Deep Learning and Practice
Lab 6: RNN Seq2Seq

李韡
0556157
fm.bigballon@gmail.com

Introduction

In this project, we are going to build a LSTM structure to do the copy experiment.

I used Tensorflow to implement sequence-to-sequence models to solve the copy task.
It has reasonable accuracy rate when training length >= testing length.
And also reasonable when training length < testing length (train= 20 + 10 padding, test = 30), test sequence = 50 cost lots of iteration to train and got an undesirable result.

I tested many optimizers(Adam, SGD, Momentum, RMSProp etc.)
And I also tested some RNN cells (BasicLSTMCell, GRUCell, MultiRNNCell etc.)

Experiment setup

- **Training hyperparameters:**
  - **Optimizer:**
    - GradientDescentOptimizer(lr)
    - MomentumOptimizer(lr, 0.9, use_nesterov=True)
    - **AdamOptimizer**(lr)
    - **RMSPropOptimizer**(learning_rate=0.0006, momentum=0.9)
  - batch size: **128 (also test 64 and 256)**
  - Iteration: **10000/15000/100000**
  - Sequence length: 10/20/30/50
  - Hidden size: 500
  - Embedding size: 100

Result

Training length		20	Training length		30
Testing length	10	99.3%	Testing length	20	99.0%
	20	99.0%		30	99.2%
	30	99.2%		50	4.25%

■ **Model 1: Train length 20: Testing length: 10 & 20**

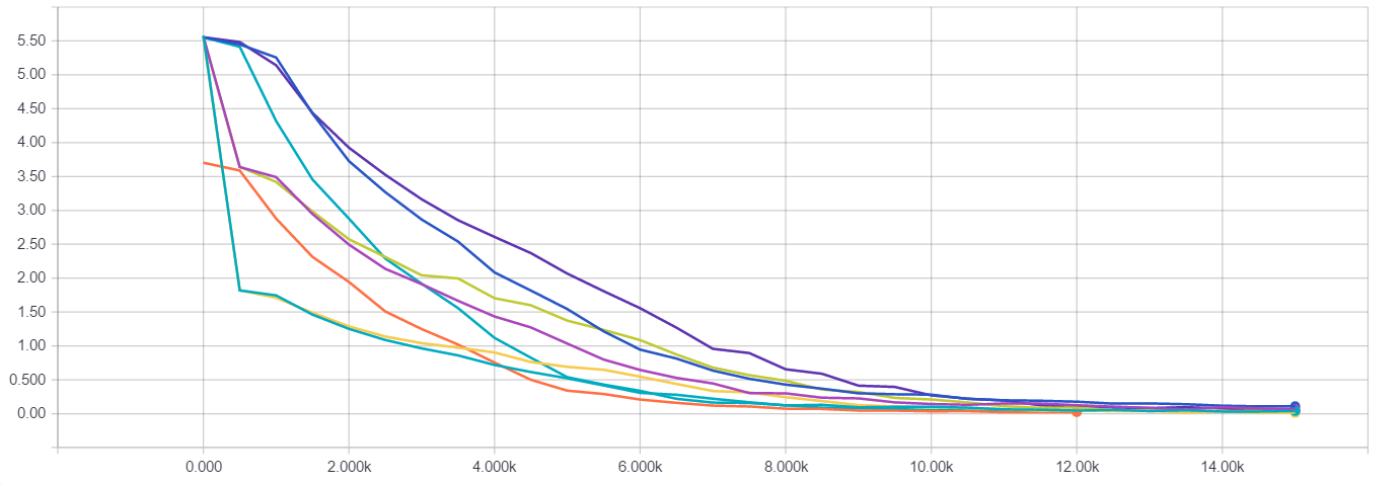
```
test:10/20, itrations: 13500, test_loss: 0.01628, test_acc: 100.00000%.
----------------------------------------------------------------------
test:20/20, itrations: 13500, test_loss: 0.01684, test_acc: 99.50000%.
----------------------------------------------------------------------
test:10/20, itrations: 14000, test_loss: 0.00695, test_acc: 100.00000%.
----------------------------------------------------------------------
test:20/20, itrations: 14000, test_loss: 0.00604, test_acc: 100.00000%.
----------------------------------------------------------------------
test:10/20, itrations: 14500, test_loss: 0.01012, test_acc: 100.00000%.
----------------------------------------------------------------------
test:20/20, itrations: 14500, test_loss: 0.02048, test_acc: 99.50000%.
```

■ **Model 2: Train length 20 + 10 padding Testing length: 10 & 20 & 30**

```
----------------------------------------------------------------------
test:10/30, itrations: 14500, test_loss: 0.01418, test_acc: 99.66667%.
----------------------------------------------------------------------
test:20/30, itrations: 14500, test_loss: 0.04188, test_acc: 98.66667%.
----------------------------------------------------------------------
test:30/30, itrations: 14500, test_loss: 0.05186, test_acc: 99.00000%.
----------------------------------------------------------------------
test:10/30, itrations: 15000, test_loss: 0.01329, test_acc: 99.33333%.
----------------------------------------------------------------------
test:20/30, itrations: 15000, test_loss: 0.03830, test_acc: 98.00000%.
----------------------------------------------------------------------
test:30/30, itrations: 15000, test_loss: 0.03911, test_acc: 99.22222%.
----------------------------------------------------------------------
```

