Lab 20

Today, we begin our journey into deep learning, by gathering the pieces of a neural net. Today'a goals are:

0. Understand the components of a perceptron
1. Code a threshold function
2. Employ stochastic gradient descent to update information inside our perceptron

Acknowledgement This lab has been streamlined to limit the amount of math-to-python translation. This streamlining was done in consultation with Kathleen Hablutzel's notes from the course.

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["group"] = -1*new_data["group"]
new_data_np = new_data.to_numpy()
```

Artificial Neurons

The idea behind neural networks is to build a system that works similar to how our brains function. Brains are a complex network of *neurons* (https://en.wikipedia.org/wiki/Axon#/media/File:Neuron.svg) which -- thanks to neuroscience -- we understand to work as follows:

0. Take in information via their dendrites (https://en.wikipedia.org/wiki/Dendrite)
1. Synthsize this information via the nucleus
2. Activate (or not) and send this sythesis down the axon

Artificial neurons work very much in this manner. Beginning with the simplest example, a classification perceptron works as follows:

0. The information that is taken is the *input* variables that we feed it.

1. To synthesize this information, we use a weighted sum. The weights are determined through training
2. Finally, we use a threshold function to determine the positive or negative class assignment. This is the *output.*

Like a brain neuron that is constantly updating and learning, our preceptrons also update constantly by comparing the guessed classes to the true classes. Today we will focus on coding just one perceptron before we consider larger examples.

First Perceptron

Training a perceptron is an iterative training process. In this process, we are aiming to set:

- The weights for the inputs
- The **bias** unit

Weighting function

Note: I find the first image in [this blog post (https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975)](https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975) to be particularly helpful for visualizing what is going on in this part.

The role of the weighting function is to determine how much attention we should pay to each input to build a successful classifier. This means that for each input, we have a weight w. The output of the weighting function is a sum of the weights times their respective input. For a given data point d with n variables represented as $(d_1, d_2 \ldots d_n)$, we write

$$\begin{aligned} \widetilde{WF}(d) &= (w_1 * d_1) + (w_2 * d_2) + \cdots + (w_n * d_n) \\ &= \Sigma_i (w_i * d_i) \end{aligned}$$

The result is a weighted sum of each variable of the input. The higher a weight, the more attention that we pay to the corresponding input in our perceptron.

In the below code block, create a function `weight_function` that takes in weights and **one** data point (as a row) and computes the weighted sum for that data point. Note that the number of weights is the is the same as the number of variables in the data.

In [ ]:
```
# Add your weight_function here
def weight_fuction(weights, datapoint):
    """Compute the weighted sum"""
    pass #return weighted_sum
```

Thresholding

So far we have completed the "sensing" and "synthesizing" steps of a neuron, with our inputs then collated through the weighting function. Our next step is to decide whether to send a "positive" or "negative" class assignment. This is determined via a *thresholding function.* Simply put, if our weighted sum is above a value θ, then we assign the positive class and if it is less than θ, then we assign a negative class.

In the below code block, write a function `thresh` that takes in one weighted sum and one singular value for θ. The output should be either `-1` and `1` where the negative class is assigned when the weighted sum is less than θ and where the positive class is assigned otherwise.

```
In [ ]: # Add your thresh function here
        def thresh(weighted_sum, thresh):
            pass #return assigned_class
```

Bias unit

To do this thresholding, we have to have a value in mind for θ. When we consider that we are setting both the weights for our weighted sum as well as the value that they will be compared against, this all feels a bit circular.

Also, instead of thresholding based on some tuned value, it would easier if we could just have the positive class relate to when the weighted sum is positive and the negative class when the weighted sum is negative. In symbols, instead of $WF(d) \geq \theta$, we want to make the comparison: $WF(d) - \theta \geq 0$.

Here we use a trick to give us an adjusted weighting function to make this more desirable comparison possible. For each data point, *pad* it with a 1. So a given data point $d = (d_1, d_2 \ldots d_n)$ becomes $(1, d_1, d_2 \ldots d_n)$. This padded vector allows us to adjust our weighted sum as follows:

$$WF(d) = (-\theta * d_0) + \Sigma_{i=1}^{n}(w_i * d_i)$$

If we view $(-\theta)$ as the zeroth weight, then we have our weighting and our thresholding in the same step. Although, we can view $(-\theta)$ as a weight, this goes by another name: the **bias unit.**

Create a function `weighted_sums` that takes in weights (including the bias value) and **one** data point (as a row) It should pad the data vector and compute the expanded weighted sum for that data point.

Then create an updated `threshold` function that assigns a positive or negative class based on the value of the weighted sum.

Typically $WF(d)$ is plainly referred to as z and the thresholded value is written as $\phi(z)$.

