CS405 Machine Learning

Lab #1 CNN

Lab (75 points):

Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: rom the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply. ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. In this lab, you need to learn the architecture of the CNN and write your own CNN minist classifier. The dataset has been illustrated in the pre lab. You need submit the your code and compare the performance between the model provided in the pre lab and your own CNN model.

Instruction

A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for minist dataset could have the architecture [Input-Conv-Pool-Conv-Pool-FC]. In more detail:

Input layer

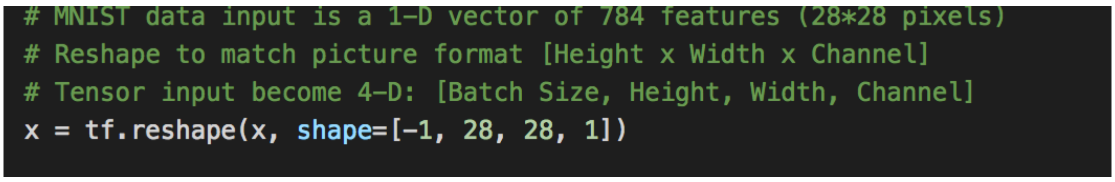
The methods in the layers module for creating convolutional and pooling layers for two-dimensional image data expect input tensors to have a shape of [*batch_size*, *image_height*, *image_width*, *channels*] by default. This behavior can be changed using the *data_format* parameter; defined as follows:

- *batch_size*. Size of the subset of examples to use when performing gradient descent during training.
- *image_height*. Height of the example images.
- *image_width*. Width of the example images.
- *channels*. Number of color channels in the example images. For color images, the number of channels is 3 (red, green, blue). For monochrome images, there is just 1 channel (black).

- *data_format*. A string, one of channels_last (default) or channels_first. channels_last corresponds to inputs with shape (batch, ..., channels) while channels_first corresponds to inputs with shape (batch, channels, ...).

Here, our MNIST dataset is composed of monochrome 28x28 pixel images, so the desired shape for our input layer is [*batch_size*, 28, 28, 1].

To convert our input feature map (features) to this shape, we can perform the following reshape operation:

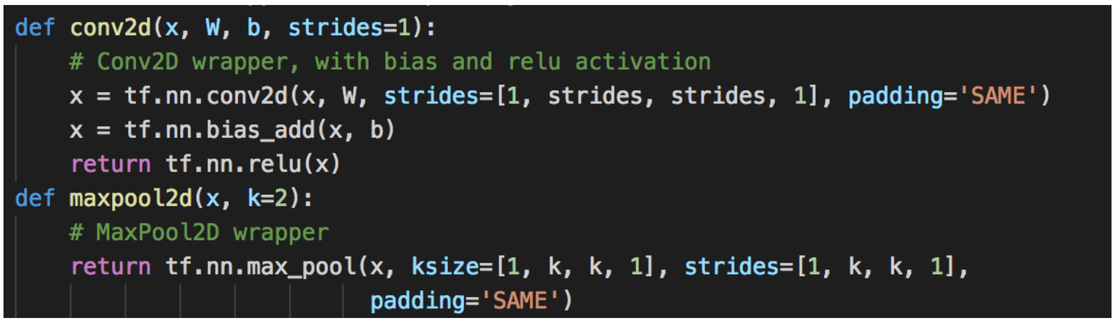


```
# MNIST data input is a 1-D vector of 784 features (28*28 pixels)
# Reshape to match picture format [Height x Width x Channel]
# Tensor input become 4-D: [Batch Size, Height, Width, Channel]
x = tf.reshape(x, shape=[-1, 28, 28, 1])
```

