**TensorFlow build Guide**

**A. Construction Phase.**

**1. Import the TensorFlow library**

Ex) import TensorFlow as tf

**2. Specify the number of inputs and outputs, and hidden neurons in each layer**

Ex) n\_inputs=28\*28 (# MNIST)

N\_hidden1 = 300

N\_hideen2 = 100

N\_outputs=10

**3. Make placeholder each variable**

Ex) x = tf.placeholder(tf.float32, shape=(None, n\_inputs), name=”X”)

Y = tf.placeholder(tf.float32, shape=(None), name=”y”)

* Important thing is the shape of tensor. (n\_inputs 자리 and y’s shape)
* We don’t know yet how many instances each training batch will contain. So ,the shape of X is (None, n\_inputs)
* Y will be a 1D tensor (normally). But don’t know the size of the training batch at this point, so the shape is (None)

**4. Create the actual neural network**

-1) Ex) def neuron\_layer(X, n\_neurons, name, activation=None) :

With tf.name\_scope(name):

N\_inputs=int(X.get\_shape()[1])

Stddev = 2 / np.sqrt(n\_inputs + n\_neurons)

Init = tf.truncated\_normal((n\_inputs, n\_neurons), stddev=stddev)

W = tf.Variable(init, name=”kernel”)

B = tf.Variable(tf.zeros([n\_neurons]), name=”bias”)

Z = tf.matmul(X,W) + B

If activation is not None :

Return activation(Z)

Else :

Return Z

* Create a name scope using the name of the layer. It will contain all the computation nodes for this neuron layer. This is optional, but if you want use TensorBoard, then you have to use it.
* Get the number of inputs by looking up the input matrix’s shape and getting the size of the second dimension.
* Next three lines create a W variable that will hold the weights matrix (layer’s kernel). It will be N-D tensor containing all the connection weight between each input and neuron. In this reason, the shape of W is n\_inputs, n\_neurons.
* In the initial phase, W is initialized randomly, using truncated\_normal(Gaussian) distribution with a standard deviation of 2/√n\_input+n\_neurons to fast converge.
* It is important to initialize connection weights randomly for all hidden layers to avoid any symmetries that the Gradient Descent algorithm would be unable to break.
* Next is biases. Initialized to 0 with one bias parameter per neuron.
* Create subgraph to compute Z=XW+B (In TensorFlow, every computation carried by subgraph). This vectorized implementation will efficiently compute the weighted sum of the inputs plus the bias term for each and every neuron in the layer.
* \*Broadcasting : See <https://deeplizard.com/learn/video/6_33ulFDuCg>. But if you don’t want, then just know that the goal of broadcasting is to make the tensors have the same shape. So, we can know that broadcasting arise when two Tensor have different shape.
* The last line is optional. It means it depend on your model.

-2) Ex) with tf.name\_scope(“dnn”) :

Hidden1 = neuron\_layer(X, n\_hidden1, name=”hidden1”, activation = tf.nn.relu)

Hidden2 = neuron\_layer(hidden1, n\_hidden2, name=”hidden2”, activation = tf.nn,relu)

Logits = neuron\_layer(hidden2, n\_outputs, name = “outputs)

* To creates a fully connected layer, just use tf.layer.dense() or tf.contrib.layers.fully\_connected(). If you want just see : <https://www.tensorflow.org/api_docs/python>. And in the tensorflow 2.0, contrib module is not available.
* So, the code become

with tf.name\_scope(“dnn”) :

Hidden1 = tf.layers.dense(X, n\_hidden1, name=”hidden1”, activation = tf.nn.relu)

Hidden2 = tf.layers.dense (hidden1, n\_hidden2, name=”hidden2”, activation = tf.nn,relu)

Logits = tf.layers.dense (hidden2, n\_outputs, name = “outputs)

**5. Create Cost function**

Ex) with tf.name\_scope(“loss”) :

Xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

Loss = tf.reduce\_mean(Xentropy, name=”loss”)

**6. Create optimizer function**

Ex) learning\_rate = 0.01

with tf.name\_scope(“train”):

Optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

Training\_op = optimizer.minize(loss)

**7. Defined how to evaluate the model**

Ex) with tf.name\_scope(“eval”):

Correct = tf.nn.in\_top\_k(logits, y, 1)

Accuracy = tf.reduce\_mean(tf.case(correct, tf.float32)

**8. Create a node to initialize all variable**

Ex) init = tf.global\_variable\_initializer()

Saver = tf.train.Saver() [optional]

**B. Execution Phase**

**1. Load your data set**

Ex) from tensorflow.examples.tutorials.minst import input\_data

Minst = input\_data.read\_data\_sets(“경로”)

* Minst data set. First, import module you need to load your data. And load your data.

**2. Define the number of epochs and size of mini-batches**

Ex) n\_epochs = 40

Batch\_size = 50

**3. Create train loop**

Ex) With tf.Session() as sess:

Init.run()

For epoch in range(n\_epochs):

For iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

Sess.run(training\_opm feed\_dict={X : X\_batch, y: y\_batch})

Acc\_train = accuracy.eval(feed\_dict={X: X\_batch, y : y\_batch})

Acc\_val = accuracy.eval(feed\_dict={X:mnist.validation,images, y:mnist.validation.labels})

Print(epoch, “train accuracy :”, Acc\_train, “val accuracy:”, Acc\_val)

Save\_path = saver.save(sess, “경로”) [optional]

* This code opens a TensorFlow session, and it runs the init node that initializes all the variables
* Then it runs the main training loop. At each epoch, the code iterates through a number of mini-batches that corresponds to the training set size.
* Each mini-batch is fetched via next\_batch() method, and then the code simply runs the training operation, feeding it the current mini-batch input data and targets.
* At the end of each epoch, the code evaluates the model on the last mini-batch and on the full validation set, and it prints put the result.
* The model parameters are saved to disk!!

**4. Reuse model to make predictions [optional]**

Ex) With tf.Session() as sess:

Saver.restore(sess, “경로”)

X\_new\_scaled = [… # some new images (scaled from 0 t o1)]

Z = logits.eval(feed\_dict={X:X\_new\_scaled})

Y\_pred = np.argmax(Z, axise=1)

* To reuse model, **create some construction phase** but **change the execution phase like that**
* First, the code loads the model parameters from disk.
* Then it loads some new images that you want to classify [Remember to apply the same feature scaling as for the training data]
* Then the code evaluates the logits node.
* If you wanted to know all the estimated class probabilities, you would need to apply the softmax() function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit values (using the argmax() function dose the trick)