ADL HW1

Q1 : Data processing

- **vacab and embedding**
  Intent cls
  1. load data from file_name.json
  2. create intent2idx which is a dictionary {intent : intent's idx}, and save as save_path/intent2idx.json
  3. create vocab by common word, where vocab.token2idx is a dictionary {token : token's idx}
  4. save vacab as vocab_save_path/vocab.pkl
  5. load pre-trained glove (http://nlp.stanford.edu/data/glove.840B.300d.zip)
  6. create embedding, embedding[token's idx] = token's vector from glove.
  7. save embedding as output_dir/embeddings.pt

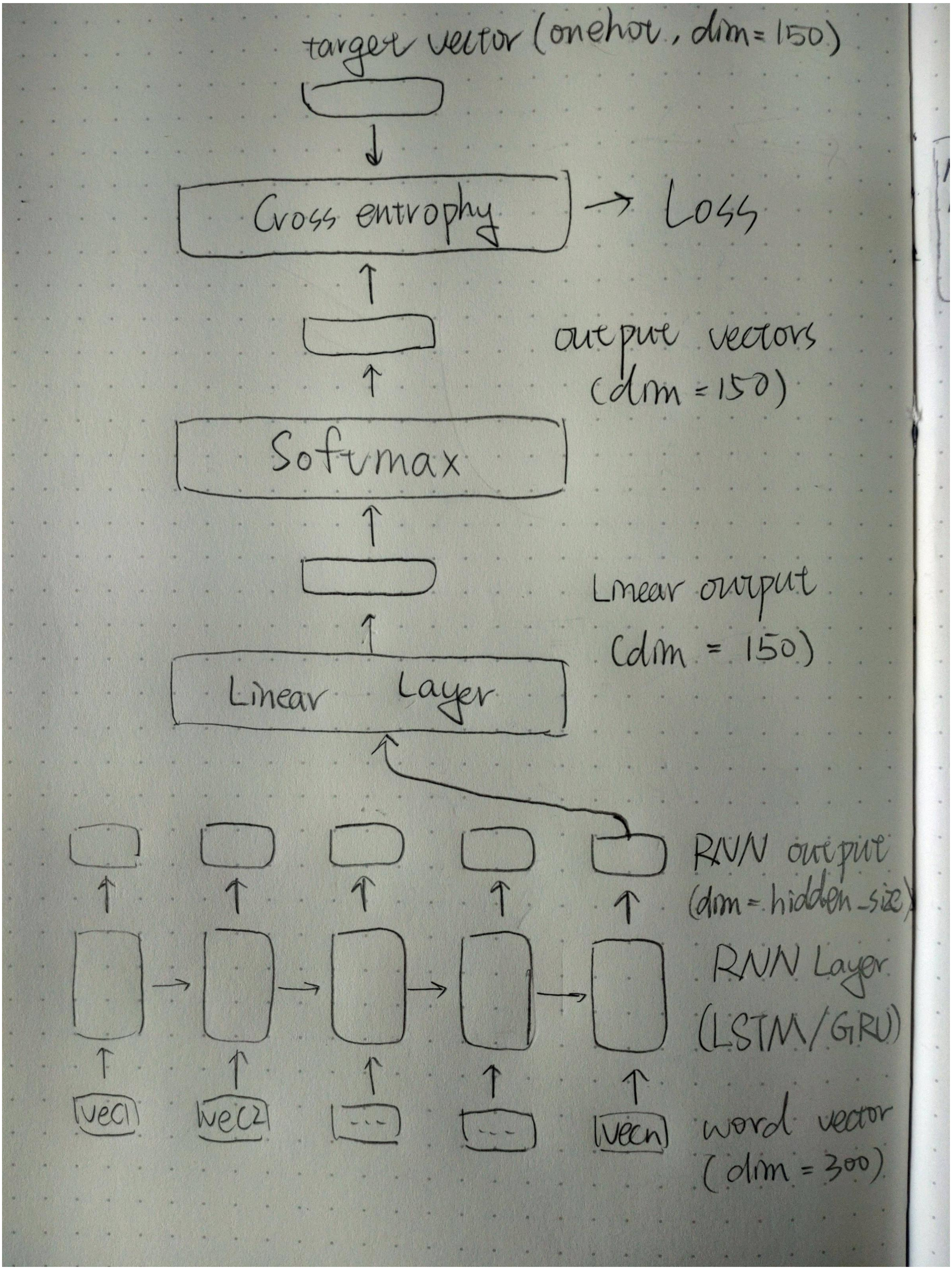
  Slot tag
  1. load data from file_name.json
  2. create vocab by common word, where vocab.token2idx is a dictionary {token : token's idx}
  3. save vacab as vocab_save_path/vocab.pkl
  4. load pre-trained glove (http://nlp.stanford.edu/data/glove.840B.300d.zip)
  5. create embedding, embedding[token's idx] = token's vector from glove.
  6. save embedding as output_dir/embeddings.pt
- **From row data to tensor of batch**
  Intent cls
  1. Load data from file_name.json
  2. Load files saved by preprocess
  3. Create dataset by data, and build dataloader by dataset
  4. To iterate the dataloader, get iterator i. i is a dictionary, i['text'] is a list of str(sentence), i['intent'] is a list of str(target class) (there are batch_size elements)
  5. Split i['text'] by space and got every word vector by embedding using vocab.token2idx
  6. Sort them by sentence's length and store the length
  7. Padded them by torch.utils.rnn.pad_sequence(), size = (batch_size, max length, 300)
  8. Packed the padded sequence by torch.utils.rnn.pack_padded_sequence() as the model's input
  9. Create one hot vector for each intent by intent2idx. Stack them into a tensor, size = (batch_sze, 150), store them as target tensor

