INTRODUCTION TO NEURAL NETWORKS

ARDEN RASMUSSEN

Abstract. Machine Learning is a rapidly expanding subject of research. At the center of this research are neural networks. Neural networks provide researchers with tools and methods for the construction of artificial intelligence. These AIs can be trained to ground breaking levels of thinking, frequently out reasoning leading humans in the subject. This paper provides a short introduction to the concept of neural networks, with emphasis on the mathematics of the neural networks.

1. Introduction

A neural network is a computational model, that can be used for advanced decision making. The original influence for neural networks, comes from biology. This paper will attempt to provide an introductory comprehension of neural networks as a whole. This includes the three primary types of network, their structures, how they are trained, and how they may be implemented.

The three primary types of neural network that we will discuss are *Feed Forward*, *Convolutional*, and *Recurrent*. These three types of networks are frequently used in modern research. Most networks can be considered as some combination of these types of networks. Thus having an understanding of these three core networks types will greatly improve ones understanding of the current research of neural networks.

2. Structure

The first thing to understand with respect to neural networks is what all the terminology means. To do this we explain explicit what the building blocks of a neural network are. We explain how these blocks are assembled in the later sections. Generally each neural network is constructed from *layers*, and each layer is constructed from *neurons* and *activation functions*. This structure is fairly universal among neural networks. The cases when a network differs from this generalized structure is outside of the scope of this introduction, and requires independent research.

2.1. **Neuron.** The most basic component of a neural network is the *neuron*. A single neuron takes input from many sources and produces a single output value. A single neuron takes inputs from other neurons or from external sources. For each input value the neuron assigns a weight associated with how important that value is relative to the other inputs. Then the neuron applies a function f, to the weighted sum of the input values.

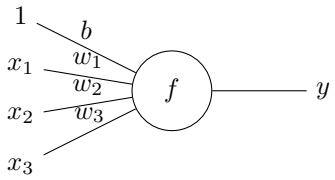

Figure 1. A single neuron acting on inputs with provided weights w_i.

In figure 1, we see that the neuron takes in four inputs, $x_1, \ldots, x_3, 1$. Then for each input there is an associated "importance" to that input with respect to the other inputs $w_1, \ldots, w_3, b$. This value is the weight of that input. The neuron takes the inputs and the weights of the inputs and applies the function f on the weighted sum of the inputs

$$f(w_1x_1 + w_2x_2 + w_3x_3 + b).$$

We can then generalize this formula for any number of input neurons to be

$$f\left(\sum_{i=1}^{N} w_ix_i + b\right). \tag{1}$$

The output of this neuron is either passed along to other neurons as inputs, or is the output of our network, and is given to the user.

The function f that the neuron applies to the weighted sum is called the *activation function*. The activation function is a non-linear function used to introduce non-linearity into the neural network. Without the non-linearity then the network would be just a bunch of linear operations, which could be represented as a single matrix, by the introduction of the

Date: April 17, 2019.

non-linearity we enforce that there must be many steps. This will increase the ability of the network to do more complex operations in the future. We will go into further detail on activation functions in a different section.

There are three main *types* of neurons that we will consider.

Input Neurons: Input neurons take information from external inputs, such as image data, and then passes that data on to the other neurons in the network. The input neurons do no computation, they just pass on their information to later neurons.

Hidden Neurons: Hidden neurons have no connection directly to input or output data. They only have what is provided to them from their previous layer, and preform the computation that was explained in section 2.1. Then the output value is passed on, either to more hidden neurons or to the output neurons.

Output Neurons: The output neurons are responsible to taking the values passed to them by the previous layer of neurons and processing the values into an understandable interpretation, such as a probability. They then return that data to the user.

We will uses these three types of neurons to construct the neural network.

2.2. **Activation Function.** Activation functions are a method to cause some non-linearity. This importance comes into play when we have multiple neurons connecting into one another. Since each neuron individually is linear, then this connection of neurons will also be linear. That would mean that it could all be modeled as a single neuron. This is the reason as to why we need to impose the activation functions, to cause non-linearity in between the different neurons, so that the function as a whole is not linear. This means that each neuron can be trained to be important in the network.

There are a number of common functions that are used for the activation functions. *Sigmoid*, *tanh*, *ReLU*, and other variations on *ReLU*. We will describe each of these possible activation functions in more detail below. Through usage, it has been found that a good default is to use *ReLU*.

