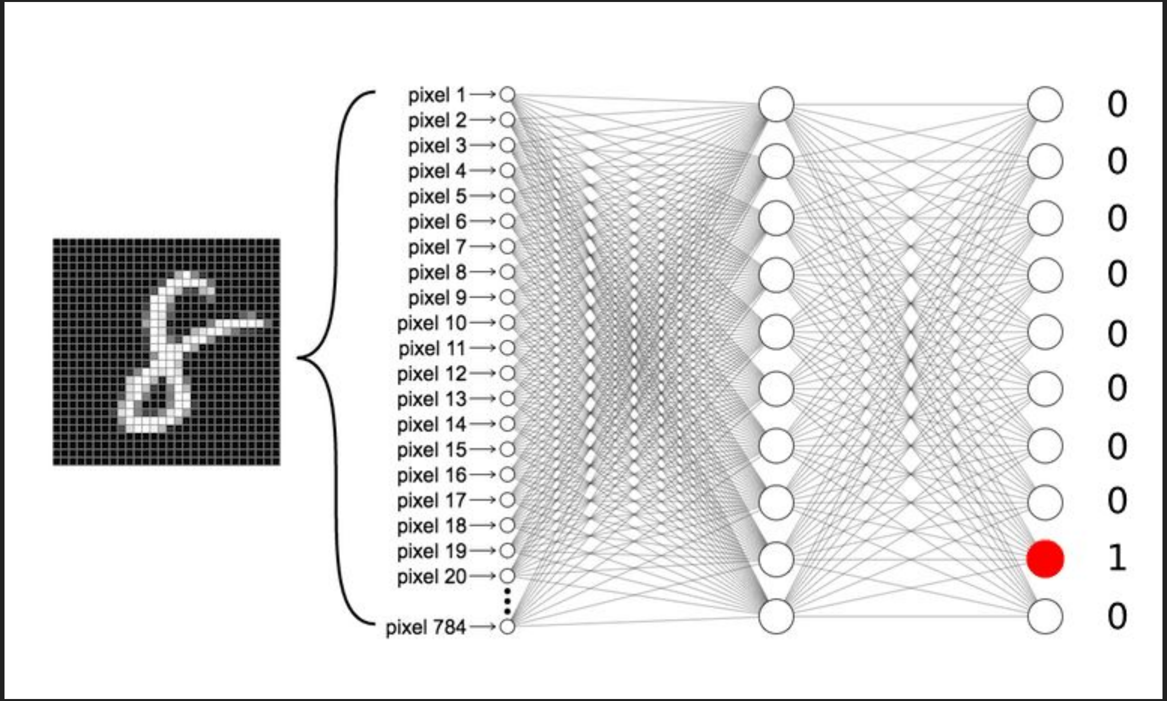
Intro to Tensorflow part 2

Classifying MNIST using fully connected NN



Importing libraries and loading mnist data

```
import tensorflow as tf

# Useful for n-dimensional array operations
import numpy as np

# Useful for plotting graphs/images
import matplotlib.pyplot as plt
```

```
pip install tensorflow

pip install numpy

pip install matplotlib
```

