PART 2 TensorFlow

5. Workshop 3 - RNN 序列資料處理

REFERENCE

1. Tom Hope, Yehezkel S. Resheff, Itay Lieder, "**Learning TensorFlow - A Guide to building Deep Learning Systems**", `Chapters 5` , O'Reilly (2017) (pdf) [https://goo.gl/iEmehh (https://goo.gl/iEmehh)](https://goo.gl/iEmehh) `[ Code ]` : [https://github.com/gigwegbe/Learning-TensorFlow (https://github.com/gigwegbe/Learning-TensorFlow)](https://github.com/gigwegbe/Learning-TensorFlow)
2. bigDataSpark Forum 檔案：**Basics of TensorFlow Programming-20180809.ipynb** [https://www.facebook.com/groups/753114451505938/permalink/1213353432148702/ (https://www.facebook.com/groups/753114451505938/permalink/1213353432148702/)](https://www.facebook.com/groups/753114451505938/permalink/1213353432148702/)

Introduction to Recurrent Neural Networks

The basic idea behind RNN models is that each new element in the sequence contributes some new information, which updates the current state of the current state of the model.

A fundamental mathematical construct in statistics and probably, which is often used as building block for modelling sequential pattern via machine learning is the Markov chain model. We tend to view our data sequences as "chains", with each node in the chain dependent in some way on the previous node, so that "history" is not erased but carried on.

RNN models are the based on this notion of chain structure. As the name implie, recurrent neural nets apply some form of "loop." At some point in time t, the network observes an input x(t)(a word in a sentence) and update its "state vector" to h(t) from the previous vector h(t-1). When we process new input (the next word), it will be done in some manner that is dependent on h(t) and thus on the history of the sequence (the previous words we've seen affect our understanding of the current word). Recurrent structure can simply be viewed as one long unrolled chain, with each node in the chain performing the same kind of processing "step" based on the "message" it obtains from the output of the previous node.

Vanilla RNN Implementation

We introduce some powerful, fairly low-level tools that Tensorflow provides for working with sequence data, which you can use to implement your own systems. We begin with our basic model mathematically. This mainly consists of defining the recurrence structure - the RNN update step. The update step for our simple vanilla RNN is h(t) = tanh(W(x)x(t) + W(h)h(t-1) + b) where W(h),W(x) and b are weight and bias variables. tanh(.) is the hyperbolic tangent function that has its range in [-1,1] and

MNIST image as sequences

From the previous chapter the architecture of convolutional neural networks makes use of the spatial structure of images, it is revealing to look at the structure of images from different angles by trying to capture in some sense the "generative process" that created each image. Intuitively, this all comes down to the notion that nearby areas in images are somehow related, and trying to model this structure. In our MNIST data, this just means that each 28 * 28 pixel image can be viewed as sequence of lengh 28, each element in the sequence a vector of 28 pixels. Then the temporal dependencies in the RNN can be imaged as a scanner head, scanning the image from top to buttom(rows) or left to right (columns).

We start by loading data, defining some parameters, and creating placeholders for our data:

In [1]:

```
import tensorflow as tf
# for the old-version usage of TensorFlow, such as tensorflow.examples.tutorials.mnist
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./data", one_hot=True)

#Define some parameters
element_size = 28
time_steps = 28
num_classes = 10
batch_size = 128
hidden_layer_size = 128

# Where to save TensorBoard model summaries
LOG_DIR = "logs/RNN_with_summaries"

# Create placeholders for inputs, labels
_inputs = tf.placeholder(tf.float32, shape=[None, time_steps, element_size], name="inp

y = tf.placeholder(tf.float32, shape=[None, num_classes], name="labels")

```

```
/Users/macmini1/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: F
utureWarning: Conversion of the second argument of issubdtype from `float` t
o `np.floating` is deprecated. In future, it will be treated as `np.float64
== np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

Extracting ./data/train-images-idx3-ubyte.gz
Extracting ./data/train-labels-idx1-ubyte.gz
Extracting ./data/t10k-images-idx3-ubyte.gz
Extracting ./data/t10k-labels-idx1-ubyte.gz
```

In [2]:

```
batch_x, batch_y = mnist.train.next_batch(batch_size)
# Reshape data to 28 sequence of 28 pixels
batch_x = batch_x.reshape((batch_size, time_steps, element_size))
```

