Works based on NNLM

Pingwei Sun

NEU-NLP Lab

Abstract

During the second stage of our course, I have done work based on Neural Network Language Model. Both a FFN and a RNN are built up. As for further research, I have done some related works based on those models, which includes experiments on hyperparameters, comparing the performance of various models, introducing grad-clip and pre-train method. The details of the works are presented as follows.

1 Introduction

With the improvement of computer processing ability, people have been exploring the method of processing natural language by programs. One of the most important tasks is to create a language model, which can effectively generate a mathematical model according to our natural language.

The natural language model has gone through stages of the grammatical model, the statistical model and now stepped into the neural network stage.

Different from the previous dictionary-based theory, **NNLM**(Bengio et al., 2000) will transform natural language into a set of machine knowledge systems by learning from data. There are no artificial assumptions during the whole process and will also save a lot of storage costs.

However, it did not consider the sequential information of language. To make models able to form memory, **RNN**(Mikolov) was launched representing history by neurons with recurrent connections.

Based on NNLM, I have conducted the following experiments:

- Explore the impact of hyperparameters(step, embedded size, hidden size) on the performance of the model.
- Compare performances of different models.
- Introduce pretrained embedding parameters into the model.

2 Methods

2.1 Vocabulary Construction

Here I customize the method of constructing the vocabulary by myself. First, a document is opened by the program. Then the words are separated by space(ways of splitting can be changed by calling various APIs) and stored in a *frequency dict* in the pattern of {*word:frequency*}.

Then we need to build two *dicts*, *word2vecotr and vector2word*, to generate a dual-direct index. Special words such as *[pad], [unk], [sos], [eos]* come first, and dual-direct index pairs are built according to the *frequency dict*.

2.2 Dataset Construction

To feed the model with appropriate data, I rewrite the *Dataset* method. Specifically, the input corpus is translated into numbers according to the previous dual index sentence by sentence. Then, *[[n-input], [target]* corpus list are generated from sentences, following the instruction of *n-step*, and those being unqualified in length will be padded in front. With the customized collate-fn, all records in the corpus list are packaged as tuples, gained by *Dataloader* and finally make up batches.

2.3 Pretrain word embedding

Pretraining word embedding (Mikolov et al., 2013) is used in my program. With the help of *gensim*, I gained a word-to-vector list based on the whole *PTB* data set, which can be used to initialize the embedding layers in models. A test has also been done to see how it works.

2.4 Facilitation for experiments

To make it more convenient to conduct my test, I use *argparse API* to change and save the settings of hyperparameters. In addition, *tensorboard* is called to visualize the training process.

Parameters		Value/ppl				
n-step	Value	3	5	7	9	10
	ppl	1.010	1.011	1.015	1.021	1.024
Hidden size	Value	64	128	256	512	1024
	ppl	1.014	1.011	1.010	1.011	1.014
Embedding size	Value	64	128	256	300	500
	ppl	1.010	1.010	1.010	1.011	1.011

Table 1: Experiments on hyperparameters. All these groups are conducted on the RNN model. Only one hyperparameter is changed per group and others stay same as the standard one.

3 Experiments

3.1 Influence of hyperparameters

In this part, several experiments are conducted to find out the influences of hyperparameters on model performance, including ***learning rate, step length, embedding size, and hidden size***. Besides, three optimizers based on different algorithms are applied.

Here I define a **standard setting** as following, *[batch size=512, optimizer=Adam, lr=1e-3, n-step=5, hidden size=512, embedding=256]*. All the controlled experiments are conducted based on it.

Optimizer comparison			
Optimizer	SGD	RMSprop	Adam
ppl	11.62	1.983	1.011

Learning rate comparison			
Learning rate	1e-2	1e-3	1e-4
ppl	1.272	1.011	1.019

Table 2: Influence of optimizer and learning rate.

In terms of the optimizer, three different algorithms are applied, representing the classical method and adaptive learning rate respectively. It is obvious (Table 2) that optimizers can adapt learning rates work much better than SGD. Though the learning rate can be adjusted during the process, it is important to choose a basic rate, which is compatible with the optimizer.