Sigmoid: The sigmoid activation function is defined as

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}.$$

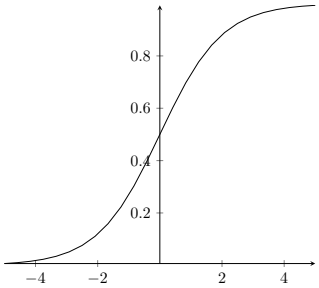

FIGURE 2. $\frac{1}{1+e^{-x}}$

The sigmoid function takes any value input to it and squashes it into the range of $(0, 1)$. However, there are some issues that arise with the sigmoid activation function. If the input is small or large, then the output becomes saturated, and the gradient of the function at those points is zero. This becomes an issue later when using back propagation. The other issue is that the output is not centered at zero, this will again cause issues later with back propagation.

tanh: The tanh activation function is similar to the sigmoid function, but is constructed to be zero centered. It is defined as

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$

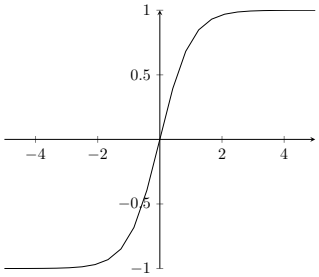

FIGURE 3. $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

The tanh function has the same issue as with the sigmoid function in that it can become saturated at large or small values, thus taking the gradient to zero. However, this does solve the issue with the zero centering. The output of this function will be zero centered.

ReLU: The ReLU is an acronym for Rectified linear unit. This is commonly the default activation function. It is defined as

$$f(x) = \max(0, x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}.$$

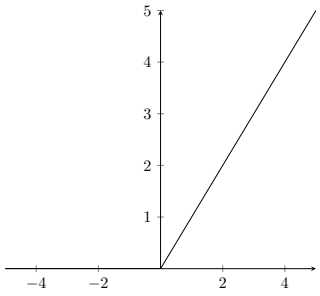

FIGURE 4. $\max(0, x)$

The ReLU function runs into many of the same issues as the other activation functions. First it is possible to the neuron to "die" if the input is less than zero, then the derivative will be zero, and that neuron is effectively dead. This can be very bad in training. It is also not zero centered, just like the sigmoid function. However both of these issues are usually left to the neural network to deal with, then the speed increase that is provided by the simplicity of the function makes this significantly better than the other possibilities. This is so much faster because there are no computationally expensive computations in the function, so it allows the network to execute much faster. It has also been shown that using ReLU allows networks to converge faster than other activation functions.

2.3. **Layer.** Neural networks are organized into layers. Each layer contains a single type of neuron, and can only interact with the layer before it and the one after it.

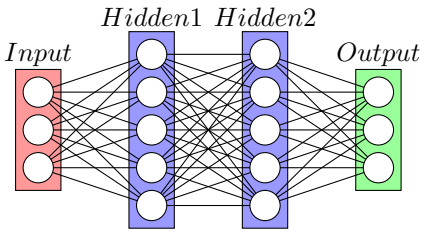

FIGURE 5. Example structure of a two layer neural network, with the four layers highlighted in different colors.

In figure 5 an example of how each layer of neurons interact with one another can be seen. Note how each neuron of the previous layer is now an input for every neuron of the current layer. We call this type of network a "two layer" network even though it clearly has four layers. This is because the input and output layers must always be present, and so it is not important if we count them or not, thus the important number is the number of hidden layers in the network.

There are many different types of layers that can be utilized together in a neural network. This introduction will provide a brief introduction to many of the commonly implemented types of layers, but will only go into specific detail and implementation of a select few of the layer types.

Input: The input layer reads the data from the user and then passes that on to the next layer in the network. This layer does no computation on its own, it is only used as a house keeping tool to keep everything concise and organized.

Output: The output layer takes the values that are determined by the previous layers of the network, and provides them to the user.

Dense: A dense or fully connected layer is what we will mainly be focusing on. A dense layer means that each neuron in the layer is connected to every neuron in the previous layer and every neuron in the next layer. For this introduction to neural networks, this is where the core of the computation is preformed. The computation of this layer is

$$f((W \cdot V) + b)$$

where f is the activation function for the layer, W is the matrix of weights for the dense layer, sometimes called the kernel, V is the vector of input values passed to the layer by the previous layer, and b is the vector of bias values that are applied to the product of the weights and input.

Activation: This layer is a tool that can be used to apply special activation functions, beyond what is normally applied by the dense layer. This means that this layer simply computes

$$f(V)$$

where f is some activation function, and V is the vector of input values.

Dropout: Dropout is a more advanced tool that is used to prevent over fitting during the training stages. It randomly sets some percentages of the input values to 0 before passing them on.

Reshape: This is a utility layer that is used for when the input data may not be a strict vector of values. For example if the input is an image with a red, green, and blue channel, then the input would be a three dimensional matrix of values. The reshape layer allows the user

to reshape the data into whatever format is necessary for the later layers of the network. So to pass an image into a dense layer, one would reshape the three dimensional matrix into a one dimensional matrix or vector.

Convolutional: The convolutional layer is a key part of most image recognition neural networks. The convolutional layer applies a matrix of weights over a small region of the input values, then by some means of moving the region that it will apply the matrix, the process is repeated to cover the entirety of the input values. There are many nuances of this type of layer that must be considered for proper implementation.

Pooling: The pooling layer is intended to quickly reduce the scale of the data that is begin passed through the network, by some method. The pooling layer takes the input values and outputs a significantly reduced size of output values. There are two ways that this commonly done. Either by taking the *max* of several values in the input layer, or by taking the *avg* of those values. Either way, this layer takes many values and condenses them down into a single value, either though maximum, or averaging.

