Project Report

Project Title: COMP 4901K Project3 — — Language Model

Name: CHANG Yingshan

Student ID: 20413368

Content

Abstract

This project aims at building a neural network language model, which, given the previous words in a sentence, is capable of predicting the last word. In this report, I am going to discuss three main aspects: tricks on hyperparameter tuning, implementation of neural network language model with a new model architecture, and further model variations with insightful findings.

Tricks on hyperparameter tuning

The original network implemented by two LSTM layers in the skeleton code already has really good performance. At the very beginning, I tried to boost performance mainly by tuning hyperparameters. During this process, I summarized a few tips for hyperparameter tuning, which also apply to other models with different architecture.

1. Hidden size

Large hidden size means large number of parameters. The original hidden size is 500, which is prone to overfitting especially when I increase the number of epochs. I reduced hidden size while keeping other hyperparameters fixed (embedding = 100, epochs = 3, dropout = 0.3) and the results are as follows.

	Embedding_size=100 Hidden_size = 500 Dropout = 0.3 Epochs = 3	Embedding_size=100 Hidden_size = 400 Dropout = 0.3 Epochs = 3	Embedding_size=100 Hidden_size = 300 Dropout = 0.3 Epochs = 3
Train_loss	2.6101	2.6497	2.7081
Val_loss	2.6163	2.6432	2.6790
Perplexity	13.70	14.07	14.58
Score on validation set	1.9500	2.0291	2.1114

When the network is less “powerful”, increasing hidden size is conducive to training efficiency and performance, since larger hidden size makes it easier to capture and emulate the relationship between input and output.

However, when a relatively “powerful” model is used on this small training set, overfitting becomes a knotty problem. Larger hidden size would make the model prone to fitting the characteristics of the training data so well that the model is not generalizable. For instance, when I replaced the first LSTM layer by Bi-LSTM layer, there was serious overfitting. From the table below, it is clear that small hidden size

has greater effect of preventing overfitting.

	Embedding_size=200 Hidden_size = 200 Dropout = 0.8 Epochs = 1	Embedding_size=200 Hidden_size = 100 Dropout = 0.8 Epochs = 1	Embedding_size=200 Hidden_size = 80 Dropout = 0.8 Epochs = 1
Train_loss	3.9040	4.4706	4.6503
Val_loss	2.4127	3.4019	3.7546
Perplexity	11.16	30.02	42.72
Score on validation set	8.7394	6.5337	6.1802

2. Embedding size

The embedding layer transforms each word representation from one-hot vector to a dense vector of fixed length specified by embedding size. I do not provide pre-trained word2vec so the vectors of each embedding get updated while training the whole network. In a word, embedding size controls how long a vector we want to use to represent a word. Noted that the embedding size in skeleton code is 100, which is a little bit small, I tried to increase embedding size while keeping all other hyperparameters fixed to see if it helps. The results are as follows.

	Embedding_size=100 Hidden_size = 300 Dropout = 0.3 Epochs = 5	Embedding_size=200 Hidden_size = 300 Dropout = 0.3 Epochs = 5	Embedding_size=300 Hidden_size = 300 Dropout = 0.3 Epochs = 5
Train_loss	2.7081	2.6785	2.6261
Val_loss	2.6790	2.6575	2.6160
Perplexity	14.58	14.27	13.70
Score on validation set	2.1114	2.0606	2.0123

It can be concluded from the table above that increasing embedding size helps improve performance. When we use a longer vector to represent each work, it is natural that words related to the same context can be more easily distinguished from words related to other contexts.

On the other hand, embedding size cannot be too large. Increasing embedding size also means more parameters to train and slower computing speed. In the extreme case, when the embedding size is almost equal to the vocabulary size, all word vectors become sparse again and this goes against our original intention to use word embedding. During my later model building and hyperparameter tuning work, I used embedding_size = 200 for most of cases.

