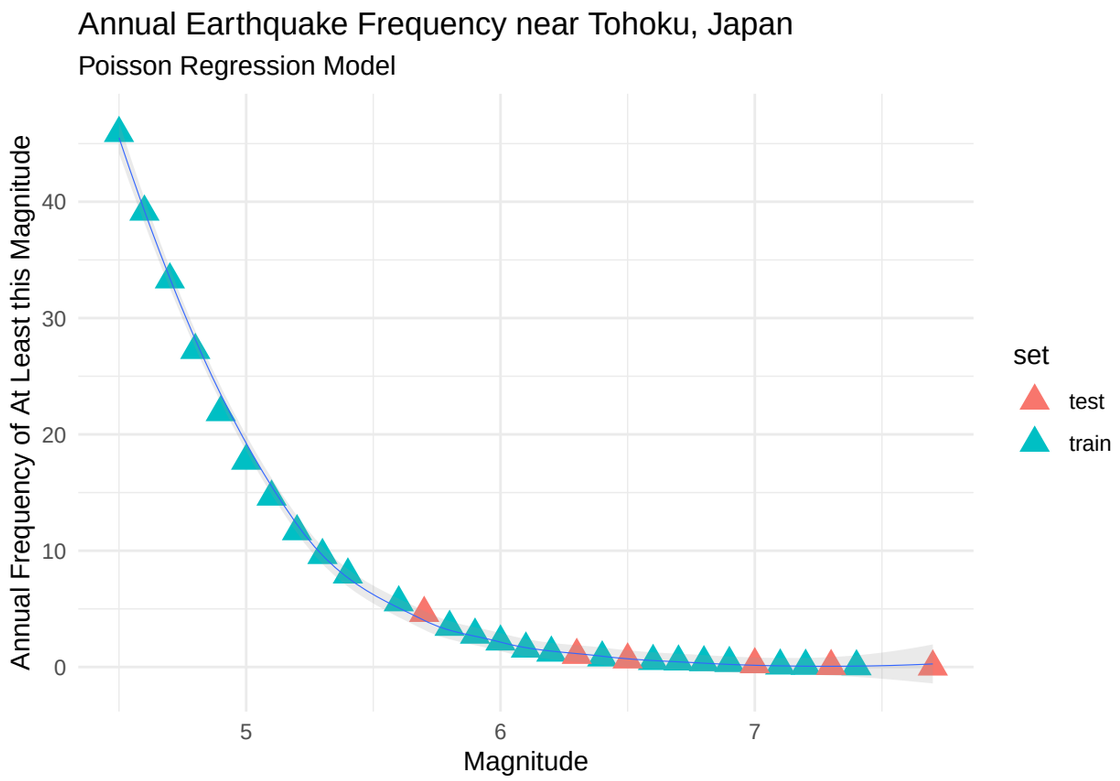
Traditional Poisson Regression Model

```
pois <- glm(freqc ~ mag, data = train, family = "poisson")
summary(pois)
```

```
##
## Call:
## glm(formula = freqc ~ mag, family = "poisson", data = train)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -0.42303  -0.17543  -0.02528   0.08986   0.22294
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  13.1813     0.7305   18.04   <2e-16 ***
## mag          -2.0651     0.1489  -13.87   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 373.58713  on 23  degrees of freedom
## Residual deviance:   0.71975  on 22  degrees of freedom
## AIC: Inf
##
## Number of Fisher Scoring iterations: 3
```

```
#---append relevant predictions to the training and test datasets
train$fitted <- predict(pois, type = "response")
test$fitted <- predict(pois, type = "response", newdata = data.frame(mag = test$mag))

#--combine data sets for plot
plot <- train[,c("mag","freqc","set","fitted")] %>%
            rbind(test[,c("mag","freqc","set","fitted")])

#---plot the data
ggplot(plot, aes(x = mag)) +
  geom_point(size = 4, shape = 17, aes(y = freqc, color = set)) +
  geom_smooth(aes(y = fitted), size = .2, alpha = .2) +
  theme_minimal() +
  labs(x = "Magnitude",
       y = "Annual Frequency of At Least this Magnitude",
       title = "Annual Earthquake Frequency near Tohoku, Japan",
       subtitle = "Poisson Regression Model")
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```




```
# TEST ERROR - MSE
poissontable <- data.frame(Model = "Poisson Regression",
                           # Hidden = NA,
                            TestErr = mean((test$freqc - test$fitted)^2))
                            #TrainErr =  sum((pois[["residuals"]])^2))
poissontable
```

```
##                Model    TestErr
## 1 Poisson Regression 0.05834858
```

Cross validation scheme to generate Test Error for Poisson regression. Due to the size of the data, distributions of TestError were skewed. So, median was taken to measure the final CV score for the network.

