**MLP (MNIST, Tensorflow)**

In this tutorial, we will use MNIST data to practice Multi Layer Perceptron with Tensorflow.

In [1]:

**#import** **tensorflow** **as** **tf**

**import** **tensorflow**.compat.v1 **as** **tf**

**tf**.disable\_v2\_behavior()

**import** **numpy** **as** **np**

**from** **IPython.display** **import** Image

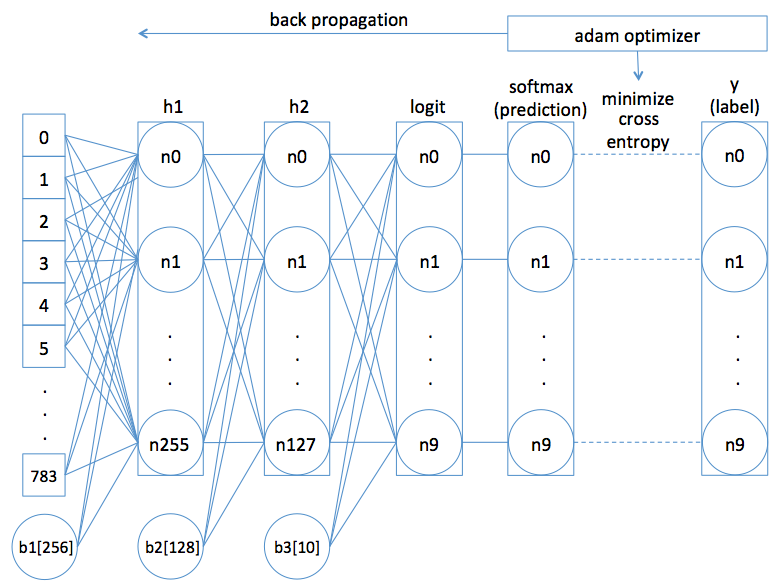
**MLP Architecture**

here is the overview of MLP architecture we will implement with Tensorflow

In [2]:

Image(url= "https://raw.githubusercontent.com/minsuk-heo/deeplearning/master/img/simple\_mlp\_mnist.png", width=500, height=250)

Out[2]:



**Collect MNIST Data**

In [3]:

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

In [4]:

print(x\_train.shape)

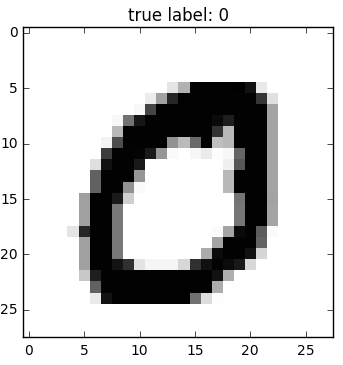
print(x\_test.shape)

(60000, 28, 28)

(10000, 28, 28)

train data has **60000** samples  
test data has **10000** samples  
every data is **28 \* 28** pixels

below image shows 28\*28 pixel image sample for hand written number '0' from MNIST data.  
MNIST is gray scale image [0 to 255] for hand written number.



**Split train data into train and validation data**

Validation during training gives advantages below,  
1) check if train goes well based on validation score  
2) apply **early stopping** when validation score doesn't improve while train score goes up (overcome **overfitting**)

In [5]:

x\_val = x\_train[50000:60000]

x\_train = x\_train[0:50000]

y\_val = y\_train[50000:60000]

y\_train = y\_train[0:50000]

In [6]:

print("train data has " + str(x\_train.shape[0]) + " samples")

print("every train data is " + str(x\_train.shape[1])

+ " \* " + str(x\_train.shape[2]) + " image")

train data has 50000 samples

every train data is 28 \* 28 image

In [7]:

print("validation data has " + str(x\_val.shape[0]) + " samples")

print("every train data is " + str(x\_val.shape[1])

+ " \* " + str(x\_train.shape[2]) + " image")

validation data has 10000 samples

every train data is 28 \* 28 image

28 \* 28 pixels has gray scale value from **0** to **255**

In [8]:

*# sample to show gray scale values*

print(x\_train[0][8])

[ 0 0 0 0 0 0 0 18 219 253 253 253 253 253 198 182 247 241

0 0 0 0 0 0 0 0 0 0]

each train data has its label **0** to **9**

In [9]:

*# sample to show labels for first train data to 10th train data*

print(y\_train[0:9])

[5 0 4 1 9 2 1 3 1]

test data has **10000** samples  
every test data is **28 \* 28** image

In [10]:

print("test data has " + str(x\_test.shape[0]) + " samples")

print("every test data is " + str(x\_test.shape[1])

+ " \* " + str(x\_test.shape[2]) + " image")

test data has 10000 samples

every test data is 28 \* 28 image

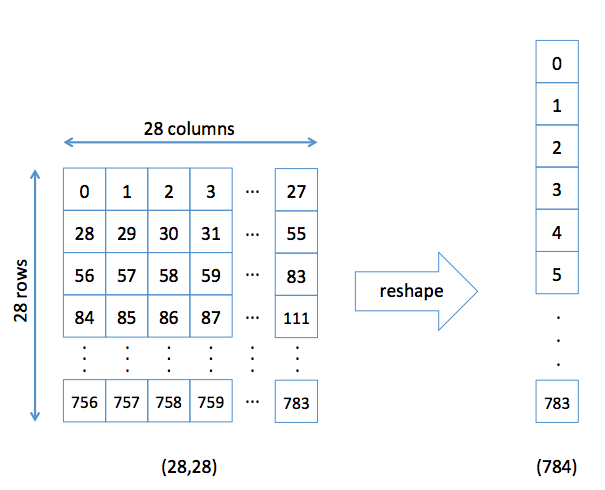
**Reshape**

In order to fully connect all pixels to hidden layer,  
we will reshape (28, 28) into (28x28,1) shape.  
It means we flatten row x column shape to an array having 28x28 (756) items.

In [11]:

Image(url= "https://raw.githubusercontent.com/minsuk-heo/deeplearning/master/img/reshape\_mnist.png", width=500, height=250)

Out[11]:



In [12]:

x\_train = x\_train.reshape(50000, 784)

x\_val = x\_val.reshape(10000, 784)

x\_test = x\_test.reshape(10000, 784)

print(x\_train.shape)

print(x\_test.shape)

(50000, 784)

(10000, 784)

In [13]:

x\_train[0]

Out[13]:

array([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 18, 18, 18,

126, 136, 175, 26, 166, 255, 247, 127, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 30, 36, 94, 154, 170, 253,

253, 253, 253, 253, 225, 172, 253, 242, 195, 64, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 49, 238, 253, 253, 253,

253, 253, 253, 253, 253, 251, 93, 82, 82, 56, 39, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 219, 253,

253, 253, 253, 253, 198, 182, 247, 241, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

80, 156, 107, 253, 253, 205, 11, 0, 43, 154, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 14, 1, 154, 253, 90, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 139, 253, 190, 2, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 190, 253, 70,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 35,

241, 225, 160, 108, 1, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 81, 240, 253, 253, 119, 25, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 45, 186, 253, 253, 150, 27, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 16, 93, 252, 253, 187,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 249,

253, 249, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 46, 130,

183, 253, 253, 207, 2, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 39, 148,

229, 253, 253, 253, 250, 182, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24, 114,

221, 253, 253, 253, 253, 201, 78, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 23, 66,

213, 253, 253, 253, 253, 198, 81, 2, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 171,

219, 253, 253, 253, 253, 195, 80, 9, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 172,

226, 253, 253, 253, 253, 244, 133, 11, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

136, 253, 253, 253, 212, 135, 132, 16, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0], dtype=uint8)

**Normalize data**

normalization usually helps faster learning speed, better performance  
by reducing variance and giving same range to all input features.  
since MNIST data set all input has 0 to 255, normalization only helps reducing variances.  
it turned out normalization is better than standardization for MNIST data with my MLP architeture,  
I believe this is because relu handles 0 differently on both feed forward and back propagation.  
handling 0 differently is important for MNIST, since 1-255 means there is some hand written,  
while 0 means no hand written on that pixel.

