Q1: Data Processing

1. intent_cls.sh:
    a. Tokenize the data and create a vocabulary index:
        i. Split the sentences in the “text” property of each JSON object in both train.json and val.json into words, and store those words into a Counter.
        ii. Compute the frequency of each word.
        iii. Extract only the top-10,000 most frequent words.
        iv. Save those common words into a dictionary, called “*token2idx”*, that maps them to integers starting from 2. (0 and 1 are saved for PAD and UNK respectively.)
    b. Use GloVe as the pretrained embeddings.
        i. The downloaded GloVe Embedding contains 300-dimensional text-encoded vectors.
        ii. Make a dictionary, called “*glove”*, that maps those common words to their corresponding embedding vector representations.
        iii. Get the corresponding embedding vector of each word in “*token2idx”* from “*glove”*. As for PAD and UNK, generate 300-dimensional random vectors as their embedding representations. Store those embedding vectors as a list, called *“embeddings”.*
    c. Convert the intents to numbers.
        i. Store the “intent” property of each JSON object in a Set.
        ii. Store all the intents in a dictionary, called *“intent2idx”*, that maps each intent to an integer starting from 0.
2. slot_tag.sh:
    a. Tokenize the data and create a vocabulary index:
       Same as part-a of intent_cls.sh, except that here we only extract the top-10,000 most frequent words.
    b. Use GloVe as the pretrained embeddings.
       Same as part-b of intent_cls.sh.
    c. Convert the tags to numbers.
        i. Split the “tags” property of each JSON object into a list of words(tags). Store those tags in a set.
        ii. Create a dictionary, called *“tag2idx”*, that maps each tag to an integer starting from 0.

Q2. Intent Classification Model

1. Model Description:
    a. First layer - Word Embedding
       This layer computes a 300-dimensional vector representation of a word.

b. Second layer - Bidirectional GRU
    i. Forward layer: $h_t = GRU_{fwd}(w_t, h_{t-1})$, where w_t is the word embedding of the t-th token and h_{t-1} is the hidden state from the previous time step $t - 1$.
    ii. Backward layer: $h'_t = GRU_{bwd}(w_t, h'_{t-1})$, feed the sequence in reverse order, where w_t is the word embedding of the t-th token and h'_{t+1} is the hidden state from the timestep $t + 1$.

c. Third layer - Bidirectional GRU
    i. Forward layer: $h2_t = GRU_{fwd}(h_t, h2_{t-1})$, where h_t is the hidden state of the t-th token from the previous GRU_{fwd} layer and $h2_{t-1}$ is the hidden state from the previous time step $t - 1$.
    ii. Backward layer: $h2'_t = GRU_{fwd}(h'_t, h2'_{t+1})$, where h'_t is the hidden state of the t-th token from the previous GRU_{bwd} layer and $h2'_{t+1}$ is the hidden state from timestep $t + 1$.

   Forward $h2_t \oplus h2'_1$ to the next layer, where $\oplus$ indicates the operation of concatenating two tensors.

d. Fourth layer - Dropout
   Randomly zeroes some of the elements of the input tensor with probability p=0.1 using samples from a Bernoulli distribution.

e. Fifth layer - Linear
   $y = Wx + b$
    - y= output tensor with the same size as the number of classes of intents
    - W= weight matrix
    - x= output of the dropout layer
    - b= bias

f. Sixth layer - LogSoftmax
   Denote the output of the linear layer as $z = [z_1, z_2,..., z_N]$, where N is the number of classes of intents. The LogSoftmax layer outputs the logarithm of the probabilities that the current sequence s_i maps to each class, which can be expressed as:

   $$log(P(y = j|x)) = log(\frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_n}}),\ j = 1, 2,..., N$$

- $P(y = j|x)$ denotes the probability that the current sequence s_i maps to the j-th class
- z_j is the output of the linear layer
- N is the number of classes of intents

2. Model Performance:
   Public score on Kaggle: 0.90133
3. Loss Function: Negative log-likelihood (NLL)

$$\ell(x, y) = \frac{1}{N}\sum_{n=1}^{N} l_n,\ l_n = - x_{n,y_n}$$

- x is the input
- y is the target
- $N$is the batch size

4. The optimization algorithm, learning rate, and batch size.
   a. The optimization algorithm: Adam with 1e-6 weight decay
   b. Learning rate: 1e-3
      MultiStepLR scheduler with milestones [3,5,7] and gamma 0.5.
   c. Batch size: 32

