ADL Hw1 Report

B08902029 資工三 陳咏誼

Q1 Data processing

1. How do you tokenize the data.

I use the example code.

1. Record the amount of appreance of every token(word).
2. Extract the top N frequently used words. $N =$ vocabulary size.
3. Open GloVe and save the embedding for the top N frequently used words.
4. Save the embedding to `embedding.pt`

2. The pre-trained embedding you used.

I use GloVe with 840B tokens and 300 embedding dimension.

Q2: Describe your intent classification model.

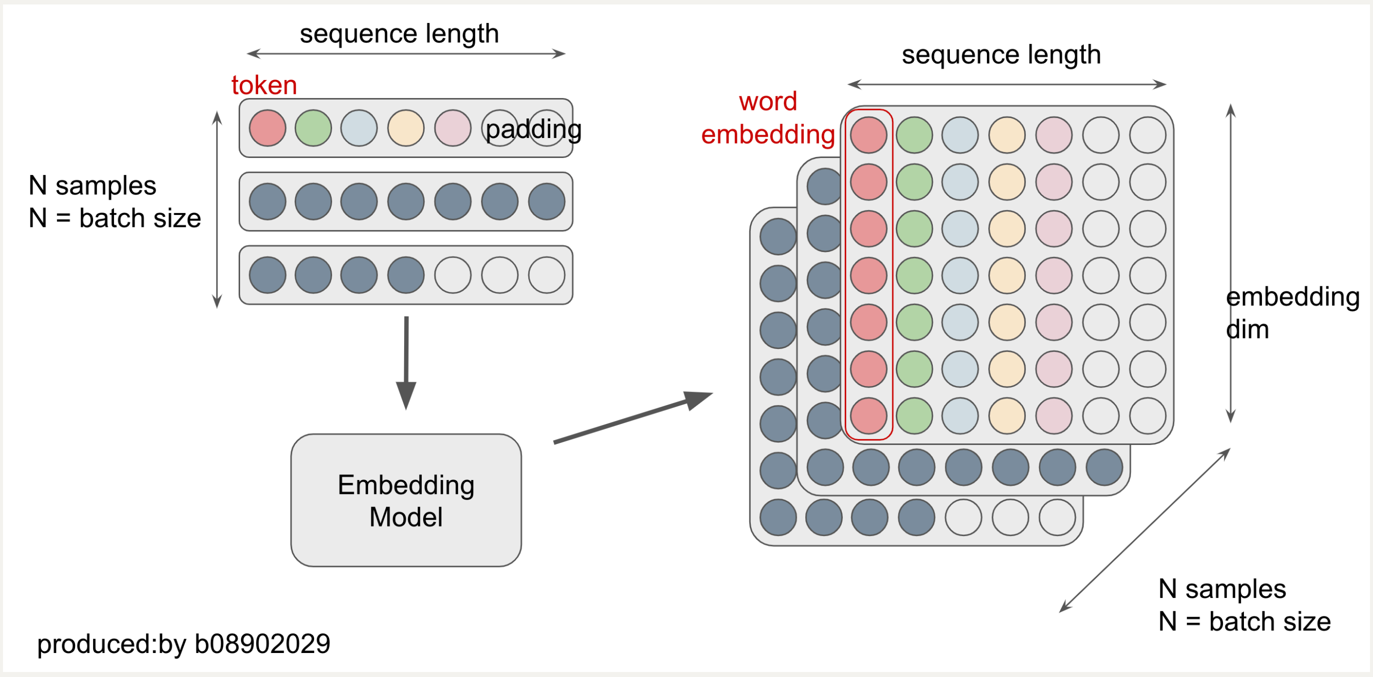
1. Model

The classification model includes 3 blocks: Embedding model, LSTM model, Fully connected layers.

1-1. Embedding model

Embedding model converts every token to a vector(word embedding).

- shape of input = batch_size $\times$ sequence_len
- shape of output= batch size $\times$ sequence_len $\times$ embedding_dim (embedding_dim = 300)
- returned value: output of Embedding model



1-2. LSTM model

I use `torch.nn.LSTM` as encoder, and pass the **hidden states of the latest tokens** of each sample to

