Homework 2: Learning on Sequenece Data

Haoyu Wang
Department of Electronic Engineering
Tsinghua University
wanghy22@mails.tsinghua.edu.cn

For all models in the homework, we choose SGD optimizer and "ReduceLROnPlateau" scheduler with factor=0.5 and patience=2. Other common hyper-parameters are listed in Tab. 1.

Versions of python and pytorch are 3.9 and 1.10.2, respectively.

Table 1: Common Hyper-parameters

n_{voc}	L_{seq}	BS_{train}	BS_{eval}	dropout
33278	35	20	10	0.5

1 Part One: RNN

1.1 RNN for Language Modeling:

Multi-layer GRU network is implemented in "model.py". We first implement one-layer GRU by using existing *torch.nn.GRU* module. Then we stack each layer to form the multi-layer GRU. The training and evaluate code are implemented in "main.py". In this part, we train a two-layer GRU network with wiki-text2 dataset. The hyper-parameter settings are listed in Table. 2. The final loss and perplexity are listed in Tab 3. The training and valid curves are plotted in Fig. 1.

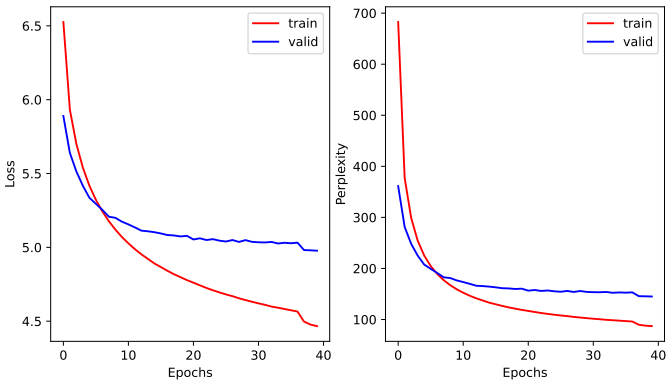


Figure 1: training and validation curves of GRU

In Fig. 1, it can be seen that

1. In the beginning of training, the valid loss are even lower than train loss, which indicates the valid dataset are relatively easier for the model in this period.

Preprint. Under review.

2. After about 20 epochs, training loss continues to decrease but the valid loss stops. Valid loss and perplexity keeps around 5.05 and 155 in the last 20 epochs, which means overfitting happens during the training. It indicates that GRU has relatively poor generalization capability.
3. There is small steep descent of loss and perplexity near the end of training, which is a result of "ReduceLROnPlateau" scheduler.

1.2 LSTM Implementation:

In this part, we implement multi-layer LSTM in "model.py". We first implement the forward function of one-layer LSTM based on the equations in Equ. 1, where $\sigma(\cdot)$ means sigmoid function. Then, we stack each layer to form a multi-layer LSTM. Here, we train a two-layer LSTM network with wiki-text2 dataset. The hyper-parameter settings are listed in Table. 2. The final loss and perplexity are listed in Tab. 3. The training and valid curves are plotted in Fig. 2.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{1}$$
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{3}$$
$$g_t = \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g) \tag{4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{5}$$
$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

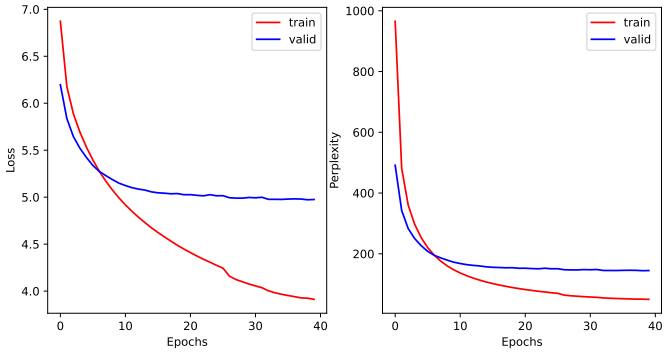


Figure 2: training and validation curves of LSTM

In Fig. 2, it can be seen that

1. Similar to GRU, the valid loss are lower than train loss in the beginning.
2. Additionally, after about 15 epochs, training loss continues to decrease but the valid loss stops. Valid loss and perplexity keeps around 4.97 and 144 in the last 25 epochs, which means overfitting happens during the training. Compared with GRU model in Sec. 1.1, although the hidden dimensions are higher and the gate control mechanism are more complex, LSTM does not achieve a meaningful increment in valid perplexity. Therefore, pure RNN based models are not so efficient in language modeling.