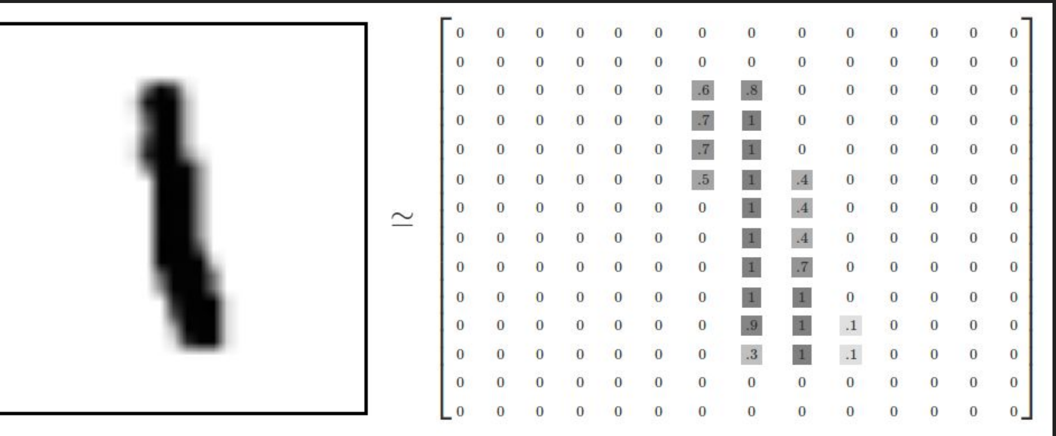
```
# Helper class for importing MNIST dataset
from tensorflow.examples.tutorials.mnist import input_data

# Loads MNIST dataset with one-hot encoding
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Setting up hyperparameters

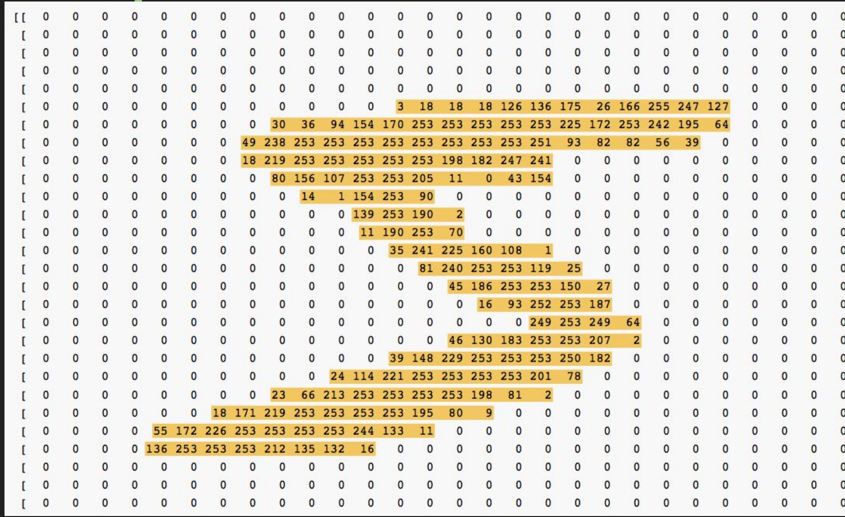
```
# Define number of input neurons in the network (image_size * image_size = 784 neurons)
image_size = 28

# Define number of output neurons in the network (output goes from 0 - 9, 10 neurons)
num_labels = 10

# No. of neurons in the first hidden layer
num_neurons_hidden_layer1 = 10

# Controls the rate of change of weights in neurons
learning_rate = 0.05

# No. of times the network has to see the same data during training
num_iterations = 1000

# For batch-wise training, each batch will have 100 images
batch_size = 100
```

A closer look at the data

```
# Get random 100 images (batch_size=100) and their corresponding ground-truth from the training set
input_batch, labels_batch = mnist.train.next_batch(batch_size)

