Handwritten Number Recognition with TFLearn and MNIST

In this notebook, we'll be building a neural network that recognizes handwritten numbers 0-9.

This kind of neural network is used in a variety of real-world applications including: recognizing phone numbers and sorting postal mail by address. To build the network, we'll be using the **MNIST** data set, which consists of images of handwritten numbers and their correct labels 0-9.

We'll be using TFLearn, a high-level library built on top of TensorFlow to build the neural network. We'll start off by importing all the modules we'll need, then load the data, and finally build the network.

In [1]:

```
# Import Numpy, TensorFlow, TFLearn, and MNIST data
import numpy as np
import tensorflow as tf
import tflearn
import tflearn.datasets.mnist as mnist
```

Retrieving training and test data

The MNIST data set already contains both training and test data. There are 55,000 data points of training data, and 10,000 points of test data.

Each MNIST data point has:

1. an image of a handwritten digit and
2. a corresponding label (a number 0-9 that identifies the image)

We'll call the images, which will be the input to our neural network, **X** and their corresponding labels **Y**.

We're going to want our labels as *one-hot vectors*, which are vectors that holds mostly 0's and one 1. It's easiest to see this in a example. As a one-hot vector, the number 0 is represented as [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], and 4 is represented as [0, 0, 0, 0, 1, 0, 0, 0, 0, 0].

Flattened data

For this example, we'll be using *flattened* data or a representation of MNIST images in one dimension rather than two. So, each handwritten number image, which is 28x28 pixels, will be represented as a one dimensional array of 784 pixel values.

Flattening the data throws away information about the 2D structure of the image, but it simplifies our data so that all of the training data can be contained in one array whose shape is [55000, 784]; the first dimension is the number of training images and the second dimension is the number of pixels in each image. This is the kind of data that is easy to analyze using a simple neural network.

In [2]:

```
# Retrieve the training and test data
trainX, trainY, testX, testY = mnist.load_data(one_hot=True)
```

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

Visualize the training data

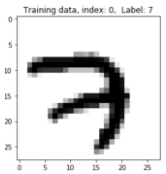
Provided below is a function that will help you visualize the MNIST data. By passing in the index of a training example, the function `show_digit` will display that training image along with it's corresponding label in the title.

In [3]:

```
# Visualizing the data
import matplotlib.pyplot as plt
%matplotlib inline

# Function for displaying a training image by it's index in the MNIST set
def display_digit(index):
    label = trainY[index].argmax(axis=0)
    # Reshape 784 array into 28x28 image
    image = trainX[index].reshape([28,28])
    plt.title('Training data, index: %d,  Label: %d' % (index, label))
    plt.imshow(image, cmap='gray_r')
    plt.show()

# Display the first (index 0) training image
display_digit(0)
```



Building the network

TFLearn lets you build the network by defining the layers in that network.

For this example, you'll define:

1. The input layer, which tells the network the number of inputs it should expect for each piece of MNIST data.
2. Hidden layers, which recognize patterns in data and connect the input to the output layer, and
3. The output layer, which defines how the network learns and outputs a label for a given image.

Let's start with the input layer; to define the input layer, you'll define the type of data that the network expects. For example,

```
net = tflearn.input_data([None, 100])
```

would create a network with 100 inputs. The number of inputs to your network needs to match the size of your data. For this example, we're using 784 element long vectors to encode our input data, so we need **784 input units**.

Adding layers

To add new hidden layers, you use

```
net = tflearn.fully_connected(net, n_units, activation='ReLU')
```

This adds a fully connected layer where every unit (or node) in the previous layer is connected to every unit in this layer. The first argument `net` is the network you created in the `tflearn.input_data` call, it designates the input to the hidden layer. You can set the number of units in the layer with `n_units`, and set the activation function with the `activation` keyword. You can keep adding layers to your network by repeated calling `tflearn.fully_connected(net, n_units)`.

Then, to set how you train the network, use:

```
net = tflearn.regression(net, optimizer='sgd', learning_rate=0.1,
loss='categorical_crossentropy')
```

Again, this is passing in the network you've been building. The keywords:

- `optimizer` sets the training method, here stochastic gradient descent
- `learning_rate` is the learning rate
- `loss` determines how the network error is calculated. In this example, with categorical cross-entropy.

Finally, you put all this together to create the model with `tflearn.DNN(net)`.

In [4]:

```
# Define the neural network
def build_model():
    # This resets all parameters and variables, leave this here
    tf.reset_default_graph()

    # Inputs
    net = tflearn.input_data([None, trainX.shape[1]])

    # Hidden layer(s)
    net = tflearn.fully_connected(net, 128, activation='ReLU')
    net = tflearn.fully_connected(net, 32, activation='ReLU')

    # Output layer and training model
    net = tflearn.fully_connected(net, 10, activation='softmax')
    net = tflearn.regression(net, optimizer='sgd', learning_rate=0.01,
loss='categorical_crossentropy')

    model = tflearn.DNN(net)
    return model
```

In [5]:

```
# Build the model
model = build_model()
```

```
WARNING:tensorflow:From //anaconda3/envs/tensorflow/lib/python3.5/site-
packages/tflearn/summaries.py:46 in get_summary.: scalar_summary (from
tensorflow.python.ops.logging_ops) is deprecated and will be removed after 2016-
11-30.
Instructions for updating:
Please switch to tf.summary.scalar. Note that tf.summary.scalar uses the node
name instead of the tag. This means that TensorFlow will automatically de-
duplicate summary names based on the scope they are created in. Also, passing a
tensor or list of tags to a scalar summary op is no longer supported.
```

Training the network

Now that we've constructed the network, saved as the variable `model`, we can fit it to the data. Here we use the `model.fit`method. You pass in the training features `trainX` and the training targets `trainY`. Below I set `validation_set=0.1` which reserves 10% of the data set as the validation set. You can also set the batch size and number of epochs with the `batch_size`and `n_epoch` keywords, respectively.

Too few epochs don't effectively train your network, and too many take a long time to execute. Choose wisely!

In [6]:

```
# Training
model.fit(trainX, trainY, validation_set=0.1, show_metric=True, batch_size=100,
n_epoch=100)
```

```
Training Step: 49500  | total loss: 0.06119
| SGD | epoch: 100 | loss: 0.06119 - acc: 0.9824 | val_loss: 0.10663 - val_acc:
0.9705 -- iter: 49500/49500
Training Step: 49500  | total loss: 0.06119
| SGD | epoch: 100 | loss: 0.06119 - acc: 0.9824 | val_loss: 0.10663 - val_acc:
0.9705 -- iter: 49500/49500
--
```

Testing

After you're satisified with the training output and accuracy, you can then run the network on the **test data set** to measure it's performance! Remember, only do this after you've done the training and are satisfied with the results.

A good result will be **higher than 95% accuracy**. Some simple models have been known to get up to 99.7% accuracy!

In [7]:

```
# Compare the labels that our model predicts with the actual labels

# Find the indices of the most confident prediction for each item. That tells
us the predicted digit for that sample.
predictions = np.array(model.predict(testX)).argmax(axis=1)

# Calculate the accuracy, which is the percentage of times the predicated
labels matched the actual labels
actual = testY.argmax(axis=1)
test_accuracy = np.mean(predictions == actual, axis=0)

# Print out the result
print("Test accuracy: ", test_accuracy)
```

```
Test accuracy:  0.9704
```