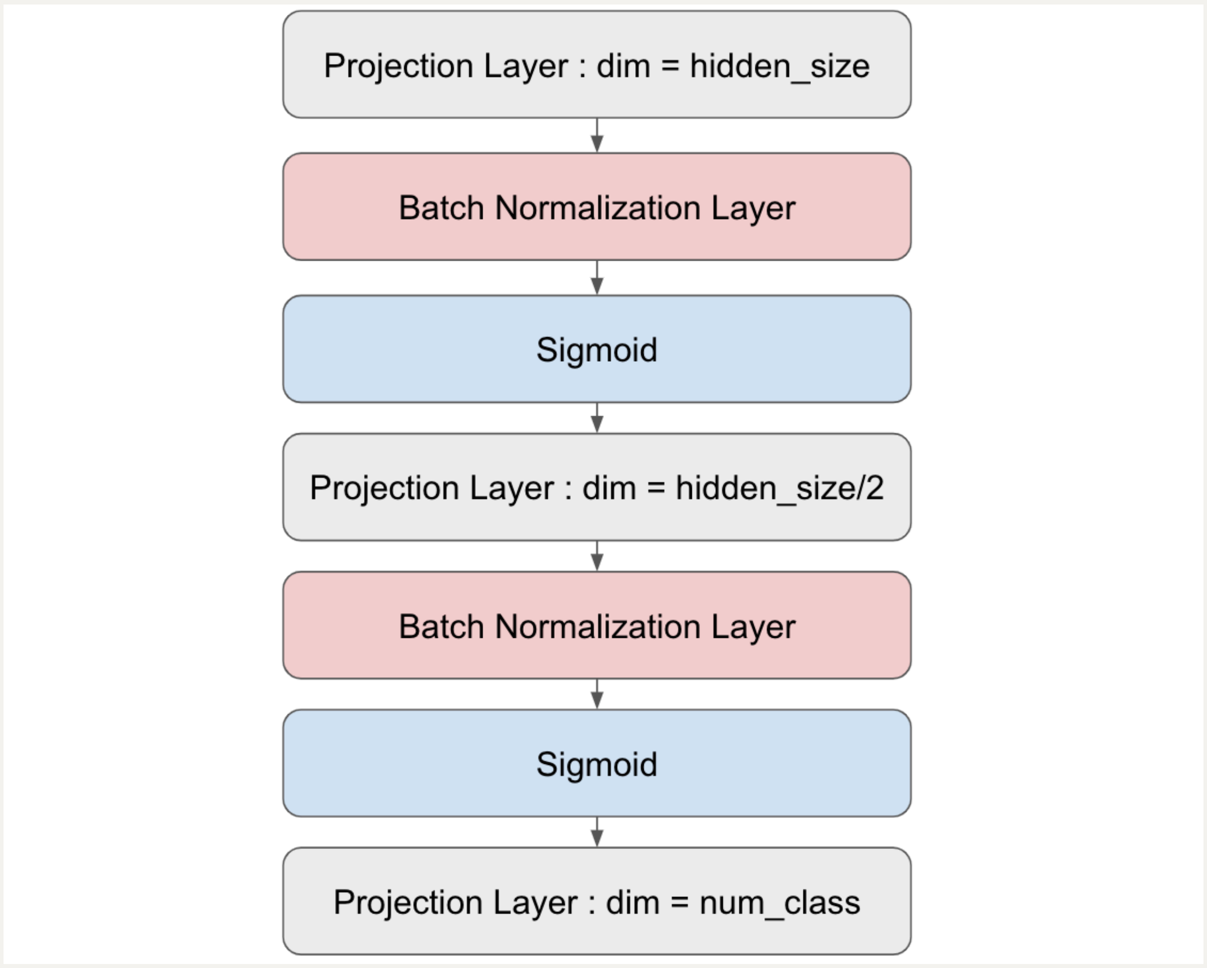
$h_t, c_t = LSTM(w_t, h_{t-1}, c_{t-1})$, where w_t is the word embedding of the t-th token and h_0, c_0 are zero matrices.

- Parameters:
    - hidden_size = 512
    - dropout = 0.3
    - bidirectional = True
    - num_layers = 2
- shape of input = batch size $\times$ sequence_len $\times$ embedding_dim

- shape of output of LSTM model = batch_size $\times$ seq_len $\times$ (2 $\times$ hidden_size)
- returned value: h_N, where N = batch_size. (shape = batch_size $\times$ (2 $\times$ hidden_size))

1-3. Fully-connected layers

- shape of input = batch_size $\times$ (2 $\times$ hidden_size)
- shape of output = batch_size $\times$ num_class (num_class = 150)


Projection Layer : dim = hidden_size
Batch Normalization Layer
Sigmoid
Projection Layer : dim = hidden_size/2
Batch Normalization Layer
Sigmoid
Projection Layer : dim = num_class

2. Performance

This base model:

- valid_acc: 0.926 ; valid_loss: 0.3405570055189681
- test_acc: 0.90177

Final best result:

- public test acc: 0.93244
- private test acc: 0.92933

3. Loss function

- CrossEntropy

4. Other

- batch_size = 32
- optimization algorithm = Adam
- learning_rate = 0.001

Q3: Describe your slot tagging model.

1. Model

Similar to intent classification, the slot tagging model includes 3 blocks: Embedding model, LSTM model, Fully connected layers.

1-1 Embedding model

Same as the embedding model in intent classification model.

1-2. LSTM model

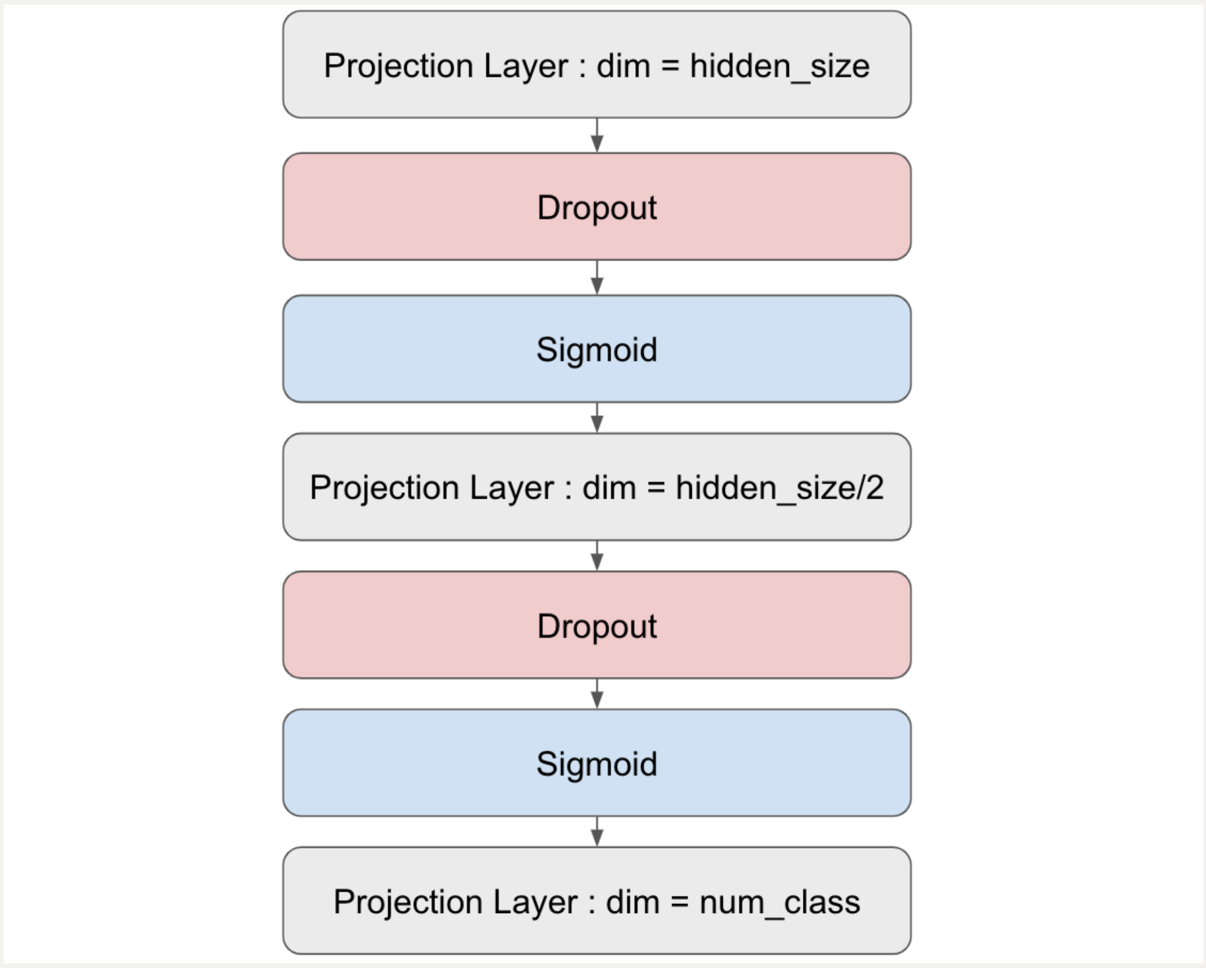
$h_t, c_t = LSTM(w_t, h_{t-1}, c_{t-1})$, where w_t is the word embedding of the t-th token and h_0, c_0 are zero matrices.

