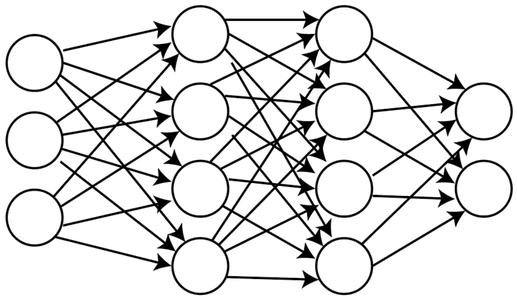
Project T Team Ferda: MultiLayer Perceptrons

Alec Zhou, Conor O'Shea, Jordan Byck

Figure 1:



1 What is a Multilayer Perceptron?

A Multilayer Perceptron is a form of model used for machine learning tasks. When we show you the structure, they may seem overwhelming, but once you understand how the various parts come together, you'll find it stunningly simple, and maybe even a little beautiful. Multilayer Perceptrons were one of the first machine learning models created, and now have many different forms used for various tasks. That will be covered in further lectures, but for now lets just figure out how the simplest form works... The underlying intuition of an MLP is that it is an *artificial neural network*. This just means it's a digital attempt at mimicking the structure and usage of the neurons in animal brains. It comes with no surprise then that MLP and its derivatives are exceptional at tasks such as image and speech recognition, as well as machine translation. MLPs are also useful since they can easily learn non-linear models.

2 History of Multilayer Percpetrons (MLPs)

The Multilayer Perceptron algorithm was developed at the Cornell Aeronautical Laboratory by **Frank Rosenblatt**, initially designed to be a physical machine. The first manifestation was created for image recognition, where the neurons are randomly connected to 400 photocells. *Weights* were stored in potentiometers and updated using electric motors. We will explain what these *weights* are later.

3 MLP Inputs and Outputs

A multilayer perceptron is a *feedfoward artificial network*, meaning that there are no cyclical connections between neurons. Before we dive into the various components of a MLP, let's first look at it as a black box and examine which outputs and inputs work best with them.

3.1 Inputs

The input to an MLP should be a dataset that is numerical. If the data is categorical (eg. dog vs. cat), it can be **one hot encoded**. Ideally, the dataset should also have underlying patterns that we are trying to train our MLP to pick up on.

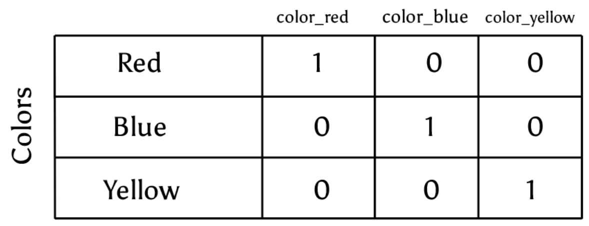
3.1.1 One Hot Encoding

One hot encoding is the task of turning categorical data into numerical data, which can subsequently be used by in machine learning models. This is because numerical functions and matrix operations used by an MLP cannot be performed on categorical data.

In the process of one hot encoding, the various categories for a variable are spread out into binary variables which correspond to that class appearing in that data. As a toy example (visualized in *Figure 2*), let us one hot encode 4 cars that have some of the colors red, blue, or yellow. We can see that each *car* data point corresponds to three binary variables that indicate whether any of *Red, Blue, or Yellow* are present. When the categorical variable contains the value that matches the binary variable, the binary variable is 1.