Q3. Slot Tagging Model

1. Model Description:
   a. First Layer - Word Embedding
      This layer computes a 300-dimensional vector representation of a word.
   b. Second Layer - 256-dimensional Bidirectional LSTM
      i. Forward Layer: $h_t, c_t = LSTM_{fwd}(w_t, h_{t-1}, c_{t-1})$, where w_t is the word embedding of the t-th token, h_{t-1} and c_{t-1} are the hidden state and cell state from the previous time step $t - 1$.
      ii. Backward Layer: $h'_t, c'_t = LSTM_{bwd}(w_t, h'_{t+1}, c'_{t+1})$, feed the sequence in reverse order, where w_t is the word embedding of the t-th token, h'_{t+1} and c'_{t+1} are the hidden state and cell state from the timestep $t + 1$.
   c. Third Layer - 256-dimensional Bidirectional LSTM
      i. Forward Layer:$h2_t, c2_t = LSTM_{fwd}(h_t, h2_{t-1}, c2_{t-1})$, where h_t is the hidden state of the t-th token from the previous $LSTM_{fwd}$ layer and $h2_{t-1}, c2_{t-1}$ are the hidden state and cell state from the previous time step $t - 1$.
      ii. Backward Layer:$h2'_t, c2'_t = LSTM_{bwd}(h'_t, h2'_{t+1}, c2'_{t+1})$, where h'_t is the hidden state of the t-th token from the previous $LSTM_{bwd}$ layer

and $h2'_{t+1}, c2'_{t+1}$ are the hidden state and cell state from the previous time step $t + 1$.

Forward $[h2_1 \oplus h2'_1, h2_2 \oplus h2'_2, ..., h2_t \oplus h2'_t]^T$ to the next layer, where where $\oplus$ indicates the operation of concatenating two tensors.

d. Fourth Layer - Dropout
Randomly zeroes some of the elements of the input tensor with probability p=0.1 using samples from a Bernoulli distribution.

e. Fifth Layer - Linear
$y = Wx + b$
   - y= output tensor with the same size as the number of classes of tags
   - W= weight matrix
   - x= output of the dropout layer
   - b= bias

f. Sixth Layer - LogSoftmax
Denote the output of the linear layer as $z = [z_1, z_2, ..., z_N]$, where N is the number of classes of tags. The LogSoftmax layer outputs the logarithm of the probabilities that the current word w_i maps to each class, which can be expressed as:
$$log(P(y = j|x)) = log(\frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_n}}),\ j = 1, 2, ..., N$$
   - $P(y = j|x)$ denotes the probability that the current word w_i maps to the j-th class
   - z_j is the output of the linear layer
   - N is the number of classes of tags

2. Model Performance:
Public score on Kaggle: 0.79088

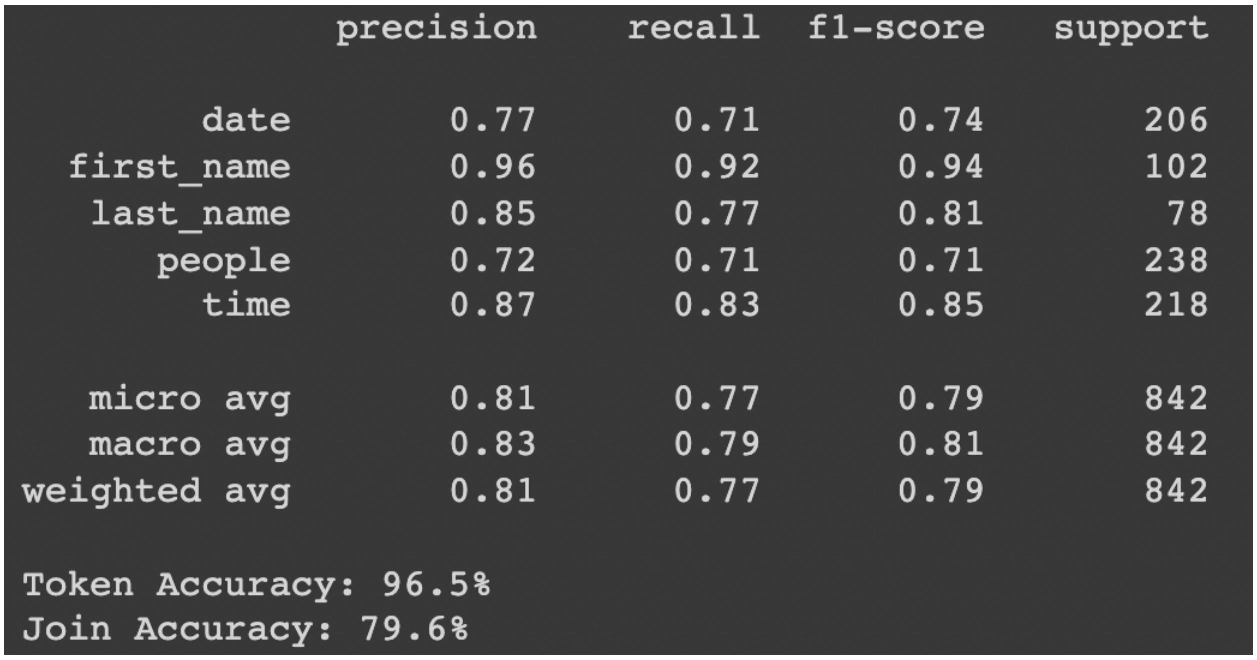
3. Loss Function: Negative log-likelihood (NLL)
$$\ell(x, y) = \frac{1}{N}\sum_{n=1}^{N} l_n,\ l_n = - x_{n,y_n}$$
   - x is the input
   - y is the target
   - $N$is the total number of words in a batch