- Parameters:
    - hidden_size = 512
    - dropout = 0.1
    - bidirectional = True

    - num_layers = 2
- shape of input= batch size × sequence_len × embedding_dim
- shape of output of LSTM model = batch_size × seq_len × (2 × hidden_size)
- returned value: output of LSTM model.

1-3. Fully-connected layers

- shape of input = batch_size × sequence_len × (2 × hidden_size)
- shape of output = batch_size × sequence_len × num_class (num_class = 9)


Projection Layer : dim = hidden_size
Dropout
Sigmoid
Projection Layer : dim = hidden_size/2
Dropout
Sigmoid
Projection Layer : dim = num_class

2. Performance

This base model:

- valid_acc: 0.813 ; valid_loss: 0.10475349848275073
- test_acc: 0.74177

Final best result:

- public test acc: 0.80697
- private test acc: 0.82368

3. Loss function

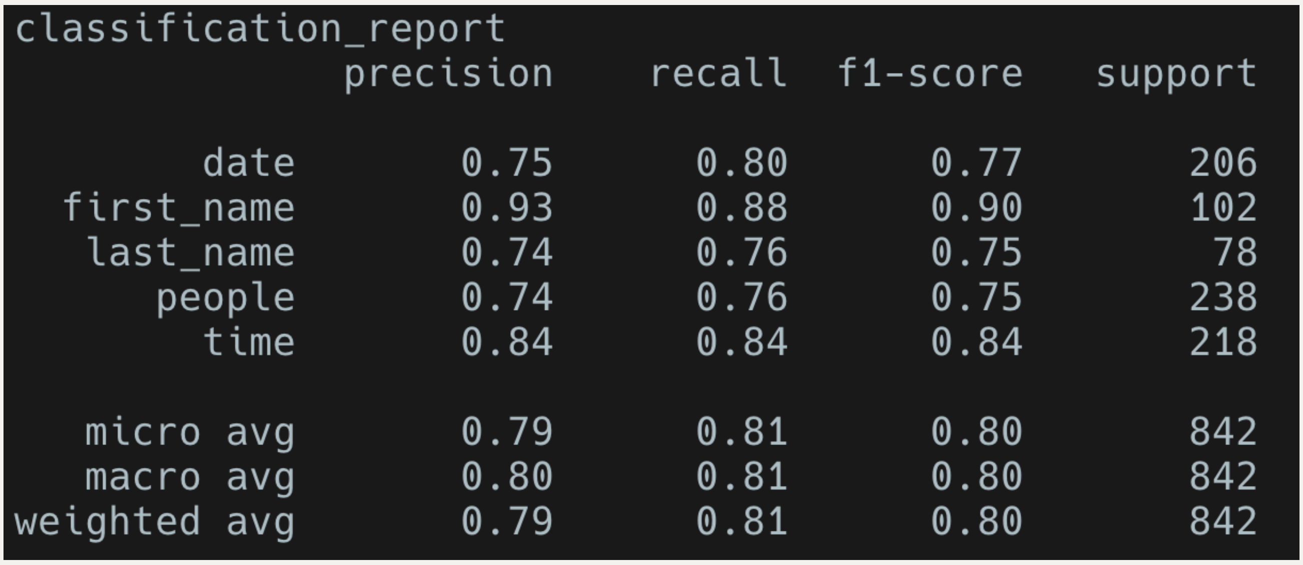
The shape of logits is $batch\_size \times sequence\_len \times num\_class$. And I see the slot tagging task as the classification task for every token, so I use `torch.nn.CrossEntropyLoss` function.

4. Other

- batch_size = 32
- optimization algorithm = Adam
- learning_rate = 0.001

Q4 Sequence Tagging Evaluation

1. Classification report of model in Q3



```
classification_report
              precision    recall  f1-score   support

        date       0.75      0.80      0.77       206
  first_name       0.93      0.88      0.90       102
   last_name       0.74      0.76      0.75        78
      people       0.74      0.76      0.75       238
        time       0.84      0.84      0.84       218

   micro avg       0.79      0.81      0.80       842
   macro avg       0.80      0.81      0.80       842
weighted avg       0.79      0.81      0.80       842
```

2. Difference between evaluation methods

- Token accuracy
    - Every token is regarded as an entity.
    - E.g.

```
Groud Truth: [B-people, I-people, O, O, B-date, I-
date]
Predicted:   [B-people, B-people, O, O, B-date, I-
date]
Token accuracy = 5 / 6
```

- Joint accuracy
    - A sample is regarded as an entity.
        - If any token in the sample is predicted wrong, the sample is considered "predicted wrong".
    - E.g.

```
Groud Truth: [B-people, I-people, O, O, B-date, I-
date]
Predicted:   [B-people, B-people, O, O, B-date, I-
date] (Wrong)
Joint accuracy = 0 / 1
```

- seqeval
    - Definition
        - $p(x)$ indicates the predicted tag of input token x
        - $t(x)$ indicates the ground-truth tag of input token x
    - Support: the amount of tags of types
    - Precision: Num($p(x) = y$ and $t(x) = y \ \forall x$) / Num($p(x) = y \ , \forall x$) , where $y \in$ {date, first_name, last_name, people, time}
    - Recall: Num($p(x) = y$ and $t(x) = y \ , \forall x$) / Num($t(x) = y \ , \forall x$) , where $y \in$ {date, first_name, last_name, people, time}
    - F1 score: $2 \times \frac{recall \times precision}{recall + precision}$

Q5 Compare with different configurations

I compare different configuration in Q3(slot tag task) as follows.