	Red	**Blue**	**Yellow**
My car	1	0	0
Sister's car	0	0	1
Brother's car	0	1	0
Clown car	1	1	1

Figure 2: Example of one hot encoding. Since *Red* is present in the first row, *color_red* = 1. *Color_blue* and *color_yellow* = 0 since they do not correspond with this value.



		color_red	color_blue	color_yellow
Colors	Red	1	0	0
	Blue	0	1	0
	Yellow	0	0	1

3.2 Outputs

An MLP ultimately outputs the values or vector of values required by the problem. MLPs are often used in:

- **Regression Problem** – desire a single or multiple values closest to the desired output
- **Binary Classification Problem** – desire a single or multiple values split among two classes
- **Multi-Class Classification** – desire a single or multiple values split among multiple classes

4 The Math Behind Perceptrons

A **perceptron** is a single node in the structure of a Multilayer Perceptron. Within the functioning of a perceptron there are a few key players which are essential both understand alone, and how they ultimately come together.

- **Weight** (w) – connection between two neurons which holds a value
- **Bias** (b) – also a weight, that helps account for factors not covered by the weights
- **Activation** $\phi(z)$ – a small transformation applied to the incoming summed values z (we'll cover how this summation works more later).

Ok, now let's see them in action.
Below we have the function for a single neuron.

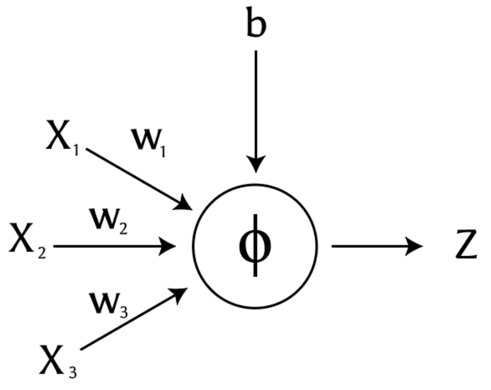
$$\phi(b + \sum_{i=1}^{n} x_i \cdot w_i) = Z \tag{1}$$

For each input, the dot product between it and its weight is calculated. These dot products are summed up to create a *weighted sum*. The beauty in a *weighted sum* is that each of the incoming inputs are effectively scaled by its weight. **MLPs 'learn' by iteratively adjusting weights and seeing how these adjustments match out desired output down the line.**

In case our weighted sum is not enough, the *bias term* b is then an extra positive or negative nudge we can give to our weighted sum (before passing it into the activation function $\phi(z)$). **MLPs also learn by iteratively adjusting and testing bias terms.**

Figure 3 illustrates the visual intuition behind this formula acting like a neuron. The dot product of each input x_i and its corresponding weight w_i are computed, summed and transformed by an activation function $\phi(x)$ to produce an output Z. Let's examine two activation functions, which will be covered in more detail in later lectures.

Figure 3: The structure of a single perceptron. The perceptron below has three inputs X_1, X_2, and X_3, however, there could be an n number of inputs and their corresponding weights.



5 Activation Functions

Activation Functions are applied to each perceptron to transform the weighted sum and bias term into values that are easier to train subsequent weights on. The simplest activation function is the linear activation function, which applies no transformation. Networks made from solely linear activation functions are easy to computationally train, but cannot be trained to properly map complex functions.

5.1 ReLu Function

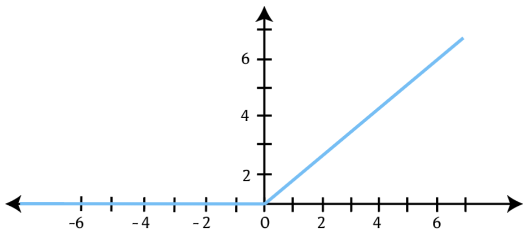
ReLu stands for Rectified Linear Unit. Networks that use ReLus for hidden layers are referred to as rectified networks. If the function value is less than 0, it will output 0, otherwise if the function value is greater than 0, it will simply pass the function input. The ReLu activation function can be defined as a piecewise linear function:

$$ReLu(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

While, in code, it is much simpler. This makes it easier to compute on many nodes too.

$$ReLu(x) = max(0, x) \tag{2}$$

Figure 4: The ReLu function graphed.