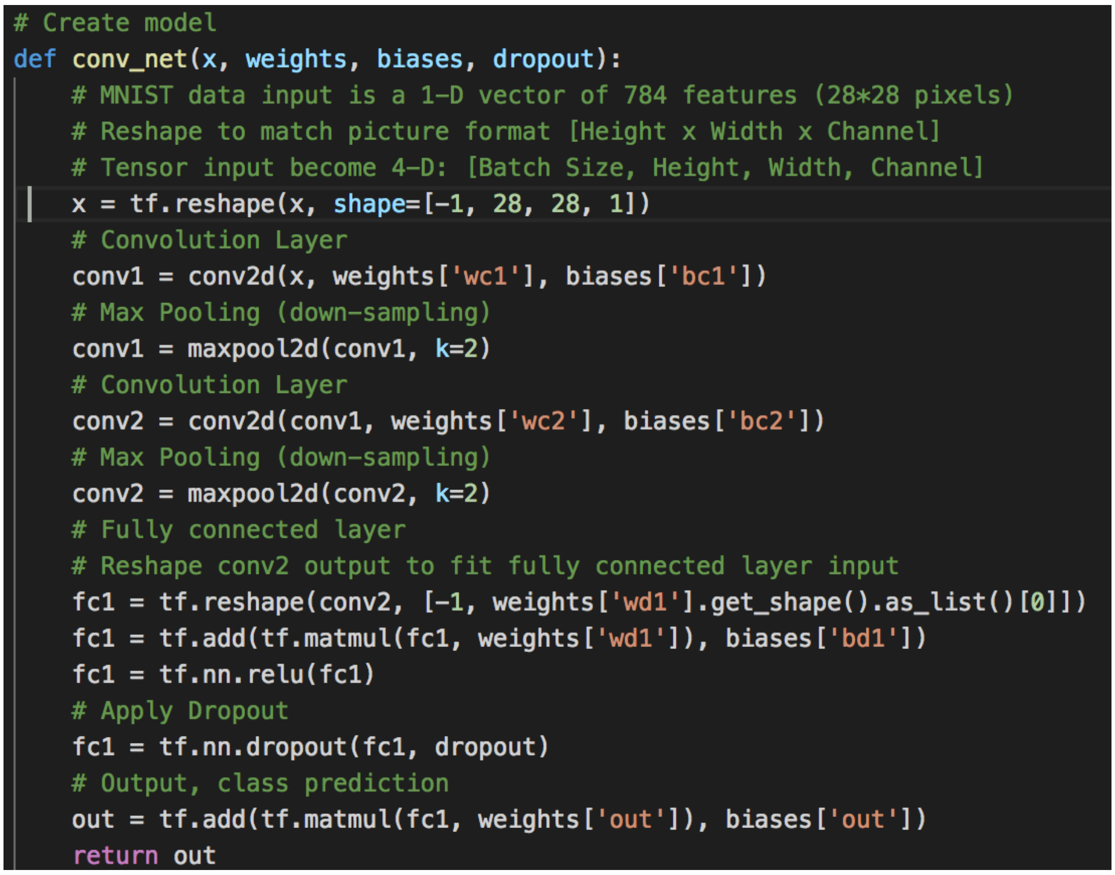
Note that we've indicated -1 for batch size, which specifies that this dimension should be dynamically computed based on the number of input values in features["x"], holding the size of all other dimensions constant. This allows us to treat batch_size as a hyperparameter that we can tune. For example, if we feed examples into our model in batches of 5, features["x"] will contain 3,920 values (one value for each pixel in each image), and input_layer will have a shape of [5, 28, 28, 1]. Similarly, if we feed examples in batches of 100, features["x"] will contain 78,400 values, and input_layer will have a shape of [100, 28, 28, 1].

Convolutional Layer #1-Pooling Layer #1-Convolutional Layer #2-Pooling Layer #2-Fully Convolutional Layer

