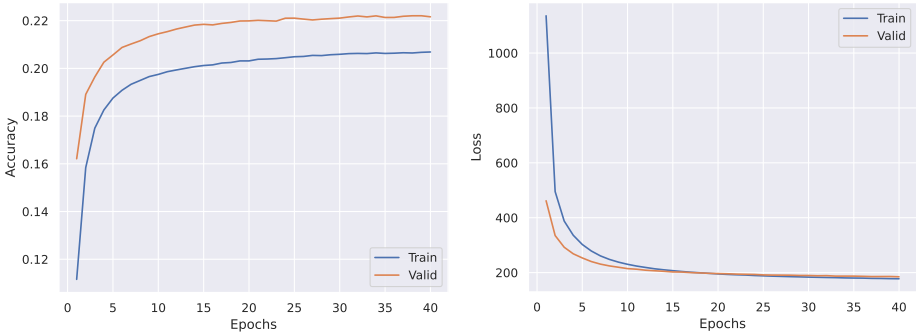
Homework 2

Zhe Xie, 2022310854

November 9, 2022

1 Part One (RNN)

1.1 RNN for Language Modeling.



(a) GRU Accuracy

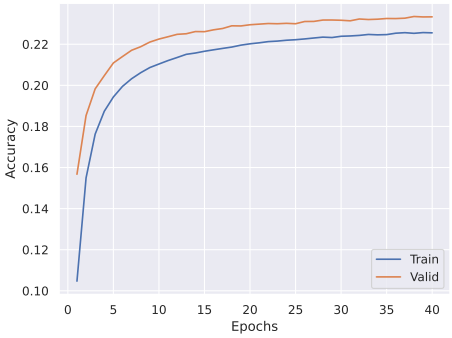
(b) GRU Loss

Figure 1: Loss and Accuracy Curves of GRU Model

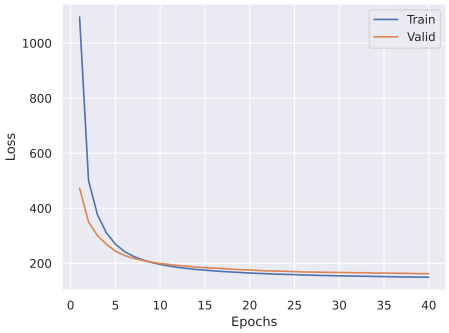
In this section, I train the language model on the given dataset with the `GRU` [1] model, implemented with `torch.nn.GRU` in PyTorch. Figure 1 shows the training and validation results. The accuracy curves show the proportion of predicted words that exactly match the labeled words. It can be found that the training accuracy is even lower than the validation accuracy, which is interesting. This may indicate that the default GRU model has additional room for improvement for this dataset. In this model, I set the `max_sql` (the maximal sequence length in training) to 35, which is the default value in the code, and the `nlayers` of the GRU model to 6. The validation accuracy of the GRU model is 0.226.

1.2 LSTM Implementation

In this section, I implement the LSTM [3] module with PyTorch (without using the `torch.nn.LSTM` module). Figure 2 shows the result of training and validation curves. It can also be found that the validation accuracy is higher than the



(a) LSTM Accuracy



(b) LSTM Loss

Figure 2: Loss and Accuracy Curves of LSTM Model

training accuracy, but their gap is much smaller than it is in the GRU model. This indicates that GRU, although more lightweight, does not fit as well as LSTM in this data. The LSTM model achieves a validation accuracy of 0.233, which is higher than the GRU model.

1.3 Question Answering

- **Advantage of character-level RNN:** The size of the vocabulary will be much smaller than its size in the word-level RNN. This may effectively avoid the problem that the huge vocabulary makes word embedding difficult.
- **Disadvantage of character-level RNN:** Each word has a more explicit semantic meaning, but the letters do not. RNN can only learn the meaning of each word through their dependencies, and it is difficult to get effective semantic information through character embeddings. This will make the learning of text difficult.

2 Part Two (Transformer)

2.1 Question Answering

1. **Why use multiple attention heads instead of one?**

   Each word usually contains different semantic information. If only one attention head is used, attention weights can only be calculated based on a single semantic information. If multiple attention heads are used, different attention weights can be assigned based on different semantic information, thus better allowing the attention model to learn the semantic information of the text.

