Lab 21

Today, we continue our journey into deep learning, by networking our perceptons. Today'a goals are:

1. Define network in the context of deep learning
2. Code an activation function
3. Detail the differences between the activation and threshold function

Today we are going to be following aspects of this blog post.

Imports for today

In [ ]:

```
%matplotlib inline

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

import numpy as np
from numpy import linalg as LA
import pandas as pd

import random
```

In [ ]:

```
# Import the lab 14 data to be an example

new_data = pd.read_csv("lab14data.csv", sep = ",")
new_data_np = np.genfromtxt("lab14data.csv", delimiter=',', skip_header=1)
```

Networks

Along with our artificial neurons, *networks* are the other central ingredient to our neural networks. Before we discuss what a neural network is, we begin with a few basic network terms:

- Nodes - These are objects that we want to form connections between
- Edges - These are the connections between the nodes
- Directional Edges - Edges can have direction, or a "to" side and "from" side. These are often used to represent flow of information.

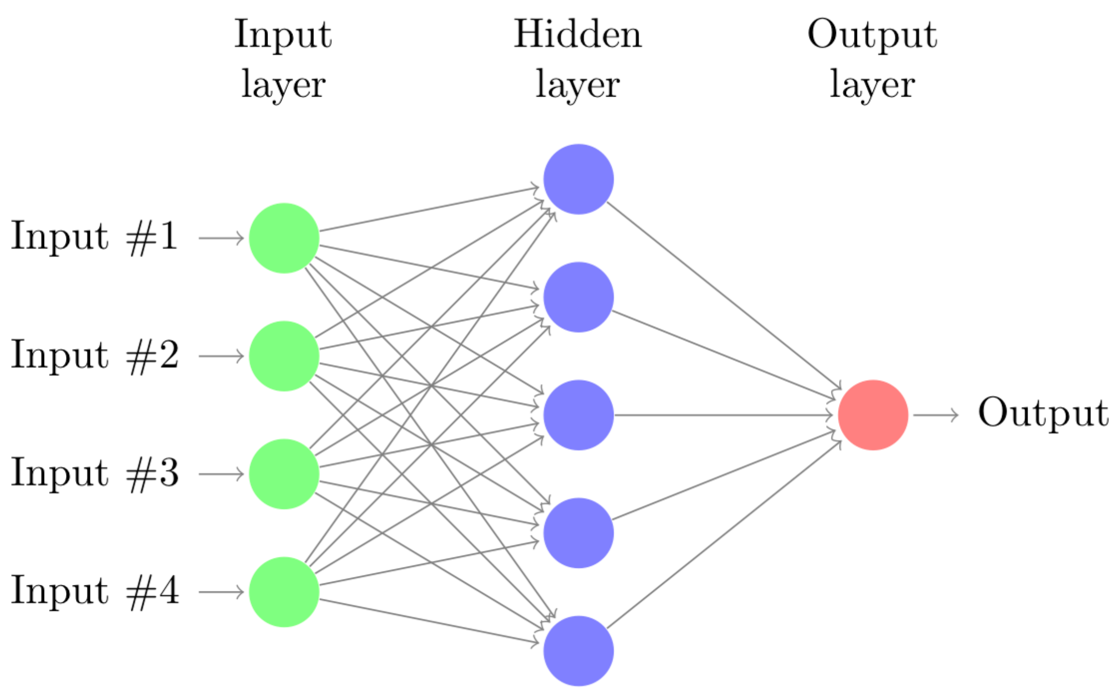
With networks that only have directional edges, we can sometimes arrange the nodes into groups where the first group has edges directly connected only to the second group, and then the second group is connected to the third and so on. This grouping is a *partitioning* of the nodes. (Networks with this kind of partitioning have special properties, but details about these special networks are outside the scope of this course.)

