PA4 RNN Report

计 72　刘光哲　2017011329

2018.11.06

目录

1 Environment

Operating System: macOS 10.14.1 Mojave
Python: Python 3.6.6
tensorflow: tensorflow 1.11.0
CPU: Intel Core i7-8750H, Memory: 16 GB

2 Code Review

2.1 Model construction

According to the slides "Deep Learning Framework" of this course, the embed part should be:

```
self.embed_input = tf.nn.embedding_lookup(embed, self.index_input)
```

Th implement of Bidirectional-RNN comes from the official document of Tensorflow, I referred to the document of bidirectional-dynamic-rnn and the tutorial of dynamic-rnn, and implemented it like this:

```
outputs, states = tf.nn.bidirectional_dynamic_rnn(cell_fw,cell_bw,self.embed_input,
 sequence_length=self.texts_length, dtype=tf.float32, scope="rnn")
```

2.2 self-attention machanism

Self-attention machanism is just an application of using tensorflow for matrix calculation.
But in this algorithm, there are several tricks:

1. matrix W_1 and W_2 is reversed (compared with the explaination of PA4), so I have to transpose each matrix multiply and finally transpose back.
2. tensorflow.matmul doesn't directly support tensors with batch_size, so I have to reshape these matrix for matmul.

```
with tf.variable_scope('logits'):
    Ws1 = tf.get_variable("Ws1", [2 * num_units, param_da])
    Ws2 = tf.get_variable("Ws2", [param_da, param_r])
    A1 = tf.tanh(tf.matmul(tf.reshape(H, [-1, 2 * num_units]), Ws1))
    A = tf.nn.softmax(tf.matmul(A1, Ws2))
    A = tf.reshape(A, [batch_size, -1, param_r])
    M = tf.matmul(tf.transpose(A, perm=[0, 2, 1]), H)
    flatten_M = tf.reshape(M,
        shape=[batch_size, param_r * 2 * num_units])
    logits = tf.layers.dense(flatten_M, num_labels, activation=None,
        name='projection')

identity = tf.reshape(tf.tile(tf.diag(tf.ones([param_r])), [batch_size, 1])
    , [batch_size, param_r, param_r])
self.penalized_term = tf.norm((tf.matmul(tf.transpose(A, perm=[0, 2, 1]), A)
    - identity)) ** 2
```

2.3 GRUCell

I learned this technique from the realization of seq2seq (https://github.com/eske/seq2seq.git).

In this part, the basic technique we used to realize the Cell is use the functions given by tf.layers.

By concat the two vector input and state, we don't have to define two tensors W and U to realize it.

In my code, there is $c = tanh((W, U) \cdot (inputs, r \times state))$, and $h_{new} = u \times state + (1 - u) \times c$.

```
with vs.variable_scope("candidate"):
    c = tf.layers.dense(tf.concat([inputs, r * state], 1), self._num_units,
     activation=self._activation, use_bias=True)
new_h = u * state + (1 - u) * c
```

2.4 BasicLSTMCell

I learned this technique from the realization of seq2seq (https://github.com/eske/seq2seq.git).

In this part we connect inputs and h, and get the output from a dense linear layer.

In my code, $(i, ct, f, o) = (W_i, U_i, W_c, U_c, W_f, U_f, W_o, U_o) \cdot (inputs, h) + (b_i, b_c, b_f, b_o)$, and $c_{new} = c \times sigmoid(f + b) + sigmoid(i) \times tanh(ct)$, $h_{new} = tanh(c_{new}) \times sigmoid(o)$.

```
val = tf.layers.dense(tf.concat([inputs, h], 1), 4*self._num_units, use_bias=True)
i, ct, f, o = tf.split(value=val, num_or_size_splits=4, axis=1)
new_c = (c * sigmoid(f + self._forget_bias) + sigmoid(i) * self._activation(ct))
new_h = tf.tanh(new_c) * sigmoid(o)
```

3 Experiments

I did the following experiments to show I finished the algorithm and can solve the problem.