# print a 1D array with pixel value (for eg [0, 1, 0, 0, 1, 1, 1, 0, 1, 0,...]
print(input_batch[5])
```



Pixel values are between 0 and 1



Usually the range is 0 - 255

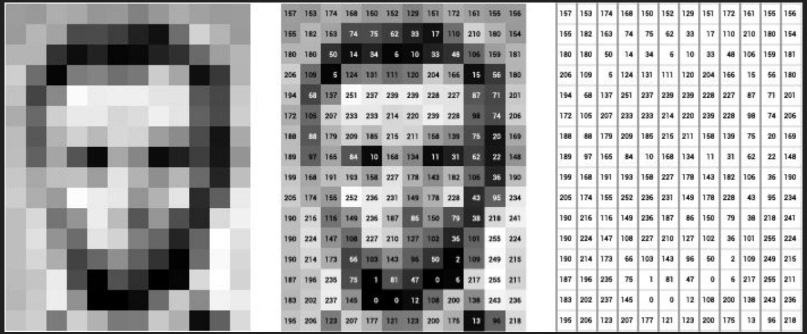
Data Preprocessing

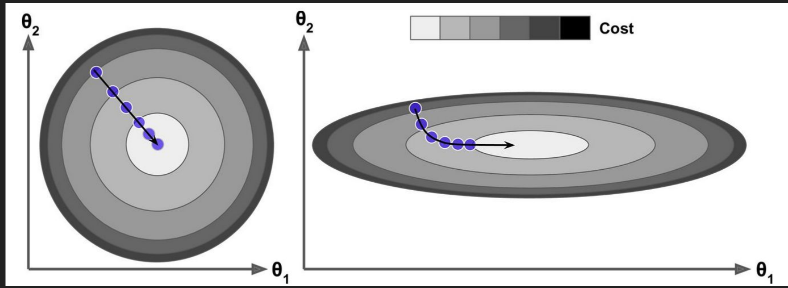
MNIST dataset is already normalized

# Data Normalization
pixel_value = pixel_value / 255

# We can do the same operation on a numpy array of images
image_array = image_array / 255

Why we need data normalized?
1) To standardize the input by bringing it to the same scale
2) Gradient descent (our optimizer) will converge faster





A closer look at the data

```
# Get random 100 images (batch_size=100) and their corresponding ground-truth from the training set
input_batch, labels_batch = mnist.train.next_batch(batch_size)

# print a 1D array with pixel value (for eg [0, 1, 0, 0, 1, 1, 1, 0, 1, 0,...]
print(input_batch[5])

# To plot the image we need to reshape the 1D array to 2D array of shape 28x28
plt.imshow(np.reshape(input_batch[5], [28, 28]), cmap='gray')
print('Sample Input')
plt.show()

print('Sample output (one hot encoding)')
print(labels_batch[5])
```

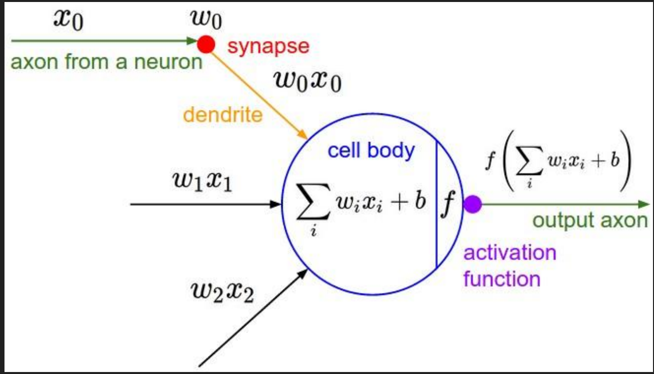
Build the graph

```
# Define placeholders

# placeholder to store batch training data per iteration. Shape = [None, 784]
training_data = tf.placeholder(tf.float32, [None, image_size*image_size])

# placeholder to store batch labels per iteration. Shape = [None, 10]
labels = tf.placeholder(tf.float32, [None, num_labels])
```

Build the graph

```
# Variables to be tuned. These are the learned parameters.

# We initialize the weights with random values from a normal distribution using tf.truncated_normal()
# While training, our optimizer will update the weight values for us

# Weights and bias for the hidden layer. Shape = [784, 10]
W1 = tf.Variable(tf.truncated_normal([image_size*image_size, num_neurons_hidden_layer1], stddev=0.1))
# Shape = [10]
b1 = tf.Variable(tf.constant(0.1, shape=[num_neurons_hidden_layer1]))