1.3 Question Answering:

Based on reading in [1] and my understanding, some advantages and disadvantages of character-level RNN can be summarized as:

1. **Advantages:** (1) The size of dictionary is decreased from n_{voc} to n_{chara}. For instance, the vocabulary amount of wiki-text2 is 33278, which means the probability output of each word

is passed through a $n_{voc} = 33278$-dim softmax. The extremely large dictionary size makes the cross-entropy loss difficult to converge. For a character-level RNN, however, it only predicts the probability of n_{chara} characters, which is easier to converge. (2) Meanwhile, character-level RNN can save the parameters in the final linear layers $\lfloor \frac{n_{nvoc}}{n_{chara}} \rfloor$ times. (3) Character-level RNN are more suitable for character-level tasks, for instance, translate a uncased sequence into a cased sequence [1].

2. **Disadvantages:** (1) Character-level RNN makes the long-distance dependencies more difficult. The distance between two characters becomes word length times of the word-level RNN. (2) Since words are splitted into characters, it makes model to learn word-level feature more difficult. (3) If attention is used in character-level RNN, the attention matrix size grows $\mathcal{O}(wordLen^2)$ of the word-level RNN, which is not memory-efficient.

Table 2: Hyper-parameters of RNN

network	layer	d_h	d_{em}	l_{ini}	l_{min}	epochs
GRU	2	200	200	2	0.01	40
LSTM	2	650	200	2	0.01	40

Table 3: Performance of RNN

network	traing loss	train PP	valid loss	valid PP
GRU	4.467	87.052	4.978	145.085
LSTM	3.912	50.011	4.972	144.361

2 Part Two: Transformer

2.1 Question Answering:

Q1: why use multiple attention heads instead of one?

Multiple head attention allows the model to jointly attend to information at different positions from different representation subspaces. For instance, the model can force different heads to learn different aspects of sequences (e.g., subject-predicate, referential relationships, etc). Then the model can aggregate different kinds of attention to learn the sequence better.

Q2: why dividing $\sqrt{d/h}$ before applying softmax function?

Consider the dot product of nth query and mth key in ith head

$$a_i(q_{i,n}, k_{i,m}) = q_{i,n}^T k_{i,m} \tag{7}$$

where query $q_{i,n} \in \mathbb{R}^{d/h}$ and key $k_{i,m} \in \mathbb{R}^{d/h}$. Assume each element in $q_{i,n}$ and $k_{i,m}$ are independent and with zero mean and 1 variance. Then $\text{Var}(a_i(q_{i,n}, k_{i,m})) = d/h$. Therefore, to avoid variance explosion in deep attention layers, we need to normalize the variance of the dot-product output. After dividing $\sqrt{d/h}$, the dot-product variance $\text{Var}(a_i(q_{i,n}, k_{i,m})/\sqrt{d/h}) = 1$, which is the same with the input. Thus, we can ensure all calculation outputs vary in an acceptable range.

2.2 Multihead Attention:

In this part, we implement the mutiple-head attention module in "mha.py". Here, multi-head attention is calculated based on tensor operations, which is forwarded in parallel. The results are depicted in Fig. 3. Here, the relative errors of self-attention, masked-self-attention, attention output are $1.4e^{-4}, 1.4e^{-4}, 1.1e^{-4}$, respectively. It indicates that code implementation is correct.

2.3 Transformer for Language Modeling:

In this part, we construct a transformer language model, which is based on *torch.nn.Transformer* [2]. The functions of different components in *torch.nn.Transformer* are summarized as follows.