```
k <- 500
TestErr <- NULL
H_med <- NULL


for(i in 1:k){
  df <- eq[sample(nrow(eq)), ]

  # Extract 80% of data into train set and the remaining 20% in test set
  train_test_split <- 0.8 * nrow(df)
  train <- df[1:train_test_split,]
  test <- df[(train_test_split+1): nrow(df),]

  pois <- glm(freqc ~ mag, data = train, family = "poisson")

  #---make predictions
  testpreds <- predict(pois, type = "response", newdata = data.frame(mag = test$mag))

  #---Mean difference between predictions and actual values for test set
  TestErr[i] <- mean((test$freqc - testpreds)^2)
}

H_med <- median(TestErr)
poissontable_cv <- na.omit(data.frame(Model = "Poisson Regression", TestErr = H_med))
poissontable_cv
```

```
##                Model   TestErr
## 1 Poisson Regression 0.1862851
```

Multi-Layer Perceptron

Cross validation scheme to measure MLP hidden layer size. Due to computational limitations, only 50 of each network could be built.

It is found that the network converges faster on the log scale, so the data is transformed first.

```
k <- 50
TestErr <- NULL
H <- NULL
H_med <- NULL

for(j in c(3,6,9,10,20)){

  for(i in 1:k){
    df <- eq[sample(nrow(eq)), ]

    # Extract 80% of data into train set and the remaining 20% in test set
    train_test_split <- 0.8 * nrow(df)
    train <- df[1:train_test_split,]
    test <- df[(train_test_split+1): nrow(df),]

    train$freqc_log <- log10(train$freqc) #log transform for faster convergence
    # test$set <- "test"
    # train$set <- "train"

    mlp <- neuralnet(freqc_log ~ mag,
                     stepmax = 1e+06,
                     data = train,
                     hidden = c(j))

    #---exponentiate predictions to coerce back to standard scale
    testpreds <- 10^predict(mlp,newdata = data.frame(mag = test$mag))

    #---Mean difference between prediction and actual values for test set
    TestErr[i] <- mean((test$freqc - testpreds)^2)
  }

  H_med[j] <- median(TestErr)


}


mlptable_cv <- na.omit(data.frame(Model = "MLP", TestErr = H_med))
mlptable_cv
```

```
##    Model   TestErr
## 3    MLP 0.2441865
## 6    MLP 0.2818492
## 9    MLP 0.1785652
## 10   MLP 0.2633863
## 20   MLP 0.1706793
```

Code to generate each size MLP

Note on MLP for count data

It is worth noting here that a comparable MLP model to Poisson regression may be more appropriate for the task at hand. Particularly, the networks used in [45] to compare the performance of a neural network Poisson regression model with its traditional counterpart among various data sets were attempted for this example. According to the paper, a two-layer neural network is used that uses hyperbolic tangent in the hidden layer and the exponential function in the output. To accommodate count data, the loss function based on negative log likelihood criterion for Poisson regression is:

$$E_D = -\sum_{i=1}^{N} [-\hat{y_i} + y_i log(\hat{y_i})]$$

And so the code for the `neuralnets` package would look as follows:

```
mlp <- neuralnet(freqc ~ mag,
                 stepmax = 1e+06,
                 data = train,
                 hidden = c(5),
                 act.fct = "tanh",
                 err.fct = function(x, y) { -(-x+y*log(x))})

predictions <- predict(mlp, newdata = data.frame(mag = test$mag)) %>% exp()
```

Note that outputs would be exponentiated in order to generate predictions. This is because hyperbolic tangent is the activation for the hidden layer. In the paper, the exponential function was used to output predictions in the same way a traditional Poisson regression calculates.

Bayesian Reguarized Neural Network

Using the `brnn` package [46] to create the ideal two-layer neural network with several simulated trials. Due to computational limitations, only 50 of each network could be built.