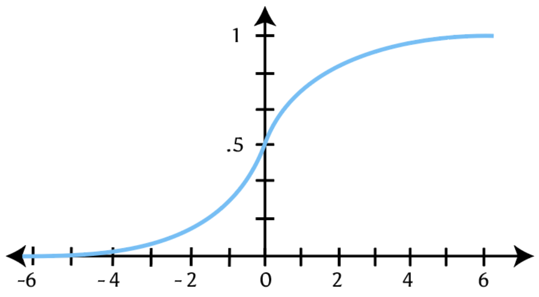


5.2 Sigmoid Functions

Sigmoid Functions are function which have a sigmoid or S shape. Large input values of a sigmoid function become larger, map to 1, while small input values map to 0. The sigmoid function essentially squeezes the inputs $0 \leq x \leq 1$, placing less emphasis on large negative or positive outlier values. This specific sigmoid function (the logistic function) is mathematically defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3}$$

Figure 5: The sigmoid or logistic function graphed



6 Layers

A layer is a group of perceptrons which are bundled together to perform computation on the previous inputs. Since each perceptron is connected to each in the previous and following layers, a Multilayer Perceptron is then **fully connected**.

6.1 Input Layers

An input layer contains all of the inputs being passed into the Multilayer Percreptron, which **are required to be in numerical form** – if the data is catagorical, we can transfrom it into numerical values as shown before through *one hot encoding.*

6.2 Hidden Layers

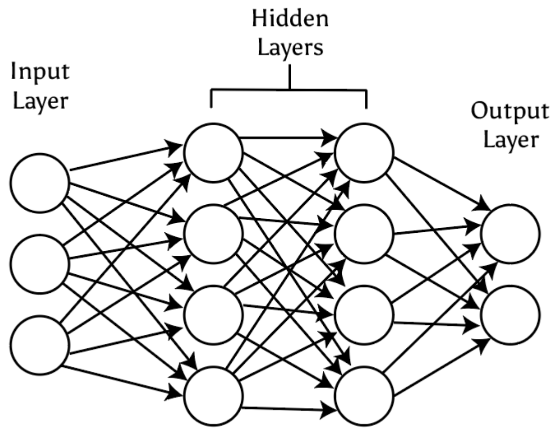
Hidden layers are the part of the MLP which we modify. We can alter the number of hidden layers d or the size of each layer n to have our MLP best learn the function to map the inputs onto the outputs.

6.3 Output Layers

The output layer's form is dictated by the nature of the output we desire. Let's go back over the problems we listed before and see how their output layers would look.

- **Regression Problem**
  - **Single Neuron** – Desire to regress on a single dimension; the estimated miles per gallon of a vehicle.
  - **Multiple Neurons** – Desire to regress on multiple dimensions; the estimated spatial coordinates of a satellite.
- **Binary Classification Problem**
  - **Single Neuron** – Desire to classify a single variable to one of two classes; is this movie going to win an academy award?
- **Multi-Class Classification**
  - **Multiple Neurons** – Desire to find the probabilities that an output is in different classes; what is the probability the person in this picture is smiling, crying, laughing?

Figure 6:



7 Batch Size and Learning

In later lectures, we will cover the this topic in more detail, particularly the concept of *backpropogation*, but for now, we'll dip our toes into how models learn the correct weights and biases. Primarily by implementing what we learned in week 2 for linear models – *batch size selection* and *gradient descent*, but now with their more sophsticated cousin, the MultiLayer Perceptron.

MLPs are also trained using gradient descent, which means that choosing the correct batch size to train our model on is key.

If the batch size is 64, this means that 64 samples will be used to estimate the error gradient before updating the model weights.

A training epoch is a single pass through the training dataset.

- **Batch Gradient Descent** – the batch size is equal to the entire dataset.
- **Mini-Batch Gradient Descent** – The batch size is more than 1, but not the entire dataset.
- **Stochastic Gradient Descent** – the batch size is set to 1.

It's important to note that even though stocastic gradient descent is often used to describe the algorithm, regardless of the batch size.