Urvika Gola
585 Assignment 5

1. A single neuron with a sigmoid activation function is essentially a linear classifier. It can only separate data that is linearly separable. If the data in train.dat is linearly separable, then a single neuron can "memorize" it by finding the appropriate linear boundary. However, if the data is not linearly separable, a single neuron will not be able to perfectly classify all instances in train.dat. In the given question we had linearly separable data that is why we could able to make neuron memorize train.dat. We had data that was reasonably linearly separable in the given question. The model can still identify a decision boundary that fairly divides the classes, as evidenced by the maximum accuracy of 80%; however, there are some situations in which the linear decision boundary is not perfect in dividing the classes.

2. The same logic applies to test.dat as it does to train.dat. If the data in test.dat is linearly separable, then a single neuron can classify it correctly. However, it's important to note that "memorizing" the test set is not the goal.If a model merely memorizes the training data without learning the underlying patterns, it may perform poorly on the test data due to overfitting.

3. As the number of epochs increases, the sigmoid function's error usually performs better as compared to sigmoid function's error.
(a) 3.1 Empirically, the error calculated from the sigmoid's function often yields better or comparable accuracy to the classifier's output. When learning rate = 0.025 and epochs = 50 epochs, maximum accuracy = 80% with sigmoid function which is higher than classifier function's error of 68%.

(b) Non-empirical Explanation:
The classifier decides between two options, 0 or 1. If it uses these numbers to figure out its mistakes, it gets a clear message about what it got wrong, but this might not show how close the guess was to being right. When the model predicted wrong but it was very close to the correct answer, this yes-or-no answer doesn't help much because it doesn't show how big the mistake was.

But if it uses the sigmoid function, which gives a number **between** 0 and 1, to find errors, the classifier can see how sure it was about its guess. This number helps the classifier learn in a more detailed way, adjust weights, which can make the model's prediction better.

```
(venv) (base) ugola@ugola 05-nns-intro-Urvika-gola % python3 implementation/neuron.py data/train.dat data/test.dat 0.005 15
data/train.dat data/test.dat 0.005 15
Result: Accuracy on test set using classifier error: 72.0%
**************************
Result: Accuracy on test set using sigmoid error: 68.0%
**************************
(venv) (base) ugola@ugola 05-nns-intro-Urvika-gola % python3 implementation/neuron.py data/train.dat data/test.dat 0.01 50
data/train.dat data/test.dat 0.01 50
Result: Accuracy on test set using classifier error: 55.0%
**************************
Result: Accuracy on test set using sigmoid error: 76.0%
**************************
(venv) (base) ugola@ugola 05-nns-intro-Urvika-gola % python3 implementation/neuron.py data/train.dat data/test.dat 0.025 25
data/train.dat data/test.dat 0.025 25
Result: Accuracy on test set using classifier error: 72.0%
**************************
Result: Accuracy on test set using sigmoid error: 76.0%
**************************
(venv) (base) ugola@ugola 05-nns-intro-Urvika-gola % python3 implementation/neuron.py data/train.dat data/test.dat 0.025 50
data/train.dat data/test.dat 0.025 50
Result: Accuracy on test set using classifier error: 68.0%
**************************
Result: Accuracy on test set using sigmoid error: 80.0%
**************************
```

4. Weights for Useless Attributes:

Useless Attributes are not likely to yield any useful data or predictive ability. Because of this, during training, ML models will typically give these useless attributes almost ~0 weights. These near-zero weights indicate that these attributes are not used by the model in its prediction process. Due to noise or tiny correlations, the weights may not be precisely zero, but they are very close to zero.

Answer 2:

Getting Started:
I started off with using Word2Vec as I had used that before in Text Retrieval coursework. I had experience working with Spotify Annoy Library which is neural based embedding and gives better embedding than Word2Vec. But for simplicity I used Word2Vec by downloading the GoogleNews-vectors-negative300.bin.gz and importing it to my project. Next I moved on to using One Hot Encoding of both XPOS and UPOS tags.

First out of many Solutions tried:
With these two things in place, I was checking the Word2Vec encoding of the top 3 words from stack and the buffer to be used as one feature for the Model. However, As I have never created any ML model before it was getting very difficult for me to create one, with so many features. But, I finally created one, and while running it took so much time to train the model, and initially it was due to infinite loops for eg. stack is not getting empty because no right-arc operation is being predicted, so stack remains full forever.

