MATH4828B Machine Learning for Natural Language Processing

**Project 3 – Language Model**

# Report

**Name:** TSE, Ho Nam
**Student ID:** 20423612
**Dev Set Score:** 1.3916

## Contents



# 1 Architectures

There are two main types of architecture that I tried, and they are (i) LSTM and (ii) TCN.

## 1.1 LSTM

Long Short Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) that aims to solve the problem of gradient explosion and diminishing problems in simple RNN. Comparing with simple RNN, LSTM can be trained using a longer sequence and usually performs better. In a LSTM cell (Figure 1), there are four gates $i, f, o, g$ and two hidden states $c_t, h_t$. The equations for forward pass is as follows, with $\odot$ being the elementwise multiplication, $\sigma$ being the sigmoid activation and tanh being the tanh activation.

$$\begin{bmatrix} i \\ f \\ o \\ g \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \mathbf{W} \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix}, \qquad c_t = f \odot c_{t-1} + i \odot g, \qquad h_t = o \odot \tanh(c_t)$$

The intuitive idea behind these equations is that, $i$ governs how much information of $x_t$ should be written to $c_t$ and $f$ governs how much information of $c_{t-1}$ should be forgotten.

$g$ governs how much $x_t$ is reach revealed to $c_t$ and $o$ governs how much $c_t$ is written to $h_t$. This new model works better than simple RNN because of the gradient highway along the $c_t$ line which help solves the problem of gradient issues.

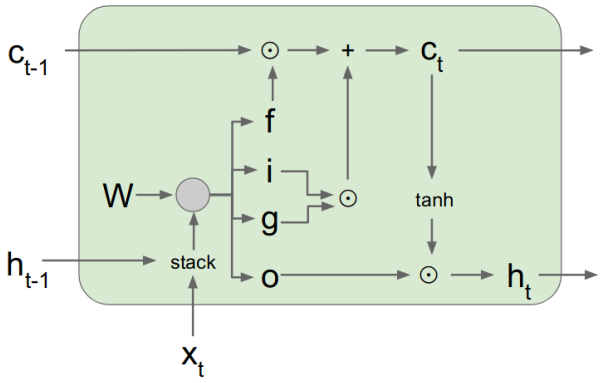


Figure 1: A LSTM cell.[1]

Even LSTM is a better choice than simple RNN, it has a lot more parameters than simple RNN and generally it takes longer to train these models. Therefore, another alternative is explored for this project.

## 1.2 TCN

Temporal Convolutional Network (TCN) is a novel Convolutional Neural Network (CNN) modified to work on temporal domain introduced in early 2018. It uses a sequence of dilated convolutional layers, normalization layers, dropout layers and ReLU activations. From the authors[2] of TCN, their experimental results show TCN outperforms simple RNN, LSTM and GRU in different sequence modelling tasks significantly. It can also exhibit a longer memory, and has stable gradients.

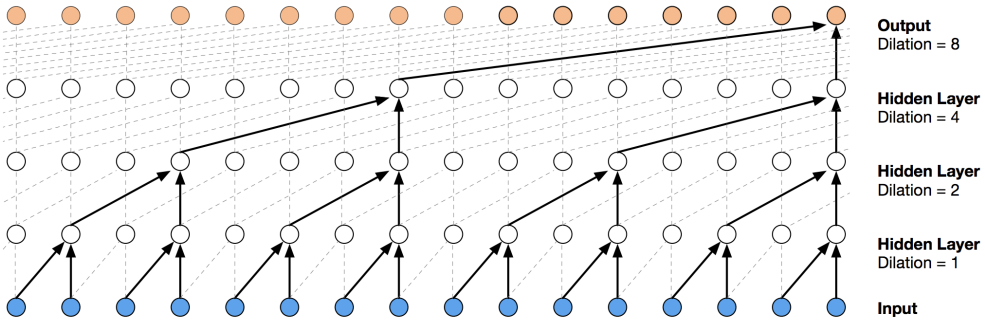


Figure 2: TCN strcuture.[3]

[1] From Stanford cs231n lecture notes.

[2] Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun."An empirical evaluation of generic convolutional and recurrent networks for sequence modeling." arXiv preprint arXiv:1803.01271 (2018).

[3] From `https://github.com/philipperemy/keras-tcn`.