Testing loss of Model 2 opt = RMSPropOptimizer lr = 0.0002 & lr = 0.0003

Name	Smoothed	Value	Step	Time	Relative
logs_train20+10padding+lr=0.0002/test_10	0.03903	0.03903	15.00k	Tue Apr 18, 03:54:26	9m 8s
logs_train20+10padding+lr=0.0002/test_20	0.06685	0.06685	15.00k	Tue Apr 18, 03:54:26	9m 8s
logs_train20+10padding+lr=0.0002/test_30	0.1092	0.1092	15.00k	Tue Apr 18, 03:54:26	9m 8s
logs_train20+10padding+lr=0.0003/test_10	0.01329	0.01329	15.00k	Tue Apr 18, 09:14:52	9m 19s
logs_train20+10padding+lr=0.0003/test_20	0.03830	0.03830	15.00k	Tue Apr 18, 09:14:52	9m 19s
logs_train20+10padding+lr=0.0003/test_30	0.03911	0.03911	15.00k	Tue Apr 18, 09:14:52	9m 19s
logs_train30+nopadding+lr=0.0003/test_20	0.02288	0.02288	12.00k	Tue Apr 18, 09:33:18	7m 26s
logs_train30+nopadding+lr=0.0003/test_30	0.03840	0.03840	12.00k	Tue Apr 18, 09:33:18	7m 26s



- **Model 3: Train length 30: Testing length: 20 & 30**

```
test:20/30, itrations: 11500, test_loss: 0.02130, test_acc: 99.33333%.
--------------------------------------------------------------------------------
test:30/30, itrations: 11500, test_loss: 0.03241, test_acc: 99.22222%.
--------------------------------------------------------------------------------
test:20/30, itrations: 12000, test_loss: 0.02288, test_acc: 99.00000%.
--------------------------------------------------------------------------------
test:30/30, itrations: 12000, test_loss: 0.03840, test_acc: 99.22222%.
--------------------------------------------------------------------------------
```

- **Model 4: Train length 30 + 20 padding Testing length: 50**

```
--------------------------------------------------------------------------------
test:20/50, itrations: 100000, train_loss: 0.03987, test_acc: 46.09375%.
test:30/50, itrations: 100000, train_loss: 0.03987, test_acc: 99.27083%.
test:50/50, itrations: 100000, train_loss: 0.03987, test_acc: 4.25000%.
--------------------------------------------------------------------------------
```

Other experiments (Bonus)

In my first implement, I got the following result:

```
test:10/30, itrations: 9500, train_loss: 0.01514, test_acc: 85.23438%.
test:20/30, itrations: 9500, train_loss: 0.01514, test_acc: 99.88281%.
test:30/30, itrations: 9500, train_loss: 0.01514, test_acc: 5.00000%.
--------------------------------------------------------------------------
test:10/30, itrations: 10000, train_loss: 0.00973, test_acc: 86.79688%.
test:20/30, itrations: 10000, train_loss: 0.00973, test_acc: 99.57031%.
test:30/30, itrations: 10000, train_loss: 0.00973, test_acc: 4.55729%.
```

Then I modified the code:

```
enc_inp = [tf.placeholder(tf.int32, shape=(None,),name="inp%i" % t)  for t in range(seq_length)]
dec_inp = [tf.placeholder(tf.int32, shape=(None,),name="dec%i" % t)  for t in range(seq_length)]
weights = [tf.placeholder(tf.float32, shape=(None,),name="weight%i" % t)  for t in range(seq_length)]
```

```
# set feed dictionary
def set_feed_dict(used_len):
    X = [np.append(np.random.randint(1, 257, size=used_len),
        np.zeros((seq_length-used_len,), dtype=np.int)) for _ in range(batch_size)]
    D = np.full([batch_size, seq_length], 257)
    W = np.full([batch_size, seq_length], 1)

    # W = np.full([batch_size, used_len], 1)
    # W = np.append(W, np.zeros([batch_size, seq_length - used_len]), axis=1)

    feed_dict = {enc_inp[t]: X[t] for t in range(seq_length)}
    feed_dict.update({dec_inp[t]: D[t] for t in range(seq_length)})
    feed_dict.update({weights[t]: W[t] for t in range(seq_length)})
    return feed_dict, X
```

Then I get the reasonable accuracy rate when training length < testing length:

```
--------------------------------------------------------------------------
test:10/30, itrations: 15000, test_loss: 0.01329, test_acc: 99.33333%.
--------------------------------------------------------------------------
test:20/30, itrations: 15000, test_loss: 0.03830, test_acc: 98.00000%.
--------------------------------------------------------------------------
test:30/30, itrations: 15000, test_loss: 0.03911, test_acc: 99.22222%.
--------------------------------------------------------------------------
```

We can also using MultiRnnCell [MultiRNNCell([BasicLSTMCell(memory_dim)] * 3)] to get get the reasonable accuracy rate.

Discussion

About RNN cells:

In order to get a reasonable accuracy rate when training length < testing length, I just test the following cells.

```
# set cell of RNN
def set_cell():
    return BasicLSTMCell(memory_dim)
    # return GRUCell(memory_dim)
    # return MultiRNNCell( [ BasicLSTMCell(memory_dim)] * 3 )
    # return MultiRNNCell( [ GRUCell(memory_dim)] * 3 )
```

Basic cell is [BasicLSTMCell], and using MultiRnnCell (BasicLSTMCell *3) can got better reults, Another cell I used is [GRUCell], which can get a better result than [BasicLSTMCell].

About Optimizers:

SGD got a bad result, so l changes it to the following opt

```
# test all kinds of optimizer
def set_optimizer(lr):
    # return tf.train.AdamOptimizer(lr)
    # return tf.train.GradientDescentOptimizer(lr)
    # return tf.train.MomentumOptimizer(lr, 0.9, use_nesterov=True)
    return tf.train.RMSPropOptimizer(learning_rate=lr, momentum=0.9)
```

Adam & RMSProp have good performance