On the Functionality of the Multilayer Perceptron Classifier Neural Network Architecture

Landon Buell

18 May 2020

1 Introduction

In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts published *A Logical Calculus of the Ideas Immanent in Nervous Activity* [6], which paved the way for the fields of computational neuroscience and early mathematical models of the brain. McCulloch and Pitts proposed that one could loosely mimic the behavior of a brain as a series of mathematical calculations. They realized that the smallest known element of the brain, a *neuron*, acts as a simple on/off switch, much like computers have been designed to. Furthermore, single neurons by themselves do very little, but instead it is the organization of these cells into collections that make up functional networks [1, 4, 6].

The mathematical model could be augmented by trading the simple Boolean (T/F) value, with a continuously ranging number referred to as an *activation*. By producing different types of artificial neurons and establishing a series of synapses, it then became apparent that different organization of these elements could produce networks of solving increasingly advanced problems [2]. To further expand upon this, in 1957, Frank Rosenblatt created a neural network architecture called a Multilayer Perceptron (MLP), which uses layers of interacting neurons to create a single mathematical model [1, 7]. This gives it a simple feed-forward design which has experimentally shown to perform well with tasks related to image and speech recognition [2, 5].

In general, the goal of an classification MLP is to produce some approximation function, F^* that maps an array or vector of inputs, $\vec{x}$ to a set of discrete classes: $0, 1, 2, ..., k-1$, using a set of parameters $\hat{\theta}$ [1, 2, 7]. For a *k-bins* classifier, we notate this as:

$$F^* : \left\{\vec{x}, \hat{\theta}\right\} \rightarrow \left\{0, 1, 2, ..., k-2, k-1\right\} \tag{1}$$

This function is in turn composed of layers of smaller repeating functions, which is what gives rise to the *network* concept of the model. It is the chains of these smaller functions that allow for the producing of the approximate function F^* to be created and solve a vast range of problems.

2 Architectural Considerations

To build a Multilayer Perceptron (MLP), artificial neurons need to be grouped into multiple structures, each called a *layer*. A single layer is modeled mathematically as a column vector, $\vec{x}$ by convention:

$$\vec{x}^{(l)} = \big[x_0, x_1, x_2, ..., x_{n-2}, x_{n-1}\big]^T \tag{2}$$

The superscript (l) is used to indicate the layer number of which this vector represents. Each element inside this vector is a neuron in the MLP model, and the value of that element is the neuron's *activation* [1]. In term of numerical handling, this structure is an array of floating-point numbers. A neural network can have any number of these layers in them, with any number of neurons in each layer. The number of these hidden layers is often referred to as the *network depth* and the number of neurons in each layer is often called the *neuron density* or *network width*.

In addition to these layers, called *hidden layers*, a neural network also requires an *input layer* and an *output layer*[2]. As the name implies, the input layer is where initial arrays of data are passed into the network. This array is often called a *feature vector* as each element is a feature derived from a potentially larger data set. Typically, this feature vector is denoted by $\vec{x}^{(0)}$ is also the $\vec{x}$ object in equation (1).

In a network with L layers, the output array is also a vector, $\vec{x}^{(L-1)}$ which encodes the final decision made by the network model. In a k-bins classifier, there are k neurons, each one corresponding to a different class. The neuron with the highest activation value corresponds to the networks prediction for what class the particular sample, $\vec{x}^{(0)}$ belongs to. If the output, $\vec{x}^{(L-1)}$ is appropriately normalized, then then activations can be interpreted percentage probabilities of which class the sample belongs to:

$$P\big(\vec{x}^{(L-1)}\big) = \frac{1}{||\vec{x}^{(L-1)}||_2}\big[x_0, x_1, x_2, ..., x_{k-2}, x_{k-1}\big]^T \tag{3}$$

In this network of L layers, there are $L-2$ hidden layers whose activations are given by the entries in $\vec{x}^{(1)}$, $\vec{x}^{(2)}$, ... , $\vec{x}^{(L-3)}$, $\vec{x}^{(L-2)}$. And again, the input layers and output layers are given by $\vec{x}^{(0)}$ and $\vec{x}^{(L-1)}$ respectively. Throughout the duration of the model's construction, training and evaluation, these objects remain as column vectors / arrays of real floating point values.