3. Regularizer

Large weights are the most obvious symptom of overfitting. Regularization penalizes weights with large magnitudes, which prevents gradient exploding and overfitting. This is especially helpful when number of epochs is large. I trained the neural network with one Bi-LSTM layer with kernel_regularizer_l2 (*keras.regularizers.Regularizer* [1]) = 0, 5e-6 and 10e-6, respectively. The results are summarized below, which indicates that kernel regularizer has a little effect in preventing overfitting.

	Embedding_size=200 Hidden_size = 100 Dropout = 0.8 Epochs = 1 Kernel_regularizer = 0	Embedding_size=200 Hidden_size = 100 Dropout = 0.8 Epochs = 1 Kernel_regularizer = 5e-6	Embedding_size=200 Hidden_size = 100 Dropout = 0.8 Epochs = 1 Kernel_regularizer = 8e-6
Train_loss	2.8980	2.9117	2.9135
Val_loss	1.8987	1.9732	1.9712
Perplexity	6.68	7.18	7.15
Score on validation set	4.8614	4.8562	4.8135

New Model Architecture

1. Embedding layer

The embedding layer learns dense vector representation of each word. After the training of the whole network, the resulting embedding vectors would be capable of capturing context of a word in a document as well as semantic and syntactic similarity. It nearly has become the standard of every neural network in NLP. I set the embedding layer trainable so that it is trained with the whole network. I do not need to provide pre-trained weights as input.

2. Dense layer

I directly add a dense layer (fully connected) after the embedding layer. It learns features from all the combinations of the features in the previous layer. Of course, due to a large number of parameters required by the fully connected layer, it is computationally expensive. But it is worthwhile since there is perceptible improvement.

3. Residual Network

This is the core of my model so I am going to talk into details.

- **Introduction**

Deep residual network first took the deep learning word by storm when Microsoft Research released *Deep Residual Learning for Image Recognition* [2]. This led to 1st-place winning in all five tracks of the ImageNet and COCO 2015 competitions, which covered image classification, object detection and semantic segmentation. The robustness of ResNet has since been proven by various deep learning tasks in other domains including speech and language [3].

- **Why residual network**

When it comes to neural network design, the trend in the past few years pointed in a single direction: deeper. For many applications, the deeper the neural network, the better the performance, provided that they can be properly trained. The first intuition when designing a deep network may be to simply stack many of the typical building blocks such as convolutional layer, LSTM, GRU or fully-connected layer together. This works, but performance quickly diminishes as the depth of the neural network grows. This issue arises from the way in which weights in the network are trained through backpropagation. The gradient of a single weight must be propagated backwards from the very top all the way down to the bottom. With the traditional network, this gradient becomes slightly diminished as it is passed through each layer [4]. So the problem is to design a network in which the gradients would more easily reach all the layers.

- **How does ResNet work**

A residual network solves the problem of vanishing gradients by connecting (concatenate) the output of previous layers to the output of new layers. For instance, in a traditional network, the activation layer is defined as follows:

$$y = f(x)$$

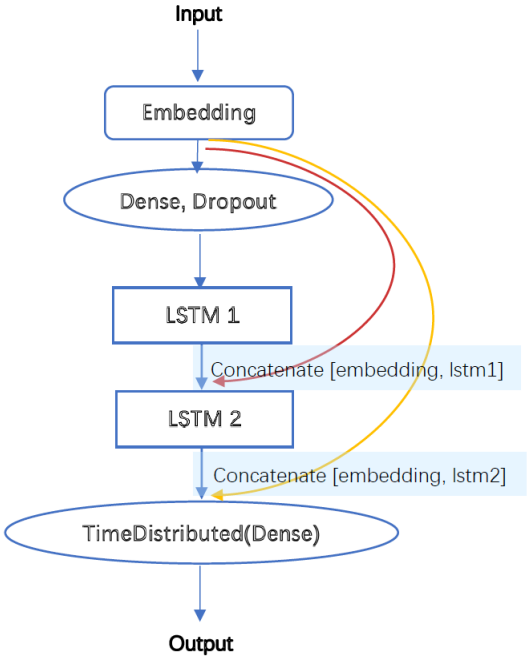
f is usually a nonlinear function. When the signal is sent backwards, the gradient must be always passed through f, which can cause trouble due to the nonlinearities. Instead, at each layer the ResNet implements:

$$y = f(x) + x$$

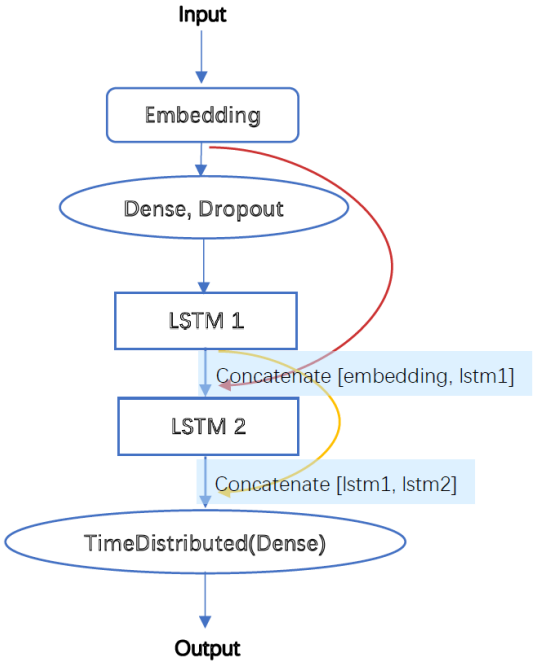
The "+x" at the end is a "shortcut", it allows the gradient to pass backwards directly [4]. This simplifies the optimization of the network at no cost in a sense that if one layer is "useless", the next layer at least receives the output of the previous layer which is not passed through the problematic layer. Thus, each block is responsible for fine-tuning the output of a previous block, instead of having to generate the desired output from scratch.

- **Performances**

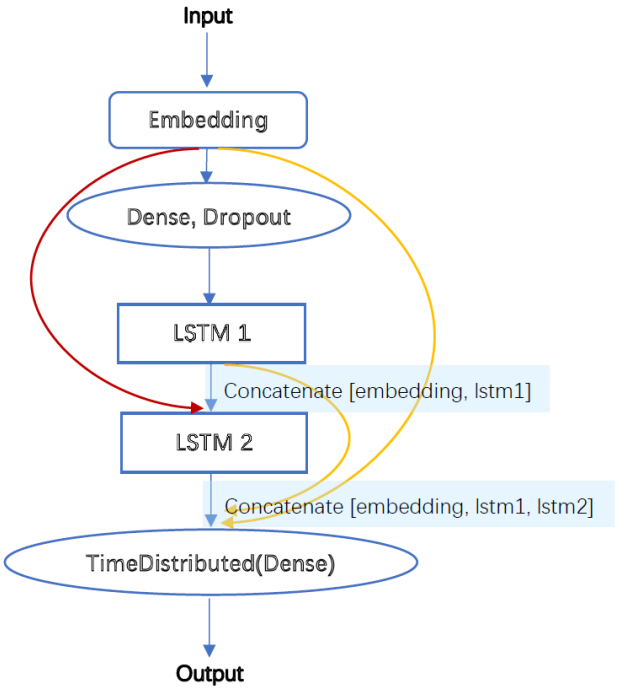
I test the performances using three model architectures with different "shortcut" implementation. The model architectures and corresponding results are shown in the following graphs.



