

Report

Student Name: 陳耕宇

Student ID: B07902063

Q1: Data processing

I use the sample code from TA, so the following description is based on the file `preprocess.sh`, `preprocess_intent.py` and `preprocess_slot.py`.

The preprocessing script `preprocess.sh` first downloads and unzip the GloVe embedding from [Stanford-GloVe](#). Next it executes `preprocess_intent.py` and `preprocess_slot.py`.

`preprocess_intent.py`

The python code first reads the training and validation data, collecting all the possible intents, give each of them an integer index, and store the results in file `./cache/intent/intent2idx.json`. The indices are later used in training as the label for each sample. Also, for each sample in training and validation set, all the words in texts are updated into a python `Counter()` container named `words`, which helps count the occurring times of each word.

Next it builds the vocabulary set and embeddings. Note only the first `vocab_size` most common words in `words` are used and stored in this stage, where the default value for `vocab_size` is 10000. The vocabulary set is a python class `Vocab`, which is defined in `utils.py`. This class index all the input words with an integer from 2. The index value `0` and `1` are not used, as they are reserved for padding token and unknown token, respectively. The `Vocab` class also supports methods including but not limited to `tokens()` (listing all the tokens/words), `encode()` (providing the indices of a given list of tokens/words), `encode_batch()` (encoding a batch of list of tokens/words, while padding each list to a given length). The `Vocab` is saved as file `./cache/intent/vocab.pkl`.

The embedding is built based on GloVe. It searches all given words in the vocabulary set. If the word is in GloVe, the embedding vector of this word is used; otherwise, it picks a vector with random coefficients. The result embedding is a list of vectors, where the *i*-th element denotes the embedding vector of the *i*-th word in the vocabulary set. The embedding is then transformed to a PyTorch tensor and saved as file `./cache/intent/embeddings.pt`.