3 Feed-Forward System

Just by having systems of discrete layers, there is no way for the network to behave as a single model. Thus, we need to connect adjacent layers with synapses. These synapses, also called *weights* allow each activation in any given layer to be a linear combination of the activation in the previous layer, with an addition *bais* constant added [1, 5]. Keeping in mind the mathematical equivalent to vectors: we can then say that the i-th element in some layer $(l+1)$ is given by:

$$x_i^{(l+1)} = b_i^{(l)} + \sum_j w_{ij}^{(l)} x_j^{(l)} \tag{4}$$

Where $b_i^{(l)}$ is the bias entry, $w_{ij}^{(l)}$ is the weight coefficient that joins neuron x_j to neuron x_i, and x_j is the activation of the j-th neuron. Notice how the combination of elements in layer (l), given by the upper index, combine to produce the i-th neuron activation in layer $(l+1)$.

Suppose there are m elements in layer (l) and n elements in layer $(l+1)$. We can then express all activations in layer $(l+1)$ by expanding out the sum into a system of linear combinations:

$$\begin{aligned}
x_0^{(l+1)} &= b_0^{(l)} + w_{0,0}^{(l)} x_0^{(l)} + w_{0,1}^{(l)} x_1^{(l)} + \ldots + w_{0,m-1}^{(l)} x_{m-1}^{(l)} \\
x_1^{(l+1)} &= b_1^{(l)} + w_{1,0}^{(l)} x_1^{(l)} + w_{1,1}^{(l)} x_1^{(l)} + \ldots + w_{1,m-1}^{(l)} x_{m-1}^{(l)} \\
\vdots \quad & \qquad \vdots \\
x_{n-1}^{(l+1)} &= b_{n-1}^{(l)} + w_{1,0}^{(l)} x_{n-1}^{(l)} + w_{n-1,1}^{(l)} x_{n-1}^{(l)} + \ldots + w_{n-1,m-1}^{(l)} x_{m-1}^{(l)}
\end{aligned} \tag{5}$$

To make this more convenient to read and express mathematically, the entirety of the system (5) can be expressed as a single matrix vector product. The elements $x_j^{(l)}$ and $x_i^{(l+1)}$ can be concatenated in column vectors, $\vec{x}^{(l)}$ and $\vec{x}^{(l+1)}$ respectively:

$$\begin{aligned}
\vec{x}^{(l)} &= \left[x_0^{(l)}, x_1^{(l)}, x_2^{(l)}, ..., x_{m-2}^{(l)}, x_{m-1}^{(l)}\right]^T \\
\vec{x}^{(l+1)} &= \left[x_0^{(l+1)}, x_1^{(l+1)}, x_2^{(l+1)}, ..., x_{n-2}^{(l+1)}, x_{n-1}^{(l+1)}\right]^T
\end{aligned} \tag{6}$$

Similarly, we can take all elements $w_{ij}^{(l)}$ and organize them into a single matrix object, which we call $\hat{W}^{(l)}$, defined as:

$$\hat{W}^{(l)} = \begin{bmatrix}
w_{0,0}^{(l)} & w_{0,1}^{(l)} & w_{0,2}^{(l)} & \cdots & w_{0,m-2}^{(l)} & w_{0,m-1}^{(l)} \\
w_{1,0}^{(l)} & w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,m-2}^{(l)} & w_{1,m-1}^{(l)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
w_{n-2,0}^{(l)} & w_{n-2,1}^{(l)} & w_{n-2,2}^{(l)} & \cdots & w_{n-2,m-2}^{(l)} & w_{n-2,m-1}^{(l)} \\
w_{n-1,0}^{(l)} & w_{n-1,1}^{(l)} & w_{n-1,2}^{(l)} & \cdots & w_{n-1,m-2}^{(l)} & w_{n-1,m-1}^{(l)}
\end{bmatrix} \tag{7}$$