Hyperparameters	Embedding_size = 200 Hidden_size = 400 Kernel_regularizer = 8e-6 Dropout = 0.3 epochs = 6
Train_loss	2.4549
Val_loss	2.5245
Perplexity	12.05
Score on validation set	1.8634



Hyperparameters	Embedding_size = 200 Hidden_size = 400 Kernel_regularizer = 8e-6 Dropout = 0.3 epochs = 6
Train_loss	2.4390
Val_loss	2.5371
Perplexity	12.04
Score on validation set	1.7378



Hyperparameters	Embedding_size = 200 Hidden_size = 400 Kernel_regularizer = 8e-6 Dropout = 0.3 epochs = 6
Train_loss	2.3947
Val_loss	2.5053
Perplexity	11.74
Score on validation set	1.6662

4. TimeDistributed layer

TimeDistributed layer is used on LSTM to keep one-to-one relations on input and output [5]. It is wrapped around a dense layer after the second LSTM. In my implementation, I did not modify it and this layer just remained the same as skeleton code.

Further Discussion and Finding

1. Bidirectional LSTM

- **Introduction**

In language modeling of predicting a word in a sentence, sometimes we would like to use both left and right context to see how well the word fits in the sentence. Previously I thought bidirectional LSTM might work better than unidirectional LSTM. This can easily be done by wrapping a "Bidirectional" outside LSTM layer. It will return two sets of hidden vectors where one is the output of forward LSTM and the other is the output of backward LSTM. The prediction of the target word will be based on the combination of two sets of hidden vectors.

- **Implementation**

```
lstm_out_1 = Bidirectional(LSTM(units=hidden_size,
                                dropout = drop,
                                recurrent_dropout = drop,
                                kernel_regularizer=keras.regularizers.l2(8e-6),
                                return_sequences=True))(drop_1)
```

- **Result**

Hyperparameters	Embedding_size = 200 Hidden_size = 100 Kernel_regularizer = 8e-6 Dropout = 0.85	Embedding_size = 200 Hidden_size = 100 Kernel_regularizer = 8e-6 Dropout = 0.85 Epochs = 1
Train_loss	2.6003	3.1644
Val_loss	1.3039	2.3187
Perplexity	3.64	10.14
Score on validation set	3.6964	4.1713

- **Explanation**

It is discovered that bidirectional LSTM is too easy to overfit. In order to prevent overfitting, I reduced hidden size to only 100 and raised dropout rate to 0.85, but the result is still not satisfactory. It is also notable that when the bidirectional LSTM achieved its minimum validation loss, the training loss is already higher than the 60 baseline. The training loss can reach 0.5 by increasing hidden size or reduce dropout rate, but the validation loss will shoot up without going further down.

This issue may due to the small size of training set. There is only 90K records in the training set. So it might by easy for Bi-LSTM to be trained to completely fit the training data.

2. Self-Attention

- **Introduction**

The essence of Attention Mechanism is that they are an imitation of the human sight

mechanism. When the human sight mechanism detects an item, it will not scan the entire scene end to end. Instead, it will always selectively focus on parts of the visual space to acquire information according to personal needs. Similarly, in natural language, the occurrence of each word in a sentence will only depend on certain parts of the previous context. In Self-Attention, before outputting the predicted target word, each word in the context needs to undergo attention computation. The goal is to learn the dependencies between the target word and its previous context [6]. The internal structure of the sentence can also be captured using that information.

- **Implementation**

Keras has a package (*keras-self-attention* [7]) for implementing Self Attention. It supports multiple attention types and the one I used is multiplicative attention.

```
att = SeqSelfAttention(attention_width=10,
                       attention_type=SeqSelfAttention.ATTENTION_TYPE_MUL,
                       attention_activation='tanh',
                       kernel_regularizer=keras.regularizers.l2(8e-6),
                       use_attention_bias=False,
                       name='Attention')(lstm_out_1)
```

- **Result**

Hyperparameters	Embedding_size = 200 Hidden_size = 100 Kernel_regularizer = 8e-6 Dropout = 0.5 Epochs = 1
Train_loss	2.6003
Val_loss	1.3039
Perplexity	3.64
Score on validation set	3.6964

- **Explanation**

Just like Bi-LSTM, it again seems to overfit a lot. After adding the self-attention layer, the training loss decreased dramatically while the validation loss rocketed up. This may also result from relatively small size of training set.