# 2 Experiments

There were two phases of experiments. During the first phase, I used the provided sample code for generating training and validation data; while in the second phase, I reprogrammed these parts. There were in total around 200 sets of training records.

## 2.1 Training with Provided Code

There were multiple types of architectures attempted during this phase. They are constructed by LSTM, and/or TCN layers mentioned in the previous section. All models have a Embedding layer with various intermediate layer(s), then is finally fed into a Dense layer with softmax activation. The models include:

| Model Type | Architecture |
|---|---|
| `lstm` | Embedding → LSTM1 → Dense |
| `lstm2` | Embedding → LSTM1 → LSTM2 → Dense |
| `lstm3` | Embedding → LSTM1 → LSTM2 → LSTM3 → Dense |
| `lstm_embed` | Embedding → LSTM1<br>LSTM1 + Embedding → Dense |
| `tcn` | Embedding → TCN1 → Dense |
| `tcn_embed` | Embedding → TCN1<br>TCN1 + Embedding → Dense |
| `tcn_lstm` | Embedding → LSTM1<br>Embedding → TCN1<br>TCN1 + LSTM1 → Dense |
| `tcn_lstm_embed` | Embedding → LSTM1<br>Embedding → TCN1<br>TCN1 + LSTM1 + Embedding → Dense |
| `tcn_lstm12_embed` | Embedding → LSTM1 → LSTM2<br>Embedding → TCN1<br>TCN1 + LSTM1 + LSTM2 + Embedding → Dense |
| `tcn_lstm2_embed` | Embedding → LSTM1 → LSTM2<br>Embedding → TCN1<br>TCN1 + LSTM2 + Embedding → Dense |
| `tcn12_lstm12_embed` | Embedding → LSTM1 → LSTM2<br>Embedding → TCN1 → TCN2<br>TCN1 + TCN2 + LSTM1 + LSTM2 + Embedding → Dense |

Table 1: List of attempted models.

For parameter tuning, I mainly tweak the embedding size, hidden size (size of $h_t, c_t$ in LSTM and number of kernels used in TCN) and dropout rate. Here some selected results are shown.

| Model Type | Embedding Size | Hidden Size | Dropout Rate | Log Loss | Score |
|---|---|---|---|---|---|
| lstm | 100 | 512 | 0.5 | 1.7566 | 90% |
| lstm | 100 | 512 | 0.2 | 1.7676 | 90% |
| lstm | 100 | 256 | 0.2 | 1.7812 | 90% |
| lstm | 100 | 256 | 0.5 | 1.8009 | 80% |
| lstm | 100 | 128 | 0.2 | 1.8620 | 80% |
| lstm | 100 | 128 | 0.5 | 1.9194 | 60% |
| lstm_embed | 100 | 256 | 0.2 | 1.7413 | 90% |
| lstm_embed | 100 | 128 | 0.2 | 1.7811 | 90% |
| lstm_embed | 100 | 256 | 0.5 | 1.7887 | 80% |
| lstm_embed | 100 | 128 | 0.5 | 1.7980 | 80% |
| lstm2 | 100 | 500 | 0.3 | 1.8401 | 80% |
| lstm2 | 100 | 500 | 0.4 | 1.8668 | 80% |
| lstm2 | 100 | 500 | 0.5 | 1.8830 | 80% |
| lstm2 | 100 | 500 | 0.2 | 1.8861 | 80% |
| lstm2 | 100 | 500 | 0.6 | 1.9672 | 60% |
| lstm3 | 100 | 500 | 0.3 | 1.8723 | 80% |
| lstm3 | 100 | 500 | 0.5 | 1.8924 | 80% |
| lstm3 | 100 | 500 | 0.2 | 1.9035 | 60% |
| lstm3 | 100 | 500 | 0.4 | 1.9099 | 60% |
| lstm3 | 100 | 500 | 0.6 | 1.9784 | 60% |
| tcn | 100 | 512 | 0.5 | 1.8067 | 80% |
| tcn | 100 | 128 | 0.2 | 1.8217 | 80% |
| tcn | 100 | 512 | 0.2 | 1.8362 | 80% |
| tcn | 100 | 256 | 0.2 | 1.8472 | 80% |
| tcn | 100 | 256 | 0.5 | 1.8505 | 80% |
| tcn | 100 | 128 | 0.5 | 1.9228 | 60% |
| tcn_embed | 100 | 256 | 0.2 | 1.8105 | 80% |
| tcn_embed | 100 | 256 | 0.5 | 1.8287 | 80% |
| tcn_embed | 100 | 128 | 0.5 | 1.8428 | 80% |
| tcn_embed | 100 | 128 | 0.2 | 1.8497 | 80% |
| tcn_lstm | 100 | 512 | 0.5 | 1.7346 | 90% |
| tcn_lstm | 100 | 256 | 0.2 | 1.7385 | 90% |
| tcn_lstm | 100 | 256 | 0.5 | 1.7424 | 90% |
| tcn_lstm | 100 | 512 | 0.2 | 1.7569 | 90% |
| tcn_lstm | 100 | 128 | 0.2 | 1.7640 | 90% |
| tcn_lstm | 100 | 128 | 0.5 | 1.7677 | 90% |
| tcn_lstm_embed | 256 | 256 | 0.5 | 1.7036 | 100% |
| tcn_lstm_embed | 100 | 256 | 0.5 | 1.7130 | 100% |
| tcn_lstm_embed | 100 | 512 | 0.5 | 1.7365 | 90% |
| tcn_lstm_embed | 128 | 256 | 0.5 | 1.7439 | 90% |
| tcn_lstm_embed | 100 | 128 | 0.2 | 1.7516 | 90% |
| tcn_lstm_embed | 100 | 256 | 0.2 | 1.7563 | 90% |
| tcn_lstm_embed | 100 | 512 | 0.2 | 1.7861 | 80% |