  Slot tag
  1. Load data from file_name.json
  2. Load files saved by preprocess
  3. Create dataset by data (data is different from intent cls, so I join the tokens by space), and build dataloader by dataset
  4. To iterate the dataloader, get iterator i. i is a dictionary, i[tokens] is a list of str(sentence), i['tags'] is a list of str(target class) (there are batch_size elements)
  5. Create packed padded sequence as Intent cls, and use it as model's input, size = (batch_size, max length, 300)
  6. Create the mask where size = (batch_size, max length) and mask[i][j] = 0 if model's input[i][j] is padded, else mask[i][j] = 1

7. For every tokens, I get the tag label by tag2idx, and build a target which is a pad_sequence of size = (batch_size, max length, 1). target[i][j] = [tag label of the coresponded word]. It will be 0 for the padded words

- **Out of vocab**
  For every word don't exist in vocab, I suppose it to be zero vector.

Q2 : Describe your intent classification model



(a) model

- RNN Layers
  parameter :
  --model_type : 0 for basic RNN, 1 for LSTM, 2 for GRU
  --input_size : dimention of word vector, default = 300
  --num_layers : number of rnn layer, default = 1
  --dropout : use an rnn layers as dropout layer(num_layers must bigger than 1)
  --hidden_size : the size of hidden layers, size of each output vector = hidden_size, default 512

  denote max sentence's length of a batch = max_l
  input : pack_padded_sequence of data size = (batch_size, max_l, 300)
  output : pack_padded_sequence of data size = (batch_size, max_l, hidden_size)

  For every sentence, I only choose the last output of rnn layer.
  For example, the ith sentence with length=j gets output[i - 1][j - 1] (-1 because start form 0)

  After choosing, output becomes tensor with size = (batch_size, hidden_size)

- Linear Layer
  parameter :
  --in_features : size of each input vector, set hidden_size
  --out_features : size of each output vector, set 150

  input : tensor with size = (batch_size, hidden_size)
  output : tensor with size = (batch_size, 150)

- Softmax layer
  parameter :
  --dim : softmax aloge with which dimention, because the input size = (batch_size, 150) which is 2D, set dim = 1 (start with 0)

  input : tensor with size = (batch_size, 150)
  output : tensor with size = (batch_size, 150), sum of elements of output[i] = 1