4. The optimization algorithm, learning rate, and batch size.
   a. The optimization algorithm: Adam with 1e-6 weight decay
   b. Learning rate: 1e-3
   MultiStepLR scheduler with milestones [3,5,7] and gamma 0.5.

c. Batch size: 32

Q4. Sequence Tagging Evaluation

a. Evaluation of the model:



```
              precision    recall  f1-score   support

        date       0.77      0.71      0.74       206
  first_name       0.96      0.92      0.94       102
   last_name       0.85      0.77      0.81        78
      people       0.72      0.71      0.71       238
        time       0.87      0.83      0.85       218

   micro avg       0.81      0.77      0.79       842
   macro avg       0.83      0.79      0.81       842
weighted avg       0.81      0.77      0.79       842

Token Accuracy: 96.5%
Join Accuracy: 79.6%
```

b. Comparing different evaluation methods:

Define the following three metrics:

- True positives (tp): number of labels of a class that are predicted correctly
- False positives (fp): number of predictions of a class that are predicted wrongly
- False negatives (fn): number of predictions of a class whose true labels don't belong to the class.

Precision, recall, and f1-score are computed per class and defined as follows:

- $precision = \frac{tp}{tp+fp}$
- $recall = \frac{tp}{tp+fn}$
- $f1 - score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$

Token accuracy and join accuracy are computed as follows:

- $token\ accuracy = \frac{\#\ of\ correctly\ predicted\ tokens}{total\ \#\ of\ tokens}$
- $join\ accuraccy = \frac{\#\ of\ correctly\ predicted\ sequences}{total\ \#\ of\ sequences}$

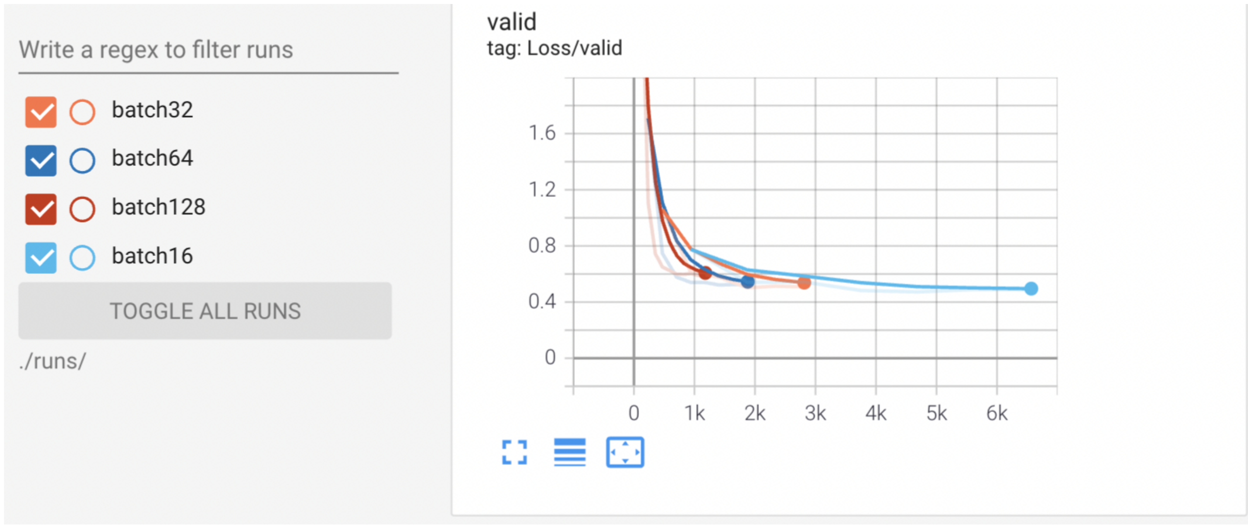
Compared to token accuracy and join accuracy, F1-score could take into account how the data is distributed, and therefore, is more suitable when the data is imbalanced.