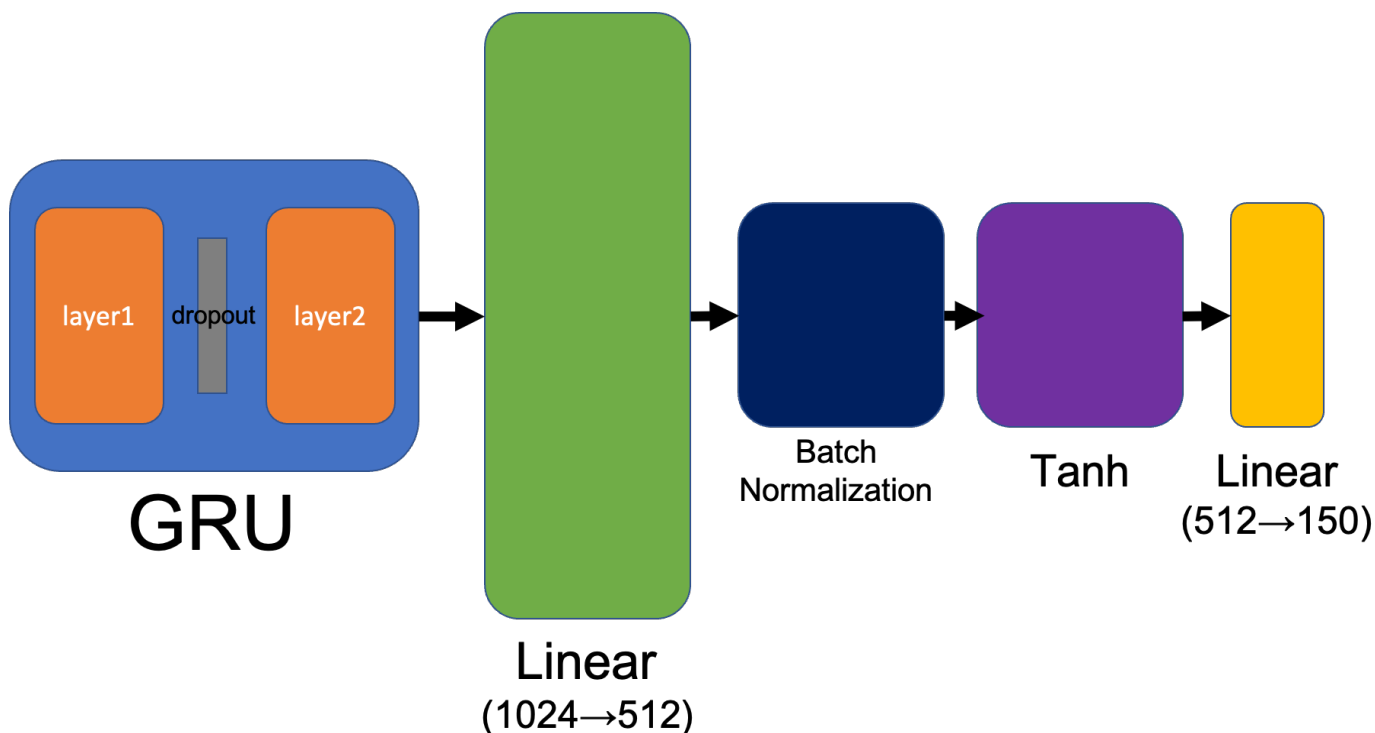
`preprocess_slot.py`

The python code first reads the training and validation data, collecting all the possible tags, give each of them an integer index, and store the results in file `./cache/slot/tag2idx.json`. The indices are later used in training as the label for each token. Next for each sample in training and validation set, all the tokens are updated into a python `Counter()` container named `words`, which helps count the occurring times of each token. Next it builds the vocabulary set and embeddings with `words`, which is basically the same routine as the case in `preprocess_intent.py`.

Q2: Describe your intent classification model

Model Architecture

The model architecture is a GRU network concatenated with two linear layers. The given sample first goes through a GRU, which has input size 300, hidden size 512, 2 layers, a 0.1-probability dropout layer in the middle GRU output layer, and is bi-directional. The output of GRU, sized 1024, then goes through a linear layer with size (1024, 512). Soon after a batch normalization and a `Tanh` activation layer is performed, the sample then goes through the second linear layer with size (512, 150). The output is a batch of vectors, each of which can be viewed as a probability distribution over all possible intents after a log-softmax.



Performance

The public score on kaggle is 0.92355, and the private score is 0.93244.

Loss Function

I use the `CrossEntropyLoss()` function in PyTorch, which computes the negative log of the tensor value corresponding to the correct label and normalized to sum to 1 within a batch.

