Tap-Delay Neural Networks and Recurrent Neural Networks

Tre' R. Jeter

November 7, 2023

Abstract

Tap-Delay Neural Networks (TDNNs) and Recurrent Neural Networks (RNNs) are two distinct architectures used in sequence modeling tasks. TDNNs are feedforward networks that include layers with fixed time delays, designed to capture temporal dependencies in sequential data. In contrast, RNNs are designed to process sequential data with internal memory, making them well-suited for tasks with variable-length sequences and complex temporal dependencies. For grammar contexts like $\{0^+, 1\}$, TDNNs are expected to perform well. TDNNs excel at capturing short-range dependencies and patterns in sequential data, which is particularly suitable for recognizing simple grammatical structures. On the other hand, RNNs might overcomplicate the task for such straightforward grammatical rules due to their ability to capture long-range dependencies, making them less efficient in this specific context. Therefore, it is expected that TDNNs will outperform RNNs in this assignment.

1 Setup

We attempt to compare the performance of a TDNN and RNN on the $\{0^+, 1\}$ grammar, where + means one or more zeros. There are infinite strings that could belong (and not belong) to the grammar. For simplicity, we randomly generate 500 strings of length 60 with at most 10 consecutive 0's that belong to the grammar and another 500 strings of length 60 that do not belong to the grammar (more than 2 consecutive 1's). Therefore, we have a total of 1000 strings of length 60 that could belong to one of two classes. In early experimentation, the training set contained strings of length 60. However, the testing set contained strings of length 81 (80 for strings + 1 for label). Because of this, during training, each generated string of length 60 is also padded while preserving their original label. This allows the input of the models during training and testing to be consistent. Each generated string is one-hot encoded to represent the two classes. After this, I implement an 80/20 split to create training and validation sets. The labels themselves are converted into integers for easy mapping.

2 Tap-Delay Neural Network

2.1 Architecture

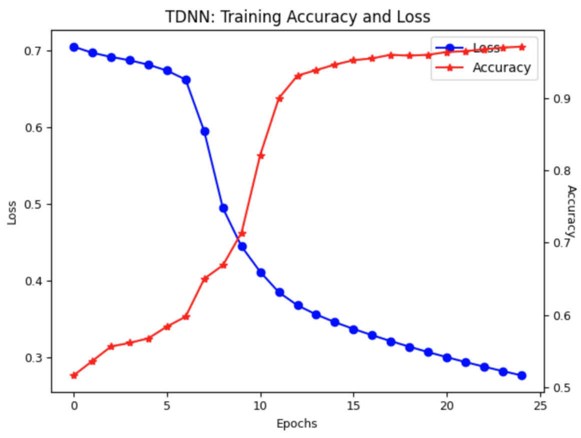
The TDNN architecture includes an input layer with input shape of 80×2. 80 represents the generated strings of length 60 that are padded to 80 as discussed in Section 1 while 2 represents the number of classes (belong to the grammar or not). The next layer is a 1D convolutional layer with 1 filter and delay line size of 5 with the ReLU activation function. This is followed by a flattening layer and dense layer with 1 neuron using the Sigmoid activation function. The optimizer used is Adam and the loss is calculated with the binary cross entropy loss function. In another case, the same architecture is followed only changing the delay line size to 20 for a comparison between memory depths and performance of a TDNN.