# Weights and bias for the output layer. Shape = [10, 10]
W2 = tf.Variable(tf.truncated_normal([num_neurons_hidden_layer1, num_labels], stddev=0.1))
# Shape = [10]
b2 = tf.Variable(tf.constant(0.1, shape=[num_labels]))
```

Build the Neural Network



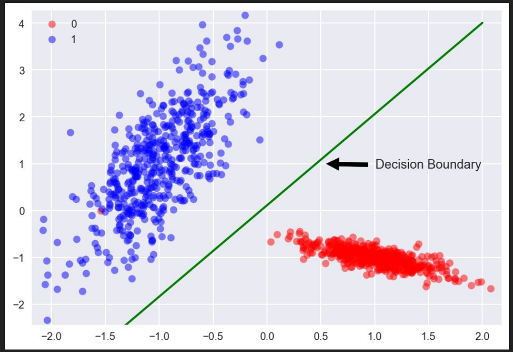
Two functions in the neuron

1) Linear Function: Wx + b

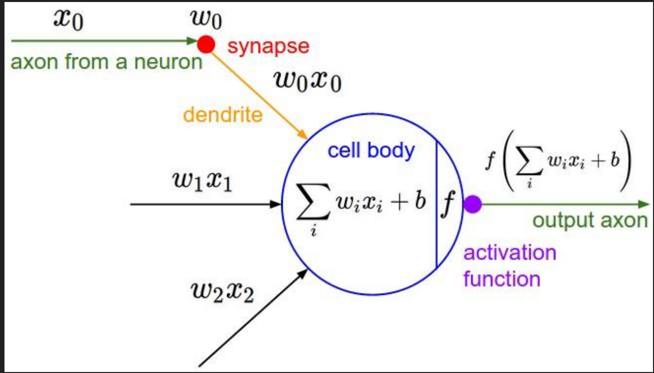
Wx + b is basically y = Mx + c, equation of a straight line, where M is the slope/gradient and c is the y-intercept

Wx+b is preferred because:

- It's easier to work with
- Straight lines are useful to model decision boundaries



Build the Neural Network

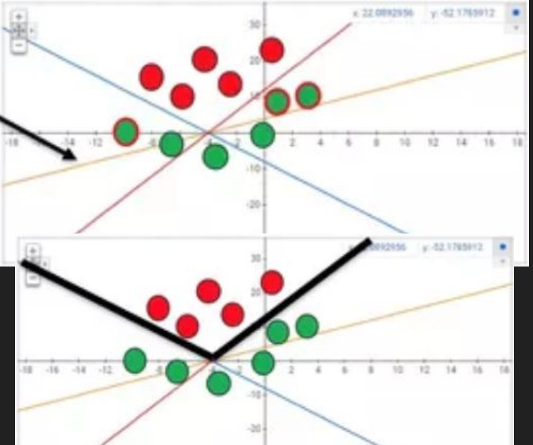


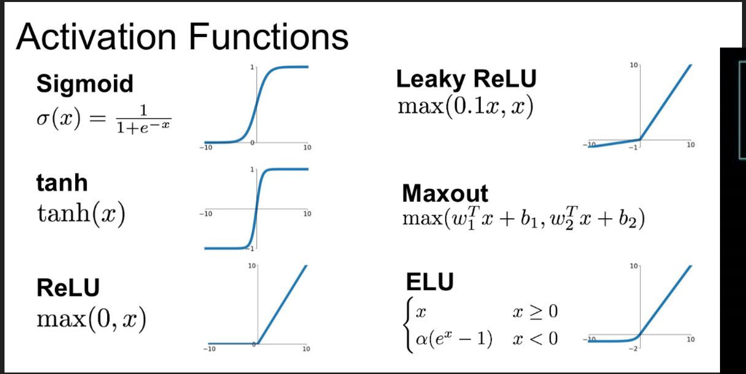
Two functions in the neuron

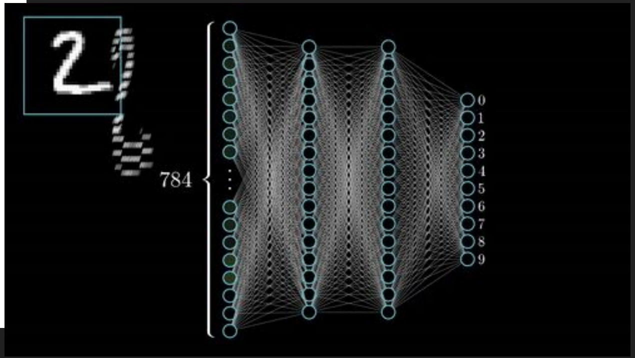
2) Activation function: σ(Wx + b)

σ activates neuron based on the output of the linear function

Creates non-linearity in the network

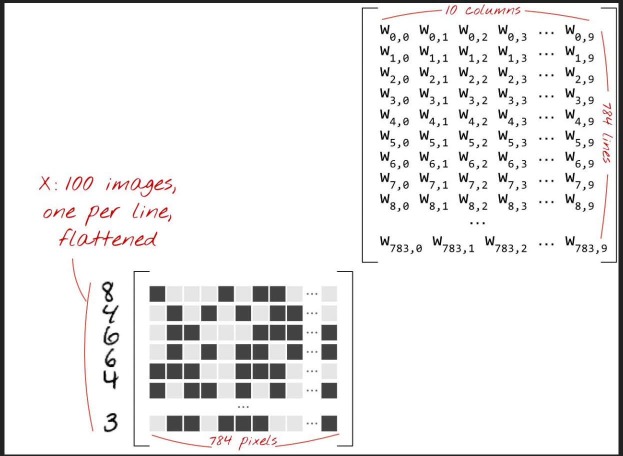






Build the Neural Network

