ADL HW1 Report R11944004

Student ID: R11944004

Department: GINM 11

Name: 李勝維

Q1: Describe how do you use the data for intent_cls.sh, slot_tag.sh:

How do you tokenize the data

I use the provided sample code to tokenize the data.

For intent classification, each sentence is splited into tokens by whitespace (using str.split() method).

As for slot tagging, the sentences are pre-tokenized to avoid ambiguity with the labels.

After tokenization, we count the frequency of each word in the corpus, and selects top 10K frequent words as our dictionary. <PAD> and <UNK> tokens are added afterwards.

The pre-trained embedding you used

The word2vec embeddings I use is GloVe: Global Vectors for Word Representation. The embeddings are trained on Common Crawl dataset with 840B tokens, results in 2.2M vocabulary size and 300-dim vectors: glove.840b.300d.zip

Q2: Describe your intent classification model.

Model Architecture

The model consists of three learnable components: word embeddings, Bi-LSTM, and a linear classifier.

First, the input words are transformed into word vectors by pre-trained word embeddings.

Second, the embeddings are fed into a two-layer bidirectional LSTM with 512 dimension hidden states and 20% dropout rate.

Third, I obtain the sequence representation by averaging all token representations in the sequence.

Finally, I fed the sequence representation into a one-layer linear classifier and softmax to predict the intent of the sentence.

Formally, given an input sequence:

$$X = \{x_0, x_1, ..., x_T\}, \text{where T is the sequence length}$$

The model can be represented as:

$$\begin{gathered} W = \{w_0, w_1, ...w_T\} = Embedding(X) \\ h_t, c_t = forward\text{-}LSTM(w_t, h_{t-1}, c_{t-1}) \\ h'_t, c'_t = backward\text{-}LSTM(w_t, h'_{t+1}, c'_{t+1}) \\ h_{seq} = \frac{1}{T}\sum\{(h_0, h'_0), (h_1, h'_1), ..., (h_T h'_T)\} \in \mathbb{R}^{1024} \\ logits = linear\_clf(h_{seq}) \in \mathbb{R}^{150} \\ pred\_intent = argmax(softmax(logits)) \end{gathered}$$

where W represents the word embeddings, h_t and h'_t represents the hidden state of forward LSTM and backward LSTM for t-th token respectively, and (h_t, h'_t) represents contatenation.

Performance

The model scored **0.90266** accuracy on kaggle public test data.

Loss function

The loss function I use is the cross entropy between the ground truth intent and the model's prediction.

$$loss = CrossEntropyLoss(softmax(logits), gt)$$

where gt stands for intent ground truth.

Optimization algorithm

I use Adam as my optimizer, with:

- learning rate = 1e-3
- batch size = 128

Besides Adam, I also use one cycle lr scheduler introduced in [1708.07120] Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates (arxiv.org) with "pct_start" = 0.3, the percentage of the cycle spent increasing the lr, to control the learning rate.

Q3: Describe your slot tagging model.

Model Architecture

Similar to Q2, the model consists of three learnable components: word embeddings, Bi-LSTM, and a linear classifier.

First, the input words are transformed into word vectors by pre-trained word embeddings.

Second, the embeddings are fed into a two-layer bidirectional LSTM with 512 dimension hidden states and 20% dropout rate.

Finally, I fed the each token representation into a one-layer linear classifier and softmax to predict the tag of the token.

Formally, given input sequence:

$$X = \{x_0, x_1, ..., x_T\}, \text{where T is the sequence length}$$

The model can be represented as:

$$\begin{aligned} W &= \{w_0, w_1, ...w_T\} = Embedding(X) \\ h_t, c_t &= forward\text{-}LSTM(w_t, h_{t-1}, c_{t-1}) \\ h'_t, c'_t &= backward\text{-}LSTM(w_t, h'_{t+1}, c'_{t+1}) \\ H &= \{(h_0, h'_0), (h_1, h'_1), ..., (h_T, h'_T)\} \in \mathbb{R}^{T \times 1024} \end{aligned}$$

$$\begin{aligned} &\text{for each token, } H_t = (h_t, h'_t) \\ &\text{do} \\ &\ logits = linear\_clf(H_t) \in \mathbb{R}^9 \\ &\ pred\_tag = argmax(softmax(logits)) \\ &\text{end do} \end{aligned}$$

where W represents the word embeddings, h_t and h'_t represents the hidden state of forward LSTM and backward LSTM for t-th token respectively, and (h_t, h'_t) represents contatenation.

