Exploring a Convoluted Neural Netowrk¶

In this notebook, we will use a simple convoluted neural network to predict digits with around a 97% accuracy using the mnist dataset that contains 60,000 handwritten digits. We accomlish this by using `Tensorflow`, Google's machine learning API, so start by importing `Tensorflow` and a few other necessary libraries.

Contributers: Daniel Silver

Credit to Dyllan Elliot for the code used to import the mnist data

In [24]:

```
import sys
import tensorflow as tf
import time
import numpy
import matplotlib.pyplot as plt
import cv2
import os
import gzip
```

Importing Training and Test Data¶

This example will assume that all of the mnist data is located in the local directory of this program in .gz files. The files can be found directly on the mnist website here:http://yann.lecun.com/exdb/mnist/

In [28]:

```
# MNIST datasets
MNIST_TRAIN_IMG = 'train-images-idx3-ubyte.gz'
MNIST_TRAIN_LAB = 'train-labels-idx1-ubyte.gz'
MNIST_TEST_IMG = 't10k-images-idx3-ubyte.gz'
MNIST_TEST_LAB = 't10k-labels-idx1-ubyte.gz'
def load_mnist(path, kind='train'):


    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte.gz'
                               % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte.gz'
                               % kind)

    with gzip.open(labels_path, 'rb') as lbpath:
        labels = numpy.frombuffer(lbpath.read(), dtype=numpy.uint8,
                               offset=8)

    with gzip.open(images_path, 'rb') as imgpath:
        images = numpy.frombuffer(imgpath.read(), dtype=numpy.uint8,
                               offset=16).reshape(len(labels), 784)

    return images, labels
# import nnist training images
data_path=""
X_train, Y_train = load_mnist(data_path, kind='train')
X_test, Y_test = load_mnist(data_path, kind='t10k')
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
```

HyperParameters¶

The HyperParameters that are used are depicted below.

In [29]:

```
# Hyperparameters
n_epochs = 10
learning_rate = .0001
batch_size = 50
```

Parameters¶

The Parameters that are used are depicted below. Note: The amount of nodes in each layer increases as features throughout the convolution become more pronounced from one layer to the next. accuracy_frequency is used to conserve computer memory. Decrease this number if you are limited on memory allocation.

In [6]:

```
# Parameters
layer1_output_nodes = 32
layer2_output_nodes = 64
layer3_output_nodes = 128
dense_layer1_output_nodes = 1024
dense_layer2_output_nodes = 2048
img_width = 28
img_height = 28
stride = 1
classes = 10
accuracy_frequency = 100
```

Place Holders:¶

The placeholder, x, is used to store the shape for the input features. In this case, the input features are 28 by 28 pixel values that can on one color value as this is only representing black and white, so the dimensionality of this parameter is 1. Hence the value `1` in `tf.placeholder('float', [None, 28, 28, 1]`. The y placeholder will only store labels for the actual value of the digits, stored as ints from 0 to 9.

In [7]:

```
x = tf.placeholder('float', [None, 28, 28, 1], name="x")
y = tf.placeholder('int32', [None], name="labels")
```

Creating the Model Architecture¶

In this example, we have 3 convoluted layers and 3 dense layers. Note: There does not need to be the same amount of layers as dense layers.

The Convoluted Layers¶

We create each convoluted layer one at a time using our `create_new_conv_layer(input, num_filters, filter_shape, pool_shape, stride, name)` function. For each layer we will pass in the **previous layer** for input. The first input to create the first layer will be `x` because `x`, once fed, will be storing our feature data. We use a filter of shape `[5,5]` and a pool of shape `[2,2]`, with a stride of size`1`.

In [31]:

```
# creates a convoluted layer for use in a convoluted network
# Args:
#   - input: the input data for the layer, use previous layer if there is one
#   - num_filters: the number fo features feeding into the layer
#   - filter_shape: a 2 element array where the first represents width of filer, and second represents height. ex. [5,5] a 5 x 5 filter
#   - pool_shape: a 2 element array where the first represents width of the pool, and second represents height. ex. [2,2] a 2 x 2 pool
#   - pool_stride: the size of the stride for the filer
# Returns:
#   - a new convoluted layer


def create_new_conv_layer(input, num_filters, filter_shape, pool_shape, stride, name):
    out_layer = tf.layers.conv2d(input, filters=num_filters, kernel_size=[filter_shape[0], filter_shape[1]],
                                 strides=[stride, stride], padding='SAME', activation=tf.nn.relu)
    pool_size = [pool_shape[0], pool_shape[1]]
    strides = [2, 2]
    out_layer = tf.layers.max_pooling2d(
        out_layer, pool_size=pool_size, strides=strides, padding='SAME')
    return out_layer
```

In [32]:

```
layer1 = create_new_conv_layer(
    x, layer1_output_nodes, [5, 5], [2, 2], stride, "layer1")
layer2 = create_new_conv_layer(
    layer1, layer2_output_nodes, [5, 5], [2, 2], stride, "layer2")
layer3 = create_new_conv_layer(
    layer2, layer3_output_nodes, [5, 5], [2, 2], stride, "layer2")
```

The Dense Layers¶

We again create our desnse layers one at a time, this time using our `create_dense_layer(input, units, namespace)` function. For each layer we will pass in our **previous layer** for input. Deep layers must take in a vectorized tensor, so me must flatten our last convulted layer. Our final dense layer must output with a number of nodes equal to the number of classes to be able to classify our images as 1 of 10 digits. We then use the **softmax** function to normalize our outputs. **Softmax** is equivalent to `tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis)`.