And finally, we define a bias vector, $\vec{b}^{(l)}$ for layer (l), such that:

$$\vec{b}^{(l)} = \left[b_0^{(l)}, b_1^{(l)}, b_2^{(l)}, ..., b_{n-2}^{(l)}, b_{n-1}^{(l)}\right]^T \tag{8}$$

With these conventions established, we can then express the system (5) as a matrix-vector product:

$$\vec{x}^{(l+1)} = \vec{b}^{(l)} + \hat{W}^{(l)}\vec{x}^{(l)} \tag{9}$$

This linear equation is again, a compact way of expressing of expressing the linear relationship between activations in layer (l) and layer $(l+1)$. However, this relationship only allows for a linear model as all activations are made of linear combinations. This is part of the justification as to why we have chosen to model layers of neuron activations as elements in column vectors in equation (2).

Often, in many-real world problems, models confined to provide linear descriptions or create linear decision boundaries are often insufficient descriptions of solutions [3]. To combat this, a *non-linear activation function*, f, is added to the matrix vector product in equation (9). This changes equation (9) to become [2, 5, 7] :

$$\vec{x}^{(l+1)} = f^{(l)}\Big(\hat{W}^{(l)}\vec{x}^{(l)} + \vec{b}^{(l)}\Big) \tag{10}$$

For conventional reasons, the bias vector has been moved to the other side of the matrix-vector product. This is also often represented as two separate processes:

- The *Pre-activation* linear portion:

$$\vec{z}^{(l)} = \hat{W}^{(l)}\vec{x}^{(l)} + \vec{b}^{(l)} \tag{11}$$

- The *final-activation* portion:

$$\vec{x}^{(l+1)} = f^{(l)}\Big(\vec{z}^{(l)}\Big) \tag{12}$$

Where $f^{(l)}$ is the activation function for the l-th layer. $\vec{x}^{(l+1)}$ has shape $n \times 1$, $\hat{W}^{(l)}$ has shape $n \times m$, $\vec{x}^{(l)}$ has shape $m \times 1$ and $\vec{b}^{(l)}$ has shape $n \times 1$. Note that the inclusion of an activation function by layer, $f^{(l)}$ in eqn. (10) changes the linear combination in eqn. (4) to instead become:

$$x_i^{(l+1)} = f^{(l)}\Big(b_i^{(l)} + \sum_j w_{ij}^{(l)} x_j^{(l)}\Big) \tag{13}$$

As the upper indexes imply, this is how the activations in the *next* sequential layer are produced. By repeating this equation $L-1$ times in an L-layered network, information from the feature vector $\vec{x}^{(0)}$ can be successively transformed into the final decision output of the network $\vec{x}^{(L-1)}$. Each application of equation (10) produces activations in each of the hidden layers. Note that in this L-layer network, there are $L-1$ weight matrices and $L-1$ bias vectors as well. Every single element in all of these arrays can be concatenated to form a larger object, $\hat{\theta}$. It is the combination of the elements in $\hat{\theta}$ that allow the network to map an input vector to an output vector as expressed in equation (1).

For a fully connected multilayer perceptron model with L layers, we can construct an algorithm to execute the forward pass process. The algorithm returns two array-like object containing all final activations (10), and pre-activation function (9) values. The final object in the list of the output of the network $\vec{x}^{(L-1)}$ (also referred to as y^*). For a pseudo-code execution, see alg.(3) [2]. Note that this algorithm uses features from a single training samples, a practical implementation should include batches of samples.