| Model Type | Embedding Size | Hidden Size | Dropout Rate | Log Loss | Score |
|---|---|---|---|---|---|
| `tcn_lstm_embed` | 100 | 128 | 0.5 | 1.7952 | 80% |
| `tcn_lstm12_embed` | 256 | 256 | 0.2 | 1.6785 | 100% |
| `tcn_lstm12_embed` | 128 | 256 | 0.5 | 1.6844 | 100% |
| `tcn_lstm12_embed` | 256 | 128 | 0.5 | 1.7210 | 100% |
| `tcn_lstm12_embed` | 256 | 256 | 0.5 | 1.7244 | 100% |
| `tcn_lstm12_embed` | 128 | 128 | 0.2 | 1.7262 | 100% |
| `tcn_lstm12_embed` | 128 | 128 | 0.5 | 1.7305 | 100% |
| `tcn_lstm12_embed` | 128 | 256 | 0.2 | 1.7557 | 90% |
| `tcn_lstm12_embed` | 256 | 128 | 0.2 | 1.7659 | 90% |
| `tcn_lstm2_embed` | 256 | 256 | 0.5 | 1.7322 | 100% |
| `tcn_lstm2_embed` | 128 | 256 | 0.5 | 1.7373 | 90% |
| `tcn_lstm2_embed` | 256 | 128 | 0.5 | 1.7405 | 90% |
| `tcn_lstm2_embed` | 128 | 128 | 0.2 | 1.7614 | 90% |
| `tcn_lstm2_embed` | 256 | 128 | 0.2 | 1.7726 | 90% |
| `tcn_lstm2_embed` | 128 | 256 | 0.2 | 1.7785 | 90% |
| `tcn_lstm2_embed` | 256 | 256 | 0.2 | 1.7840 | 80% |
| `tcn_lstm2_embed` | 128 | 128 | 0.5 | 1.7926 | 80% |
| `tcn12_lstm12_embed` | 128 | 256 | 0.5 | 1.7154 | 100% |
| `tcn12_lstm12_embed` | 128 | 256 | 0.2 | 1.7304 | 100% |
| `tcn12_lstm12_embed` | 128 | 128 | 0.2 | 1.7561 | 90% |
| `tcn12_lstm12_embed` | 256 | 256 | 0.5 | 1.6899 | 100% |
| `tcn12_lstm12_embed` | 256 | 128 | 0.2 | 1.744 | 90% |
| `tcn12_lstm12_embed` | 256 | 256 | 0.2 | 1.7633 | 90% |
| `tcn12_lstm12_embed` | 256 | 128 | 0.5 | 1.7322 | 100% |
| `tcn12_lstm12_embed` | 128 | 128 | 0.5 | 1.749 | 90% |

Table 2: Selected results trained with skeleton code. All models listed are trained with 25 epochs (may be early stopped, usually stops within 10 epochs) and batch size 32.