Neural networks have such a structure. In deep learning, we call each of these groups *layers.*

- The first layer is the **input layer** where each variable is its own node.
- The last layer is the **output layer** where we get our final predictions
- All the other layers are **hidden layers**

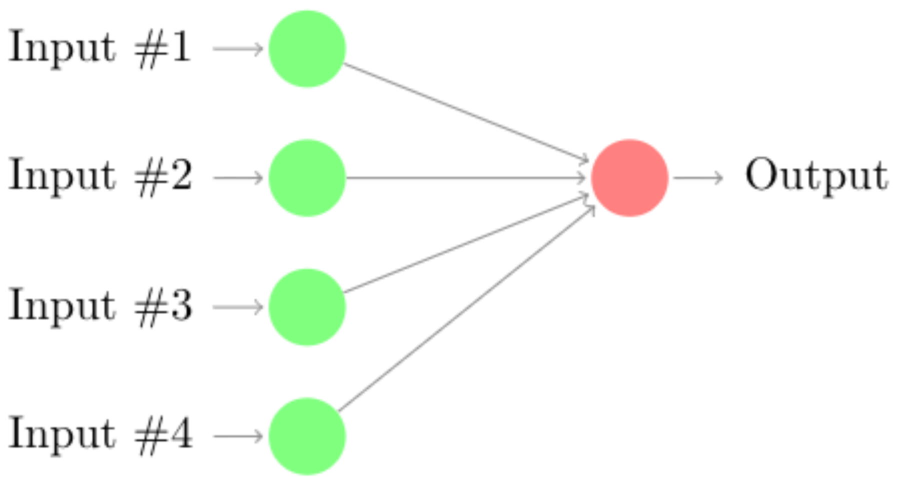
In the case of neural networks, the strength of the connections between the nodes in each layer are the weights.

Consider the below image from this website:



First Perceptron as a NN

So what do we mean by "the strength of the connections between the nodes in each layer are the weights"? Let's look at an example, we know: the perceptron:



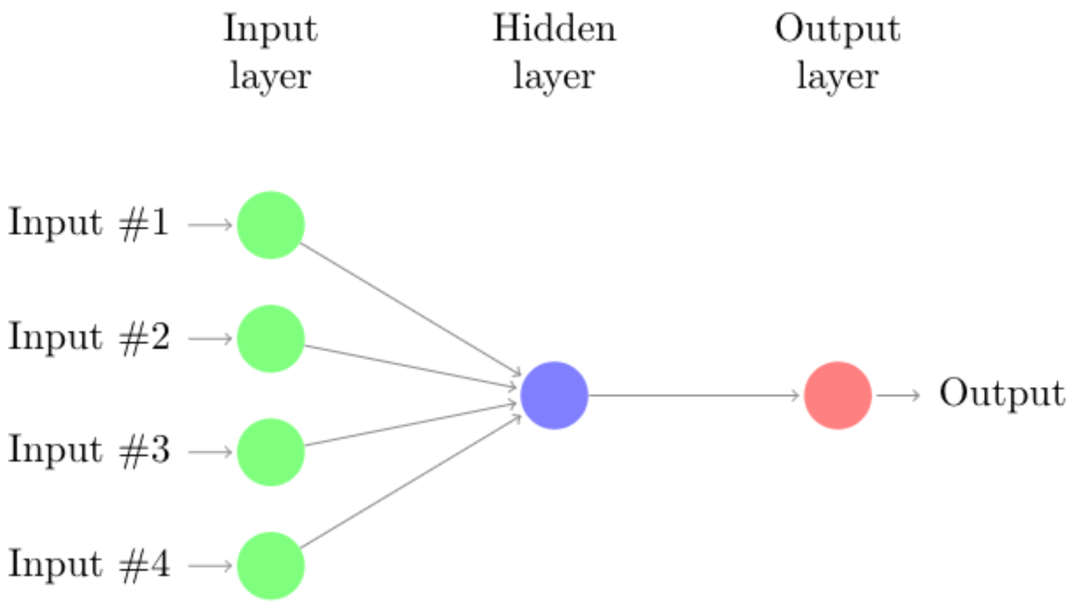
So we have our inputs (in green) and our output. The arrows represent the weights that get used within the red dot to determine the output. In this view, there are no "hidden" parts. We know what goes in (green) and have a prediction (in red). There's no mystery within; hence no hidden layer.