Compared with the former factors, when hyperparameters inside models are changed, results seem to fluctuate less (Table 1). Specifically, when we change the length of the task, the ppl rises as the step grows. As for the hidden size and embedding size, they are more likely to affect model convergence rate or even cause overfitting if not set correctly.

3.2 Comparison of models

Here I compare the performances of various models, such as FFN, RNN, and LSTM. All those models share the same embedding layer construction and hyperparameters are set following the standard one except *n-step*, which is designated as 10.

Model performance		
Model	ppl	Time/epoch(seconds)
FFN	1.203	17s
RNN	1.026	6s
RNN with grad clip	1.025	8s
LSTM	1.024	11s
BiLSTM	1.022	21s

Table 3: Performance of various models following the standard setting. Running time is calculated on RTX 6000, ignoring the time to load the text.

During this part of the experiment, the models' performances are roughly the same, except FFN. According to Table 3, FFN runs slower and its *ppl* is the worst. In terms of the sequential model, RNN is more efficient, but LSTM has the potential to obtain more accurate results. I use the standard setting prepared for RNN during this experiment. If time permits, some adjustments would be done on settings for LSTM to better complete the prediction task.

Additionally, I introduce grad clipping into the training of RNN. It is not obvious from the results, but it definitely helps model fit faster.

3.3 Pretrained word embedding

Aiming at gaining an efficient word-to-vector list, skip-gram work is done based on the whole *PTB* data set. After training that predict model, the word2vector list is saved and then used to initialize the embedding layers of those models in 3.2.

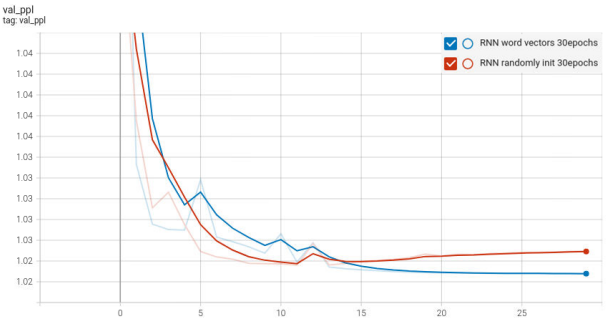

Figure 1: Utility of pretraining

To see how it works, RNN is taken as an example. Both models in figure 1 run under standard settings for thirty epochs. It is clear that RNN initialized randomly becomes overfitting after training for 15 epochs, while, the blue curve, standing for the pretrained one decreases steadily and finally reaches the *ppl* of 1.022. In contrast, the red one has only reached 1.024.

4 Conclusion

During the second stage of the course, I build up various neural network models and conducted experiments to figure out relations between certain hyperparameters and model performance. In addition, I compare models by their performance and try some popular tricks, such as pretraining, to make them work better.

Finally, I would like to give thanks to our tutors and seniors. I have learned a lot and had a pretty good time in the last week!

References

Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems*, 13.

Tomas Mikolov. Recurrent neural network based language model.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

A Records of experiments

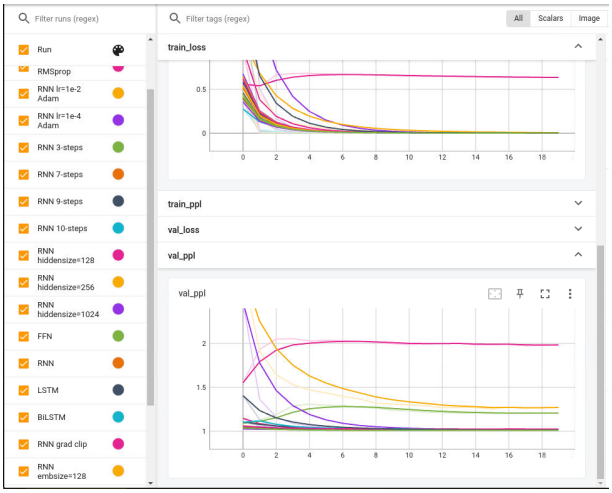

Figure 2: Records of experiments