Algorithm 1 Feed-Forward System in a fully-connected Multilayer perceptron model for a single sample.

```
Require: Network depth of $L$ layers
Require: Object of weighting matrices: $\hat{W}^{(i)}$ with $i \in [0, 1, ..., L-3, L-2]$
Require: Object of bias vectors: $\vec{b}^{(i)}$ with $i \in [0, 1, ..., L-3, L-2]$
Require: Object of activation functions: $\vec{f}^{(i)}$ with $i \in [0, 1, ..., L-3, L-2]$
Require: Raw input to network: $\vec{a}$
  final-activations $\leftarrow \{\}$
  pre-activation $\leftarrow \{\}$
  $x^{(0)} \leftarrow \vec{a}$
  for $l = 0, 1, 2, ..., L-2, L-1$ do
    $\vec{z}^{(l)} \leftarrow \vec{b}^{(l)} + \hat{W}^{(l)}\vec{a}^{(l)}$
    $\vec{x}^{(l+1)} \leftarrow f^{(l)}\big(\vec{z}^{(l)}\big)$
    pre-activations.insert($\vec{z}^{(l)}$)
    final-activations.insert($\vec{x}^{(l+1)}$)
  end for
  return pre-activations: $\big\{\vec{z}^{(0)}, \vec{z}^{(1)}, ..., \vec{z}^{(L-2)}\big\}$
  return final-activations: $\big\{\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(L-1)}\big\}$
```

Both arrays that are returned by the alg. (3) contain $L-1$ elements. The j-th element in the *final-activations* array is the element produced by applying an activation function to the j-th element of the *pre-activations* array. These correspond to layers $l+1$ and l respectively.

4 Back Propagation System

The logic of using and input $\vec{x}^{(0)}$ and set of parameters $\hat{\theta}$ to produce an output into a set of discrete bins 0 thorough $k-1$ becomes worrisome when one considers the amount of parameters that make up the object $\hat{\theta}$, and furthermore how to produce these parameters such that the resulting approximation F^* shows experimental validity. To remedy this, the model needs some way of using inputs, with known outputs, to find the elements of $\hat{\theta}$ that allow similar, but non-identical inputs to be mapped to correct outputs.

We construct the object $\hat{\theta}$ to be some multidimensional array-like object that contains all of the parameters in the model. very generally, it can be thought of as a concatenation of all weighting and bias elements for all of the layers.

$$\hat{\theta} \equiv \left[\hat{W}^{(0)}, \vec{b}^{(0)}, \hat{W}^{(1)}, \vec{b}^{(1)}, ..., \hat{W}^{(L-2)}, \vec{b}^{(L-2)}\right] \tag{14}$$

Where $\hat{W}^{(l)}$ and $\vec{b}^{(l)}$ are given in eqns. (7) and (8) respectively. For smaller networks, $\hat{\theta}$ may contain several hundred or thousand elements, and for larger deep-learning networks, may contain tens to hundreds of thousands of elements [2, 5]. It is the combinations of these elements that allow for some input given $\vec{x}^{(0)}$ to be transformed into an output $\vec{x}^{(L-1)}$.

To do this, we treat the behavior of a neural network as a multidimensional optimization problem. The goal of this would then be to find the set of parameters $\hat{\theta}$ that produces the smallest possible output of some *error function*, [2, 3, 7]. This may also be called a *cost function* or *loss function* as well. This function , J, takes the output of the network $\vec{x}^{(L-1)}$ (which we will refer to as y^*) to represent a predicted or approximated output) on a given training sample, and compares it to the expected output for that sample $\vec{y}$, and maps it to a real number, which then measures the *cost* of that particular sample such that:

$$J : \{y^*, y\} \rightarrow \{\mathbb{R}\} \tag{15}$$

The larger the value outputted from this function J, the further the model's prediction differs from the expected value. Thus, this function can be through of as a measurement of how *poor* the network's prediction was. It then becomes appropriate to make the goal of the neural network to *optimize* J, with respect the parameters $\hat{\theta}$ to allow for a generally low cost value for a given input.