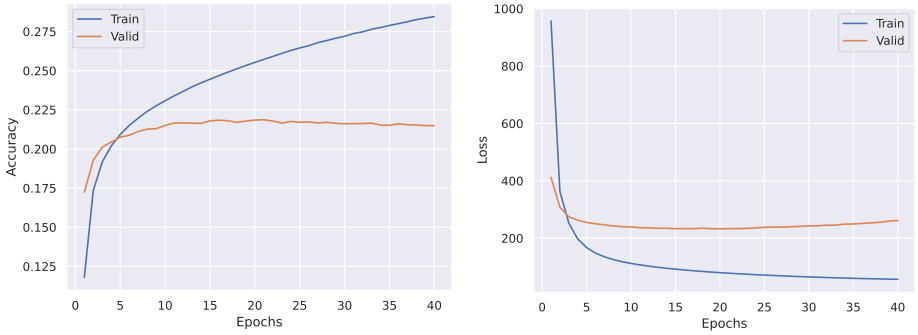
2. **Why divide by $\sqrt{d/h}$ before applying the softmax function?**

   In single-head attention, the QK^T is divided by $\sqrt{d}$ to keep the variance expectation of the attention weights to 1. However, in multi-head attention, the attention weights of different heads are calculated respectively, which means that the dimension of $Q_h K_h^T$ for each head is d/h. So in multi-head attention, we need to divide the $Q_h K_h^T$ by $\sqrt{d/h}$ to keep the variance and make the training process more steady.

2.2 Multi-head Attention

See `mha.py` in the submission.

2.3 Transformer for Language Modeling



(a) Transformer Accuracy (b) Transformer Loss

Figure 3: Loss and Accuracy Curves of Transformer LM Model

I implement the Transformer [4] Language Model with `nn.TransformerEncoder` in PyTorch. The training and validation curves are demonstrated in Figure 3. We can find that the training accuracy is geting higher during training, while the validation accuracy shows performance degradation after epoch 15. The final validation accuracy of the Transformer model is only 0.215, which is much lower than LSTM and GRU. This may be due to an overfitting of the Transformer model on this small dataset. I will further analyze this phenomenon in Transformer in Section 3. Please refer to the source code in the submission for more details.

2.4 Masking in Transformer LM

In Transformer LM, mask is necessary to avoid the leakage of labels [4]. Without masks, the Transformer may directly assign the attention weight of the next word x_{t+1} to 1, which simply copy the words in the input to the output. This

prevents the model from learning any valid information about the semantics and makes it difficult to use it in practice.

I use the following code the generate the mask in Transformer:

```
attn_mask = torch.tril(torch.ones(max_sql, max_sql, dtype
    =torch.bool))
attn_mask = attn_mask.float().masked_fill(~attn_mask, -np
    .inf).masked_fill(attn_mask, 0.0)
```

This code produces a mask matrix with the masked values filled with `-inf` and other values filled with `0`, according to the definition in PyTorch[1].

In the mask matrix, the values in the lower triangular part (including the diagonal part) are filled with `-inf`, which means that the attention weight in this part should **not** be considered in the calculation. In this way, for each word, all subsequent input words are masked to avoid the label leakage.

2.5 Attention Visualization

from the pressure . They have also been observed to strike at each other with mouths closed . Occasionally , the combatants will tire and break off the fight by " mutual consent " ,

(a) Attention weight for word "been"

from the pressure . They have also been observed to strike at each other with mouths closed . Occasionally , the combatants will tire and break off the fight by " mutual consent " ,

(b) Attention weight for word "observed"

from the pressure . They have also been observed to strike at each other with mouths closed . Occasionally , the combatants will tire and break off the fight by " mutual consent " ,

(c) Attention weight for word "off"

Figure 4: Attention weights for several typical words. Red colors represents different weights (darker denotes larger weights) and green colors represent the current word (to be predicted).

Figure 4 shows the attention weights for several typical words. With this weights, we can find that:

- The Transformer LM can correctly learn the positional information among words. The words in the neighborhood usually have larger attention weights.

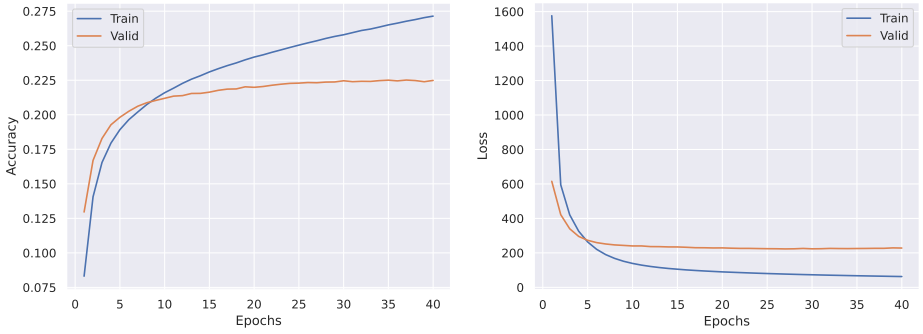
[1] https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html#torch.nn.Transformer.forward

- Transformer can learn part of the syntax information. Figure 4a shows the attention weight of word "been". We can find that the word "have" has larger attention weight than the other words, which means that the model effectively learn something about the grammar of the passive voice.