Problems/Results:
My Model was making wrong prediction of left-arc always. When I debugged the code to resolve these situation, the head prediction was wrong as it was always predicting [-1] value for all heads array. So after days of doing the same thing fixing it and observing no results change, I read research papers discussed in class and some articles on how to create Models, I learned it required input and output dimensions, I know that outputs were 3, which are SHIFT, LEFT-ARC, and RIGHT-ARC, but I was unsure about the input dimensions. Ater applying some heuristics, eventually my y features were coming empty.

```
def _train(self):
    print("Training started...")
    start_time = time.time()

    X, y = [], []
    for index, sentence in enumerate(self.train):
        print(index)
        gold_actions = self.get_gold_actions(sentence)
        if not gold_actions:  # Ensure gold actions are not empty
            continue
        features = self.extract_features(sentence)
        X.extend(features)
        y.extend(gold_actions)

        # Log progress every 100 sentences
        if (index + 1) % 10 == 0:
            print(f"Processed {index + 1} sentences...")

    if not y:  # Check if y is empty after processing all sentences
        raise ValueError("No labels found for training. Check the data processing steps.")
```

Another Solution:
Eventually I moved to working with implementing a basic perceptron model. I created a perceptron class that contains methods to predict output based on weights and inputs and to train the perceptron using labeled input data. I have defined a neural network with three hidden layers and a dropout layer to prevent overfitting. The network uses ReLU activation for hidden layers and softmax for the output layer. It is trained with training data. It encodes POS tags into one-hot encoding and uses these as features to train a neural network. I am not using XPOS tags now as it was getting more complicated and taking time to train as feature list was growing, as there were unique 50 XPOS as compared to unique 18 UPOS.

Model Training and Parsing Logic:
The one-hot encoded vectors for the UPOS tags of the top five elements on the stack are intended to be stored in the lists. To train the oracle, the perceptron receives the individual feature vectors for each of the top five stack elements concatenated into a single feature vector. This trains the model to predict the action. So, in short, UPOS tags as one-hot vectors and feeds them into the FFN model to learn parsing decisions (SHIFT, LEFT-ARC, and RIGHT-ARC). Vectors representing the top 5 elements of the stack and buffer are concatenated to serve as features for the model. With these basic infrastructure in place, I was able to get accuracy of 41% with only perceptron.

I tweaked my model a bit more by adding a new feature that considers the top elements in the buffer too instead of just the stack. This managed to raise my accuracy to 47%.

I took inspiration from other students who were using multilayer perceptron, I got intimidated a bit as even to implement a single layer perceptron for a beginner with ML like myself took so many days. But to improve the accuracy I had to try something because 47% was less than my HW4 accuracy. I watched some youtube tutorials on multilayer perceptron implementation, using TensorFlow and Keras.

Final Solution:
Here's the basic outline of the model I put together:

1. Start with a layer of 128 neurons, then cut the chances of overfitting with a 70% dropout.
2. Add another 64-neuron layer, followed by the same dropout.
3. Do that one more time.
4. Finally, end with a layer of 3 neurons that decides among SHIFT, LEFT-ARC, and RIGHT-ARC, which are 3 the actions my parser can take.
5. For figuring out what features to feed the model, I used one-hot encoding for the UPOS tags from the words in the parser's current stack and buffer, taking the top five from each.

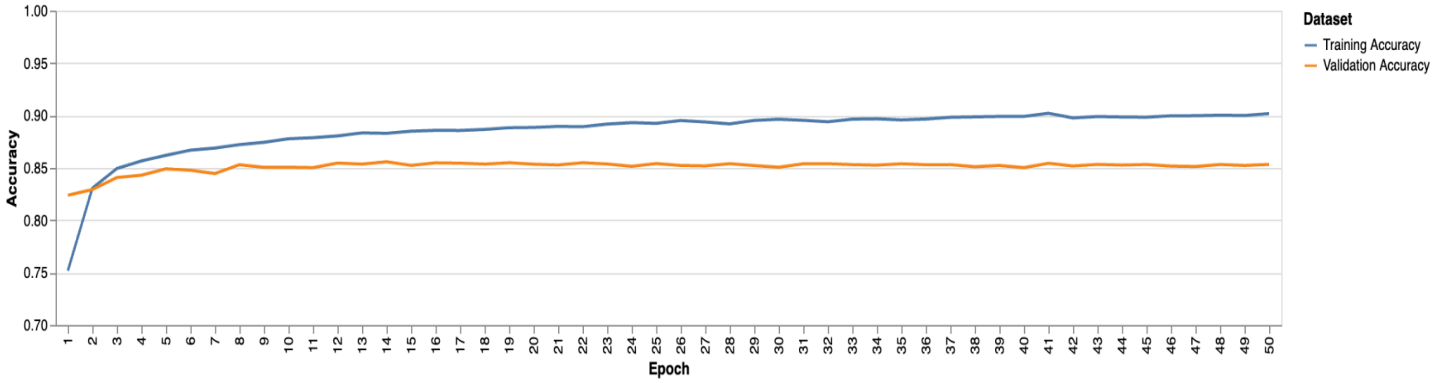
The model is trained to predict parser actions (SHIFT, LEFT-ARC, RIGHT-ARC) using a neural network that processes the parts-of-speech of sentence words. It uses one-hot encoding for features, the Adam optimizer to refine its predictions, and a sparse categorical cross-entropy loss function to improve accuracy. Dropout layers are included to avoid overfitting, making the model generalize better. The softmax function at the output layer assigns probabilities to each possible parser action, contributing to the model's improved accuracy in predicting the correct transitions.