Performance

The model scored **0.72493** joint accuracy on kaggle public test data.

- After tuning of Q5, performance is boosted to **0.78820**

Loss function

The loss function I use is cross entropy, which is calculated based on the ground truth tag and the model's prediction for each token.

$$loss = CrossEntropyLoss(softmax(logits), gt)$$

where gt stands for tag ground truth.

Optimization algorithm

I use Adam as my optimizer, with:

- learning rate = 1e-3
- batch size = 128

Besides Adam, I also use one cycle lr scheduler introduced in [1708.07120] Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates (arxiv.org) with "pct_start" = 0.3, the percentage of the cycle spent increasing the lr, to control the learning rate.

Q4: Sequence Tagging Evaluation

```
joint acc: 0.779000000000000
token acc: 0.963249271321759
              precision    recall  f1-score   support

        date       0.75      0.77      0.76       206
  first_name       0.94      0.89      0.91       102
   last_name       0.85      0.85      0.85        78
      people       0.71      0.71      0.71       238
        time       0.79      0.83      0.81       218

   micro avg       0.78      0.79      0.78       842
   macro avg       0.81      0.81      0.81       842
weighted avg       0.78      0.79      0.78       842
```

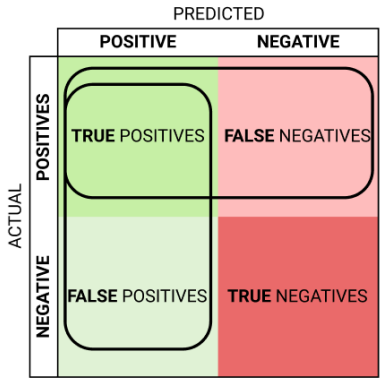
Joint accuracy vs token accuracy

Joint accuracy = (correctly predicted **sequence**) / (number of all sequences)

Token accuracy = (correctly predicted **token**) / (number of all tokens)

For joint accuracy, even as little as a single difference between the predicted sequence and target sequence, the accuracy will be 0; as for token accuracy, the accuracy will be (n-1) / n, where n is the sequence length.

Precision, recall and f1-score



TP = True Positive, FP = False Positive, TN = True Negative and FN = False Negative for short.

Precision = TP / (TP + FP) = TP / (model predicts as positive)

Precision is the ratio of how many samples that the model predicts as positive are positive.

Recall = TP / (TP + FN) = TP / (positive samples)

Recall is the ratio of how many positive samples, that are predicted as positive by the model.

F1-score = 2 * (Precision * Recall) / (Precision + Recall), which is the harmonic mean of the two.

F1-score considers precision and recall at the same time.

Micro, macro and weighted average

First, we know that “support” means the occurrences of each class in ground truth.

Micro average: Each **sample** contributes to the mean equally. In other words, calculate metrics globally by counting the total true positives, true negatives, false negatives and false positives.

Macro average: Each **class** contributes to the mean equally. In other words, calculate metrics for each class, and find their unweighted mean.

Weighted average: Calculate metrics for each class, and find their mean weighted by the size of support set for each class.

Q5: Compare with different configurations

Since slot tagging task is much more difficult than sequence classification task and limitations on computing resource (colab only 😭), I will conduct hyperparameter search only on slot tagging.

TL;DR

- Implement CNN-BiGRU and experiment different CNN architectures
- Experiment different settings of:
    1. hidden_size
    2. num_layers
    3. Dropout ratio
    4. RNN design (Bi-GRU / LSTM / Bi-LSTM)
    5. Model architecture (overall comparision)
- After tuning, boosts kaggle public score from 0.72493 to **0.78820**

BONUS trick — CNN-BiGRU on slot tagging

Implements CNN-BiGRU, which places a CNN before RNN, the first part of the model can be describe as:

$$W = \{w_0, w_1, ...w_T\} = Embedding(X)$$
$$W' = 1D\text{-}CNN(W)$$
$$h_t, c_t = forward\text{-}GRU(w'_t, h_{t-1}, c_{t-1})$$
$$h'_t, c'_t = backward\text{-}GRU(w'_t, h'_{t+1}, c'_{t+1})$$
$$H = \{(h_0, h'_0), (h_1, h'_1), ..., (h_T, h'_T)\} \in \mathbb{R}^{T \times 1024}$$

1D-convolution settings:

- kernel size = 5
- stride = 1
- padding = 2

Training settings:

- batch size = 128
- learning rate = 1e-3 + OneCycleLR
- training epoch = 100, saves model on highest validation joint accuracy

RNN settings:

- block type = GRU
- hidden_size = 1024
- num_layers = 2
- dropout = 0.2
- bidirectional = True

#	CNN layers	Validation joint accuracy
1	0 (without CNN)	0.804
2	1	0.810
3	2	0.806

From the experiments, one layer of CNN is beneficial to the model. Though two layers of CNN performs slightly worse than one layer of CNN, it still outperforms the original GRU model.