3. Deep Feed Forward Network

The Deep Feed Forward (DFF) network, is one of the simplest networks, and so is a great choice for an introduction to how to construct a neural network. A typical representation of a DFF is depicted in figure 6. The network in the depiction has one input layer, two dense layers, and one output layer. We will used this specific network for most of the explanation, but it can easily be generalized to larger networks.

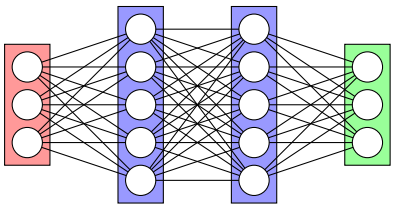

Figure 6. Deep Feed Forward Neural Network.

One key aspect of a DFF network, is that it only consists of a single input and output layer, with some number of dense layers in between the two.

3.1. Forward Propagation.

Forward propagation is the process of taking values for the input layer, and using the matrix to determine the values of the output layer. This process does not train the network in any way, it is just using the network to determine outputs for a given input. When a network is actually placed into production, this is the process that is being used.

As demonstrated in section 2.1, each neuron computes

$$f(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b).$$

And we note that here f is any activation function.

Then since each layer has many neurons, it is helpful to consider the layer as a whole instead of a single neuron at a time. For this example we will consider the first dense layer in the network. This means that each neuron is taking in 3 inputs from the three input neurons. We denote the neuron in the layer using i, using this notation, each neuron computes

$$f(w_{i1}x_1 + w_{i2}x_2 + w_{i3}x_3 + b_i) \quad i = 1, \ldots, 5.$$

Since this operation is begin done for every i, it can be easily seen that it matches a matrix expression of

$$f\left(\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}\right).$$

Where f of a vector is the same as taking f of each element individually. This notation is usually written in the form

$$f\left(W \cdot X + B\right). \tag{2}$$

Since this is the function for the first dense layer we will denote that by the use of an index. Thus the expression for the computation done by the first layer is

$$f^{(1)}\left(W^{(1)} \cdot X + B^{(1)}\right).$$

This is the final output of the values of the neurons from the first layer.

Since the second dense layer uses the first dense layer as the inputs, then the equation 2 for the first layer becomes the new X vector. For a small network such as this one, it is possible to write the complete expression for the second dense layer. We find this to be

$$f^{(2)}\left(W^{(2)} \cdot \left[f^{(1)}\left(W^{(1)} \cdot X + B^{(1)}\right)\right] + B^{(2)}\right).$$

This expression can be chained for as many hidden layers as the deep feed forward network has. That is until one reaches the output layer. In most cases the output layer also acts as a dense layer in order to reshape the data into the desired output shape.

For our example problem, we will treat the output layer exactly as another dense layer. Thus the final expression becomes

$$f^{(o)}\left(W^{(o)} \cdot f^{(2)}\left(W^{(2)} \cdot f^{(1)}\left(W^{(1)} \cdot X + B^{(1)}\right) + B^{(2)}\right) + B^{(o)}\right).$$

Looking into the details of the W and the B matrices, it can clearly be seen that the dimensions of $W^{(1)}$ is 5×3 matrix, $B^{(1)}$ is 5×1, $W^{(2)}$ is 5×5, $B^{(2)}$ is 5×1, $W^{(o)}$ is 3×5 and $B^{(o)}$ is 3×1. These dimensions were just determined by what they must be in order for the matrix multiplication to even make any sense.

An alternative way to structure a dense layer, is to add a value of 1 to the input vector, and to place the bias vector into the weights matrix. This can be written as

$$f\left(\begin{pmatrix} w_{11} & w_{12} & w_{13} & b_1 \\ w_{21} & w_{22} & w_{23} & b_2 \\ w_{31} & w_{32} & w_{33} & b_3 \\ w_{41} & w_{42} & w_{43} & b_3 \\ w_{51} & w_{52} & w_{53} & b_5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix}\right).$$

This provides the computational benefit of only needing to do a single operation of matrix multiplication, versus a matrix multiplication and vector addition operation. Because of this equivalent representation of the formula, the bias vector is often implicitly included into the weights matrix. Thus the formula would be written as

$$f(W \cdot X), \tag{3}$$

and the bias terms should be inferred to be present in the weights matrix.

3.2. **Initialization.** As it should be understood now, each dense layer must have a matrix of $N \times (M + 1)$ where N is the number of neurons in the layer, and M is the number of neurons in the previous layer. Before it is possible to train a network, we need an initial condition. This means that we must determine some method for constructing the initial values of the weight matrices.

For this process, we consider three methods

Zeros: It is reasonable thinking that half of the weights of a network will be positive, and about half will be negative. Thus it would make sense to initialize the weights as zero, which would be about the average weight. The pitfall of this method, is that each neuron would be trained exactly the same. This would means that each neuron and each weight for each neuron would be altered in the exact same way, and wouldn't allow for any variations between one another.

Small Random: A method that can be employed to avoid the issues that arise with zero initialized weights, is to initialize using small random values. This way the weights are still near zero, but each weight will be distinct, and learn different things, leading to a more powerful network.