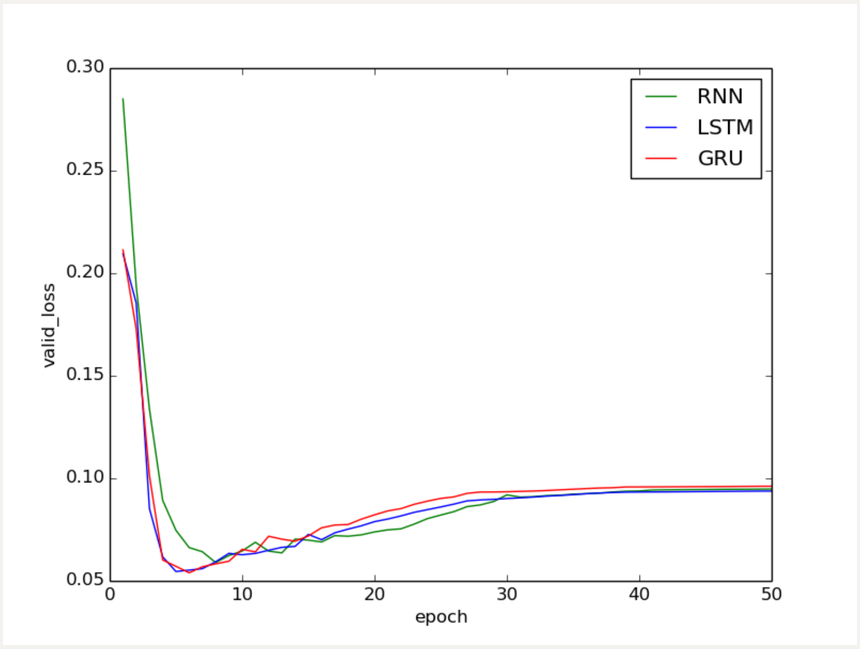
1. RNN-based: RNN vs LSTM vs GRU

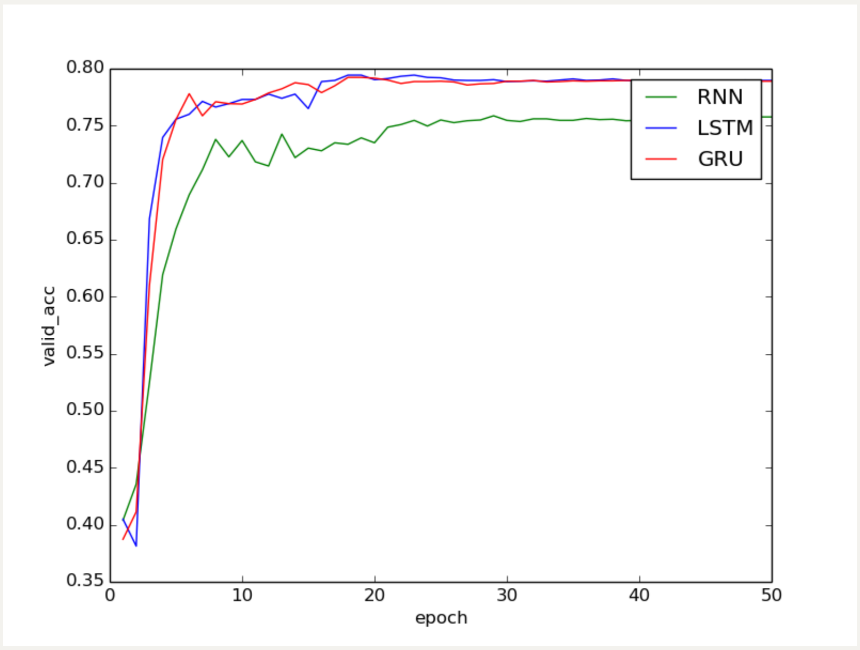
I run the comparison for three times, their **highest valid accuracy** is shown as follows.

MODEL	RNN	LSTM	GRU
Valid Acc 1	0.763	0.802	0.792
Valid Acc 2	0.774	0.801	0.808
Valid Acc 3	0.755	0.799	0.789

We can see that LSTM and GRU both has good performance compared to RNN.

In order to pick the best model between LSTM and GRU, I **average up** the results of the three tests.





Conclusions:

- Overally, LSTM outperforms GRU by a tiny disparity
- LSTM and GRU converges faster than RNN.
- They all converages within 10 epochs.
- RNN has relatively high loss and low accuracy.