```
In [1]: # Add your weighted_sums here

        def weighted_fuction(weights, datapoint):
            """Compute the weighted sum"""
            pass #return weighted_sum
```

In [ ]:
```
# Add your threshold function here

def thresh(weighted_sum, thresh):
    pass #return assigned_class
```

Updating our weights and bias

At this point, we have functions that allow us to feed weights and data in and get back classifications. But we have yet to *learn* the weights and bias. To do this, we engage with a familiar idea: stochastic gradient descent (or SGD). Using SGD, we start with some guesses for our weights and find the optimal ones by using one datapoint at a time.

Critical to any gradient descent (including SGD) is the *update step.* The updates for preceptron weights (and bias) happen in the usual manner with $W_{\text{new}} = W_{\text{old}} + \Delta W$. The change for each weight is governed by the **perceptron learning rule** (http://hagan.okstate.edu/4_Perceptron.pdf) given by:

$$\Delta W_j = \eta * (1/2) * (truth - guess) * (j^{th}(input))$$

With η as the learning rate and the $j^{th}(input)$ referring to the j^{th} variable of a datapoint.

Examing Truth and Guess

In the above set-up, we assume that we are feeding our perceptron one data point at a time. Before coding this, let's explore the potential cases and if there is a change for our weights (ie. is $\Delta W_j \neq 0$):

- True class is 1, guess is 1 $\rightarrow$ ΔW_j **changes/does not change**
- True class is 1, guess is -1 $\rightarrow$ ΔW_j **changes/does not change**
- True class is -1, guess is 1 $\rightarrow$ ΔW_j **changes/does not change**
- True class is -1, guess is -1 $\rightarrow$ ΔW_j **changes/does not change**

Putting the above all together: Will the weights change for each data point?
Your thoughts

Encoding changes in the weights

In the next two code blocks, we will code the `change_w_i` function that will adjust the i^{th} weight. Then we will code `change_w` to update all the weights in your perceptron.

As a reminder, we eventually be tuning our perceptron using SGD. This means that updating our weights will use exactly one datapoint at a time and **not** the whole dataset.

```
In [ ]: # Add your change_w_i here
        def change_w_i(data_true_cls, data_guess_cls, data_i, w_i):
            """ Updates the i^th weight based on a single data point:

            INPUTS -
                  DATA_TRUE_CLS - actual class for the data
                  DATA_GUESS_CLS - current guess for the data
                  DATA_I - the i^th variable value for a single data point
                  W_I - the i^th weight

            OUTPUT - The new i^th weight"""

            pass #return new_wi
```

```
In [ ]: # Add your change_w here
        def change_w(data_true_cls, data_guess_cls, data_row, all_weights, nu)
            """ Updates all of the weights based on a single data point:

            INPUTS -
                  DATA_TRUE_CLS - actual class for the data
                  DATA_GUESS_CLS - current guess for the data
                  DATA_ROW - a single data point
                  ALL_WEIGHTS - the current weights
                  NU - the learning rate

            OUTPUT - The new weights"""
            pass #return new_weights
```

Putting it all together

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

0. A dataset
1. The max number of epochs

This function should cycle through each epoch selecting a random (https://numpy.org/doc/stable/reference/random/generated/numpy.random.shuffle.html) datapoint each time (but not reusing one until the next epoch), and eventually outputing the weights and the bias term.

In [ ]:
```
# Add your perceptron here
def perceptron(data, max_epochs):
    # Set your learning rate
    nu = ??

    # Initial starting parameters
    weights = ???

    # Create an epoch loop
    for something in something:
        # Randomly shuffle your data
        np.random.shuffle(???)

        # Loop over shuffled data
        for thatthing in thisset:

            # Compute a guess class for your current datapoint
            # given the weights that you have
            guess = ??

            # Update your weights with this one data point
            weights = ??
```

Final Thoughts

To finish up this lab, answer the question: **Do you agree the perceptrons are kind of like brain neurons?** Share your thoughts in a post on **#lab-20-submission** channel on slack with your answer.

If your have questions from this lab, post them to #lab_questions. 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.

Resources consulted

0. *Python Machine Learning*
1. An Introduction to Python Machine Learning with Perceptrons (https://www.codementor.io/mcorr/an-introduction-to-python-machine-learning-with-perceptrons-k7pn85vfi)
2. Neural Network Design, Chapter 4 (http://hagan.okstate.edu/4_Perceptron.pdf)
3. Perceptron Learning Algorithm: A Graphical Explanation Of Why It Works (https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975)