There are many ways to optimize the parameters in a neural network,for this work, we will examine an optimizer algorithm called *Stochastic Gradient Descent* (SGD). This is an iterative method that uses repeated a passes over a data set to slowly push the cost function to a smaller and smaller value [1, 2, 5]. As the name implies, we compute the gradient of the cost function with respect to all of the parameters in the model to produce a vector. The components of this vector give the direction to which a small change the elements of $\hat{\theta}$ will result in a small reduction of the value of the loss function. Recall the gradient vector of some function h of variables x_0 , x_1 , ... ,x_{n-2} , x_{n-1} is given by:

$$\nabla_{\vec{x}}\Big[h(x_0, x_1, ..., x_{n-2}, x_{n-1})\Big] = \Big[\frac{\partial h}{\partial x_0}, \frac{\partial h}{\partial x_1}, \cdots \frac{\partial h}{\partial x_{n-2}}, \frac{\partial h}{\partial x_{n-1}}\Big]^T \tag{16}$$

To build this vector, we need to use the chain rule of multivariate calculus to work *backwards* through the layers of the network to differentiate by layers, hence why the procedure is called *backwards* propagation. To ease mathematical notation, we will redefine the forward pass equation (10) into two separate procedures:

$$\vec{z}^{(l)} = \hat{W}^{(l)}\vec{x}^{(l)} + \vec{b}^{(l)} \tag{17}$$

$$\vec{x}^{(l+1)} = f^{(l)}\big(\vec{z}^{(l)}\big) \tag{18}$$

We start by differentiating the cost function J with respect to last layer of weights and biases, $\hat{W}^{(L-2})$ and $\vec{b}^{(L-2)}$. The cost, J, is a function of the output activations, $\vec{x}^{(L-1)}$ which is in turn a function of $\vec{z}^{L-2}$ in eqn. (18) which is in turn a function of both $\hat{W}^{(L-2)}$ and $\vec{b}^{(L-2)}$ in eqn. (17). To account for this layered composition, we multiply each partial derivatives in succession to produce the gradient components that correspond to the needed change in those parameters.

$$\nabla_{W^{(L-2)}}\Big[J\Big] = \frac{\partial J}{\partial x^{(L-1)}}\frac{\partial f^{(L-2)}}{\partial z^{(L-2)}}\frac{\partial z^{(L-2)}}{\partial W^{(L-2)}} \tag{19}$$

$$\nabla_{b^{(L-2)}}\Big[J\Big] = \frac{\partial J}{\partial x^{(L-1)}}\frac{\partial f^{(L-2)}}{\partial z^{(L-2)}}\frac{\partial z^{(L-2)}}{\partial b^{(L-2)}} \tag{20}$$

Additionally, we compute the gradient with respect to the $L-2$ layer of activations, which will allows for us to move down the to previous sequential layer $L-3$.

$$\nabla_{x^{(L-2)}}\Big[J\Big] = \frac{\partial J}{\partial x^{(L-1)}}\frac{\partial f^{(L-2)}}{\partial z^{(L-2)}}\frac{\partial z^{(L-2)}}{\partial x^{(L-2)}} \tag{21}$$

These three equations allow for the model to work backwards through the layers and compute the gradient of the cost function with respect to every paramater in the network. The above equations can be generalized for any layer l:

$$\nabla_{W^{(l)}}\big[J\big] = \frac{\partial J}{\partial x^{(l+1)}}\frac{\partial f^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial W^{(l)}} \tag{22}$$

$$\nabla_{b^{(l)}}\big[J\big] = \frac{\partial J}{\partial x^{(l+1)}}\frac{\partial f^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial b^{(l)}} \tag{23}$$

$$\nabla_{x^{(l)}}\big[J\big] = \frac{\partial J}{\partial x^{(l+1)}}\frac{\partial f^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial x^{(l)}} \tag{24}$$

It is important to know that due to numerical condiderations, this operation is very computationally expensive, and fine details for improving efficiency not considered in the equations above. Additionally, the results from eqns. (22), (23), and (24) are all array-like objects. Each element is the direction and value of the small change needed to push J to a successively lower value. after each element of the gradient vector is computed, we simply add the change given by the gradient to the current value of that entry. For all parameters in any layer l:

$$\begin{aligned}\hat{W}^{(l)} &= \hat{W}^{(l)} + \alpha \nabla_{W^{(1)}} \big[J \big] \\ \vec{b}^{(l)} &= \vec{b}^{(l)} + \alpha \nabla_{b^{(1)}} \big[J \big]\end{aligned} \tag{25}$$