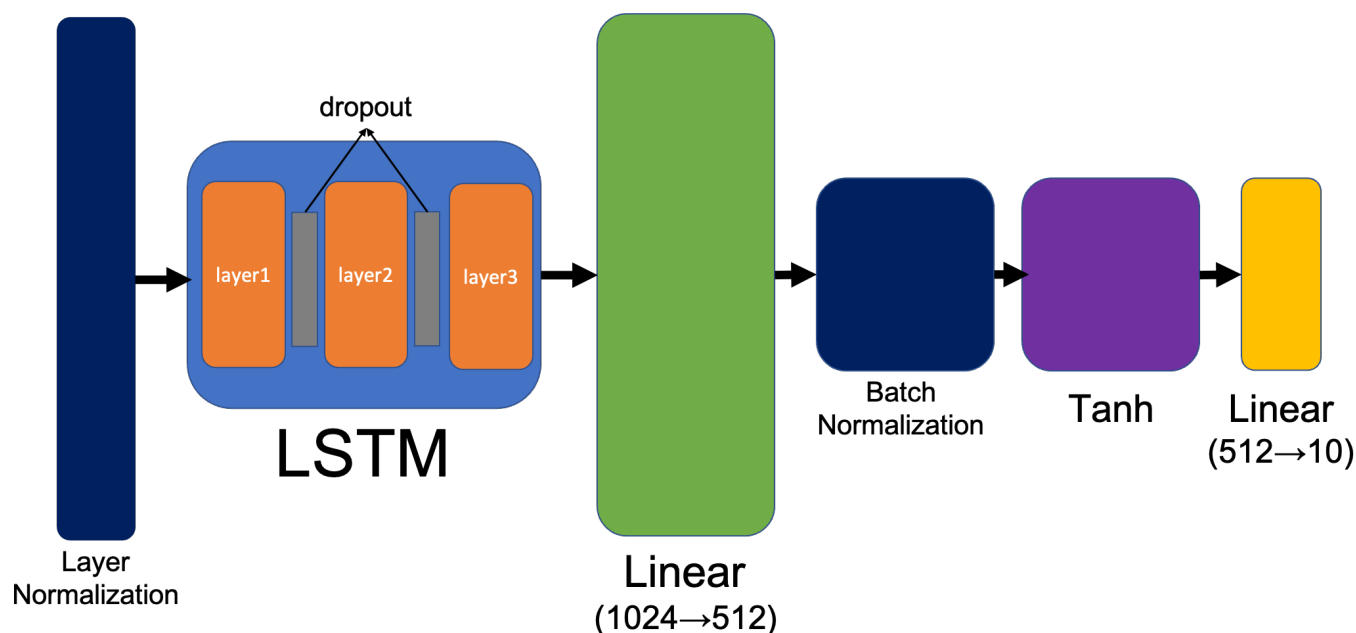
Optimization Algorithm, Learning Rate, and Batch Size

I use the `Adam` optimizer with learning rate 0.001 and betas (0.9, 0.999), which is used to compute the running average and square of gradients. The batch size in each epoch in the training is set to 128.

Q3: Describe your slot tagging model

Model Architecture

The model architecture is a LSTM network concatenated with two linear layers. The given input is first normalized within each sample. Next it goes through a LSTM, which has input size 300, hidden size 512, 3 layers, a 0.1-probability dropout layer in the middle LSTM output layer, and is bi-directional. The output of LSTM, sized 1024, then goes through a linear layer with size (1024, 512). Soon after a batch normalization among samples and a `Tanh` activation layer is performed, the sample then goes through the second linear layer with size (512, 10). The output is a batch of list of vectors, each of which can be viewed as a probability distribution over all possible tags after a log-softmax.



Performance

The public score on kaggle is 0.78605, and the private score is 0.79421.

Loss Function

I use the `CrossEntropyLoss()` function in PyTorch, which computes the negative log of the tensor value corresponding to the correct label and normalized to sum to 1 within a batch.

Optimization Algorithm, Learning Rate, and Batch Size

I use the `Adam` optimizer with learning rate 0.001 and betas (0.9, 0.999), which is used to compute the running average and square of gradients. The batch size in each epoch in the training is set to 128.

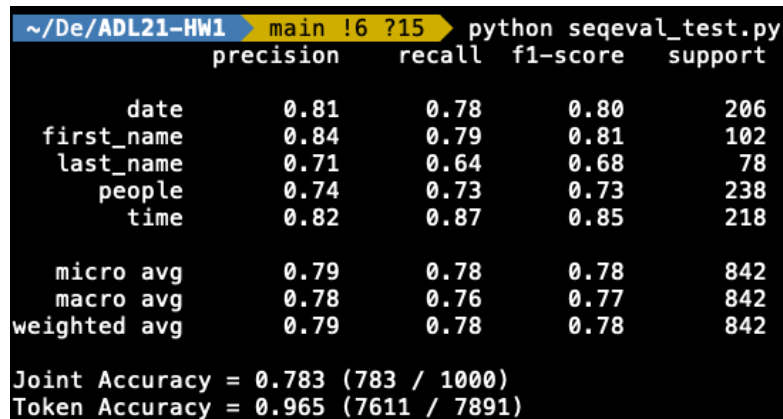
Q4: Sequence Tagging Evaluation

The model I used in the question is trained by

```
1 | python train_slot.py --num_layer 3 --optimizer Adam
```