```
k <- 50
TestErr <- NULL
H_med <- NULL

for(j in c(3,6,9,10,20,50)){

  for(i in 1:k){
    df <- eq[sample(nrow(eq)), ]

    # Extract 80% of data into train set and the remaining 20% in test set
    train_test_split <- 0.8 * nrow(df)
    train <- df[1:train_test_split,]
    test <- df[(train_test_split+1): nrow(df),]

    x <- train$mag
    y <- train$freqc %>% log10()

    #---build the model with hidden layer size i
    brnn <- brnn(y~x,neurons=j)

    #---predictions
    testpreds <- 10^predict(brnn, newdata = data.frame(x = test$mag))

    #---Mean difference between prediction and actual values for test set
    TestErr[i] <- mean((test$freqc - testpreds)^2)
  }

H_med[j] <- median(TestErr)

}
```

```
brnntable_cv <- na.omit(data.frame(Model = "BRNN", TestErr = H_med))
brnntable_cv
```

```
##    Model   TestErr
## 3   BRNN 0.1888387
## 6   BRNN 0.1949408
## 9   BRNN 0.1440102
## 10  BRNN 0.1851772
## 20  BRNN 0.1182969
## 50  BRNN 0.4055160
```

Out of all network sizes tested, the best has 20 neurons in the hidden layer.

Plot for BRNN

```
brnn <- brnn(y~x,neurons=20)   # build model
```

```
## Number of parameters (weights and biases) to estimate: 60
## Nguyen-Widrow method
## Scaling factor= 14
## gamma= 8.2201     alpha= 0.0442   beta= 904.974
```

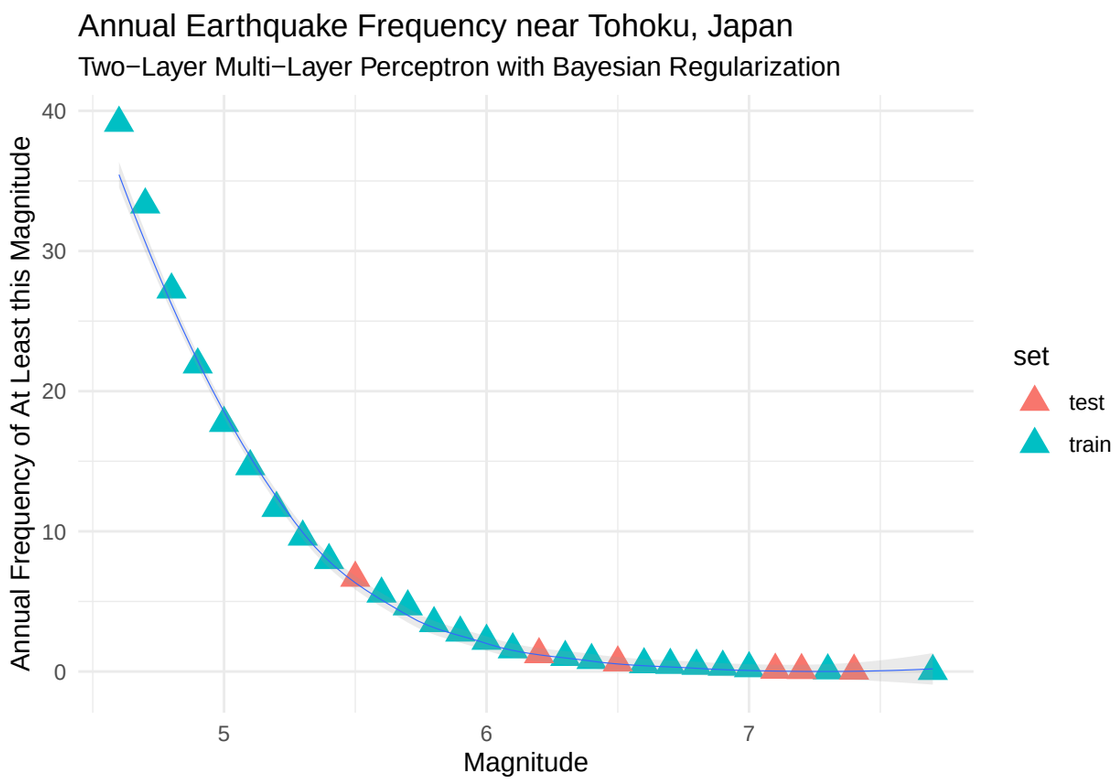
```
summary(brnn)                  # summary of model
```