3.1 Train with 3 kinds of RnnCells and Plot

3.1.1 BasicRNNCell

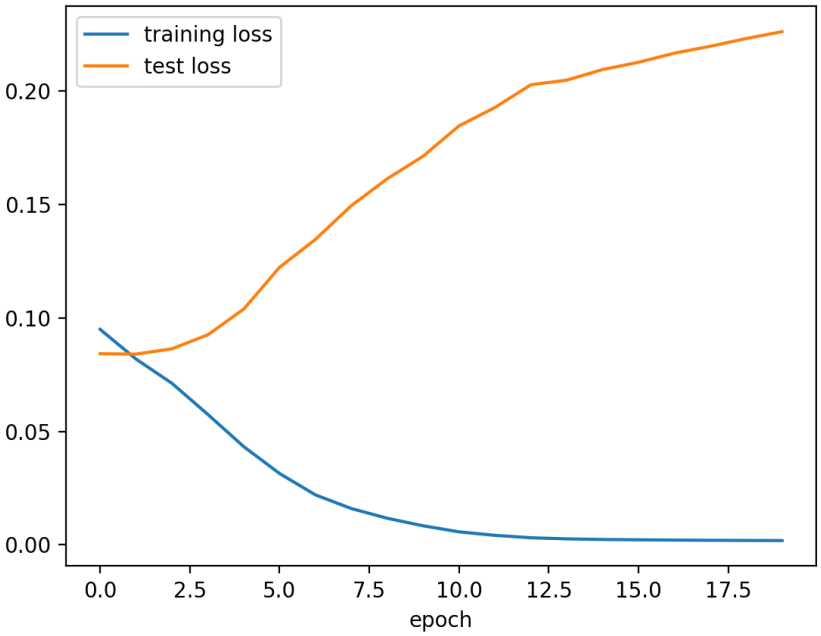
For BasicRNNCell I trained it with the following parameters:

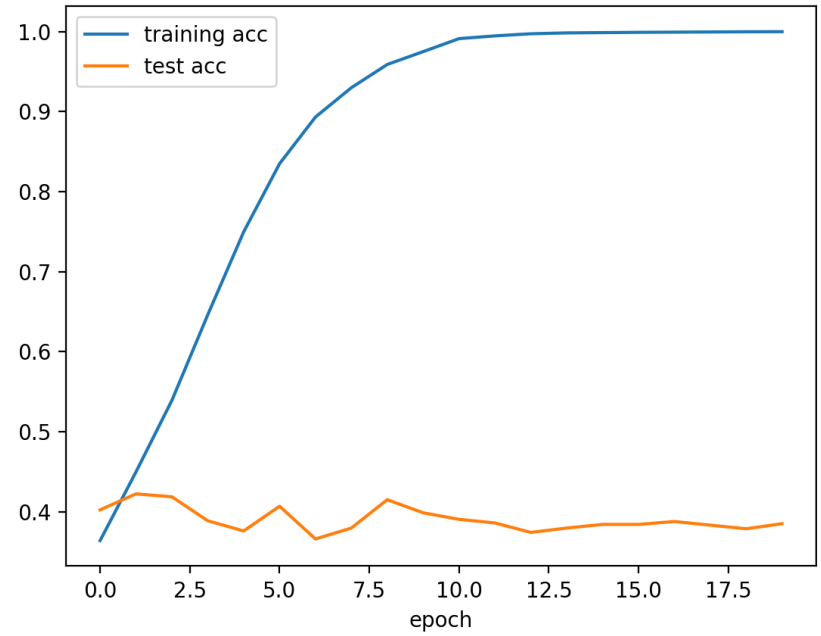
learning rate	epoch
0.025	20

The basic results are like this:

time	$\frac{time}{epoch}$	accuracy	loss
12 min	35 s	0.3851	0.2261

The result plots like this:





In this experiment, RNN converges very fast but soon it got over-fitted: the test loss is getting bigger and bigger during training.