- training
  --save_name : save the model as save_name.ckpt
  --batch_size : how many data to collect for a single optimizing
  --num_epoch : how many times to go through the train dataset

(b) preformace
Private score : 0.91333
Public score: 0.91822

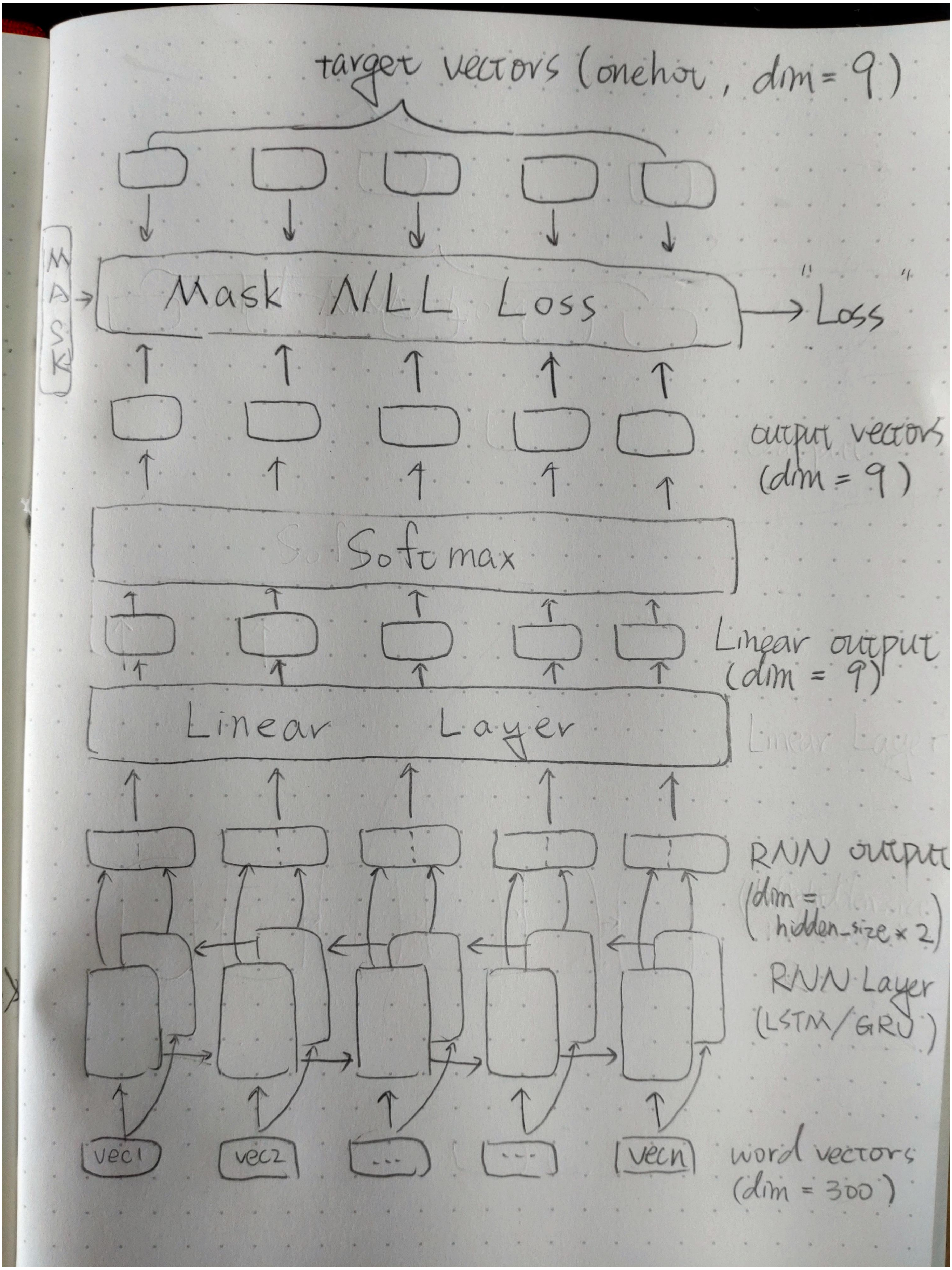
- paramiter:
  --input_size = 300
  --num_layers = 2
  --dropout = 0.4
  --hidden_size = 512
  --model_type = 1
  --batch_size = 512
  --num_epoch = 200