```
# Neural Network

# Wx + b
hidden_layer1 = tf.matmul(training_data, W1) + b1

# Activation function ReLU is applied
hidden_layer1 = tf.nn.relu(hidden_layer1)

# Wx + b
output_layer = tf.matmul(hidden_layer1, W2) + b2
```



TF Layers

```
# Weights and bias for the hidden layer. Shape = [784, 10]
W1 = tf.Variable(tf.truncated_normal([image_size*image_size, num_neurons_hidden_layer1],
stddev=0.1))
# Shape = [10]
b1 = tf.Variable(tf.constant(0.1, shape=[num_neurons_hidden_layer1]))

# Weights and bias for the output layer. Shape = [10, 10]
W2 = tf.Variable(tf.truncated_normal([num_neurons_hidden_layer1, num_labels], stddev=0.1))
# Shape = [10]
b1 = tf.Variable(tf.constant(0.1, shape=[num_labels]))

# Neural Network

# Wx + b
hidden_layer1 = tf.matmul(training_data, W1) + b1

# Activation function ReLU is applied
hidden_layer1 = tf.nn.relu(hidden_layer1)

# Wx + b
output_layer = tf.matmul(hidden_layer1, W2) + b2
```

```
hidden_layer1 = tf.layers.dense(training_data, num_neurons_hidden_layer1,
tf.nn.relu)

output_layer = tf.layers.dense(hidden_layer1, num_labels, activation=None)
```

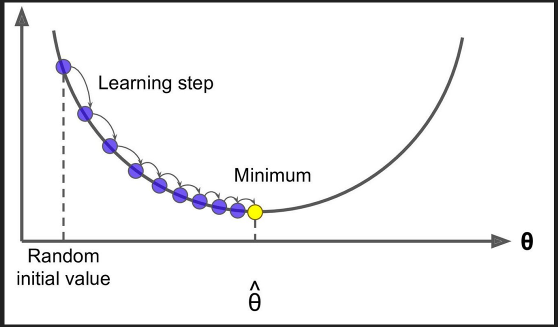
Loss Function

```
# Define the loss function
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=labels, logits=output))

# Loss function usually is a distance functions. How far away is the output produced by the network from the ground-truth?

# Softmax converts the output from the network to probabilities (in this case it also acts as an activation function)

# Cross-entropy measures the distance between probability distributions (output from softmax vs one-hot encoded ground-truth)

# Softmax is only used when we want the output as a multi-class probability distribution

# Other loss functions are  L1 Loss, L2 Loss, GAN Loss
```

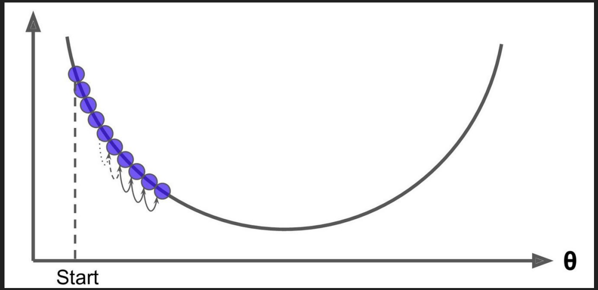
Optimizer

```
# Define optimizer

# Most commonly used optimizer is gradient descent

# Goal of the optimizer is to find the optimal weights where the loss is minimum
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)