On the other hand, in the data preprocessing stage, we divided the corpus into sentences with fixed length 10. Attention is particularly useful when we need to capture dependencies across long distance. However, when sequence length is 10, traditional RNN may have already captured the underlying relationship between words quite well.

3. Comparison of LSTM and GRU

From implementation perspective, there are two vital differences between LSTM and GRU. Firstly, GRU has 2 gates: update gate and reset gate, while LSTM has three gates: input gate, forget gate and output gate. Secondly, LSTM uses another memory unit to memorize the cell state (different from hidden state) and uses output gate to

control how much information is going to be exposed to hidden state, while GRU just exposes the full hidden content without any control.
In terms of performance, they are almost comparable to each other, but GRU is computationally more efficient since less complex structure means less parameters and less calculation. For this particular project, GRU is indeed faster than LSTM, but with a slight performance degradation.

	LSTM	GRU
Hyperparameters	Embedding_size = 200 Hidden_size = 400 Kernel_regularizer = 8e-6 Dropout = 0.3	Embedding_size = 200 Hidden_size = 400 Kernel_regularizer = 8e-6 Dropout = 0.3
Train_loss	2.3947	2.4412
Val_loss	2.5053	2.5488
Perplexity	11.74	12.05
Score on validation set	1.6662	1.7105

4. CNN

- **Introduction**

The input of cnn layer is the (dense) word embedding layer, where each row represents one word. Unlike image processing where the filters slide over the matrix both horizontally and vertically, in NLP the filters usually have the same width as the input matrix, so it can only slide vertically. The filter "size" refers to the length of the filter, which decides how many words should be involved in one convolution. After the convolution layer, 1-max pooling is used to extract the most significant feature. Therefore, the output length is fixed regardless of filter sizes and sequence length. The output of CNN is considered as part of the inputs to the LSTM later.

- **Implementation**

Since one filter only captures one feature, it is better to have multiple filters with multiple sizes. In my implementation, I used 100 filters in total, among which 40 filters have length 2, 30 filters have length 3, 20 filters have length 4 and 10 filters have length 5.

```
convs = []
filter_sizes = [2,2,2,2,3,3,3,4,4,5]
for f in filter_sizes:
      conv = Conv1D(filters=10, kernel_size = f, activation = 'tanh')(embedding)
      pool = MaxPooling1D(sequence_length-f+1)(conv)
      #flatten = Flatten()(pool)
      convs.append(pool)
merge = concatenate(convs, axis = 1)
conv_out = Dropout(0.5)(merge)
```

- **Result**

Hyperparameters	Embedding_size = 300

	Hidden_size = 300 Dropout = 0.3 epochs = 3
Train_loss	2.0714
Val_loss	1.8921
Perplexity	6.39
Score on validation set	1.8486

- **Explanation**

The performance is not improved by adding convolutional layer. Perhaps this is due to the fact that CNN does not care about the order of words. Intuitively, RNN makes more sense because they resemble how we process language: reading sequentially from left to right. However, CNN fails to process sentences in this way. It detects features without caring about the position of them. However, in this task, when we want to predict the next word, the order of words in the previous context matters much. What's more, the unsatisfactory performance of CNN may also be attributed to the use of 1-max pooling. When only the most significant feature is extracted, it is clear that many other possibly useful information is lost.

References

[1] https://keras.io/regularizers/

[2] https://arxiv.org/pdf/1512.03385.pdf

[3] https://blog.waya.ai/deep-residual-learning-9610bb62c355

[4] https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32

[5] https://www.quora.com/What-is-time-distributed-dense-layer-in-Keras

[6] https://dzone.com/articles/self-attention-mechanisms-in-natural-language-proc

[7] https://pypi.org/project/keras-self-attention/