DAY 4

Multi-layer Neural Networks in TensorFlow

TensorFlow High Level API

Convolutional Neural Networks

- In this section we will show with a simple examples different “tricks” to program your own CNN in TensorFlow:

    1. Reconstruct an original shape of an image in TensorFlow
    2. Convolutional kernels in TensorFlow
    3. Pooling layers in TensorFlow
    4. Include a softmax layer and dropout
    5. Training the model

1. Reconstruct an original shape of an image

- A typical feature of CNN’s is that they nearly always have images as inputs.

- In the previos example, the images has been transformed in a bunch of points in a vectorial space of 784 dimensions.

  The shape of mnist.train.image is a 2D Tensor of

  ```
  TensorShape([Dimension(60000), Dimension(784)])
  ```

1. Reconstruct an original shape of an image

- We will reconstruct the original shape of the image of the input data with **tf. reshape** function:

```
x = tf.placeholder("float", [None, 784])
x_image = tf.reshape(x, [-1,28,28,1])
```

Here we changed the input shape to a 4D tensor, the second and third dimension correspond to the width and the height of the image while the last dimension corresponding number of color channels, 1 in this case.

2. Convolutional kernels in TensorFlow

For example, assume that we want to use 32 kernels, each one defined by a 5x5 weight matrix W and a bias b

- In order to simplify the code we will define:

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

(*)Initialized the weights with some random noise and the bias values slightly positive

2. Convolutional kernels in TensorFlow

For this example we must define a tensor to hold this weight matrix W with the **shape [5, 5, 1, 32]**

```
W_conv1 = weight_variable([5, 5, 1, 32])
```

- first two dimensions are the size of the window,
- third is the amount of channels, which is 1 in our case.
- last one defines how many features we want to use.

Furthermore, we will also need to define a bias for every of 32 weight matrices. Using the previously defined functions we can write this in TensorFlow as follows:

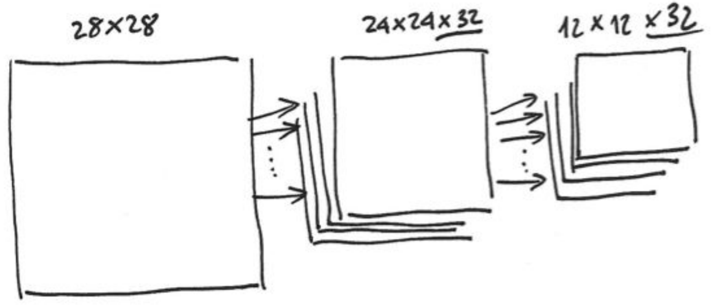
```
b_conv1 = bias_variable([32])
```

As we have already seen, there are several parameters that we have to define for the convolution and pooling layers. We will use a stride of size 1 in each dimension (this is the step size of the sliding window) and a zero padding model

3. Pooling layers in TensorFlow

In addition to the convolutional layers it is usual to be followed by a pooling layer. For example, assume that:

- we will use a 2x2 region of the convolution layer of which we summarize the data into a single point using pooling
- we will use the method called max-pooling (condensing the information by just retaining the maximum value in the 2x2 region)



(*) This leads that the 24x24 convolution result is transformed to a 12x12 space by the max-pooling layer that correspond to the 12x12 tiles, of which each originates from a 2x2 region.

3. Convolutional/Pooling in TensorFlow

- Two generic functions to be able to write a cleaner code:

```
def conv2d(x, W):

   return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


 def max_pool_2x2(x):
   return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
```

- Note: We will use a stride of size 1 in each dimension and a zero padding model.

3. Convolutional/Pooling in TensorFlow

- Code:

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

h_pool1 = max_pool_2x2(h_conv1)
```

- Note: We use ReLU activation function

3. Convolutional/Pooling in TensorFlow

- If we want to stack several layers on top of each other:

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

4. Softmax layer and Dropout

- Assume that we want to add a fully connected which will then be fed to a final softmax layer of 1024.
- The tensors required are:

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
```

(*) the first dimension of the *tensor* represents the 64 filters of size 7x7 from the second convolutional layer, while the second parameter is the amount of neurons in the layer. We will use a layer of 1024 neurons, allowing us to to process the entire image.

4. Softmax layer and Dropout

- Now we want to flatten the tensor into a vector:

(*) This is achieved by multiplying the weight matrix *W_fc1* with the flattend vector, adding the bias *b_fc1* after wich we apply the *ReLU* activation function

```
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

4. Softmax layer and Dropout

- The next step will be dropout (randomly)
    - To do this in a consistent manner we will assign a probability to the neurons being dropped or not in the code.
    - To do this we construct a placeholder to store the probability that a neuron is maintained during dropout

```
keep_prob = tf.placeholder("float")
```

- Using the function dropout tf.nn.dropout before the final softmax layer.