2.2 Training and Testing

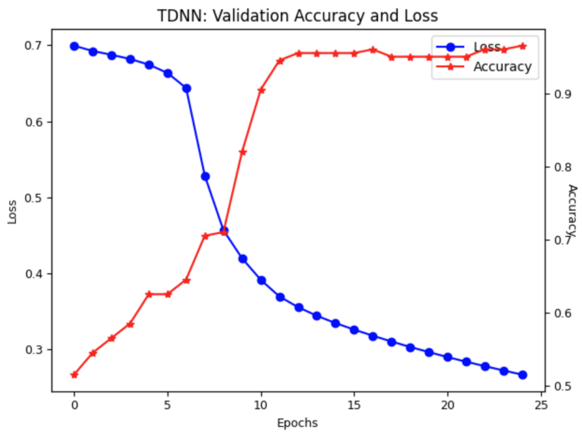
The TDNN models are trained for 25 epochs with a batch size of 32. Early stopping monitors the validation loss at a patience of 5. Training with delay line size of 5 reached a training accuracy of 97.12% while the validation accuracy reached 96.50%. When testing with this model, testing accuracy reached $\approx$94.9%. These results indicate that the model is generalizing relatively well, although it could be improved. The resulting confusion matrix can be seen in Figure 1. Figure 2 depicts the training loss/accuracy vs. epochs and the validation loss/accuracy vs. epochs. The figure shows the expected results: as loss decreases, accuracy increases.

$$\begin{bmatrix} 225 & 25 \\ 0 & 250 \end{bmatrix}$$

Figure 1: Confusion Matrix for Test Set with TDNN (delay = 5).



(a) Loss vs. Epochs + Accuracy vs. Epochs.



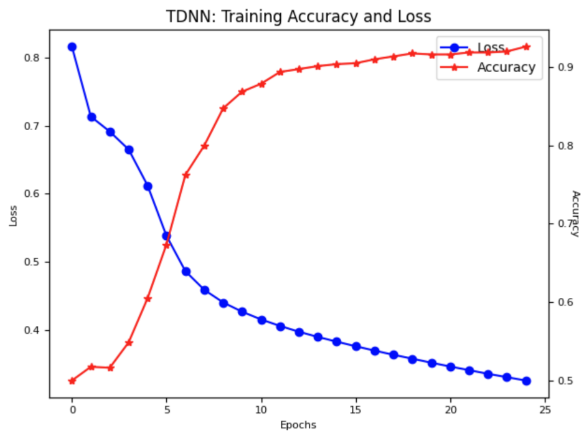
(b) Loss vs. Epochs + Accuracy vs. Epochs.

Figure 2: Training and Validation of a TDNN with Delay Line Size of 5.

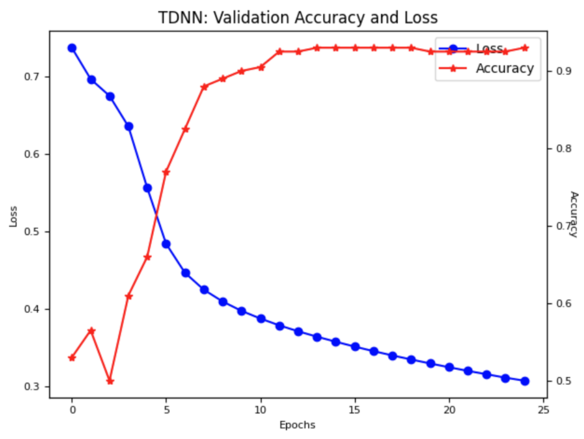
Training with delay line size of 20 reached a training accuracy of 92.62% while the validation accuracy reached 93.00%. When testing with this model, testing accuracy reached ≈92.40%. Like the previous model with delay line size of 5, these results indicate that the model is generalizing relatively well, but surely has room for improvement. Usually, with an increased memory depth, the model would be expected to generalize better. However, we can see that increasing the delay line size for this specific task is not that suitable because we are dealing with short-range sequences of data. Increasing the delay line size seems to have no positive effect in this case. The resulting confusion matrix can be seen in Figure 3. Figure 4 depicts the training loss/accuracy vs. epochs and the validation loss/accuracy vs. epochs. The figure shows the expected results: as loss decreases, accuracy increases.

$$\begin{bmatrix} 223 & 17 \\ 11 & 239 \end{bmatrix}$$

Figure 3: Confusion Matrix for Test Set with TDNN (delay = 20).



(a) Loss vs. Epochs + Accuracy vs. Epochs.



(b) Loss vs. Epochs + Accuracy vs. Epochs.

Figure 4: Training and Validation of a TDNN with Delay Line Size of 20.