The `segeval` result is generated by

```
1 | python segeval_test.py
```



	precision	recall	f1-score	support
date	0.81	0.78	0.80	206
first_name	0.84	0.79	0.81	102
last_name	0.71	0.64	0.68	78
people	0.74	0.73	0.73	238
time	0.82	0.87	0.85	218
micro avg	0.79	0.78	0.78	842
macro avg	0.78	0.76	0.77	842
weighted avg	0.79	0.78	0.78	842
Joint Accuracy	= 0.783 (783 / 1000)			
Token Accuracy	= 0.965 (7611 / 7891)			

which shows the result of `segeval` with `classification_report(scheme=IOB2, mode='strict')`, and the joint and token accuracy.

The metrics in `classification_report` represent

- precision: the number of correctly labeled entities / total number of predicted entities
- recall: the number of correctly labeled entities / total number of labeled entities
- f1-score: $(2 * \text{precision} * \text{recall}) / (\text{precision} + \text{recall})$
- support: the number of entities(date, first_name, etc.)
- micro avg: average over all the samples
- macro avg: average over all the labels
- weighted avg: same as micro avg in this case

Note that `classification_report` only shows the result about entities; for example, the accuracy about "O" flag will not be considered.

The joint accuracy is the number of correctly labeled sample / total number of all samples, which only counts how many samples in which all labels are correctly labeled. The token accuracy is the number of correctly labeled token / total number of tokens, which just counts how many tokens(regardless of samples) are correctly labeled.

As one can see, the token accuracy is high, but the precision, recall, f1-score and the joint accuracy are not so positive. In tagging problem, counting of all correctly labeled tokens has little meaning, while we consider a correctly labeled entity or correctly labeled sample more valuable.

Q5: Compare with different configurations

In this question the intent classification problem is used as example to illustrate. The training time denotes the spending time for 100 epochs, and the speed of convergence represents the number of epochs at which the model cannot perform better on validation set. The performance is characterized by validation set accuracy and private test score. All experiments are made on the MacBook Pro laptop with M1-pro chip, and number of epochs is limited to 100.

For testing different models, I tried vanilla-RNN, LSTM and GRU model for this homework. All of them are concatenated two linear layers, within which a batch normalization and `Tanh` activation layer is placed, and the `Adam` optimizer is used. The following table shows the result.

	training time (m:s)	speed of convergence (epoch)	performance (validation / private test)
vanilla RNN	22:32	20 epochs	0.031 / x
LSTM	72:32	7 epochs	0.925 / 0.90844
GRU	59:44	9 epochs	0.943 / 0.92755

As we see, the training time of vanilla RNN is a lot faster than that of LSTM and GRU, which directly stems from the complexity of model architectures, but its performance is much poorer, which is also due to the model ability to exploit data. The reason for GRU better than LSTM may be an outcome of overfitting, not sufficiently fine-tuned parameters for both models, or this problem is simply better interpreted by GRU.

For testing different optimizers, the PyTorch `SGD` and `Adam` are tried with two learning rates 0.1 and 0.001. The experiments are conducted with GRU model described above. The result is in the following table.

	learning rate	training time (m:s)	speed of convergence (epoch)	performance (validation / private test)
SGD	0.1	55:53	69 epochs	0.935 / 0.92088
SGD	0.001	57:39	Not convergent in 100 epochs	0.897 / 0.87911
Adam	0.1	59:44	Not convergent in 100 epochs	0.307 / x
Adam	0.001	59:44	9 epochs	0.943 / 0.92755

Obviously, the simple `SGD` optimizer not just converge slowly, but also suffer the problem that model cannot achieve minimum with large learning rate. On the other hand, the `Adam` optimizer effectively enhance the speed of convergence while achieve a minimum in few epochs with proper learning rate.