3 Part Three

3.1 Motivation

In this section, I focus on the modification of the Transformer model. Figure 3a shows that the Transformer suffers from overfitting. To mitigate this, I also try training the model with larger `max_sql`.



(a) Transformer Accuracy

(b) Transformer Loss

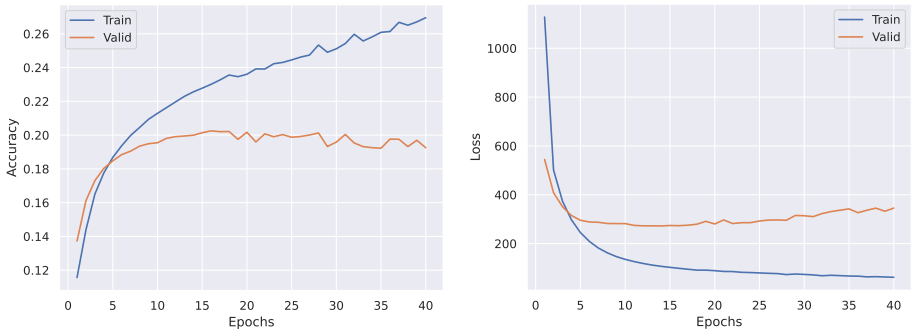
Figure 5: Loss and Accuracy Curves of Transformer LM Model (`max_sql`=100)

Figure 5 shows the training and validation curves We can find that when the sequence length becomes larger, the validation accuracy also becomes higher. However, the time complexity of the Transformer is $O(n^2 d)$, where n denotes the sequence length and d denotes the internal dimension of the model. This means that the Transformer may suffer from large computational overhead when `max_sql` is large.

3.2 Method

To address this challenge, I try to train using the Performer [2] model instead of the Transformer model, which claims to have linear time complexity $O(nd^2 \log(d))$. Performer utilizes the kernel function to approximate the computation of attention, which makes the computation faster. I use `performer_pytorch`[2] to implement the Performer LM model. Since the parallelized implementation of the Performer model is complex (additional cuda operators are need for the autoregressive Performer model), I directly used the open source implementation.

[2]https://github.com/lucidrains/performer-pytorch.git



(a) Performer Accuracy (b) Performer Loss

Figure 6: Loss and Accuracy Curves of Performer LM Model (`max_sql`=100)

Table 1: Comparison between Transformer and Performer

Model	`max_sql`	Accuracy	Avg. Time Consume (ms)
Transformer	35	**0.215**	**1.40**
Performer	35	0.197	2.36
Transformer	100	**0.225**	**1.67**
Performer	100	0.193	2.42

Figure 6 shows the training and validation curves of the Performer model and Table 1 shows the accuracy and time consume. We can find that the Performer model perform much **worse** than the Transformer model and its training speed is even **lower** than the Transformer model. This means that Performer has difficulty outperforming Transformer in this task.

One possible reason is that the approximation calculation in Performer is losing too much information. We can find that the performance of Performer is even worse when `max_sql`=100 than the model with `max_sql`=35. This is probably due to the fact that for longer sequences, the Performer model loses more information, leading to the performance degradation.

In addition, the time overhead of Performer is also related to the way it is implemented. In order to implement causal modeling in Performer, we must use a custom cuda operator instead of the original matrix multiplication calculation (See Appendix B.1 in [2]). This unoptimized computation in GPU may lead to a degradation of its time efficiency. Therefore, in order to achieve an effective performance improvement, we may need a more efficient implementation.

4 Conclusion

In this assignment, I trained GRU, LSTM, Transformer and Performer on the given dataset. For the Transformer model, I visualized the attention weights of

some typical words, which shows that the Transformer effectively learned the relationship between words and even the grammar in certain types of sentences. In the comparison between Transformer and Performer, I find that although Perform claim that it has lower time complexity, neither its time efficiency nor its performance exceeds that of Transformer. In the future, I can try to test Performer's performance on other datasets and try more efficient implementations.

References

[1] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[2] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

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

[4] 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.