3.1.2 GRUCell

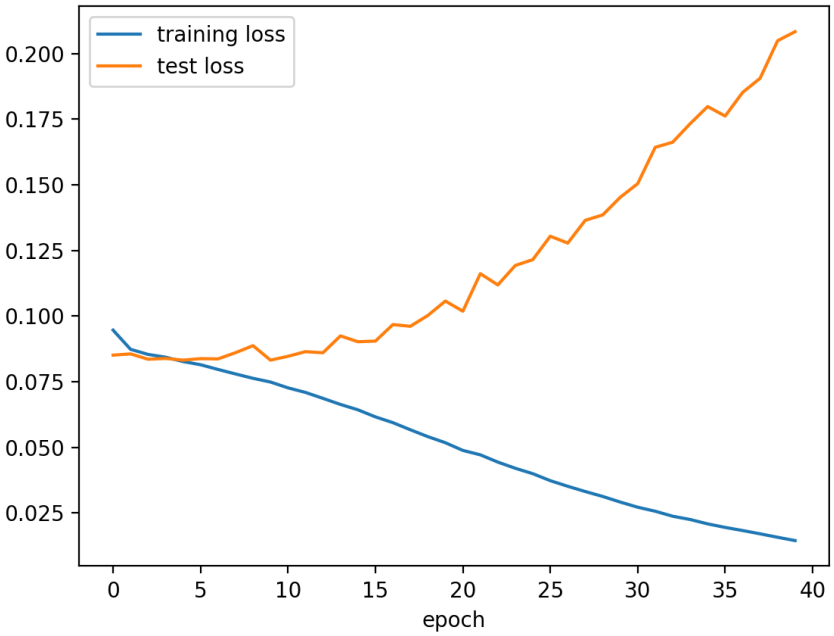
For GRUCell I trained it with the following parameters:

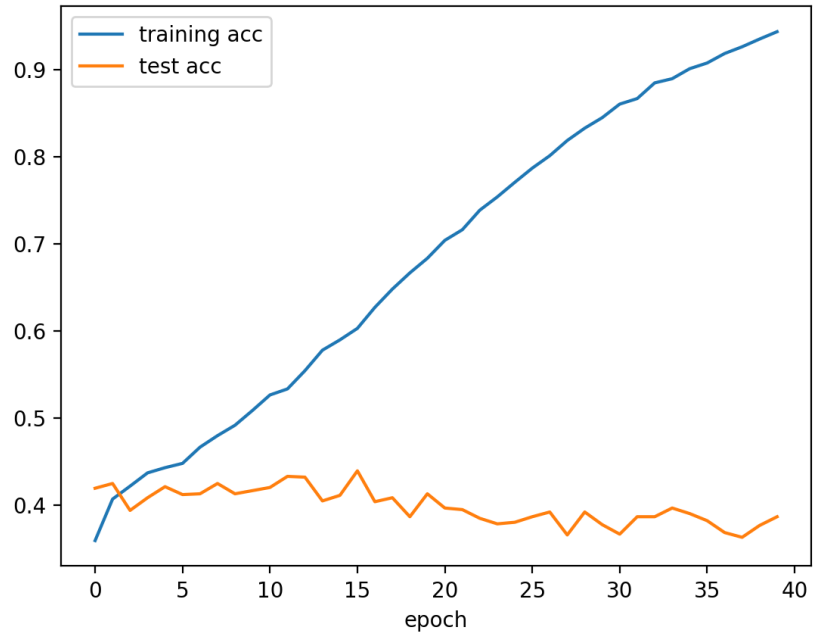
learning rate	epoch
0.025	40

The basic results are like this:

time	$\frac{time}{epoch}$	accuracy	loss
77 min	115 s	0.3869	0.2082

The result plots like this:





We can see from this experiment that RNN with GRUCell converges really slow, and spends a much longer training time.