In [33]:

```
# creates a dense layer for a convuluted netowrk
# Args:
#   - input: the vectorized input tensor
#   - units: the number of output nodes from this layer
#   - namespace: the namespace for this layer
# Returns:
#   - a new dense layer


def create_dense_layer(input, units, namespace):
    with tf.name_scope(namespace):
        return tf.layers.dense(input, units)
```

In [34]:

```
flattened = tf.contrib.layers.flatten(layer3)
dense_layer1 = create_dense_layer(
    flattened, dense_layer1_output_nodes, "fc1")
dense_layer2 = create_dense_layer(
    dense_layer1, dense_layer2_output_nodes, "fc2")
dense_layer3 = create_dense_layer(
    dense_layer2, classes, "fc2")
y_ = tf.nn.softmax(dense_layer3)
```

Cost¶

We use cross entroy to calculate our loss function which is denoted as `cost` in this example.

In [35]:

```
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=dense_layer3, labels=tf.one_hot(y, classes)))
```

Training¶

We use an AdamOptimizer to train our model. Note: There are many other valid optimizers that can be used here. I just found AdamOptimizer to work very well for this architecture.

In [36]:

```
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

Testing¶

We allocate a graph node named correct to calculate for each data point whether of not our guess was correct. We then create a graph node named accuracy to calculate the accuracy of our model after any given epoch.

In [37]:

```
correct = tf.equal(tf.argmax(tf.one_hot(tf.cast(y, 'int64'), classes), 1),
                   tf.argmax(tf.cast(y_, 'int64'), 1))
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Creating and Running the Session¶

We initialize a session and run through a certain amount of batches per epoch (The amount depends on the size of the batch as every epoch will be testing every data point once). We then run all of our training data in batches, using our optimizer to utilize its **Adam Optimization** algorithym to employ backpropogation to let our model fix itself to learn from the data by adjusting weights and biases. Afterwards, we will run our test data against our model and test our **accuracy**. This will occur for every **epoch** until the program finishes running.

In [40]:

```
# gets data for a batch from the entire dataset
# Args:
#   - batch_size: the size of the batch to fetch
#   - count: the current count within the epoch to continue from for this batch
#   - X_train: the training data
#   - Y_train: the training labels
# Returns:
#   - mini-batch of x's and y's of size batch_size

def get_mini_batch(batch_size, count, X_train, Y_train, Test=False):
    batch_x = []
    batch_y = []
    if Test and (batch_size * count + batch_size) == len(X_train):
        for i in range(batch_size - 1):
            batch_x.append(X_train[count * batch_size + i])
```

```
            batch_y.append(Y_train[count * batch_size + i])
            return batch_x, batch_y
    for i in range(batch_size):
        batch_x.append(X_train[count * batch_size + i])
        batch_y.append(Y_train[count * batch_size + i])
    return batch_x, batch_y
```

In [ ]:

```
with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for epoch in range(n_epochs):
            current_accuracy = 0
            accuracy_calculations = 0
            epoch_loss = 0
            accpoints = []
            accuracy_points = []
            iterations_per_epoch = int(len(X_train) / batch_size)
            for j in range(iterations_per_epoch):
                batch_x, batch_y = get_mini_batch(
                    batch_size, j, X_train, Y_train)
                feed = {x: batch_x, y: batch_y}
                _, c = sess.run([optimizer, cost], feed_dict={
                                x: batch_x, y: batch_y})
                epoch_loss += c
            for i in range(accuracy_frequency):
                X_test_batch, Y_test_batch = get_mini_batch(
                    int(len(X_test) / accuracy_frequency), i, X_test, Y_test, True)
                a = sess.run(accuracy, feed_dict={
                             x: X_test_batch, y: Y_test_batch})
                accuracy_points.append(a)
            print("Total Accuracy:  " + str(numpy.mean(accuracy_points)))
            accpoints.append(numpy.mean(accuracy_points))
```

Calculating Accuracy¶

```
for i in range(accuracy_frequency):
                X_test_batch, Y_test_batch = nnUtils.get_mini_batch(
                    int(len(X_test) / accuracy_frequency), i, X_test, Y_test, True)
                a = sess.run(accuracy, feed_dict={x: X_test_batch, y: Y_test_batch})
                accuracy_points.append(a)
```

Due to the fact that the test data is so large (10,000 images), it can be a burden on your computer to store it all in memory at once. By splitting in up into chunks of 1/accuracy_frequency size of the data set, we can achieve the same accuracy calculation with lower memory burden.