Some results we can see from Table 2 are as follows.

- For all pure LSTM models, the more the layers, the worse it generally performs.
- Pure TCN models outperform most of the LSTM models (except a few `lstm` ones).
- Pure TCN/LSTM models with concatenating Embedding generally performs better than those without. This is similar to what we see with ResNet.
- Concatenating all TCN, LSTM and Embedding works the best among all models. `tcn_lstm_embed` hits 100% score constantly.

Even I have reached 100% multiple times, I still wish to achieve the bonus score for validation set. Hence, I started to modify the provided skeleton code for generating data.

## 2.2 Training with Altered Code

After reading through the skeleton code, I believe this is how it works for sampling training/validation data.

1. Converts all words into IDs.
2. Fixes the input temporal dimension as 10.
3. Concatenates next sentences to fill in the remaining space if current sentence is too short, or break the current sentence into two if it is too long.

One of the advantages of RNN and TCN is they support varying temporal dimension. With step 3 above, each input sentence is longer a complete sentence since either some of them are truncated, or some unrelated sentences are mixed in with them. This could affect the performance of the model since the models are looking for contexts through the whole sentence, which now the words can be unrelated. Hence, I modified the code and now it works as follows.

1. Converts all words into IDs.
2. Gets a batch of sentences and pad 0 to shorter sentences such that all sentences in the batch has the same temporal dimension during training.

After the modification, the performance of the models improved by a lot. This implementation is not the best though, because ID 0 is given to word `L0127`, meaning in the perspective of the model, those words become `L0127`. However, this is still acceptable because the model eventually learns to neglect this word towards the end of a sentence. From the table of selected results below, we can see an improvement in performance comparing with the previous table.

| Model Type | Log Loss | Score |
|---|---|---|
| `tcn_lstm_embed` | 1.6699 | Bonus |
| `tcn_embed` | 1.7355 | 90% |
| `lstm_embed` | 1.7165 | 100% |
| `tcn_lstm12_embed` | 1.6131 | Bonus |
| `tcn12_lstm12_embed` | 1.6053 | Bonus |

Table 3: Selected empirical results trained with altered code. All models listed are trained with 1 epoch, batch size 32, embedding size 128, hidden size 256 and dropout rate 0.2.

| Model Type | Embedding Size | Hidden Size | Dropout Rate | Log Loss | Score |
|---|---|---|---|---|---|
| `lstm_embed` | 256 | 256 | 0.2 | 1.4165 | Bonus |
| `lstm_embed` | 256 | 128 | 0.2 | 1.4395 | Bonus |
| `lstm_embed` | 128 | 128 | 0.2 | 1.4457 | Bonus |
| `lstm_embed` | 128 | 256 | 0.2 | 1.422 | Bonus |
| `lstm_embed` | 128 | 128 | 0.5 | 1.4729 | Bonus |
| `lstm_embed` | 256 | 128 | 0.5 | 1.4641 | Bonus |
| `lstm_embed` | 256 | 256 | 0.5 | 1.4173 | Bonus |
| `lstm_embed` | 128 | 256 | 0.5 | 1.4414 | Bonus |
| `tcn_embed` | 256 | 256 | 0.2 | 1.5162 | Bonus |