3.1.3 BasicLSTMCell

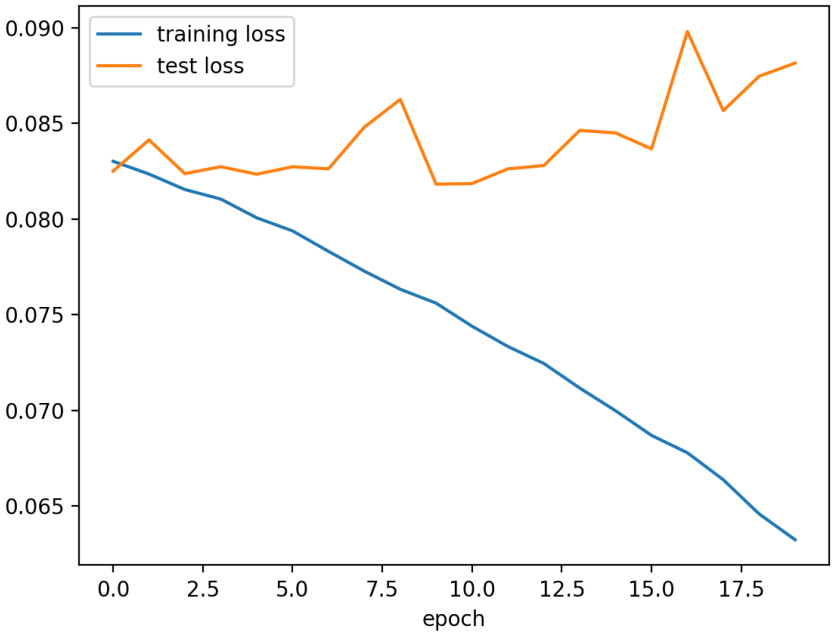
For BasicLSTMCell I trained it with the following parameters:

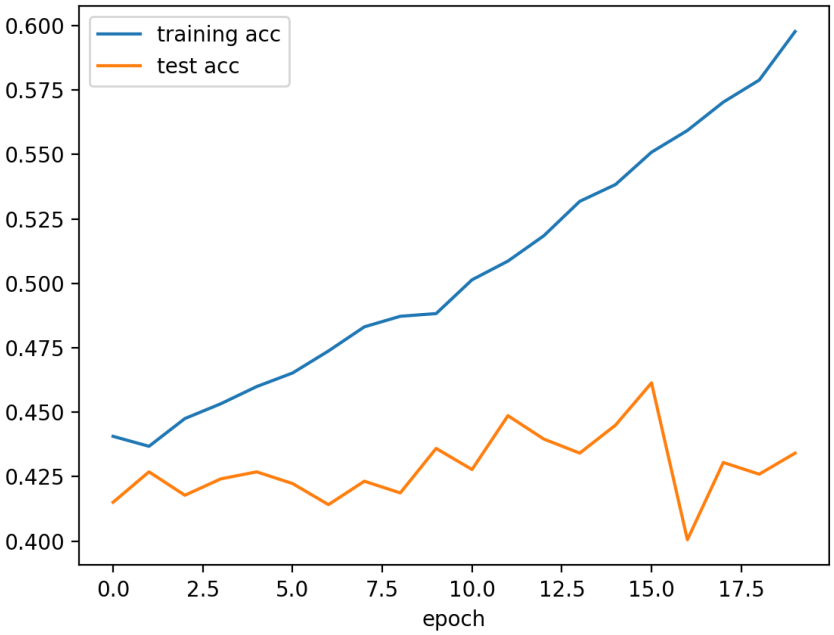
learning rate	epoch
0.1	20

I used 0.1 as learning rate considering the performance of GRUCell, the basic results are like this:

time	$\frac{time}{epoch}$	accuracy	loss
60 min	180 s	0.4341	0.0881

The result plots like this:





BasicLSTMCell needs a even longer training time than GRUCell, but it does have a better test result during training for it properly get rid of the problem of overfitting.