Within the red node, we have all the actions of the perceptron from last time: 1) gathering and 2) thresholding. If the inputs are denoted as $(x_1, x_2, \ldots, x_n)$, then the output of this red node would be given by (assuming no bias term):

$$Red = \phi(\Sigma_{i=1}^{n} w_i * x_i) \tag{1}$$

Adding a Layer

Let's consider the following network. This one has a hidden layer:



For this network, what is happening the purple and red nodes? (In other words, what is the output for red and purple?)

$$Red = ??? \tag{2}$$

$$Purple = ??? \tag{3}$$

Feed-Forward

There are two parts to a neural network: the forward part and the backward part. The first is called **feed-forward** and refers to how to push the information forward from one layer to the next.

The last piece of feed-forward that we did not talk about last time is *activation.* Essentially, instead of having the threshold part in the interior layers, we use an *activation* function that is a softer version of thresholding that tells our node how much to pass forward (in proportion the weighted sum). So instead of having the weighted sum immediately thresholded, we instead use a relaxation *until* the last layer where we do threshold into one class or another class.

There are many kinds of activation functions $\phi()$:

- Identity function: $\phi(x) = x$
- Sigmoid function: $\phi(x) = \frac{1}{1+e^{-x}}$
- ReLU: $\phi(x) = x \text{ if } x \geq 0; 0, \text{otherwise}$

In the below code blocks, code implementations for sigmoid and ReLU

In [ ]:
```
# Implementation for sigmoid
def sigmoid(x):
    pass
```

In [ ]:
```
# Implementaiton for ReLU
def ReLU(x):
    pass
```

Adding to the perceptron

Returning to your perceptron from last time, instead of the threshold function, replace it with the sigmoid function.

In [ ]:
```
# Add your edited perceptron here:
```

Backpropagation

Backpropagation is where we update our weights. As with our preceptron, we do this using a derivative. In typically backpropagation, we use the MSE as our "loss" function that we want to optimize. Considering our example network with one hidden layer, let's work through the details of how we would take this derivative for each of the weights in our network.

Recall that MSE is given by $\frac{1}{N}\sum_{i=1}^{N}(truth - predictions)^2$, where N is the number of data points. In this case the output of our **Red** function are the predictions. So our MSE is:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(truth - Red)^2 \tag{4}$$

This means that updating our weights involves taking partial derivatives of **Red** respect to our various weights

Partial derivative time!

We have our **Red** function from before:

$$\begin{aligned} Red &= \phi(w_1^{(1)} * z) \\ &= \phi(w_1^{(1)} * \phi(\Sigma_{i=1}^{n} w_i^{(0)} * x_i)) \end{aligned} \tag{5}$$

If we take the *sigmoid function* as the activation, then it is worth noting that the derivative of the *sigmoid function* $\sigma(x)$ is just $\sigma(x) \cdot (1 - \sigma(x))$. This fact makes computing our derivatives a touch easier.

However, it is important to note that due to the chain rule, the derivative of $\sigma(f(x))$ would be $\sigma(f(x)) \cdot (1 - \sigma(f(x))) \cdot \frac{d}{dx}f(x)$

Let's build a helper function that does the leading fraction for our *sigmoid* derivative `lead_frac_dev_sig` . It should take in the value of $f(x)$ for a particular x denoted `val_fx` .

In [ ]:

```
# Implementation for sigmoid
def sigmoid(val_fx):
    pass
```

We are interested in two derivatives: those for the input weights and the derivative for the weight from the hidden layer to the output layer.

Partial derivative with respect to the hidden weight

Here we are looking to update the weight between the purple and red nodes, that is:

$$\begin{aligned} \frac{d}{dw_1^{(1)}} MSE &= \frac{d}{dw_1^{(1)}} \frac{1}{N} \sum_{i=1}^{N} (truth - Red)^2 \\ &= \frac{2}{N} \sum_{i=1}^{N} (truth - Red) \frac{d}{dw_1^{(1)}} Red \end{aligned} \tag{6}$$

This means that we need to compute the partial derivative of **Red** with respect to $w_1^{(1)}$, that is:

$$\frac{d}{dw_1^{(1)}} Red = \frac{d}{dw_1^{(1)}} \phi(w_1^{(1)} * \phi(\Sigma_{i=1}^{n} w_i^{(0)} * x_i)) \tag{7}$$