The convolution layer can help capture local features and reduce the effect of noise.

Improving model by hyperparameter search

Below experiments follows the same training and CNN setting:

- batch size = 128
- learning rate = 1e-3 + OneCycleLR
- training epoch = 100, saves model on highest validation joint accuracy
- Two layers of 1D-CNN with kernel size=5, stride=1, padding=2 before RNN
  (This hyperparameter search is conducted before the search of CNN architecture, thus it uses two-layers of 1D-CNN)

Comparing different hidden_size on slot tagging

#	hidden_size	num_layers	dropout	bidirectional	Validation joint accuracy
1	128	2	0.2	True	0.784
2	256	2	0.2	True	0.787
3	512	2	0.2	True	0.796
4	1024	2	0.2	True	0.806
5	2048	2	0.2	True	0.797

The result shows that with increasing hidden_size, the model scaling up well. This is probably due to increased model complexity allows the model to handle more complicated tasks.

For model #5, the model becomes too large and overfitting occurs, thus the validation accuracy decreases.

Comparing different num_layers on slot tagging

#	hidden_size	num_layers	dropout	bidirectional	Validation joint accuracy
1	512	1	N/A	True	0.778
2	512	2	0.2	True	0.796
3	512	3	0.2	True	0.790
4	512	4	0.2	True	0.787

The result shows that from model #1 to #2, the validation accuracy increases, but after model #3, the validation accuracy decreases .

In my opinion, model #1 underfits the training data, while model #3 and #4 overfits, which means we do not have sufficient training data to train a model such large.

Comparing different dropout ratio on slot tagging

#	hidden_size	num_layers	dropout	bidirectional	Validation joint accuracy
1	512	2	0.1	True	0.784
2	512	2	0.2	True	0.796
3	512	2	0.3	True	0.788
4	512	2	0.4	True	0.789

Dropout is a regularization method. Stronger regularization usually means a more generalized model, but regularization could also makes the model underfit.

We can see that from model #1 to #2, the validation accuracy increases. The reason is model #2 is more generalized than model #1.

On the other hand, the validation accuracy decreases after model #3, this is because underfitting occurs.

Comparing different RNN design on slot tagging

#	hidden_size	num_layers	dropout	bidirectional	RNN block	Validation joint accuracy
1	512	2	0.2	True	LSTM	0.796
2	512	2	0.2	False	LSTM	0.786
3	512	2	0.2	True	GRU	0.795

Unidirectional LSTM performs worse than bidirectional LSTM, this is mainly due to its lower model complexity. Plus, slot tagging benefits from bidirectional informations.

GRU performs similarly compared to LSTM (difference is within the margin of error), but GRU has less trainable parameters and trains faster than LSTM on my machine (16.54 vs 12.57 batch/s). This makes GRU a great alternative for LSTM.

Comparing different model architectures on slot tagging

#	hidden_size	num_layers	dropout	bidirectional	RNN block	Validation joint accuracy
1	1024	2	0.2	True	LSTM	0.806
2	1024	4	0.2	True	LSTM	0.803
3	1024	2	0.2	True	GRU	0.806

Model #2 has the highest model complexity, but overfits the training data.

Meanwhile, LSTM and GRU has similar performance, but GRU trains faster and has less trainable parameters.

Q5 Summary

Parameters:

- Training
  - batch size = 128
  - learning rate = 1e-3 + OneCycleLR
  - training epoch = 100, saves model on highest validation joint accuracy
- CNN
  - One layer of 1D-CNN
  - kernel size = 5
  - stride = 1
  - padding = 2
- RNN
  - Gated Recurrent Unit
  - hidden_size = 1024
  - num_layers = 2
  - dropout = 0.2
  - bidirectional = True

Results:

- After tuning, boosts kaggle public score from 0.72493 to **0.78820**