In [3]:

```
#This helper function taken from official TensorFlow documentation,
# simply add some ops that take care of logging summaries
def variable_summaries(var):
    with tf.name_scope('summaries'):
      mean = tf.reduce_mean(var)
      tf.summary.scalar('mean', mean)
      with tf.name_scope('stddev'):
        stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
      tf.summary.scalar('stddev', stddev)
      tf.summary.scalar('max', tf.reduce_max(var))
      tf.summary.scalar('min', tf.reduce_min(var))
      tf.summary.histogram('histogram', var)


# Weights and bias for input and hidden layer
with tf.name_scope('rnn_weights'):
        with tf.name_scope("W_x"):
            Wx = tf.Variable(tf.zeros([element_size, hidden_layer_size]))
            variable_summaries(Wx)
        with tf.name_scope("W_h"):
            Wh = tf.Variable(tf.zeros([hidden_layer_size, hidden_layer_size]))
            variable_summaries(Wh)
        with tf.name_scope("Bias"):
            b_rnn = tf.Variable(tf.zeros([hidden_layer_size]))
            variable_summaries(b_rnn)
```

In [4]:

```
def rnn_step(previous_hidden_state,x):

        current_hidden_state = tf.tanh(
            tf.matmul(previous_hidden_state, Wh) +
            tf.matmul(x, Wx) + b_rnn)

        return current_hidden_state

# Processing inputs to work with scan function
# Current input shape: (batch_size, time_steps, element_size)
processed_input = tf.transpose(_inputs, perm=[1, 0, 2])
# Current input shape now: (time_steps,batch_size, element_size)

```

In [5]:

```

initial_hidden = tf.zeros([batch_size,hidden_layer_size])
# Getting all state vectors across time
all_hidden_states = tf.scan(rnn_step,
                            processed_input,
                            initializer=initial_hidden,
                            name='states')


# Weights for output layers
with tf.name_scope('linear_layer_weights') as scope:
    with tf.name_scope("W_linear"):
        Wl = tf.Variable(tf.truncated_normal([hidden_layer_size,
                                              num_classes],
                                              mean=0,stddev=.01))
        variable_summaries(Wl)
    with tf.name_scope("Bias_linear"):
        bl = tf.Variable(tf.truncated_normal([num_classes],
                                             mean=0,stddev=.01))
        variable_summaries(bl)

```

In [6]:

```
#Apply linear layer to state vector
def get_linear_layer(hidden_state):

    return tf.matmul(hidden_state, Wl) + bl

with tf.name_scope('linear_layer_weights') as scope:
    #Iterate across time, apply linear layer to all RNN outputs
    all_outputs = tf.map_fn(get_linear_layer, all_hidden_states)
    #Get Last output -- h_28
    output = all_outputs[-1]
    tf.summary.histogram('outputs', output)

with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=o
    tf.summary.scalar('cross_entropy', cross_entropy)

with tf.name_scope('train'):
    #Using RMSPropOptimizer
    train_step = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cross_entropy)

with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(output,1))

    accuracy = (tf.reduce_mean(tf.cast(correct_prediction, tf.float32)))*100
    tf.summary.scalar('accuracy', accuracy)

```

In [7]:

```python

# Merge all the summaries
merged = tf.summary.merge_all()


#Get a small test set
test_data = mnist.test.images[:batch_size].reshape((-1, time_steps,
                                                     element_size))
test_label = mnist.test.labels[:batch_size]

with tf.Session() as sess:
    #Write summaries to LOG_DIR -- used by TensorBoard
    train_writer = tf.summary.FileWriter(LOG_DIR + '/train',
                                         graph=tf.get_default_graph())
    test_writer = tf.summary.FileWriter(LOG_DIR + '/test',
                                        graph=tf.get_default_graph())

    sess.run(tf.global_variables_initializer())

    for i in range(10000):

            batch_x, batch_y = mnist.train.next_batch(batch_size)
            # Reshape data to get 28 sequences of 28 pixels
            batch_x = batch_x.reshape((batch_size, time_steps,
                                       element_size))
            summary,_ =sess.run([merged,train_step],
                                feed_dict={_inputs:batch_x, y:batch_y})
            #Add to summaries
            train_writer.add_summary(summary, i)

            if i % 1000 == 0:
                acc,loss, = sess.run([accuracy,cross_entropy],
                                     feed_dict={_inputs: batch_x,
                                                y: batch_y})
                print ("Iter " + str(i) + ", Minibatch Loss= " + \
                      "{:.6f}".format(loss) + ", Training Accuracy= " + \
                      "{:.5f}".format(acc))
            if i % 100 == 0:
                # Calculate accuracy for 128 mnist test images and
                #add to summaries
                summary, acc = sess.run([merged, accuracy],
                                        feed_dict={_inputs: test_data,
                                                   y: test_label})
                test_writer.add_summary(summary, i)

    test_acc = sess.run(accuracy, feed_dict={_inputs: test_data,
                                             y: test_label})
    print ("Test Accuracy:", test_acc)