Q5. Compare with different configurations

a. Intent classification model - baseline (with Kaggle public score 0.88355)
   i. Batch size: 32
   ii. Hidden size: 128
   iii. Number of layers: 2
   iv. Learning rate: 1e-3 with scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[3,5,7], gamma=0.5)
   v. Weight decay: 1e-6

b. Compare with different configurations:
   NOTE:
   - If the model has not improved for 3 epochs, stop the training process.
   - The horizontal axis shows the training steps, and the vertical axis shows the validation loss.
   - The loss curves are plotted for each epoch.

   i. Varying batch size (baseline: 32)

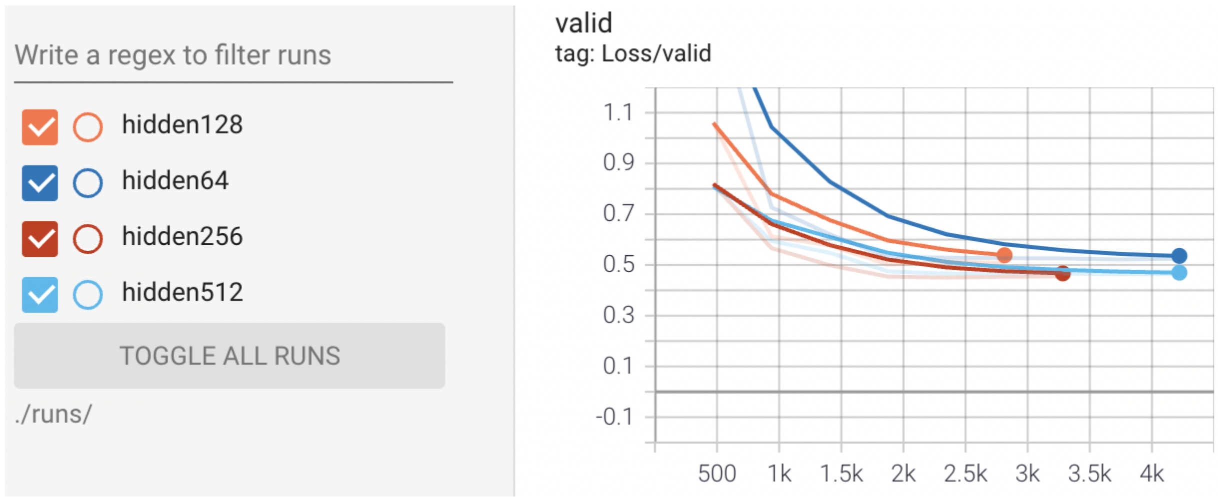


Batch size	Best validation loss
16	0.4724
32	0.5017
64	0.5216
128	0.5844

Finding: Lower batch size leads to lower validation loss.

From the plot we can see that as the batch size increases, the model converges slower.

ii. Varying hidden state size (baseline: 128)