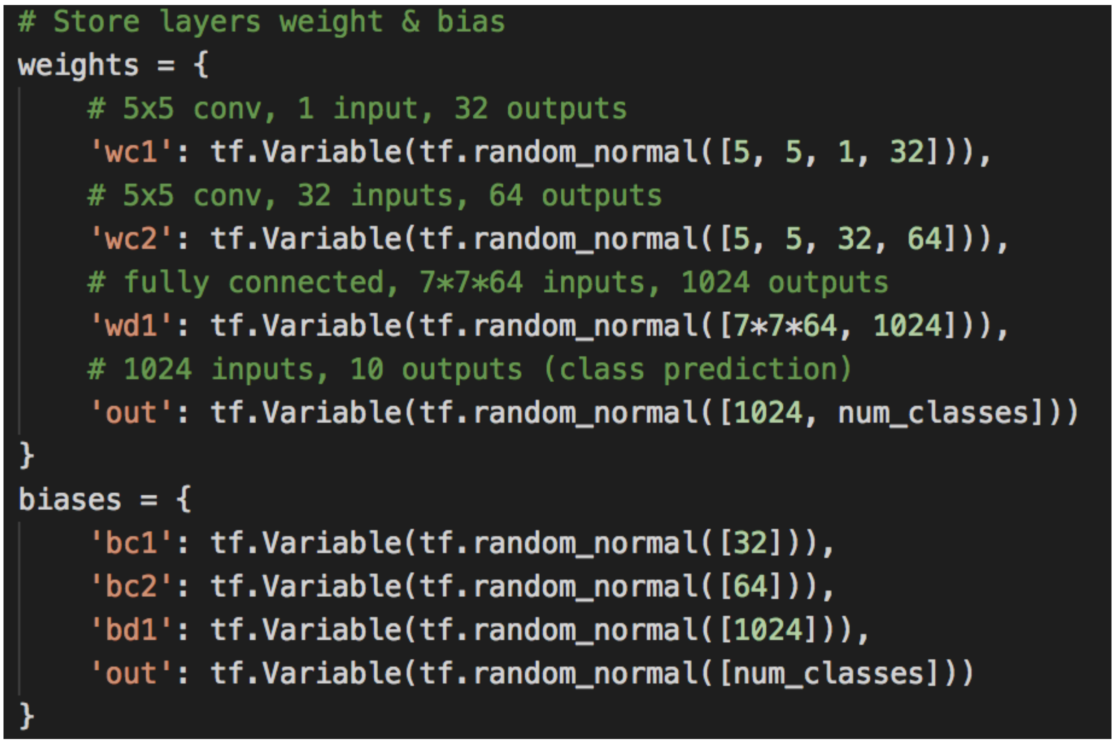
In our first convolutional layer, we want to apply 32 5x5 filters to the input layer, with a ReLU activation function, we can use the `conv2d()` method. Next, we connect our first pooling layer to the convolutional layer we just created. We can use the max_pooling2d() method in layers to construct a layer that performs max pooling with a 2x2 filter and stride of 2. We can connect a second convolutional and pooling layer to our CNN using conv2d() and max_pooling2d() as before. For convolutional layer #2, we configure 64 5x5 filters with ReLU activation, and for pooling layer #2, we use the same specs as pooling layer #1 (a 2x2 max pooling filter with stride of 2). Next, we want to add fully convolutional layer (with 1,024 neurons and ReLU activation) to our CNN to perform classification on the features extracted by the convolution/pooling layers. The example code is provided below.



```python
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
def maxpool2d(x, k=2):
    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                          padding='SAME')
```



```python
# Create model
def conv_net(x, weights, biases, dropout):
    # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)
    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)
    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)
    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

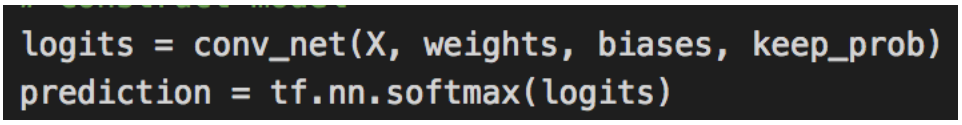


```python
# Store layers weight & bias
weights = {
    # 5x5 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    # 5x5 conv, 32 inputs, 64 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    # fully connected, 7*7*64 inputs, 1024 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    # 1024 inputs, 10 outputs (class prediction)
    'out': tf.Variable(tf.random_normal([1024, num_classes]))
}
biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

In the reshape() operation above, the -1 signifies that the *batch_size* dimension will be dynamically calculated based on the number of examples in our input data. Each example has 7 (pool2 height) *

7 (pool2 width) * 64 (pool2 channels) features, so we want the features dimension to have a value of 7 * 7 * 64 (3136 in total). The output tensor, pool2_flat, has shape [*batch_size*, 3136].

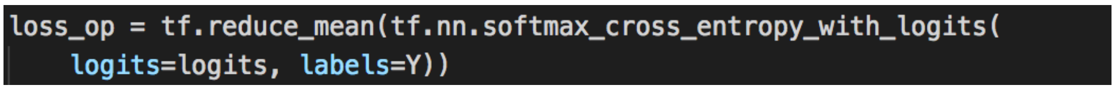
Generate Predictions



```
logits = conv_net(X, weights, biases, keep_prob)
prediction = tf.nn.softmax(logits)
```

Calculate Loss

For both training and evaluation, we need to define a loss function that measures how closely the model's predictions match the target classes. For multiclass classification problems like MNIST, cross entropy is typically used as the loss metric. The following code calculates cross entropy when the model runs in either TRAIN or EVAL mode:



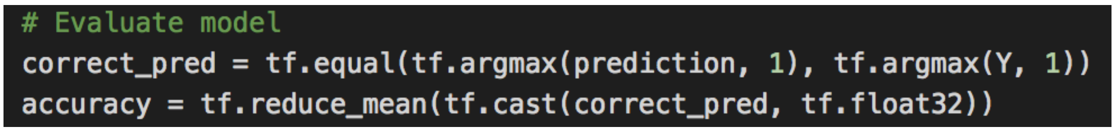
```
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
```

Let's take a closer look at what's happening above.