```
## A Bayesian regularized neural network
## 1 - 20 - 1 with 60 weights, biases and connection strengths
## Inputs and output were  normalized
## Training finished because  SCE <= 0.01
```

```
test$set <- "test"
train$set <- "train"

#---append relevant predictions to the training and test datasets
test$brnnpreds <- 10^predict(brnn, newdata = data.frame(x = test$mag))
train$brnnpreds <- 10^predict(brnn, newdata = data.frame(x = train$mag))
```

```
#--combine data sets for plot
plot_brnn <- train[,c("mag","freqc","set","brnnpreds")] %>%
            rbind(test[,c("mag","freqc","set","brnnpreds")])

#---plot the data
ggplot(plot_brnn, aes(x = mag)) +
  geom_point(size = 4, shape = 17, aes(y = freqc, color = set)) +
  geom_smooth(aes(y = brnnpreds), size = .2, alpha = .2) +
  theme_minimal() +
  labs(x = "Magnitude",
       y = "Annual Frequency of At Least this Magnitude",
       title = "Annual Earthquake Frequency near Tohoku, Japan",
       subtitle = "Two-Layer Multi-Layer Perceptron with Bayesian Regularization")
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```




```
rbind(poissontable_cv,mlptable_cv,brnntable_cv)
```

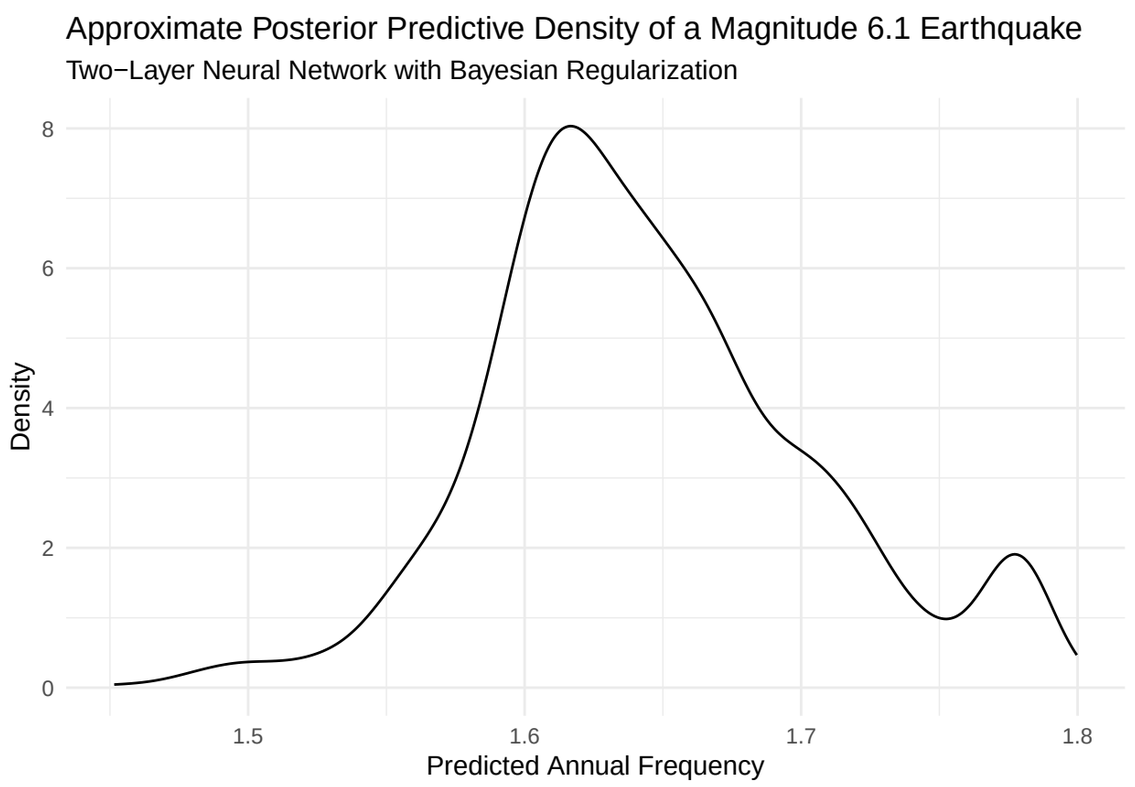
```
##                   Model   TestErr
## 1   Poisson Regression 0.1862851
## 3                   MLP 0.2441865
## 6                   MLP 0.2818492
## 9                   MLP 0.1785652
## 10                  MLP 0.2633863
## 20                  MLP 0.1706793
## 31                BRNN 0.1888387
## 61                BRNN 0.1949408
## 91                BRNN 0.1440102
## 101               BRNN 0.1851772
## 201               BRNN 0.1182969
## 50                BRNN 0.4055160
```