Where α is a scalar called the *learning rate*, typically $\alpha \in (0, 1)$ [1, 7]. It can be intuitively through of as a step size taken by the model at each back-propagation iteration. Larger values of α may cause the model to reach a minimum faster, but also cause a possible *overshoot* or missing the target minimum, while smaller values of α may cause the model to reach a more suitable minimum, at the cost of must greater computational time and resources. Often, α can be set to have a dynamic value, allowing for the step size at each iteration for a compromise between computation time and selection of a suitable value. The act details on the choice of the learning rate is topic of a different study on *model regularization* techniques [2, 3].

We can express the back-propagation process with an algorithm that executes the Stochastic Gradient Descent algorithm outlined about. Note that this algorithm uses the loss from a single training sample, a practical implementation should include batches of samples. This algorithm can only be executable after a forward pass prediction is made.

Algorithm 2 Back Propagation System in a fully-connected Multilayer perceptron model for a single sample.

```
Require: Network depth of $L$ layers , cost function $J$ , learning rate $\alpha$
Require: Objects: $\hat{W}^{(i)}$, $\vec{b}^{(i)}$ from previous feed-forward.
Require: Object of activation function derivatices: $\partial\vec{f}^{(i)}$
Require: Object of final activations by layer: $\vec{x}^{(i)}$ with $i \in [1, 1, ..., L-2, L-1]$
Require: Object of pre-actications by layer: $\vec{z}^{(i)}$ with $i \in [0, 1, ..., L-3, L-2]$
Require: Sample target output $y$ (usually one-hot-encoded)
  Array to hold weight updates: $dW \leftarrow \{\}$ (22)
  Array to hold bias updates: $db \leftarrow \{\}$ (23)
  Compute gradient of $J$ w.r.t output layer $\vec{x}^{(0)}$:
  $dx \leftarrow \nabla_{y^*}\big[J(y^*, y)\big]$
  for $l = L-2, L-3, ..., 1, 0$ do
    $db^{(l)} \leftarrow dx \odot \partial f^{(l)}\big(z^{(l)}\big)$
    $dW^{(l)} \leftarrow db^{(l)} \cdot x^{(l)}$
    dW.insert($dW^{(l)}$)
    db.insert($db^{(l)}$)
    $dx \leftarrow dW^{(l)} db^(l)$
  end for
  ASSERT: $dW^{(i)}$ has same dimension as $W^{(i)}$ for $i \in [0, 1, ..., L-3, L-2]$
  ASSERT: $db^{(i)}$ has same dimension as $b^{(i)}$ for $i \in [0, 1, ..., L-3, L-2]$
  for $l = 0, 1, ..., L-3, L-2$ do
    $W^{(i)} \leftarrow W^{(i)} + (\alpha) dW^{(i)}$
    $b^{(i)} \leftarrow b^{(i)} + (\alpha) db^{(i)}$
  end for
  return Object containing updated weights: $\hat{W}^{(i)}$ with $i \in [1, 1, ..., L-2, L-1]$
  return Object containing updated biases: $\hat{b}^{(i)}$ with $i \in [1, 1, ..., L-2, L-1]$
```

5 Summary

References

[1] Géron Aurélien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.

[2] Goodfellow, Ian, et al.*Deep Learning*. MIT Press, 2017.

[3] James, Gareth, et al. *An Introduction to Statistical Learning with Applications in R*. Springer, 2017.

[4] Levine, Daniel S. *Introduction to Neural and Cognitive Modeling*. 3rd ed., Routledge, 2019.

[5] Loy, James , *Neural Network Projects with Python*. Packt Publishing, 2019

[6] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[7] Petrik, Marek. "Introduction to Deep Learning." Machine Learning. 20 April. 2020, Durham, New Hampshire.