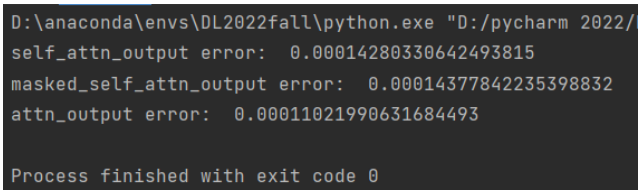


Figure 3: Relative error check for multi-head attention

1. **Transformer:** It consists of a transformerEncoder and a transformerDecoder.
2. **Transformer Encoder:** It consists of several transformerEncoderLayers. Different layers are cascaded. Its input include embedded word vector and source mask. In this task, we do not use source mask.
3. **Transformer Encoder Layer:** It consists of a self-attention block and a feed-forward block. In each block, there exists a residual short-path to avoid gradient vanishing. Additionally, layer-norms are applied after calculation of each block.
4. **Transformer Decoder:** It consists of several transformerDecoderLayers. Different layers are cascaded. Its input include embedded word vectors and output (memory) of the transformer encoder and target mask. In this task, we use casual mask to keep auto-regressive.
5. **Transformer Decoder Layer:** It consists of a self-attention block , a multihead-attention block and a feed-forward block. The self-attention block calculate the self-attention of the input. The multihead-attention block calculate the attention of input and memory. In each block, there exists a residual short-path to avoid gradient vanishing. Additionally, layer-norms are applied after calculation of each block.

Here, we implement a transformer with 2-layer encoder and 2-layer decoder. Other hyper-parameters are listed in *network original* of Tab. 4. The training and validation curves are plotted in Fig. 4. The final training and validation results are listed in *network original* of Tab. 5.

In Fig. 4, it can be seen that:

1. Similar to RNN-type language models, the valid loss are lower than train loss in the beginning.
2. Compared to RNN-type language models, difference between training loss and valid loss becomes much smaller. Additionally, final valid loss and perplexity are decreased to 4.563 and 95.870, respectively. The results indicate that transformer has better generalization performance compared to RNN-type models. It validates long-distance dependencies abilities of transformer.

Table 4: Hyper-parameters of Transformer

network	L_{en}	L_{de}	d_h	d_{em}	d_{ff}	n_{heaed}	l_{ini}	l_{min}	epochs
original	2	2	400	400	2048	8	0.1	0.0001	80

Table 5: Performance of Transformer

network	traing loss	train PP	valid loss	valid PP
original	4.465	86.896	4.563	95.870
prior knowledge	4.150	63.412	4.282	72.412

2.4 Masking:

Q1: Why masking is necessary in language model?

For casual language modeling, we need to add an attention mask in the self-attention block of decoder. Two main reasons can be summarized:

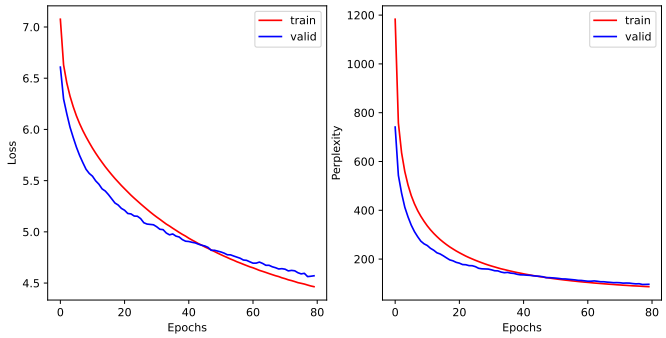


Figure 4: training and validation curves of transformer

1. For casual language modeling, we need to keep the model auto-regressive, which means we can only see the words before it when predicting a target word. By masking, the attention weight after current word is set as 0, so we can avoid the model cheating.
2. For primitive auto-regressive, we can only predict the next word after receiving the prediction of current word, which is in serial and inefficient in computation. With masking, we can easily add a masking matrix to the attention matrix before softmax. All operations are tensor parallel and is computation efficient.

Q2: How is masking implemented in my code?

In my code, I use *nn.Transformer.generate_square_subsequent_mask* to generate casual masking for self-attention. For instance, when applying *nn.Transformer.generate_square_subsequent_mask(4)*, we can obtain a masking matrix as:

$$Mask = \begin{bmatrix} 0 & -inf & -inf & -inf \\ 0 & 0 & -inf & -inf \\ 0 & 0 & 0 & -inf \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{8}$$

Consider a sentence with 4 words. After the calculation scaled-dot product QK^T, we obtain a 4×4 matrix A_{dot}, then we apply masking before softmax as follows:

$$Attn = softmax(A_{dot} + Mask) = softmax(\begin{bmatrix} a_{11} & -inf & -inf & -inf \\ a_{21} & a_{22} & -inf & -inf \\ a_{31} & a_{32} & a_{33} & -inf \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, dim = -1). \tag{9}$$

Then, attention weight $Attn(i, j) = 0$ for $j > i$, which means current word cannot see the target word after it. In this way, we can avoid the model from cheating.