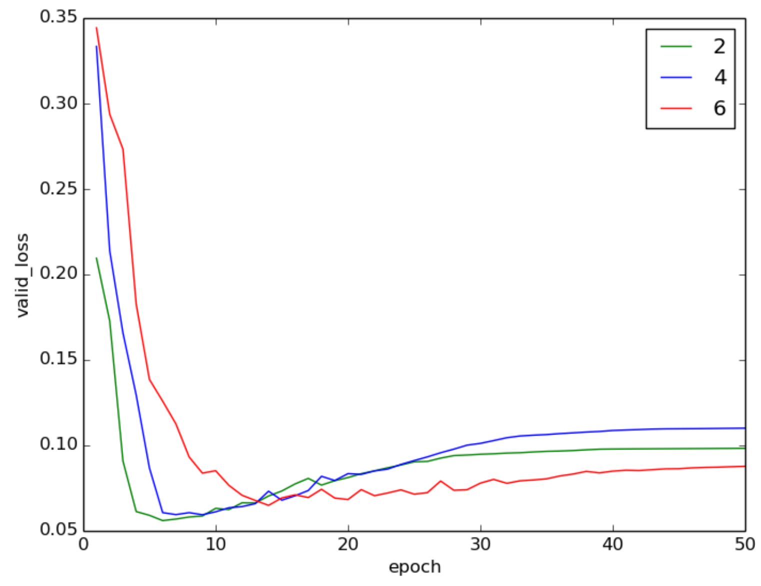
2. Number of recurrent layers

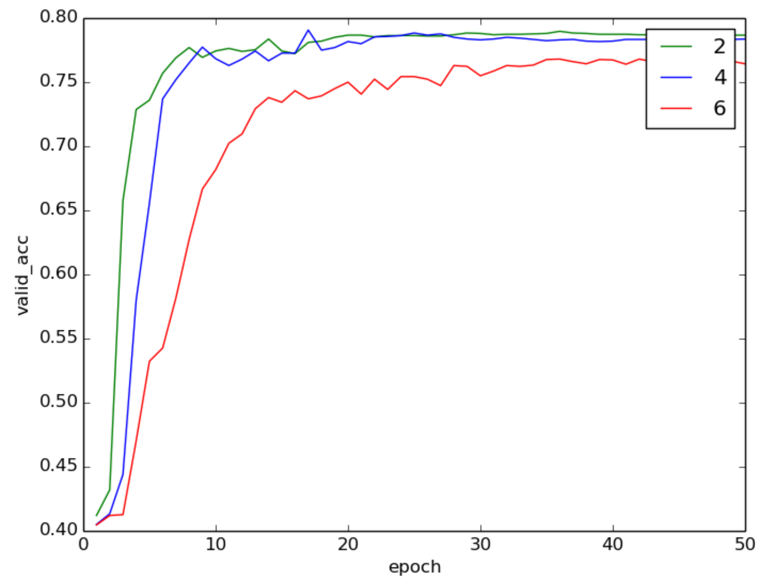
From the last comparison, I found that LSTM is the best encoder.
Then now I'd like to find the most appropriate number of its recurrent layers.

I run the comparison for three times, their **highest valid accuracy** is shown as follows.

NUM OF LAYERS	**2**	**4**	**6**
Valid Acc 1	0.787	0.79532	0.778
Valid Acc 2	0.788	0.795	0.7622
Valid Acc 3	0.8	0.794	0.776

In order to pick the best number of layers, I **average up** the results of the three tests.





Conclusions:

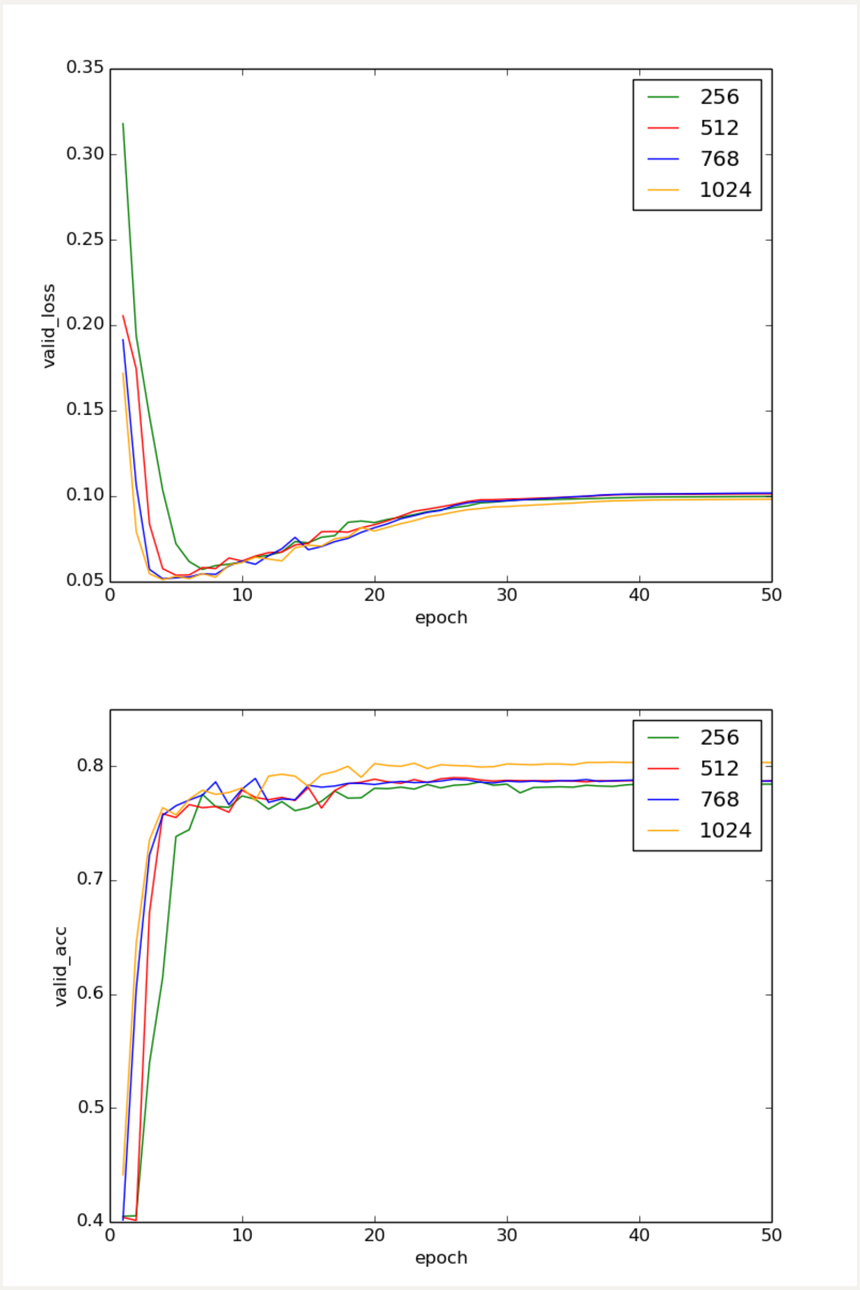
- Overaly, 2-layers outperforms 4-layers by a tiny disparity
    - However, I think 4-layers have the potential to outperform 2-layer if we mask some tokens to seek more stable performance.
- 6-layer may blur the information of input and leads to lower accuracy.
- The model with less fewer layer converges faster since it has less parameters.