Simulating 1000 networks' predictions for relative magnitudes with *neurons = 20*.

```
# initialize values
x <- train$mag
y <- train$freqc %>% log10()
n <- 1000
prediction_6.1 <- NULL

# run model and generate predcition n times
for(i in 1:n){

  brnn <- brnn(y~x,neurons=20)
```

```
  #---exponentiate predictions to coerce back to standard scale
#  prediction_9.1[i] <- 10^predict(brnn, newdata = data.frame(x = 9.1))
  prediction_6.1[i] <- 10^predict(brnn, newdata = data.frame(x = 6.1))
}

ddf_6.1 <- data.frame(iteration = 1:1000, prediction_6.1)
```

```
#---posterior predictive distribution of a magnitude 6.1
ggplot(ddf_6.1, aes(x = prediction_6.1)) +
geom_density() +
    labs(x = "Predicted Annual Frequency",
         y = "Density",
         title = "Approximate Posterior Predictive Density of a Magnitude 6.1 Earthquake",
         subtitle = "Two-Layer Neural Network with Bayesian Regularization") +
  theme_minimal()
```



Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Ismoilov Nusrat and Sung-Bong Jang. A comparison of regularization techniques in deep neural networks. *Symmetry*, 10(11):648, 2018.

[3] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[4] H. M. Dipu Kabir, Abbas Khosravi, Mohammad Anwar Hosen, and Saeid Nahavandi. Neural network-based uncertainty quantification: A survey of methodologies and applications. *IEEE Access*, 6:36218–36234, 2018.

[5] C.M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, USA, 1995.

[6] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[7] Michael A Nielsen. *Neural networks and deep learning*. Determination press, 2015.

[8] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[9] Martin Schumacher, Reinhard Roßner, and Werner Vach. Neural networks and logistic regression: Part i. *Computational Statistics & Data Analysis*, 21(6):661–682, 1996.

[10] Chaity Banerjee, Tathagata Mukherjee, and Eduardo Pasiliao Jr. An empirical study on generalizations of the relu activation function. In *Proceedings of the 2019 ACM Southeast Conference*, pages 164–167, 2019.

[11] Kamil Nar and Shankar Sastry. Step size matters in deep learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[12] I Rprop. Rprop-description and implementation details.

[13] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[14] B. Rai. *Advanced Deep Learning with R: Become an Expert at Designing, Building, and Improving Advanced Neural Network Models Using R*. Packt Publishing, 2019.

[15] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[17] David JC MacKay. Bayesian interpolation. *Neural computation*, 4(3):415–447, 1992.

[18] Eric Baum and David Haussler. What size net gives valid generalization? *Advances in neural information processing systems*, 1, 1988.

[19] David JC MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[20] Chi Dung Doan and Shie-yui Liong. Generalization for multilayer neural network bayesian regularization or early stopping. In *Proceedings of Asia Pacific association of hydrology and water resources 2nd conference*, pages 5–8, 2004.

[21] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[22] Geir K Nilsen, Antonella Z Munthe-Kaas, Hans J Skaug, and Morten Brun. Epistemic uncertainty quantification in deep learning classification by the delta method. *Neural networks*, 145:164–176, 2022.

