Project 3: Language Model

Seunghwan Cha (20155502)

December 15, 2018

Abstract

Language modelling is a task to generate a text given source text. For this particular assignment, we were given a sentence as training data and the last token of that sentence as the label. This report outlines some of the approaches I have taken to find the optimal architecture for this task. I used 4 GRU layers, Layer Normalization, Dropout, and weight tying to achieve the best result. Furthermore, I have used advanced techniques such as Early Stopping and Cyclic Learning rates to land in the optimal minima point during training.

1 Introduction

The task for this project is to predict last token given the previous tokens. We were asked to use language models we have learned during class (preferrably nueral language models). The structure of the template code is from lab 8 which first introduced language modelling using Recurrent Neural Network (RNN).

The major challenge for this project was the use of encrypted training data. Because the word tokens in the data are not actual words, I couldn't use any pretrained word embeddings such as GloVe or pretrained models to boost the performance of my model. As a result, the major objective for me was to build an efficient architecture without the help of any pretrained embeddings.

2 Theory

2.1 LSTM

As a way to solve vanishing gradient issue in RNN, Long Short-Term Memory [1] uses 4 different gates as illustrated in Figure 1. As we learned in class, this

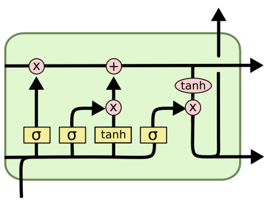
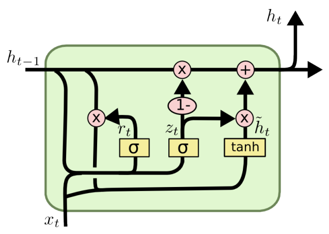


Figure 1: Diagram of LSTM Cell



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2: Diagram of GRU Cell

is the most widely used variants of RNN that could alleviate the long term dependency problem. LSTM was extensively studied during lab 8 and is the base structure of the example code.

2.2 GRU

Cho et al. proposed another variant of RNN called Gated Recurrent Unit (GRU) [2]. As shown in Figure 2, GRU removes memory unit in LSTM and reduces the number of parameters greatly. It is known that the performance of GRU is similar to that of LSTM but GRU has less number of parameters. In my final model, I stacked several layers of GRU to achieve the best perplexity.

2.3 QRNN

Team from Salesforce introduced a state of the art model called Quasi-Recurrent Neural Network (QRNN) [3]. As shown in Figure 3, QRNNs address both drawbacks of standard models: CNN and RNN. Like CNNs, QRNNs allow for parallel computation across both timestep and minibatch dimensions, enabling high throughput and good scaling to long sequences. Like RNNs, QRNNs allow the output to depend on the overall order of elements in the sequence. The result shows that the training time for QRNN is greatly faster compared to that of LSTM models as parallel computation is possible for QRNN. However, I tried running QRNN by implementing a custom layer in qrnn.py but it didn't have good result. This was mainly because there are so many hyperparameters to choose from. As the default hyperparameters are set for standard dataset such as Penn Tree Bank dataset, those values may not work well for our dataset.

2.4 Weight Tying

Weight Tying is a popular techniques used for language modeling [4]. The approach is fairly simply by just tying the weights of input word embedding (Embedding layer in Keras) and output embedding (last Dense layer inside TimeDistributed). This is beneficial because it reduces the parameter size greatly by removing the trainable weights of the last fully connected layer. Although the algorithm is simple, the implantation in Keras was not trivial. I had to replace original Dense layer with a newly defined custom layer called DenseTransposeTied which doesn't initialize its kernel/weight like the original Dense layer. One problem is that Keras doesn't allow me load the weights again

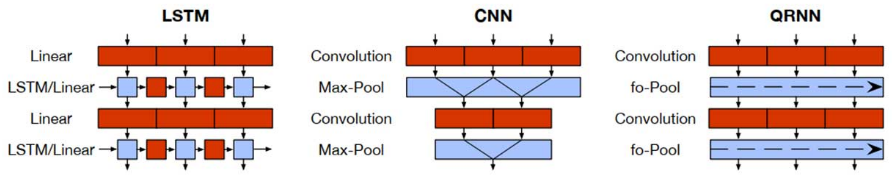


Figure 3: Block diagrams showing the computation structure of the QRNN compared with typical LSTM and CNN architectures. Red signifies convolutions or matrix multiplications; a continuous block means that those computations can proceed in parallel. Blue signifies parameter less functions that operate in parallel along the channel/feature dimension. LSTMs can be factored into (red) linear blocks and (blue) elementwise blocks, but computation at each timestep still depends on the results from the previous timestep.

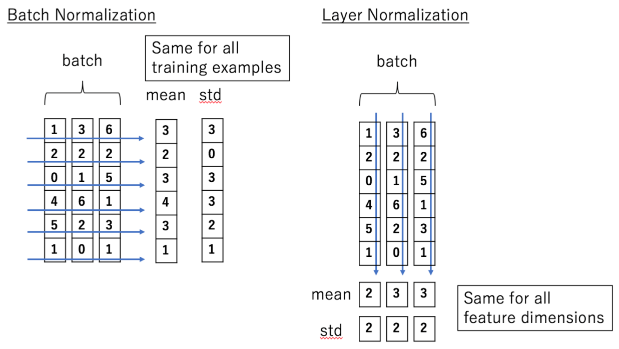


Figure 4: Comparison of Batch Normalization and Layer Normalization

once I save the trained model. As a result, I had to modify the code so that I run the inference right away after training.