3. Hidden Size

I run the comparison for three times with LSTM, num_layer=2, their **highest valid accuracy** is shown as follows.

NUM OF LAYERS	256	512	768	1024
Valid Acc 1	0.787	0.799	0.8022	0.8022
Valid Acc 2	0.782	0.7862	0.793	0.805
Valid Acc 3	0.796	0.806	0.801	0.814

I average up the three test and get the following result:



Conclusion:

- Hidden size = 1024 has the best performance.
    - If we only examine the highest valid accuracy, we may think 768 and 1024 have similar preformance, but through the graph, we can tell the difference from the averaged results.

Bonus: Other effort

I try the following method to improve Q3.

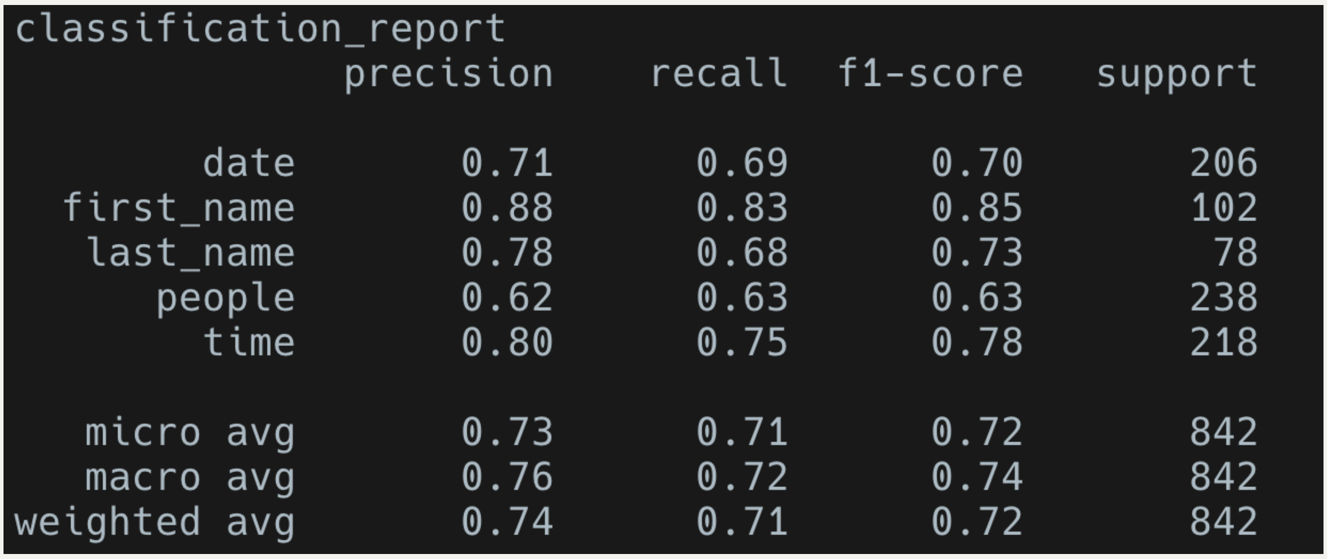
1. Preprocessing

There are originally 9 tags in `tag2idx.json`. The results above are acquired by Method1.

- Method1: Add "" label in `slot_dataset.py`
    - Label of text `["date","23rd","of","may"]` would be `["O","B-date", "I-date", "I-date", "PAD", "PAD", "PAD" ...]` instead of `["O","B-date", "I-date", "I-date", "O", "O, "O" ...]`
- Method2: Base on Method1, additionally add "", "" in `slot_dataset.py` and `class Vocab` in `untils.py`.
    - Text `["date","23rd","of","may"]` would be converted to `["[BOS]","date","23rd","of","may", "[EOS]"]`
    - Label of it would be `["BOS", "O","B-date", "I-date", "I-date", "EOS", "PAD", "PAD" ...]`
    - Note: Remember to rerun `preprocess_slot.py` to get a new vocabulary and `embedding.pt`.