[23] JT Gene Hwang and A Adam Ding. Prediction intervals for artificial neural networks. *Journal of the American Statistical Association*, 92(438):748–757, 1997.

[24] D. Salsburg. *The Lady Tasting Tea: How Statistics Revolutionized Science in the Twentieth Century*. Henry Holt and Company, 2002.

[25] Vikram Mullachery, Aniruddh Khera, and Amir Husain. Bayesian neural networks. *arXiv preprint arXiv:1801.07710*, 2018.

[26] R. McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. A Chapman & Hall book. CRC Press/Taylor & Francis Group, 2016.

[27] Michael E Tipping. Bayesian inference: An introduction to principles and practice in machine learning. *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures*, pages 41–62, 2004.

[28] A. Gelman, J.B. Carlin, H.S. Stern, D.B. Dunson, A. Vehtari, and D.B. Rubin. *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, 2013.

[29] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[30] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741, 1984.

[31] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, may 2022.

[32] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

[33] D. Wackerly, W. Mendenhall, and R.L. Scheaffer. *Mathematical Statistics with Applications*. Cengage Learning, 2014.

[34] Creager Elliot Duvenaud David Yu, Jerry and Jesse Bettencourt. Bayesian Neural Networks. 2022.

[35] F Dan Foresee and Martin T Hagan. Gauss-newton approximation to bayesian learning. In *Proceedings of international conference on neural networks (ICNN'97)*, volume 3, pages 1930–1935. IEEE, 1997.

[36] Christopher M Bishop. Bayesian neural networks. *Journal of the Brazilian Computer Society*, 4:61–68, 1997.

[37] Mariia Vladimirova, Jakob Verbeek, Pablo Mesejo, and Julyan Arbel. Understanding priors in bayesian neural networks at the unit level. In *International Conference on Machine Learning*, pages 6458–6467. PMLR, 2019.

[38] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A comprehensive guide to bayesian convolutional neural network with variational inference. *arXiv preprint arXiv:1901.02731*, 2019.

[39] Tianyuan Yu, Yongxin Yang, Da Li, Timothy Hospedales, and Tao Xiang. Simple and effective stochastic neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3252–3260, 2021.

[40] Ethan Goan and Clinton Fookes. Bayesian neural networks: An introduction and survey. *Case Studies in Applied Bayesian Data Science: CIRM Jean-Morlet Chair, Fall 2018*, pages 45–87, 2020.

[41] Liu Ziyin, Hanlin Zhang, Xiangming Meng, Yuting Lu, Eric Xing, and Masahito Ueda. Stochastic neural networks with infinite width are deterministic. *arXiv preprint arXiv:2201.12724*, 2022.

[42] Daniele Silvestro and Tobias Andermann. Prior choice affects ability of bayesian neural networks to identify unknowns. *arXiv preprint arXiv:2005.04987*, 2020.

[43] N. Silver. *The Signal and the Noise: Why So Many Predictions Fail–but Some Don't*. Penguin Publishing Group, 2015.

[44] Stefan Fritsch, Frauke Guenther, and Marvin N. Wright. *neuralnet: Training of Neural Networks*, 2019. R package version 1.44.2.

[45] Nader Fallah, Hong Gu, Kazem Mohammad, Seyyed Ali Seyyedsalehi, Keramat Nourijelyani, and Mohammad Reza Eshraghian. Nonlinear poisson regression using neural networks: A simulation study. *Neural Computing and Applications*, 18:939–943, 2009.

[46] Paulino Perez Rodriguez and Daniel Gianola. *brnn: Bayesian Regularization for Feed-Forward Neural Networks*, 2022. R package version 0.9.2.

[47] Hansapani Rodrigo and Chris Tsokos. Bayesian modelling of nonlinear poisson regression with artificial neural networks. *Journal of Applied Statistics*, 47(5):757–774, 2020.

[48] Manfred Opper and Ole Winther. A bayesian approach to on-line learning. 1999.

[49] Alessandro Chiuso. Regularization and bayesian learning in dynamical systems: Past, present and future. *Annual Reviews in Control*, 41:24–38, 2016.