In [14]:

x\_train = x\_train.astype('float32')

x\_val = x\_val.astype('float32')

x\_test = x\_test.astype('float32')

gray\_scale = 255

x\_train /= gray\_scale

x\_val /= gray\_scale

x\_test /= gray\_scale

**label to one hot encoding value**

In [15]:

num\_classes = 10

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_val = tf.keras.utils.to\_categorical(y\_val, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

In [16]:

y\_train

Out[16]:

array([[0., 0., 0., ..., 0., 0., 0.],

[1., 0., 0., ..., 0., 0., 0.],

[0., 0., 0., ..., 0., 0., 0.],

...,

[0., 0., 0., ..., 0., 1., 0.],

[0., 0., 0., ..., 0., 0., 0.],

[0., 0., 0., ..., 0., 1., 0.]])

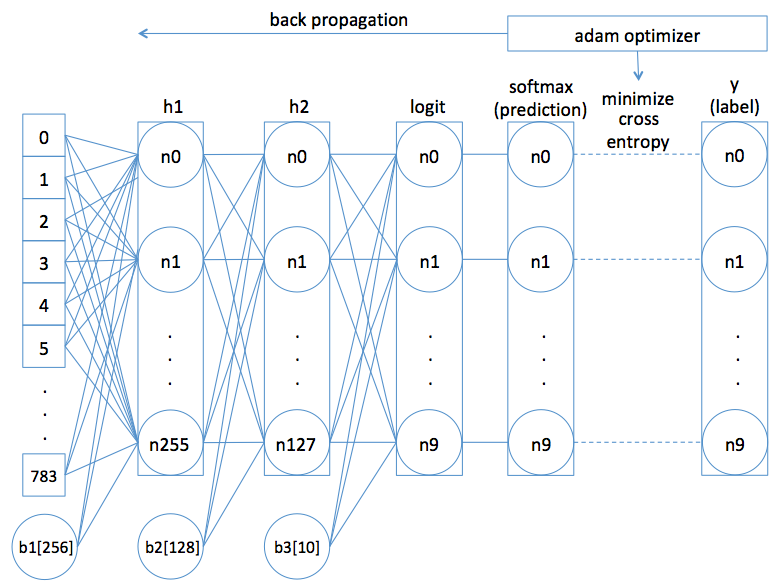
**Tensorflow MLP Graph**

Let's implement the MLP graph with Tensorflow

In [17]:

Image(url= "https://raw.githubusercontent.com/minsuk-heo/deeplearning/master/img/simple\_mlp\_mnist.png", width=500, height=250)

Out[17]:



In [18]:

x = tf.placeholder(tf.float32, [**None**, 784])

y = tf.placeholder(tf.float32, [**None**, 10])

In [19]:

**def** mlp(x):

*# hidden layer1*

w1 = tf.Variable(tf.random\_uniform([784,256]))

b1 = tf.Variable(tf.zeros([256]))

h1 = tf.nn.relu(tf.matmul(x, w1) + b1)

*# hidden layer2*

w2 = tf.Variable(tf.random\_uniform([256,128]))

b2 = tf.Variable(tf.zeros([128]))

h2 = tf.nn.relu(tf.matmul(h1, w2) + b2)

*# output layer*

w3 = tf.Variable(tf.random\_uniform([128,10]))

b3 = tf.Variable(tf.zeros([10]))

logits= tf.matmul(h2, w3) + b3

**return** logits

In [20]:

logits = mlp(x)

In [21]:

loss\_op = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(

logits=logits, labels=y))

In [22]:

train\_op = tf.train.AdamOptimizer(learning\_rate=0.01).minimize(loss\_op)

In [23]:

*# initialize*

init = tf.global\_variables\_initializer()

*# train hyperparameters*

epoch\_cnt = 30

batch\_size = 1000

iteration = len(x\_train) // batch\_size

*# Start training*

**with** tf.Session() **as** sess:

*# Run the initializer*

sess.run(init)