2.5 Attention Visualization:

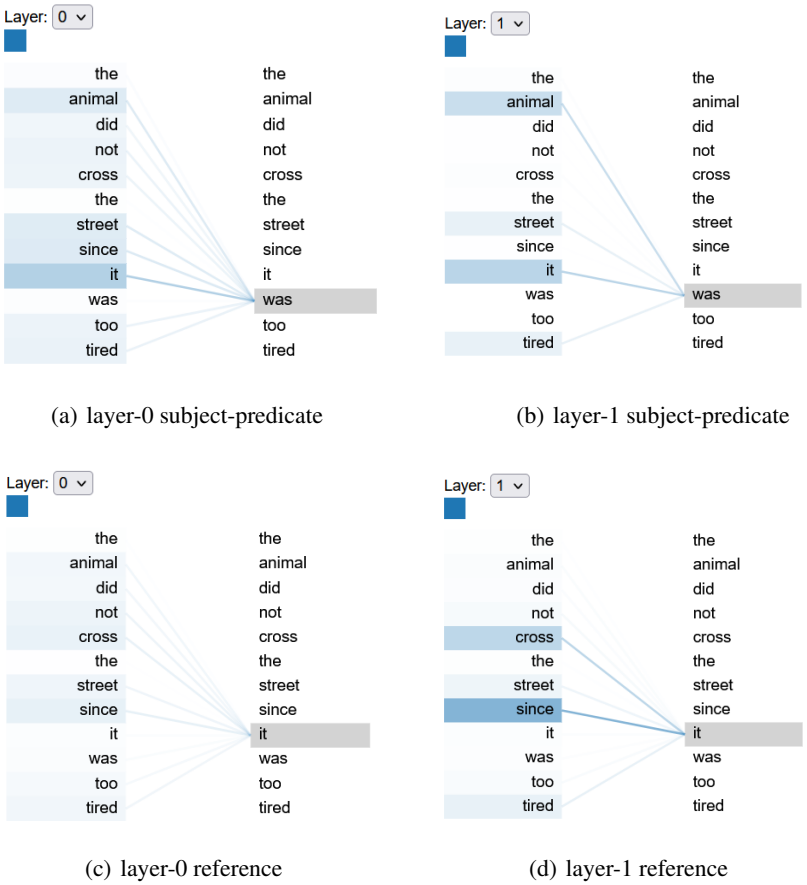
In this part, we visualize the attention weight of the trained transformer decoder. The visualization code is implemented in "attnvis.ipynb" and visualization is based on module "bertviz" [3]. We choose the sentence "*the animal did not cross the street since it was too tired*" and visualize the attention weight of source words to a given target word.

Here, it should be noted that we visualize the attention weight of the "multihead-attention" in the decoder layer, which is averaged over different heads. The results are shown from Fig. 5(a) to 5(d). It can be seen that:

1. In layer-0 (Fig. 5(a)), it can be seen that many words have relatively high attention of target word "was". In layer-1 (Fig. 5(b)), only words "animal", "street", "it" and "tired" have relatively high attention weight, which proves that the model becomes more "concentrated"

in deeper layers. Meanwhile, words "animal" and "it" have the highest attention weight, which indicates that "subject-predicate" relationship is relatively simple in linguistics and shallow transformer network can understand it well.

2. Similar to the "subject-predicate" case, attention weight is dispersed in layer-0 (Fig. 5(c)). In layer-1 (Fig. 5(d)), attention weight becomes more "concentrated". However, the model misunderstands the word "it" since it has small attention weight between word "animal" and "it". Thus, it can be inferred that the "reference" relationship is more complicated than "subjective-predictive" relationship for shallow transformer. We can further deepen the network to help transformer understand better.



(a) layer-0 subject-predicate

(b) layer-1 subject-predicate

(c) layer-0 reference

(d) layer-1 reference

Figure 5: Attention Visualization

3 Part Three: Improve Your Language Model

Comparing Fig. 4 with 2 and 1, we could find that transformer converge relatively slower than RNN-type language models. Two main reasons can be inferred: (1) transformer has much more parameters than RNN type models; (2) sequence length L_{seq} is set as 35 in the experiment, which makes it difficult to allocate attention.