```

```
Iter 0, Minibatch Loss= 2.301909, Training Accuracy= 10.15625
Iter 1000, Minibatch Loss= 1.169573, Training Accuracy= 55.46875
Iter 2000, Minibatch Loss= 0.646247, Training Accuracy= 78.90625
Iter 3000, Minibatch Loss= 0.224386, Training Accuracy= 92.96875
Iter 4000, Minibatch Loss= 0.126278, Training Accuracy= 95.31250
Iter 5000, Minibatch Loss= 0.128137, Training Accuracy= 97.65625
Iter 6000, Minibatch Loss= 0.049506, Training Accuracy= 97.65625
```

```
Iter 7000, Minibatch Loss= 0.188727, Training Accuracy= 93.75000
Iter 8000, Minibatch Loss= 0.009813, Training Accuracy= 100.00000
Iter 9000, Minibatch Loss= 0.030163, Training Accuracy= 99.21875
Test Accuracy: 99.21875
```

Visualizing the model with TensorBoard

TensorBoard is an interactive browser-based tool that allows us to visualize the learning process. To run TensorBoard, go to the command terminal and tell TensorBoard where the relevant summaries you logged are:

In [8]:

```
#tensorboard --logdir=logs/RNN_with_summaries
#tensorboard --logdir=C:\Users\other\Documents\ASE-DL-201809\ASE-20181008-TensorFlow\l
```

In [9]:

```
#If you are on Windows use:tensorboard --logdir=rnn_demo:LOG_DIR
```

TensorBoard allows us to assign names to individual log directories by putting a colon between the name and the path, which may be useful when working with multiple log directories. In such a case, we pass a comma-seperated list of log directories as follows-

In [10]:

```
#tensorboard --logdir=rnn_demo1:LOG_DIR1, rnn_demo2:LOG_DIR2
```

To start the tensorboard, go to the directory containing the log and run the tensorboard command in the terminal

In [11]:

```
#Starting TensorBoard b'39' on port 6006
#(You can navigate to http://10.100.102.4:6006)
```

TensorFlow Built-in RNN Functions

In [12]:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./data/", one_hot=True)
```

```
Extracting ./data/train-images-idx3-ubyte.gz
Extracting ./data/train-labels-idx1-ubyte.gz
Extracting ./data/t10k-images-idx3-ubyte.gz
Extracting ./data/t10k-labels-idx1-ubyte.gz
```

In [13]:

```
## ---------------------------------------------------------------------------------
##  Because of using tf.nn.dynamic_rnn(), you need to clear you computational graph.
##  You can do that by putting this line at the beginning of your script.
## ---------------------------------------------------------------------------------
tf.reset_default_graph()

element_size = 28; time_steps= 28; num_classes =10
batch_size = 128; hidden_layer_size = 128

_inputs = tf.placeholder(tf.float32,shape=[None, time_steps,
element_size],
name='inputs')
y = tf.placeholder(tf.float32, shape=[None, num_classes],name='inputs')

# TensorFlow built-in functions
rnn_cell = tf.contrib.rnn.BasicRNNCell(hidden_layer_size)
outputs, _ = tf.nn.dynamic_rnn(rnn_cell, _inputs, dtype=tf.float32)

Wl = tf.Variable(tf.truncated_normal([hidden_layer_size, num_classes],
mean=0,stddev=.01))
bl = tf.Variable(tf.truncated_normal([num_classes],mean=0,stddev=.01))



def get_linear_layer(vector):
    return tf.matmul(vector, Wl) + bl

last_rnn_output = outputs[:,-1,:]
final_output = get_linear_layer(last_rnn_output)
softmax = tf.nn.softmax_cross_entropy_with_logits(logits=final_output,
labels=y)
cross_entropy = tf.reduce_mean(softmax)
train_step = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(final_output,1))
accuracy = (tf.reduce_mean(tf.cast(correct_prediction, tf.float32)))*100
sess=tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
test_data = mnist.test.images[:batch_size].reshape((-1,time_steps, element_size))
test_label = mnist.test.labels[:batch_size]


for i in range(3001):
## for i in range(10000):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, time_steps, element_size))
        sess.run(train_step,feed_dict={_inputs:batch_x,y:batch_y})

        if i % 1000 == 0:
                acc = sess.run(accuracy, feed_dict={_inputs: batch_x,y: batch_y})
                loss = sess.run(cross_entropy,feed_dict={_inputs:batch_x,y:batch_y})
                print("Iter " + str(i) + ", Minibatch Loss= " + \
                          "{:.6f}".format(loss) + ", Training Accuracy= " + \
                          "{:.5f}".format(acc))
print("Testing Accuracy:",
        sess.run(accuracy, feed_dict={_inputs: test_data, y: test_label}))
```

```
Iter 0, Minibatch Loss= 2.300996, Training Accuracy= 15.62500
Iter 1000, Minibatch Loss= 0.148401, Training Accuracy= 96.09375
```

```
Iter 2000, Minibatch Loss= 0.122571, Training Accuracy= 96.87500
Iter 3000, Minibatch Loss= 0.072394, Training Accuracy= 98.43750
Testing Accuracy: 98.4375
```