**for** epoch **in** range(epoch\_cnt):

avg\_loss = 0.

start = 0; end = batch\_size

**for** i **in** range(iteration):

\_, loss = sess.run([train\_op, loss\_op],

feed\_dict={x: x\_train[start: end], y: y\_train[start: end]})

start += batch\_size; end += batch\_size

*# Compute average loss*

avg\_loss += loss / iteration

*# Validate model*

preds = tf.nn.softmax(logits) *# Apply softmax to logits*

correct\_prediction = tf.equal(tf.argmax(preds,1), tf.argmax(y,1))

*# Calculate accuracy*

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

cur\_val\_acc = accuracy.eval({x: x\_val, y: y\_val})

print("epoch: "+str(epoch)+", validation accuracy: "

+ str(cur\_val\_acc) +', loss: '+str(avg\_loss))

*# Test model*

preds = tf.nn.softmax(logits) *# Apply softmax to logits*

correct\_prediction = tf.equal(tf.argmax(preds, 1), tf.argmax(y, 1))

*# Calculate accuracy*

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

print("[Test Accuracy] :", accuracy.eval({x: x\_test, y: y\_test}))

epoch: 0, validation accuracy: 0.1064, loss: 9479.23924316406

epoch: 1, validation accuracy: 0.7404, loss: 487.9066563034058

epoch: 2, validation accuracy: 0.8683, loss: 20.24389074325562

epoch: 3, validation accuracy: 0.8761, loss: 11.892984285354613

epoch: 4, validation accuracy: 0.8858, loss: 9.276838760375973

epoch: 5, validation accuracy: 0.8785, loss: 8.25918293952942

epoch: 6, validation accuracy: 0.8832, loss: 7.402374687194823

epoch: 7, validation accuracy: 0.906, loss: 6.622725062370303

epoch: 8, validation accuracy: 0.9034, loss: 5.537717547416686

epoch: 9, validation accuracy: 0.8971, loss: 4.807866144180299

epoch: 10, validation accuracy: 0.8396, loss: 7.349521398544313

epoch: 11, validation accuracy: 0.9066, loss: 6.607095966339113

epoch: 12, validation accuracy: 0.8217, loss: 52.06143003463745

epoch: 13, validation accuracy: 0.8922, loss: 15.170511302948

epoch: 14, validation accuracy: 0.9016, loss: 6.205790314674376

epoch: 15, validation accuracy: 0.9036, loss: 4.978821859359741

epoch: 16, validation accuracy: 0.9083, loss: 4.454537310600279

epoch: 17, validation accuracy: 0.9112, loss: 3.8163385868072504

epoch: 18, validation accuracy: 0.9137, loss: 3.665376300811767

epoch: 19, validation accuracy: 0.9171, loss: 3.8778756713867173

epoch: 20, validation accuracy: 0.9159, loss: 3.3953911519050597

epoch: 21, validation accuracy: 0.9175, loss: 3.1974300575256356

epoch: 22, validation accuracy: 0.8809, loss: 4.378022892475128

epoch: 23, validation accuracy: 0.8764, loss: 4.933798418045042

epoch: 24, validation accuracy: 0.912, loss: 4.379148626327513

epoch: 25, validation accuracy: 0.9154, loss: 3.86887104511261

epoch: 26, validation accuracy: 0.9196, loss: 3.1193934822082525

epoch: 27, validation accuracy: 0.9205, loss: 2.864556527137757

epoch: 28, validation accuracy: 0.9095, loss: 2.6170078659057614

epoch: 29, validation accuracy: 0.9123, loss: 2.499971570968628

[Test Accuracy] : 0.9089

**MLP (MNIST, Tensorflow)**

**import** **tensorflow** **as** **tf**

**#import** **tensorflow**.compat.v1 **as** **tf**

**#tf**.disable\_v2\_behavior()

**#import** **numpy** **as** **np**

**#from** **IPython.display** **import** Image

(x\_train, y\_train),(x\_test, y\_test)=tf.keras.datasets.mnist.load\_data()

print(x\_train.shape)

print(x\_test.shape)

x\_val = x\_train[50000:60000]

x\_train = x\_train[0:50000]

y\_val = y\_train[50000:60000]

y\_train = y\_train[0:50000]

print("train data has " + str(x\_train.shape[0]) + " samples")

print("every train data is " + str(x\_train.shape[1])