Challenges:
1. ML expertise, If I had taken an ML course previously, I could have performed better and taken less time to implement it.
2. Using Word2Vec embedding was one of the initial things I started with but eventually I dropped this idea because I was unable to implement it correctly as model was getting complicated.
3. In my current implementation I am using the features for top 5 elements of Stack and Buffer, which is not ideal which do not consider the context or nuances of the whole sentence, which could be improvised if I used Word2Vec embeddings.
4. Increasing batch_size to 512 improved my accuracy, I got this suggestion from other student's excel sheet.
5. I also changed my epoch to 100 and that did not help, as with epoch as 50, I get 73.3

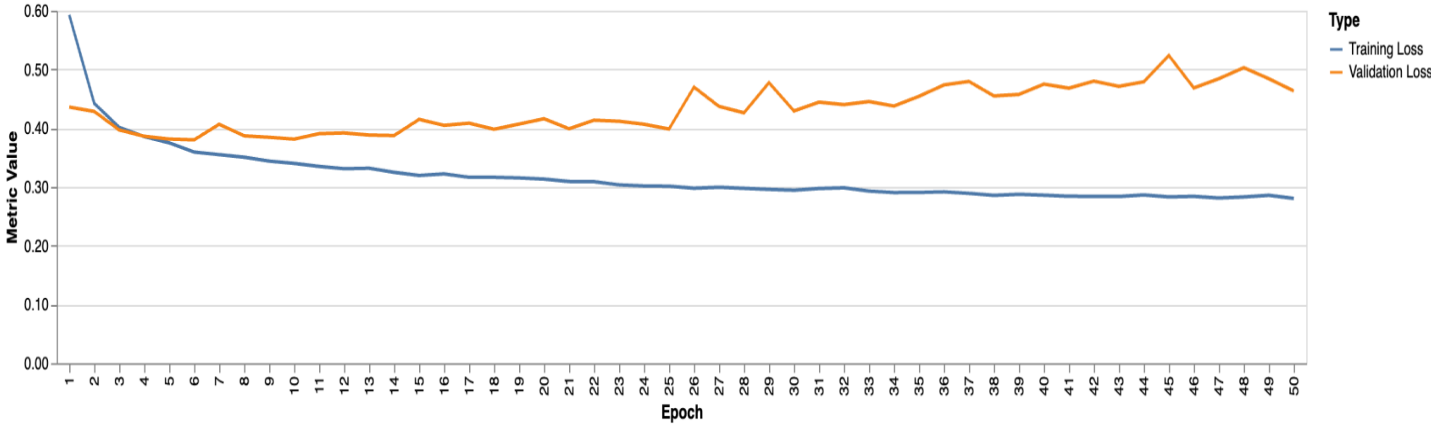
Result:

```
FancyArcStandardParser Unlabeled Accuracy (UAS): 0.733 [25429 tokens]
FancyArcStandardParser Labeled Accuracy (UAS):   0.000 [25429 tokens]
```

Graphs:



Accuracy Graph:
This above graph shows the accuracy-related performance evolution of the model over a series of training epochs.
In the iterative process of model optimization, discrete intervals are indicated by the horizontal axis, which represents epochs. Accuracy is measured on the vertical axis, which is scaled from 0 to 1. The "Training Accuracy" series in the above graph tracks how well the model predicts the training data it learns from, while the "Validation Accuracy" series shows how the model's predictive ability on a held-out dataset that was not seen during training.



Loss Graph:
The above graph shows the cost or inaccuracy of the model's predictions. The epochs are plotted on the horizontal axis to show how the situation has changed over the training iterations. This time, a smaller value indicates better performance, and the loss value is shown on the vertical axis. The model's error rate as it directly learns from the input data is displayed in the blue "Training Loss" series. On the other hand, the error rate that occurs when the model is exposed to unseen data is shown by the orange "Validation Loss" series.

On training and validation sets, accuracy and loss should ideally increase simultaneously. Divergence may indicate overfitting, a situation in which an NLP model is too adapted to the training data and is unable to function well on fresh, untested language data. Examples of this include validation metrics plateauing or getting worse while training metrics keep getting better.