3.1.4 Comparision

	training speed	converge speed	accuracy	loss	overfitting
BasicRNNCell	fast	fast	low	high	serious
GRUCell	medium	slow	low	high	serious
BasicLSTMCell	slow	slow	high	low	not seroius

3.2 My final network

3.2.1 multi-layer RNN

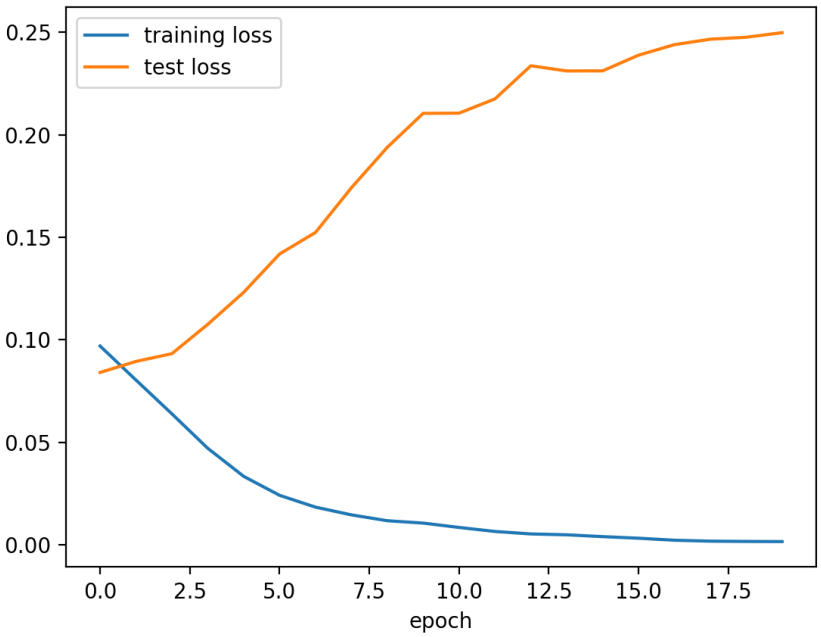
I tested multi-layer RNN with BasicRNNCell and got the following result:

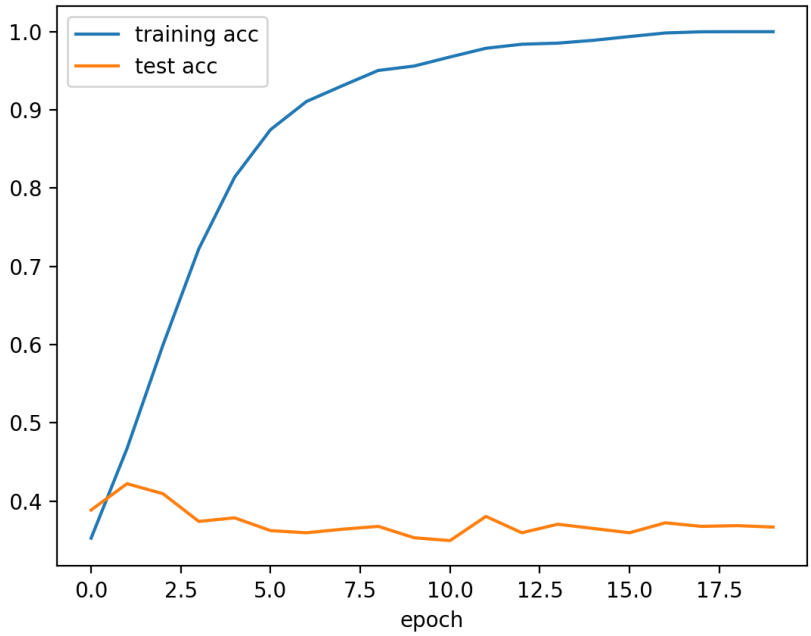
learning rate	epoch	layers
0.05	20	2

The basic results are like this:

time	$\frac{time}{epoch}$	accuracy	loss
31 min	95 s	0.3669	0.2497

The result plots like this:





It's appalling to find that the performance of multi-layer RNNs has nothing different from single-layer RNNs, except for it takes a longer training time. It still has the problem of overfitting.

3.3 Analysis

I got the following conclusions:

1. LSTMCell has better performance than BasicRNNCell and GRUCell, for it's better at dealing with overfitting.
2. RNN is not the best model for this problem, for it never gets an accuracy greater than 0.5 in my experiment.
3. Training RNN takes a longer time than CNN and MLP.
4. Official Documents are very important for learning a new tool, such as Tensorflow.