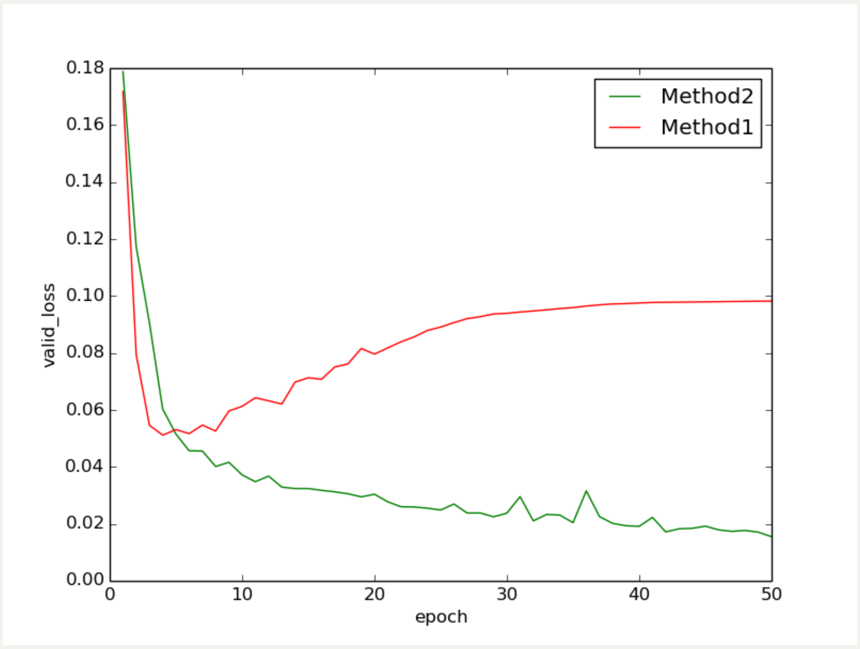
Result

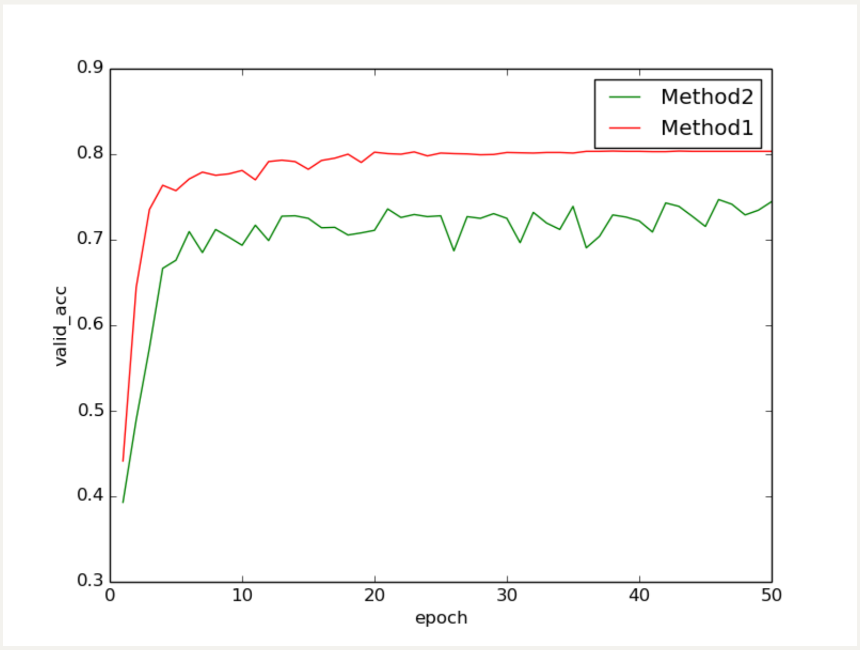
Fail. Its test score = 0.39302, with highest valid accuracy = 0.723.



```
classification_report
              precision    recall  f1-score   support

        date       0.71      0.69      0.70       206
  first_name       0.88      0.83      0.85       102
   last_name       0.78      0.68      0.73        78
      people       0.62      0.63      0.63       238
        time       0.80      0.75      0.78       218

   micro avg       0.73      0.71      0.72       842
   macro avg       0.76      0.72      0.74       842
weighted avg       0.74      0.71      0.72       842
```

I use the model with **LSTM, num_layer=2, hidden_size=1024**, repectively with Method1 and Method2, and average up the results:





Findings

- Method1 is better.
- Method2 performances very baddly. I think it might because the embedding cannot represent the special tokens well. So the tokens are consodered as content and thus mislead the model.
    - Possible solution: Use BertTokenizer which deals with [CLS] and [SEQ].
- The test accuracy is lower than half, which has a great gap with valid accuracy.
- It seems that the valid loss of Method2 hasn't reach the minimum, but the test score is too low so I give up this method.

2. Masking

Motivation

`preprocess_slot.py` only takes the words in `train.json` and `eval.json` into vocabulary. So there must exists some unknown tokens in `test.json`. So I'd like to **boost the robustness** of my model when **facing unknown tokens**.

Method

In training stage, every sample has 50% chance to be masked out 20% tokens. That is, if the 50% chance hits, 20% tokens would be replaced with `Vocab.UNK`.

E.g.: `["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]` might be converted to `["A", "B", "<unk>", "D", "<unk>", "F", "G", "H", "I", "J"]`

My intention to boost the model capability to output correct tags for unknown tokens.

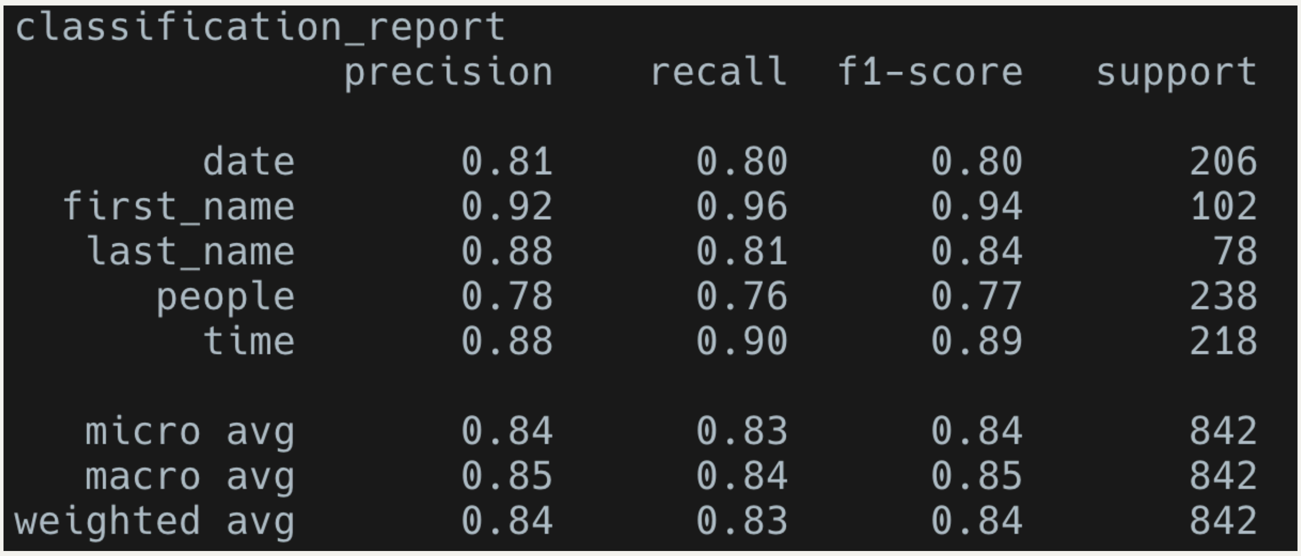
Result

SUCCESS. Test Accuracy is boosted **5.4%** compared to the model without masking.