# Evaluate accuracy
correct_prediction = tf.equal(tf.argmax(output, 1), tf.argmax(labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

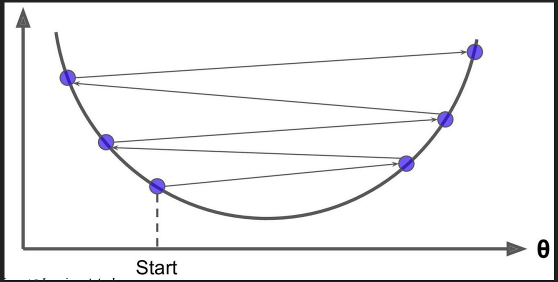
Learning Rate



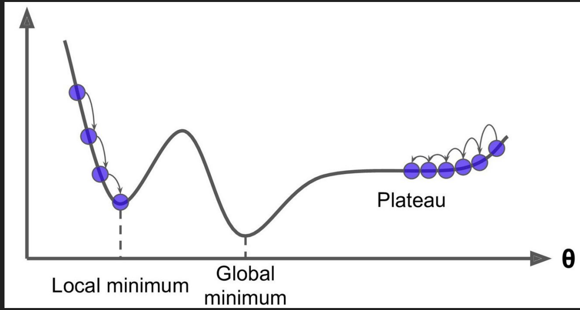
Ideal Learning Rate



Small Learning Rate



High Learning Rate



Real World Scenario

Tensorflow Session

```
# Run the training
sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(num_iterations):
  # Get the next batch
  input_batch, labels_batch = mnist.train.next_batch(batch_size)

  # Run the optimizer, feeding the current input batch and corresponding labels
  sess.run(optimizer,feed_dict={training_data: input_batch, labels: labels_batch})


  if i%100 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={training_data: input_batch, labels: labels_batch})
        print("Iteration %d, training batch accuracy %g %%"%(i, train_accuracy*100))

# Evaluate on the test set
test_accuracy = sess.run(accuracy, feed_dict={training_data: mnist.test.images, labels: mnist.test.labels})
print("Test accuracy: %g %%"%(test_accuracy*100))
```

Converting fully connected MNIST to CNN

```
hidden_layer1 = tf.layers.dense(training_data, num_neurons_hidden_layer1, tf.nn.relu)

output_layer = tf.layers.dense(hidden_layer1, num_labels, activation=None)
```

```
training_data_reshaped = tf.reshape(training_data, shape=[-1, 28, 28, 1])

hidden_layer1 = tf.layers.conv2d(training_data_reshaped, 32, 3, activation=tf.nn.relu)

output_layer = tf.layers.conv2d(hidden_layer1, 32, 3, activation=tf.nn.relu)

output_layer = tf.layers.flatten(output_layer)

output_layer = tf.layers.dense(output_layer, num_labels)
```

CNN

```
# Convert 1D array to a 2D array. Shape = [num_of_images, width, height, channel]
# If num_of_images is unknown when building the graph put -1
# Since MNIST image are grayscale, channel = 1. If we have RGB image then channel = 3

training_data_reshaped = tf.reshape(training_data, shape=[-1, 28, 28, 1])

hidden_layer1 = tf.layers.conv2d(training_data_reshaped, filters=32, kernel_size=3, activation=tf.nn.relu)

output_layer = tf.layers.conv2d(hidden_layer1, filters=32, kernel_size=3, activation=tf.nn.relu)

# Convert 2D array to 1D array for the last layer
output_layer = tf.layers.flatten(output_layer)

# Last layer is fully connected layer
output_layer = tf.layers.dense(output_layer, num_labels)
```