To make the formula a bit clearer, let $H = \phi(\Sigma_{i=1}^{n} w_i^{(0)} * x_i)$. This H represents the output from the hidden layer. Using this simplification, we have:

$$\begin{aligned} \frac{d}{dw_1^{(1)}} Red &= \frac{d}{dw_1^{(1)}} \phi(w_1^{(1)} * H) \\ &= \phi(w_1^{(1)} * H)(1 - \phi(w_1^{(1)} * H)) \frac{d}{dw_1^{(1)}} (w_1^{(1)} * H) \\ &= \phi(w_1^{(1)} * H)(1 - \phi(w_1^{(1)} * H))(H) \end{aligned} \tag{8}$$

In [ ]:

```
# Implementation for partial derivatives for hidden weight
def hidden_weight_update(curr_hidden_weight,hidden_output):
    pass
```

Partial derivative with respect to the input weights

This second derivation is even less friendly then the first one. Here we are taking a derivative with respect to the input weights, which are the inner weights in this function. Here we are looking to update the weight between the green nodes and the red one, that is:

$$
\begin{aligned}
\frac{d}{dw_i^{(0)}} MSE &= \frac{d}{dw_i^{(0)}} \frac{1}{N} \sum_{i=1}^{N} (truth - Red)^2 \\
&= \frac{2}{N} \sum_{i=1}^{N} (truth - Red) \frac{d}{dw_i^{(0)}} Red
\end{aligned} \tag{9}
$$

This time let $\beta = \Sigma_{i=1}^{n} w_i^{(0)} * x_i$. This simplifies **Red** to be: $Red = \phi(w_1^{(1)} * \phi(\beta)$. Then by the chain-rule, the partial derivative of **Red** with respect to $w_i^{(0)}$ is:

$$
\begin{aligned}
\frac{d}{dw_i^{(0)}} Red &= \frac{d}{dw_i^{(0)}} \phi(w_1^{(1)} * \phi(\beta)) \\
&= \phi(w_1^{(1)} * \phi(\beta))(1 - \phi(w_1^{(1)} * \phi(\beta))) \frac{d}{dw_i^{(0)}} (w_1^{(1)} * \phi(\beta)) \\
&= \phi(w_1^{(1)} * \phi(\beta))(1 - \phi(w_1^{(1)} * \phi(\beta)))(w_1^{(1)}) * \frac{d}{dw_i^{(0)}} \phi(\beta)) \\
&= \phi(w_1^{(1)} * \phi(\beta))(1 - \phi(w_1^{(1)} * \phi(\beta)))(w_1^{(1)}) * \phi(\beta)(1 - \phi(\beta)) * x_i
\end{aligned} \tag{10}
$$

Next let's code partial derivatives for our input weights and our hidden weight:

In [ ]:

```
# Implementation for partial derivatives for hidden weight
```

There are a lot of layers to each computation! And this is just for one update!

Tuning NNs involve a forward and backward step:

- **Forward** - Get the guesses for the classes, before going backwards to..
- **Backward** - Get updates for the weights, before going forward again...

Submitting this lab (aka "Final" Thoughts)

To finish up this lab, you will create a post on **#lab21_submissions** addressing one of the following questions:

1. What surpises you about NN? (even just from this super small example)
2. How many parameters need to be updated each time we go forward and backwards (noting that the update occurs in the backwards step)?
3. Does it surprise you that deep learning (ie. systems built out of NN) is so effective? Why or why not?

If your have questions from this lab, post them to #lab_questions with the preamble **Lab21**. If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Putting it all together

If you are super curious, try putting the pieces together from this lab to build your first full multi-layer perceptron.

Keeping the idea stochastic gradient descent in mind, code `firstNN` that takes in:

1. A dataset
2. The max number of epochs

It should:

1. Determine the right number of nodes at the input layer given the input data
2. Set initial weights for those in the first layer and from the hidden to the output layer
3. Use a threshold function at the output layer to assign the positive and negative classes
4. Update the weights each time using the derivative functions

This function should eventually output all of the weights and the predictions.

Note: Recall that this lab was designed following parts of this blog post which defines a class for the neural network. You can absolutely do this, but in our case, we only are defining one network instead of several.

In [ ]:
```python
# Add your firstNN here
```

Resources consulted

1. *Python Machine Learning*
2. Images based on this site
3. How to build your own Neural Network from scratch in Python
4. Building Neural Network from scratch
5. Sigmoid
6. Activation functions