(c) loss function
torch.nn.CrossEntropyLoss()

input = tensor, size = (batch_size, 150)
output = tensor, size = (batch_size, 150)

(d) optimization
torch.optim.Adam()
parameter:
--lr : learning rate, default = 0.001

Q3 : Describe your slot tagging model



(a) model

- RNN Layers
  parameter :
  --model_type : 0 for basic RNN, 1 for LSTM, 2 for GRU
  --input_size : dimention of word vector, default = 300
  --num_layers : number of rnn layer, default = 1
  --dropout : use an rnn layers as dropout layer(num_layers must bigger than 1)
  --hidden_size : the size of hidden layers, size of each output vector = hidden_size, default 512
  --bidirectional : always true to create a bidirectional rnn

  denote max sentence's length of a batch = max_l
  input : pack_padded_sequence of data size = (batch_size, max_l, 300)
  output : pack_padded_sequence of data size = (batch_size, max_l, hidden_size * 2)
  transform the pack_padded_sequence to tensor of size = (batch_size, max_l, hidden_size * 2) as input of Linear Layer

- Linear Layer
  parameter :
  --in_features : size of each input vector, set hidden_size
  --out_features : size of each output vector, set 150

  input : tensor with size = (batch_size, max_l, hidden_size * 2)
  output : tensor with size = (batch_size, max_l, 9)

- Softmax layer
  parameter :
  --dim : softmax aloge with which dimention, because the input size = (batch_size, max_l, 150) which is 3D, set dim = 2 (start with 0)

  input : tensor with size = (batch_size, max_l, 9)
  output : tensor with size = (batch_size, max_l, 9), sum of elements of output[i][j] = 1

- training
  --save_name : save the model as save_name.ckpt
  --batch_size : how many data to collect for a single optimizing
  --num_epoch : how many times to go through the train dataset

(b) preformace
Private score : 0.82636
Public score: 0.80643

- paramiter:
  --model_type = 2
  --input_size = 300
  --num_layers = 3
  --dropout = 0.5
  --hidden_size 256
  --batch_size = 128
  --num_epoch = 150

(c) loss function

- paramiter:
  --predict : tensor of size = (batch_size, max_l, 9), predict[i - 1][j - 1] means the probability distribution over 9 class of the jth word in ith sentence
  --target : tensor of size = (batck_size, max_l, 1), target[i - 1][j - 1][1] means the correct class label of jth word of ith sentence
  --mask : tensor of size = (batch_size, max_l), if mask[i - 1][j - 1] = 1 means jth word of ith sentence exist , else mask[i - 1][j - 1] = 0
- calculate:
  loss_before_select = -torch.log(torch.gather(predict, 2, target).squeeze())
  loss_after_select = loss.masked_select(mask).sum()

  loss_before_select is a tensor of size = (batch_size, max_l)
  loss_before_select[i][j] = -log(predict[i][j][target[i][j][1]])
  For example, a word w whose class label is 3. I get the p where p is the probability of w being 3 from predict. Finally, calculate -log(p).

  loss_after_select is simply calculate the sum of the loss_before_select[i][j] where mask[i][j] = 1
  It can avoid the useless imformation from data after padding

(d) optimization
torch.optim.Adam()
parameter:
--lr : learning rate, default = 0.001