+ " \* " + str(x\_train.shape[2]) + " image")

print("validation data has " + str(x\_val.shape[0]) + " samples")

print("every train data is " + str(x\_val.shape[1])

+ " \* " + str(x\_train.shape[2]) + " image")

*# sample to show gray scale values*

print(x\_train[0][8])

*# sample to show labels for first train data to 10th train data*

print(y\_train[0:9])

print("test data has " + str(x\_test.shape[0]) + " samples")

print("every test data is " + str(x\_test.shape[1])

+ " \* " + str(x\_test.shape[2]) + " image")

x\_train = x\_train.reshape(50000, 784)

x\_val = x\_val.reshape(10000, 784)

x\_test = x\_test.reshape(10000, 784)

print(x\_train.shape)

print(x\_test.shape)

x\_train[0]

x\_train = x\_train.astype('float32')

x\_val = x\_val.astype('float32')

x\_test = x\_test.astype('float32')

gray\_scale = 255

x\_train /= gray\_scale

x\_val /= gray\_scale

x\_test /= gray\_scale

num\_classes = 10

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_val = tf.keras.utils.to\_categorical(y\_val, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

y\_train

x = tf.placeholder(tf.float32, [**None**, 784])

y = tf.placeholder(tf.float32, [**None**, 10])

**def** mlp(x):

*# hidden layer1*

w1 = tf.Variable(tf.random\_uniform([784,256]))

b1 = tf.Variable(tf.zeros([256]))

h1 = tf.nn.relu(tf.matmul(x, w1) + b1)

*# hidden layer2*

w2 = tf.Variable(tf.random\_uniform([256,128]))

b2 = tf.Variable(tf.zeros([128]))

h2 = tf.nn.relu(tf.matmul(h1, w2) + b2)

*# output layer*

w3 = tf.Variable(tf.random\_uniform([128,10]))

b3 = tf.Variable(tf.zeros([10]))

logits= tf.matmul(h2, w3) + b3

**return** logits

logits = mlp(x)

loss\_op = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(

logits=logits, labels=y))

train\_op = tf.train.AdamOptimizer(learning\_rate=0.01).minimize(loss\_op)

*# initialize*

init = tf.global\_variables\_initializer()

*# train hyperparameters*

epoch\_cnt = 30

batch\_size = 1000

iteration = len(x\_train) // batch\_size

*# Start training*

**with** tf.Session() **as** sess:

*# Run the initializer*

sess.run(init)

**for** epoch **in** range(epoch\_cnt):

avg\_loss = 0.

start = 0; end = batch\_size

**for** i **in** range(iteration):

\_, loss = sess.run([train\_op, loss\_op],

feed\_dict={x: x\_train[start: end],

y: y\_train[start: end]})

start += batch\_size; end += batch\_size

*# Compute average loss*

avg\_loss += loss / iteration

*# Validate model*

preds = tf.nn.softmax(logits) *# Apply softmax to logits*

correct\_prediction = tf.equal(tf.argmax(preds,1), tf.argmax(y,1))

*# Calculate accuracy*

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

cur\_val\_acc = accuracy.eval({x: x\_val, y: y\_val})

print("epoch: "+str(epoch)+", validation accuracy: "

+ str(cur\_val\_acc) +', loss: '+str(avg\_loss))

*# Test model*

preds = tf.nn.softmax(logits) *# Apply softmax to logits*

correct\_prediction = tf.equal(tf.argmax(preds, 1), tf.argmax(y, 1))

*# Calculate accuracy*

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

print("[Test Accuracy] :", accuracy.eval({x: x\_test, y: y\_test}))