I use the model with **LSTM, num_layer=2, hidden_size=1024, Method1**, and run the configurations respectively 3 times.

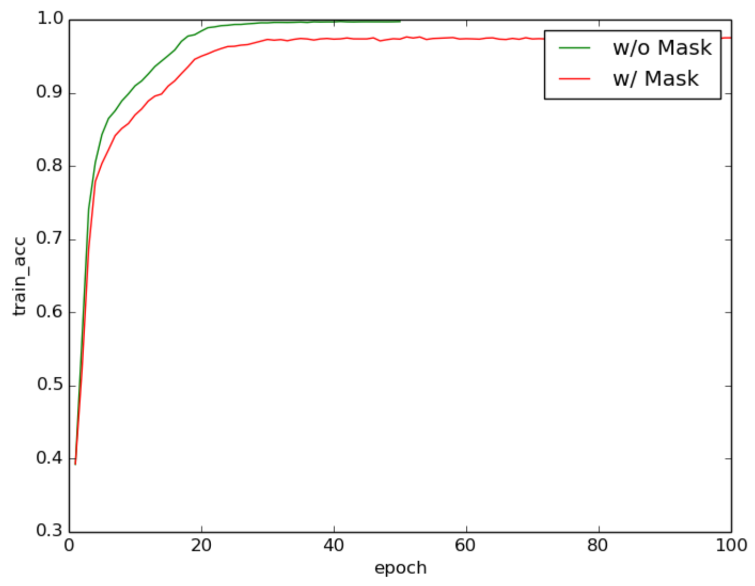
	W/O MASKING	W/ MASKING
Valid Acc 1	0.8022	0.818
Valid Acc 2	0.805	0.819
Valid Acc 3	0.814	0.837
Test Acc	0.75227	0.80697

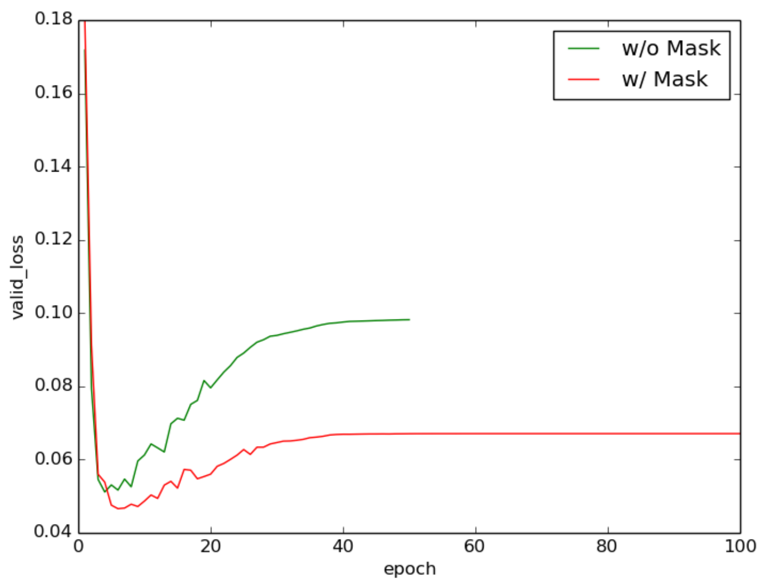
The classification report of Masking method:

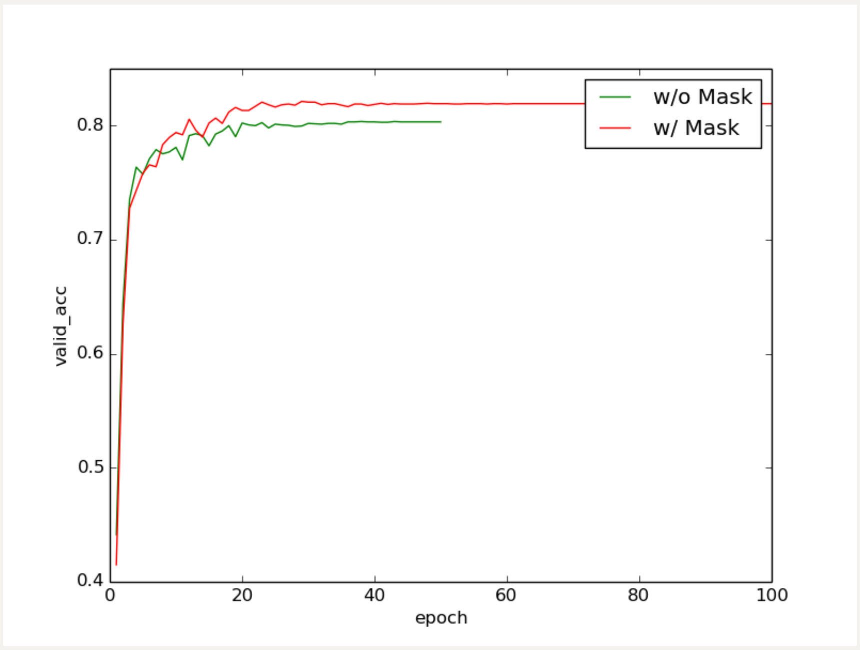


```
classification_report
              precision    recall  f1-score   support

        date       0.81      0.80      0.80       206
  first_name       0.92      0.96      0.94       102
   last_name       0.88      0.81      0.84        78
      people       0.78      0.76      0.77       238
        time       0.88      0.90      0.89       218

   micro avg       0.84      0.83      0.84       842
   macro avg       0.85      0.84      0.85       842
weighted avg       0.84      0.83      0.84       842
```

Comparison of the two configurations:







Finding

- Masking method lowers the train accuracy but boost the valid accuracy. So the model without masking is actually overfitting.
- The test accuracy boost a lot (5.4%), so it seems that there are many words in `test.json` that are not in vocabulary.

Update: This method also improve my test accuracy for intent classification task for 3.2%.