2.5 Layer Normalization

Layer Normalization [5] was proposed by Geoffery Hinton in 2016. This paper focused on introducing variant of Batch Normalization which is widely used in vision task. Normalization acts as a powerful regularization that could reduce overfitting. Nonetheless, Batch Normalization is greatly dependent on mini-batch size and is known to be difficult to applied in recurrent connections of RNN. In contrast, Layer Normalization could be applied to RNN and is known to be successful in many sequence to sequence tasks. Figure 4 illustrates the difference between Batch Normalization and Layer Normalization. As a result, I inserted Layer Normalization after every GRU layer.

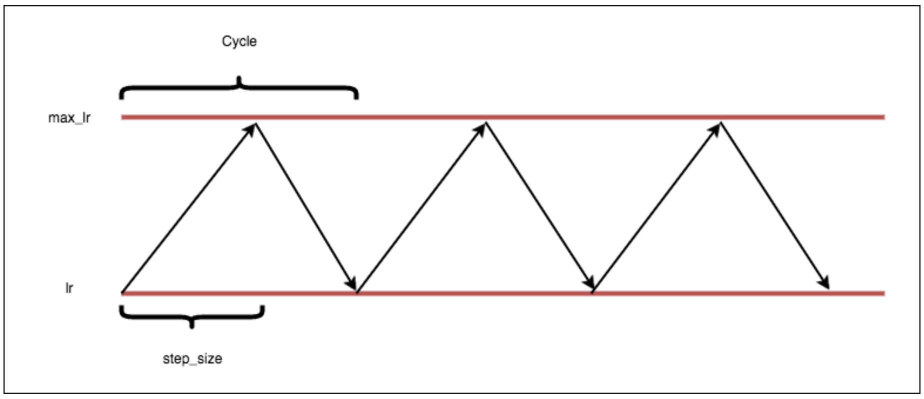


Figure 5: Triangular Learning Rate Decay.

3 Parameter Tuning

3.1 Experimental set-up

The provided code conducts hyperparameter tuning using valid.csv through TestCallback class in model.fit(). However, this is not an accurate way to do hyperparameter tuning as our only way of checking the generalization ability of our language model is through getting score from valid.csv. As a result, I prepared validation set from train.csv and used valid.csv as my testset. For my case, I used 9:1 ratio for train-validation split which was determined empirically. Hopefully, the model performing well on valid.csv should be able to perform equally well on test.csv that doesn't have the label.

I have used Keras as my main framework as most of the helper functions and callbacks were designed for Keras. Although Keras is a great framework to easily build neural networks through functional API, it is quite cumbersome to make modifications to layers. To do so, I had to define a custom layer which requires me to look into the source code of Keras to make it compatible with the original implementation.

3.2 Different Learning Rate Decay Methods

- **Cyclic Learning Rate [6]:** Learning could be often stuck at local minima if we just use regular learning rate decay method. As shown in 5, Cyclic Learning Rate regularly bumps up the learning rate so that the gradient could escape the local minima to search for better minima. I've implemented a custom callback function in clr_callback .py but it didn't perform very well.

- **ReduceLROnPlateau:** This is one of the callback function in Keras that reduces learning rate once the convergence rate in on plateau. This is known to be effective in helping gradient to approach the minima more slowly with reduced learning rate. Nonetheless, this technique was as effective as other approaches.

3.3 Early Stopping and Model Checkpoint

In order to ensure that I find the optimal hyperparameter for my model, I used early stopping and model checkpoint callback functions provided in Keras. Early stopping ends the training whenever the validation loss starts to increase. I set the patience to 2 which means that the callback will wait for 2 epochs until the ending the training process. Furthermore, I've used Model Checkpoint so that I save best model at every epoch. These two techniques greatly improved the efficiency of parameter searching.

3.4 Final Model Architecture

After several parameter searching through the aforementioned techniques and layers, I have found the best model that could exceed the 100 point baseline using validation data. I've used GRU - LayerNorm - Dropout block four times. I've added tied weight Dense Layer in the end. Both the embdding size and hidden size were 500 and dropout rate was 0.2. The training was terminated at around 18 epochs through EarlyStopping callbacks once the validation loss began to increase for 2 consecutive epochs.

4 Results and running the code

Before running the code, you need to install all the python dependencies through running pip install requirements.txt which will do the installation for you. I was able to train the model through Paperspace with 4 NVIDIA 1080Ti using Keras multi_gpu_model(). You may use single GPU to train the model by changing the number of GPU in the multi_gpu_model() function. The final perplexity on the provided validation set was **1.7244**.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[2] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[3] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *CoRR*, abs/1611.01576, 2016.

[4] Ofir Press and Lior Wolf. Using the output embedding to improve language models. *CoRR*, abs/1608.05859, 2016.

[5] Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[6] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.