| Model Type | Embedding Size | Hidden Size | Dropout Rate | Log Loss | Score |
|---|---|---|---|---|---|
| tcn_embed | 128 | 256 | 0.2 | 1.5249 | Bonus |
| tcn_embed | 256 | 128 | 0.2 | 1.5414 | Bonus |
| tcn_embed | 256 | 128 | 0.5 | 1.5772 | Bonus |
| tcn_embed | 128 | 128 | 0.2 | 1.5507 | Bonus |
| tcn_embed | 128 | 128 | 0.5 | 1.5785 | Bonus |
| tcn_embed | 128 | 256 | 0.5 | 1.5334 | Bonus |
| tcn_embed | 256 | 256 | 0.5 | 1.5479 | Bonus |
| tcn_lstm_embed | 128 | 128 | 0.5 | 1.4297 | Bonus |
| tcn_lstm_embed | 128 | 128 | 0.2 | 1.4231 | Bonus |
| tcn_lstm_embed | 128 | 256 | 0.5 | 1.4296 | Bonus |
| tcn_lstm_embed | 256 | 128 | 0.2 | 1.426 | Bonus |
| tcn_lstm_embed | 256 | 256 | 0.2 | 1.4218 | Bonus |
| tcn_lstm_embed | 128 | 256 | 0.2 | 1.4596 | Bonus |
| tcn_lstm_embed | 256 | 128 | 0.5 | 1.4223 | Bonus |
| tcn_lstm_embed | 256 | 256 | 0.5 | 1.4027 | Bonus |
| tcn_lstm12_embed | 256 | 128 | 0.2 | 1.4032 | Bonus |
| tcn_lstm12_embed | 128 | 128 | 0.2 | 1.4325 | Bonus |
| tcn_lstm12_embed | 256 | 256 | 0.2 | 1.3916 | Bonus |
| tcn_lstm12_embed | 128 | 256 | 0.2 | 1.4308 | Bonus |
| tcn_lstm12_embed | 256 | 128 | 0.5 | 1.4152 | Bonus |
| tcn_lstm12_embed | 128 | 256 | 0.5 | 1.4038 | Bonus |
| tcn_lstm12_embed | 256 | 256 | 0.5 | 1.3968 | Bonus |
| tcn_lstm12_embed | 128 | 128 | 0.5 | 1.4262 | Bonus |
| tcn12_lstm12_embed | 256 | 128 | 0.2 | 1.4235 | Bonus |
| tcn12_lstm12_embed | 128 | 128 | 0.2 | 1.4557 | Bonus |
| tcn12_lstm12_embed | 128 | 256 | 0.2 | 1.4447 | Bonus |
| tcn12_lstm12_embed | 256 | 256 | 0.2 | 1.4123 | Bonus |
| tcn12_lstm12_embed | 256 | 128 | 0.5 | 1.431 | Bonus |
| tcn12_lstm12_embed | 128 | 256 | 0.5 | 1.4218 | Bonus |
| tcn12_lstm12_embed | 128 | 128 | 0.5 | 1.4229 | Bonus |
| tcn12_lstm12_embed | 256 | 256 | 0.5 | 1.3963 | Bonus |

Table 4: Selected results trained with altered code. All models listed are trained with 100 epochs (may be early stopped, usaully stops within 10 epochs) and batch size 32.

From Tables 3 and 4, we may find these observations.

- All architectures performed significantly better even when trained with 1 epoch only. Some of them even got to bonus already.
- `lstm_embed` outperforms `tcn_embed` despite it is shown TCN are better. This could be because our task does not require a long memory so the advantages of TCN are somewhat lost.
- Combining LSTM and TCN together are better than pure LSTM/TCN.

- The best model seems to be `tcn_lstm12_embed` with embedding size 256, hidden size 256 and dropout 0.2. It would be best to check this by running the same model setups multiple times, but this is not performed due to the lack of time.

After these evaluations, `tcn_lstm12_embed` with embedding size 256, hidden size 256 and dropout 0.2 model setup is used for the inference of the test data.

# 3 Implementations

In this project, the following libraries are used.

| Name | Version | Descriptions |
|---|---|---|
| `Python` | `3.6.6` | Language used |
| `numpy` | `1.15.3` | For handling data matrices |
| `pandas` | `0.23.4` | For handling `csv` files |
| `tensorflow` | `1.12.0` | Deep learning library |
| `keras` | `2.2.4` | High level API wrapper for `tensorflow` |
| `keras-tcn` | commit `6d71e19` | `keras` implementation of TCN |

Some notes for in the implementations are:

- `keras.callbacks.EarlyStopping` is used to stop training when the validation loss does not increase for 3 consecutive epochs.
- `keras.callbacks.TensorBoard` is used to visualize the models during implementation.
- `Python` generators are used so data do not need to be fetched to memory before training. This speeds up the training time. In particular, `model.fit_generator` is used instead of `model.fit`. Unfortunately, `model.predict_generator` does not currently support varying dimension inference so `model.predict` is still used.

**End of Report**