3 Recurrent Neural Network

3.1 Architecture

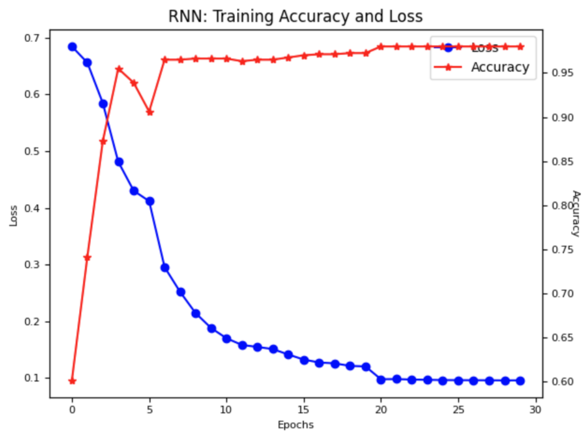
The RNN architecture includes an input layer very similar to the TDNN with an input shape of 80×2. 80 represents the generated strings of length 60 that are padded to 80 as discussed in Section 1 while 2 represents the number of classes (belong to the grammar or not). The next layer is a SimpleRNN layer with 4 neurons. Within the SimpleRNN layer, the return sequences are set to **False** and unroll is set to **True**. The return sequences variable allows the model to return the last output in the output sequence, or the full sequence. I only return the last output. Paraphrasing from the Keras website, "The unroll variable allows the model to be unrolled unless a symbolic loop is created. Unrolling can speed-up an RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences"[1]. Following the SimpleRNN layer are two dense layers. The first dense layer has 20 neurons and uses the ReLU activation function. The second dense layer has 1 neuron and uses the Sigmoid activation function. The optimizer used is Adam and the loss is calculated with the binary cross entropy loss function.

3.2 Training and Testing

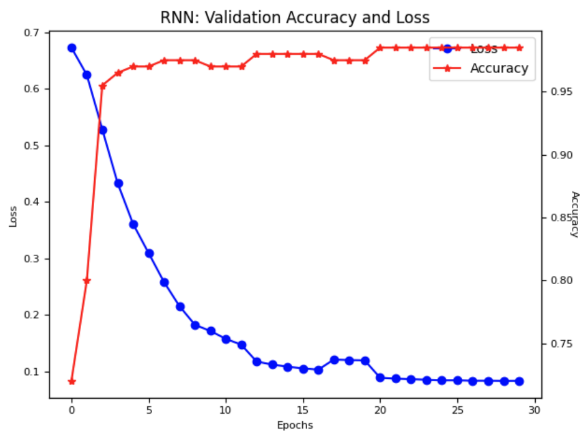
This model was trained for 30 epochs with a batch size of 32. Early stopping monitors the validation loss at a patience of 5. Training with this model reached a training accuracy of 98.00% while the validation accuracy reached 98.50%. When testing, this model reached a testing accuracy ≈87.99%. These results may infer over-fitting as the RNN trains well on the training and validation sets, but performs roughly 10-10.5% worse on the testing data. The resulting confusion matrix can be seen in Figure 5. Figure 6 depicts the training loss/accuracy vs. epochs and the validation loss/accuracy vs. epochs. The figure shows the expected results: as loss decreases, accuracy increases.

$$\begin{bmatrix} 211 & 39 \\ 21 & 229 \end{bmatrix}$$

Figure 5: Confusion Matrix for Test Set with RNN.



(a) Loss vs. Epochs + Accuracy vs. Epochs.



(b) Loss vs. Epochs + Accuracy vs. Epochs.

Figure 6: Training and Validation of an RNN.

4 Conclusion

The hypothesis for this work was that a TDNN would outperform an RNN when classifying a simple grammar $\{0^+, 1\}$. This hypothesis was upheld and proved correct for this specific case understanding that RNNs perform

[1] https://keras.io/api/layers/recurrent_layers/simple_rnn/

better with long-range sequences of data and would more than likely overcomplicate our simple task. It is noted that the RNN trained much better than both versions of the TDNN. However, both TDNNs generalize better than the RNN on the test data, hence proving why they are better suited for this specific task.