Applied Deep Learning

Homework 1

學號：ntnu_40747026S
系級：資工系 111 級
姓名：洪偉倫

Q1. Data processing

I used sample code.
As far as I understand, in preprocess_intent.py and preprocess_slot.py, we pick 10000 most common words to build the Vocab class object, which is stored in "./cache/{intent or slot}/vocab.pkl". Then, use those words to find the corresponding vector in GloVe to create the "./cache/{intent or slot}/embedding.pt".
In training program, through vocab.pkl we can encode words to a number, then use embedding.pt to convert those numbers to 300-dimension vectors.

Q2: Describe your intent classification model.

a. Model

I used concepts of BERT to build my model for this task.
For inputs, I preprocessed texts with the vocab.pkl and pad them to args.max_len (28) before feeding into the model; for output, I encoded labels to one-hot.
From the start:

```
x is encoded texts input      # Shape: (batch_size=128, text_length=28)
x = embedding(x)              # Shape: (128, 28) -> (128, 28, 300)
x = position-encoding(x)
repeat for args.number_layers times:
    x_tmp = Multi-Head-Attention(x, x, x, mask)
    x += dropout(x_tmp)
    x = layer-normalization(x)
    x_tmp = Pointwise-Feed-Forward-Network(x)
    x += dropout(x_tmp)
    x = layer-normalization(x)
x = dropout(x)
```

```
x = dense(x) for serval times       # Shape: (128, 28, 300) -> (128, 28, 1)
x = reshape(x)                      # Shape: (128, 28, 1) -> (128, 28)
x = dropout(x)
x = dense(x) for serval times       # Shape: (128, 28)-> (128, 150)
x = softmax(x)                      # Gets the classification probability
```

b. Performance

Score: 0.89288

c. Loss function

tf.keras.losses.CategoricalCrossentropy() with default settings

d. Optimization algorithm

I used Adam with tf.keras.optimizers.schedules.ExponentialDecay() which:

Initial learning rate: 2e-4

Decay rate: 0.85

Batch size: 128

Q3: Describe your slot tagging model.

a. Model

I used transformer for this task, but got a 0% accuracy and I can't figure out where's the problem...

I add the original tokens/labels with a START token/label at the beginning and an END token/label in the end, then pad them to args.max_len (37).

From the start:

```
x is encoded tokens input    # Shape: (batch_size=128, token_len=37)
y is encoded labels input    # Shape: (batch_size=128, token_len=37)

# Encoder part
x = embedding(x)             # Shape: (128, 37) -> (128, 37, 300)
x = position-encoding(x)
repeat for args.number_layers times:
    x_tmp = Multi-Head-Attention(k=x, q=x, v=x, mask)
    x += dropout(x_tmp)
    x = layer-normalization(x)
    x_tmp = Pointwise-Feed-Forward-Network(x)
    x += dropout(x_tmp)
    x = layer-normalization(x)

# Decoder part
y = embedding(y)                    # Shape: (128, 37) -> (128, 37, 300)
```

```
y = position-encoding(y)
repeat for args.number_layers times:
    y_tmp = Multi-Head-Attention(k=y, q=y, v=y, mask)
    y += dropout(y_tmp)
    y = layer-normalization(y)
    y_tmp = Multi-Head-Attention(k=y, q=x, v=x, mask)
    y += dropout(y_tmp)
    y = layer-normalization(y)
    y_tmp = Pointwise-Feed-Forward-Network(y)
    y += dropout(y_tmp)
    y = layer-normalization(y)
y = dropout(y)
y = dense(y) for serval times     # Shape: (128, 37, 300) -> (128, 37, 12)
y = softmax(y)                    # Gets the classification probability
```

b. Performance

Accuracy is 0%, so I didn't upload the prediction onto the Kaggle website.
Precisely, all output labels are O.
In loss function, I ignored the output whenever the input is a padding token, so it is not strange that the model predicts a label O for padding tokens.
However, there are END tokens in inputs, the model can't even learn to predict corresponding END labels. That is what I couldn't understand.

c. Loss function

tf.keras.losses.SparseCategoricalCrossentropy().

d. Optimization algorithm

I used Adam with tf.keras.optimizers.schedules.ExponentialDecay().

Q4: Sequence Tagging Evaluation

Q5: Compare with different configurations

For the intent classification part:

I tested with different agrs.num_layers, dropout.
For agrs.num_layers, I tested numbers 2~8 and found out that 4 gives the best performance.
For dropout, I tested 0.05, 0.1, 0.15, 0.2, and 0.25. 0.1 gives the best performance.