```
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

4. Softmax layer and Dropout

- Finally, we add the softmax layer un our example:

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

5. Training of the model

- Similar code to the previous example:

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
sess = tf.Session()

sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  sess.run(train_step,feed_dict={x: batch[0], y_: batch[1],
     keep_prob: 0.5})
```

(*) we replace the *gradient descent optimizer* with *the ADAM optimizer*

5. Training of the model

- If we want to evaluate our model ...

```
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

for i in range(20000):
  batch = mnist.train.next_batch(50)
  sess.run(train_step,feed_dict={x: batch[0], y_: batch[1],
     keep_prob: 0.5})
```

complete code of our example - 1

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
import tensorflow as tf

x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])

x_image = tf.reshape(x, [-1,28,28,1])

def weight_variable(shape):
 initial = tf.truncated_normal(shape, stddev=0.1)
 return tf.Variable(initial)
```

complete code of our example - 2

```
def weight_variable(shape):
initial = tf.truncated_normal(shape, stddev=0.1)
return tf.Variable(initial)

def bias_variable(shape):
initial = tf.constant(0.1, shape=shape)
return tf.Variable(initial)
```

complete code of our example - 3

```
def conv2d(x, W):
 return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

complete code of our example - 4

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

complete code of our example - 5

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

complete code of our example - 6

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

complete code of our example - 7

```
sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(200):
  batch = mnist.train.next_batch(50)
  if i%10 == 0:
    train_accuracy = sess.run( accuracy, feed_dict={
    x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  sess.run(train_step,feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

Class hands-on (or Homework)

- Download the code MultiLayerNeuralNetworks.py from github:
  - Accuracy?
  - Execution time problem? How to solve it?

Some TensorFlow High Level APIs

- TensorFlow Slim (now merged to tensorflow.contrib.layers)
  - Source
- TF Learn (aka Scikit Flow)
  - Source
  - Examples
- Pretty Tensor
  - Source
- Keras
  - Source
  - Documentation

TF contrib layers

- Lightweight library for defining, training and evaluating models in TensorFlow
- Simplifies complex networks definitions (Inception in TF)
- Modules are independent which adds flexibility
- Interesting for importing pre-trained models for fine-tuning
- Directly implements TensorBoard scopes for the graph definiton
- In some cases more limited to layer definition than raw TF, although they can be used alongside

TF contrib layers

- Layers
  - `convolution2d (inputs, num_outputs, kernel_size, stride=1, padding='SAME', activation_fn=nn.relu, normalizer_fn=None, normalizer_params=None, weights_initializer=initializers.xavier_initializer(), weights_regularizer=None, biases_initializer=init_ops.zeros_initializer, biases_regularizer=None, scope=None)`
  - `fully_connected (inputs, num_outputs, activation_fn=nn.relu, normalizer_fn=None, normalizer_params=None, weights_initializer=initializers.xavier_initializer(), weights_regularizer=None, biases_initializer=init_ops.zeros_initializer, biases_regularizer=None, scope=None)`
    - Initializes a ‘weight’ variable with given initializers ➝ initializers.py, init_ops.py
    - Applies the activation function (if given) at the end ➝ Activation functions
    - Initializes a ‘bias’ variable (if no batch_norm_params)
    - Applies a normalizer function (if given) with parameters ➝ batch_norm()
    - Applies regularization function (if given) ➝ regularizers.py

TF contrib layers

- Layers
  - `batch_norm (inputs, center=True, scale=False, epsilon=0.001, activation_fn=None, scope=None)`
    - tf.nn.batch_normalization (Python API)
  - `max_pool2d (inputs, kernel_size, stride=2, padding='VALID', scope=None)`
  - `avg_pool2d (inputs, kernel_size, stride=2, padding='VALID', scope=None)`
  - `dropout (inputs, keep_prob=0.5, is_training=True, scope=None)`
  - `flatten (inputs, scope=None)`
    - ‘bridge’ function for reshaping tensors before `fully_connected`
  - `repeat (inputs, repetitions, layer, *args, **kwargs)`
    - not included in v0.9

TF contrib layers

- Scopes
  - Allows to define a set of arguments for a given type of layers

```
from tensorflow.contrib.layers.python.layers import layers as slim
from tensorflow.contrib.framework.python.ops import arg_scope


with arg_scope([slim.conv2d, slim.fully_connected],
               weights_initializer=init_ops.zeros_initializer,
               biases_initializer=init_ops.random_normal_initializer(stddev=0.1)):
  net = slim.conv2d(inputs, 64, [11, 11], 4, padding='VALID', scope='conv1')
  net = slim.conv2d(net, 256, [5, 5], biases_initializer=tf.zeros_initializer, scope='conv2')
  net = slim.flatten(net)
  net = slim.fully_connected(net, 1000, activation=None, scope='fc')
```

TF contrib layers

- Homework
  - Build a neural network with the tools seen today