Our labels tensor contains a list of prediction indices for our examples, e.g. [1, 9, ...]. logits contain the linear outputs of our last layer.

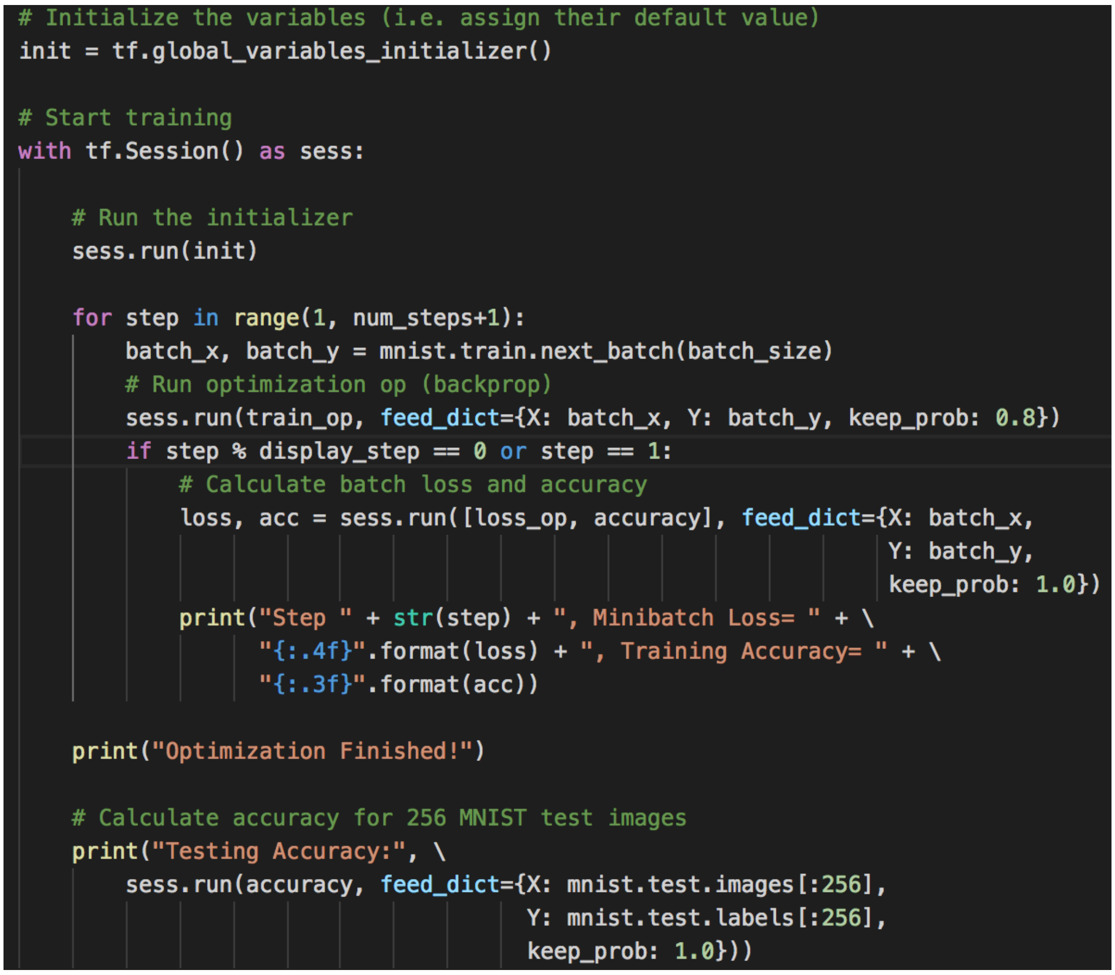
tf.losses.sparse_softmax_cross_entropy, calculates the softmax crossentropy (aka: categorical crossentropy, negative log-likelihood) from these two inputs in an efficient, numerically stable way.

Add evaluation metrics



```python
# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Training and Evaluating the CNN classifier using minist dataset.



```python
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y, keep_prob: 0.8})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y,
                                                                 keep_prob: 1.0})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 256 MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images[:256],
                                      Y: mnist.test.labels[:256],
                                      keep_prob: 1.0}))
```