# CSE 143 Assignment 3

Sanjay Shrikanth, Matthew Daxner, Collin McColl, Soren Larsen

6/3/2022

## 1 Sentiment Analysis

### 1.1 Text Classification with RNNs

*Model Description:* We implemented a Simple RNN-based text classifier that predicts whether a sequence of words have positive or negative sentiment. We followed the given sample code, which processes data from the IMDB dataset and converts each unique word to a number in the preprocessing phase. Our model takes in data. Our model embeds the integer representation of the sentences with a dimension size of 16. The neural network then puts it through a Simple RNN layer of 40 units, a Dropout layer with $p = 0.5$, and a Dense layer of 1. We use Adam as our optimizer with a learning rate of 0.01 and Binary Cross Entropy as our loss function.

*Experimental Procedure:* We began by going through the sample code and understanding how the data is loaded and processed into the input tensors. We used the textbook as reference to do so. For our model, we followed the given suggestions and created a Sequential model that has all the components. To see the model's performance on the test set, we performed processing similar to the train set on the test set and evaluated the model using accuracy as our metric.

*Performance of the Simple RNN:*

| | Train | Test |
|---|---|---|
| Accuracy | 0.4999 | 0.4852 |

### 1.2 Text Classification with LSTMs

*Model Description:* We modified our previous simple RNN model and added a LSTM layer in between the embedding and the first Dense layer. Our LSTM layer has 40 units and has the same parameters as the previous one except the learning rate, which is now 0.005 as that yielded significantly better results on the training set.

*Experimental Procedure:* After adding the LSTM Model, we first put an arbitrary number for the LSTM layer. Treating it as a hyper parameter we tried smaller values and ended up converging to 40 units, which proved to yield the best performance. We also noticed that often, the model would have high training accuracies in the middle epochs, then decrease in performance significantly. Adjusting the learning rate proved to fix that deviation, helping us achieve the best results overall.

*Performance of RNN with LSTM layers:*

| | Train | Test |
|---|---|---|
| Accuracy | 0.9187 | 0.7173 |

We observed that with the tuned LSTM layer, training accuracy improved significantly as well as testing accuracy. When tested with longer sequences(larger embedding size) we found that the accuracy's on the train set generally fell with an increased size. The test Accuracy's remained nearly the same.

*Performance of RNN with increasing embedding size:*

| | Train | Test |
|---|---|---|
| Embedded size 16 Accuracy | 0.921 | 0.7093 |
| Embedded size 64 Accuracy | 0.92 | 0.69 |
| Embedded size 128 Accuracy | 0.83 | 0.69 |

## 2 Viterbi Algorithm (Parts 2 & 3)

### 2.1 Deriving Viterbi

1. *Proof.* We are given that for every possible value of $y_j$:

$$v_j(y_j) = \max_{y_1 \cdots y_{j-1}} \left[ \sum_{i=1}^{j} s(\mathbf{x}, i, y_{i-1}, y_i) \right] \tag{1}$$

where function $s(\mathbf{x}, i, y_{i-1}, y_i)$ is the local score given sentence $\mathbf{x}$ and $\mathbf{y} = \{y_1 ... y_{j-1}\}$ is the sequence of $j-1$ tags in the sentence.

Suppose we break up the sum inside the argmax. Using laws of summation, we can break the sum into the sum of the last element plus the summation of the previous elements:

$$\sum_{i=1}^{j} s(\mathbf{x}, i, y_{i-1}, y_i) = s(\mathbf{x}, j, y_{j-1}, y_j) + \sum_{i=1}^{j-1} s(\mathbf{x}, i, y_{i-1}, y_i) \tag{2}$$

With the Markov Assumption, we know that a tag position $y_k$ depends on the sequence of tags $y_1 ... y_{k-1}$ before it. Since we are extracting $s(\mathbf{x}, j, y_{j-1}, y_j)$ from the sum, the sequence of tags we are argmax-ing over partition into $y_{j-1}$ and $y_1 ... y_{j-2}$. So, we can rewrite (1) as:

$$v_j(y_j) = \max_{y_{j-1}} \left[ s(\mathbf{x}, j, y_{j-1}, y_j) + \max_{y_1 \cdots y_{j-2}} \left[ \sum_{i=1}^{j-1} s(\mathbf{x}, i, y_{i-1}, y_i) \right] \right] \tag{3}$$

Observe that using the given equation (1):

$$\max_{y_1 \cdots y_{j-2}} \left[ \sum_{i=1}^{j-1} s(\mathbf{x}, i, y_{i-1}, y_i) \right] = v_{j-1}(y_{j-1}) \tag{4}$$

Since the recursive calls of $v_{j-1}$ handle the argmaxes of $\{y_1..y_{j-2}\}$, we only need to find the $y_{j-1}$ that maximizes (3). Therefore we can substitute (4) into the summation to get,

$$v_j(y_j) = \max_{y_{j-1}} \left[ s(\mathbf{x}, j, y_{j-1}, y_j) + v_{j-1}(y_{j-1}) \right] \tag{5}$$

□

2. Viterbi takes in a sequence of observations of size $X$ that represents sequence probabilities over time and a sequence of Hidden Markov Model states $N$ as parameters. In our case $X$ is the number of words in the sentence and $N$ is the number of possible tags. The initialization phase iterates through all the $N$ states to create the Viterbi matrix that memoizes previous sequence probabilities. This allows access to recursive calls to be done in $O(1)$ time.

   The recursive step iterates through all $X$ time steps and for each time step, examines the states at that positions. This requires $O(X \cdot N)$ update operations. In each update, the viterbi matrix entry at a specific time step and HMM state is initialized to the argmax of the previous probabilities, which is a $O(N)$ operation. Therefore, the whole recursive step takes $O(X \cdot N^2)$.

   The termination step iterates through all the states at the final time step and takes the max probability and path to output the best sequence of tags that has the highest probability of being the correct sequence. This is a $O(N)$ operation.

   For total runtime, the algorithm takes $O(N) + O(X \cdot N^2) + O(N)$. Therefore the runtime for Viterbi is $O(X\dot{N}^2)$.

## 2.2 Programming Viterbi

## 2.3 Dev Set

Our implementation of *decode* processed 51578 token with 5917 phrases. It evaluated 2123 phrases correctly out of 6599 discovered phrases.

*Viterbi performance of ner.dev Set:*

| | Output |
|---|---|
| Precision | 32.17% |
| Recall | 35.88% |
| $F_1$ | 33.92 % |

## 2.4 Test Set

Our implementation of *decode* processed 46666 token with 5616 phrases. It evaluated 1829 phrases correctly out of 6648 discovered phrases.

*Viterbi performance of ner.test Set:*

|  | Output |
|---|---|
| Precision | 27.51% |
| Recall | 32.57% |
| $F_1$ | 29.83 % |