To guarantee model effectiveness, we cannot reduce the parameters of the shallow transformer. Therefore, we need to bring in some prior knowledge, which helps transformer to allocate attention more easily. One intuitive idea is that one word correlates with its "neighbors" with high probability in many cases. Therefore, we need to measure the distance between different words and allocate more attention to the "closer" word.

Motivated by Guassian Trasnformer in [4], we define the distance $d_{i,j}$ between word i and word j as

$$d_{i,j} = w\sqrt{\frac{|i-j|}{L_{seq}}}. \tag{10}$$

Then, the attention weight considering word distance can be calculated by

$$Attn = softmax(A_{dot} + Mask + D, dim = -1), \tag{11}$$

where A_{dot} is the scaled dot product matrix between queries and keys, $Mask$ is masking matrix and $D = (-d_{i,j}^2)_{L_{seq} \times L_{seq}}$.

To validate effectiveness of prior word distance, we keep the model parameters in Table .4 and add word distance in the "multihead attention" module in each layer of decoder. w in Equ. 10 is set as 1. The training and validation curves are shown in Fig. 6 and the final results are shown in Table. 5. Comparing the results, we can find that

1. transformer with prior knowledge converges faster than the original one. It indicates that prior knowledge helps the model to allocate attention more easily.
2. transformer with prior knowledge achieves lower training and validation perplexity, which validates the effectiveness of prior knowledge.
3. Here, we visualize the multihead attention weight of decoder, as shown in Fig. 7(a) and 7(b). Comparing 7(a) and 5(b), we can find that the model can still learn subjective-predicate relationship. Unfortunately, as shown in Fig. 7(b), pure spatial-distance based prior knowledge cannot help understand reference relationship better, since reference usually has long distance dependency.

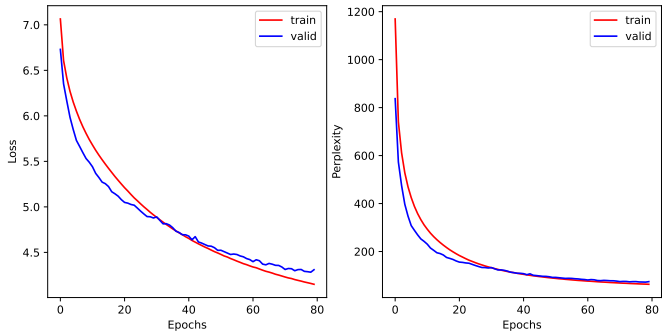
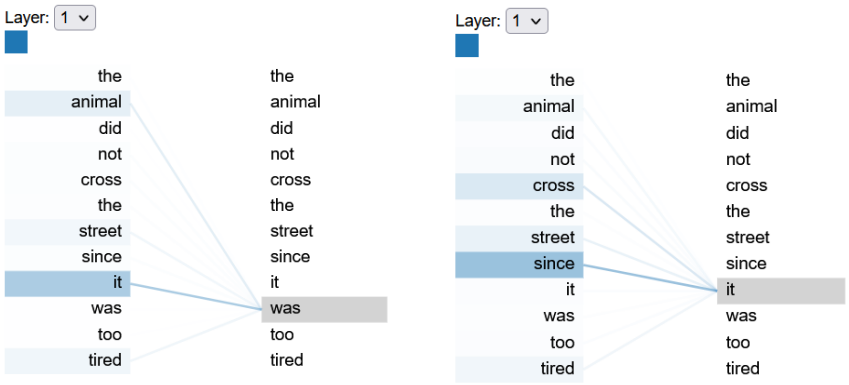


Figure 6: training and validation curves of transformer (with prior knowledge)



(a) layer-1 subjective-predicate (b) layer-1 reference

Figure 7: Attention Visualization (with prior knowledge)

References

[1] Raymond Hendy Susanto, Hai Leong Chieu, and Wei Lu. Learning to capitalize with character-level recurrent neural networks: an empirical study. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 2090–2095, 2016.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[3] Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-3007. URL `https://www.aclweb.org/anthology/P19-3007`.

[4] Maosheng Guo, Yu Zhang, and Ting Liu. Gaussian transformer: a lightweight approach for natural language inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6489–6496, 2019.