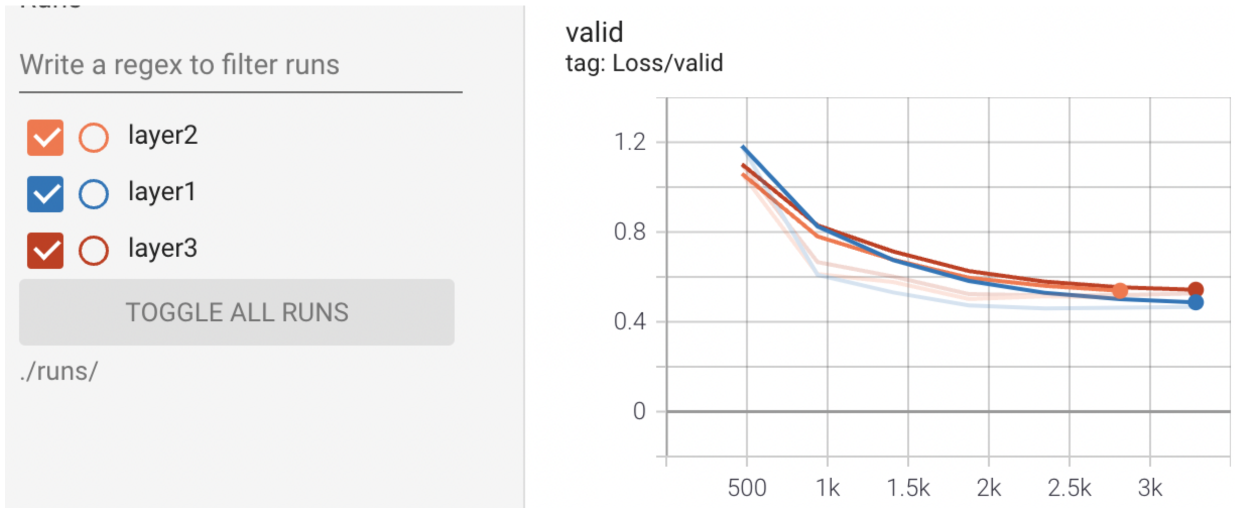


Hidden state size	Best validation loss
64	0.5358
128	0.5017
256	0.4502
512	0.4622

Finding: Larger hidden state size does not necessarily result in better performance.

When the number of hidden layers equals 2, the best performance occurs when the hidden state size equals 64. Therefore, in order to improve the model, trying smaller hidden state size may work. In addition, when the hidden state size ranges from 64 to 256, the speed of convergence decreases as the hidden state size decreases.

iii. Varying number of layers (baseline: 2)

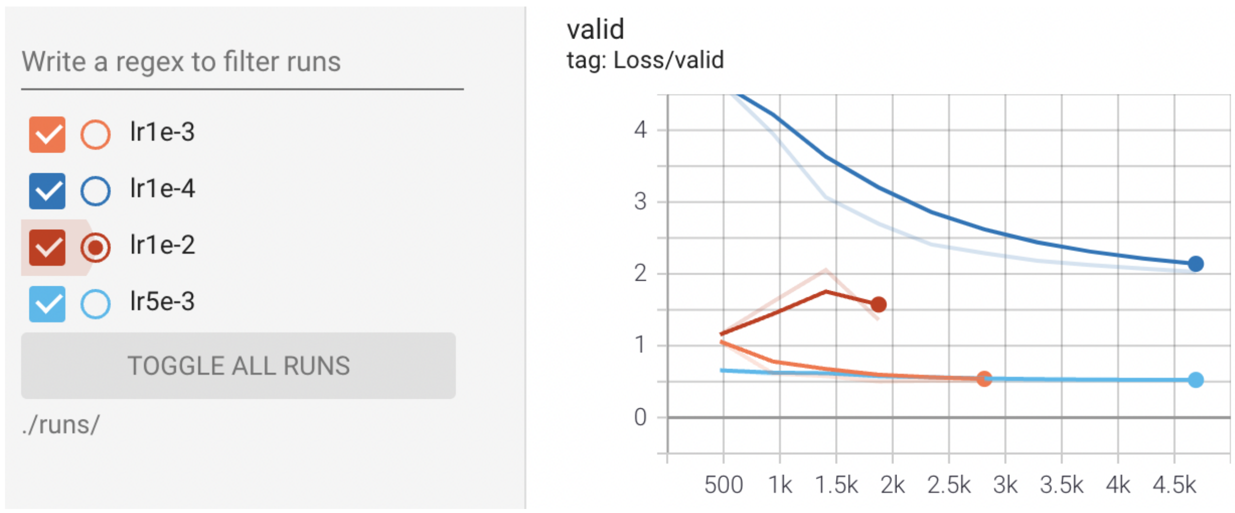


Number of layers	Best validation loss
1	0.4870
2	0.5017
3	0.5177

Finding: Smaller number of layers leads to smaller validation loss.

From the plot we can see that in this problem, when the number of layers increases, which leads to more complicated model, the performance does not improve. In addition, the number of layers does not affect the speed of convergence much.

iv. Varying learning rate (baseline: 1e-3)



Learning rate	Best validation loss
1e-4	2.028
1e-3	0.5017
5e-3	0.5211
1e-2	1.360

Finding: Larger learning rate does not necessarily result in better performance.

When the learning rate is increased to 1e-2, the model converges too quickly to a suboptimal solution. Where as when the learning rate is set at 1e-4, the training process is converging too slowly and gets stuck. Therefore, the learning rate needs to be carefully selected.