Q4 : Sequence Tagging Evaluation

```
b09902128@meow2 [/tmp2/B09902128] python repo_slot.py --eval_file ./data/slot/eval.json
--pred_file ./pred_slot.csv
              precision    recall  f1-score   support

        date       0.78      0.78      0.78       206
  first_name       0.98      0.87      0.92       102
   last_name       0.90      0.82      0.86        78
      people       0.75      0.74      0.75       238
        time       0.89      0.89      0.89       218

   micro avg       0.83      0.81      0.82       842
   macro avg       0.86      0.82      0.84       842
weighted avg       0.83      0.81      0.82       842
```

- **TP, FN, TN, FP**
  - TP (true positive) : we say it's positive and we are right.
  - FN (false negitive) : we say it's negitive and we are wrong.
  - TN (true negitive) : we say it's negitive and we are right.
  - FP (false positive) : we say it's positive and we are wrong.
- **precision**
  p = $\frac{TP}{TP+FP}$
  When **FP** is important, we should consider precision more.
- **recall**
  rc = $\frac{TP}{TP+FN}$
  When **FN** is important, we should consider recall more.
- **f1-score**
  f1-score = $2 \times \frac{p \times rc}{p+rc}$

$$\frac{p+rc}{2} \geq \sqrt{p \times rc} \Rightarrow 1 \geq 2 \times \frac{\sqrt{p \times rc}}{p+rc} \Rightarrow \sqrt{p \times rc} \geq 2 \times \frac{p \times rc}{p+rc}$$

$$\because p, rc \leq 1$$

$$\therefore \sqrt{p \times rc} \leq 1$$

$$f1 - score = 2 \times \frac{p \times rc}{p+rc} \leq 1$$

$$if \quad p = 1, rc = 1, \quad f1 - score = 1$$

- **my model**
  for data and time, precision = recall. It means the prediction is quite balance for FP and FN.
  for first_name, last_name, people precision > recall. It means the prediction gets more FN than FP.

Q5: Compare with different configurations

Intent cls

- model
  First, I test rnn/lstm/gru by default parameters and num_poch=200 batch_size=256

model	private score	public score
RNN	0.85422	0.83600
LSTM	0.90933	0.90888
GRU	0.89911	0.90577

I decide to use lstm and gru as the further rnn layer.

- dropout
  For the local test, I test the dropout = [0.1, 0.2, 0.3, 0.4, 0.5]
  dropout = 0.4 has the best preformance with eval_data. Both GRU and LSTM improve.

model(dropout = 0.4)	private score	public score
LSTM	0.91822	0.91333
GRU	0.90888	0.90577

Because LSTM is better than GRU on the two test. So I decide choose LSTM.

- hidden_size
  hidden_size = 512 or 256 has the best preformance.

hidden size	private score	public score
128	0.90800	0.90400
256	0.91466	0.90133
512(default)	0.90933	0.90888
1024	0.90177	0.90622
2048	0.89866	0.89688

- final parameter
  consider model = LSTM, hidden_size = [256, 512] and dropout = 0.4

hidden size	private score	public score
256	0.89777	0.89377
512(default)	0.91822	0.91333

The best is consider model = LSTM, hidden_size = 512, and dropout = 0.4

Slot tag

In this task, considering the training time and efficiency, I only use model=gru. LSTM is too slow and RNN has less accuracy.

The strategy is totally different from **Intent cls**. Because, I have already got accuracy above 0.99 with default gru in train_data. Thus, I start with the **dropout**.

dropout	private score	public score
none	0.79742	0.78498
0.2	0.83011	0.81983
0.4	0.79957	0.80160
0.6	0.82100	0.81662

dropout layer truely improve the model; however, the value of dropout doesn't have positive correlation to the accuracy.

I try more layers of rnn layer; however, they are all overfitting and get less score than the simpler ones.

hidden size	num_layers	dropout	private score	public score
512(default)	2(less with dropout)	0.2	0.83011	0.81983
256	7	0.4	0.80278	0.79302
256	3	0.5	0.82636	0.80643
256	7	0.2	0.82529	0.80750

In concluding, I think the parameter has little meaning for overfitting model. For overfitting model, accuracy can't represent the goodness of it. It's just pure luck. For this task case, dropout is the only way I found to solve the problem.