Create a neural network with Tensorflow. Use it for classification of MNIST or any other dataset of your choice. Divide the dataset into training, validation and test set. Report the accuracy obtained by

1) Varying the number of hidden layers from 0 to 2. Choose the number of neurons in the hidden layers such that the total number of parameters in the network remain the same.

2) Trying sigmoid and relu activation functions for the hidden layer nodes.

3) Not using any nonlinearity in the network

Your program should use the validation set to decide when to stop training. Compute the error on the validation set after every epoch. Stop the training once the error on the validation set starts increasing, or if there is negligible reduction in the error on the validation set for 5 consecutive epochs.

Summarize your results in a report. Submit the report (as a pdf file) along with the code.

Referring Class Lectures

```
[ ] import tensorflow.compat.v1 as tf # Using tensorflow version 1 explicitly
    tf.disable_v2_behavior() # Disabling version 2 routines
    import numpy as np
    mnist = tf.keras.datasets.mnist
```

```
[ ] tf.__version__
```

```
'2.3.0'
```

```
[166] (x_train, y_train),(x_test, y_test) = mnist.load_data(path = 'mnist.npz')
```

```
[167] batch_size = 100 # used for computing the grads (That is there will be 100 rows
      # for 784 examples in a 100 X 784 matrix)
      img = tf.placeholder(tf.float32, [batch_size, 784]) #input
      ans = tf.placeholder(tf.float32, [batch_size, 10])  #output
```

We will call our weights U and V for 2 layered architecture

```
[168] # # W = weights
      # # b = bias
      # W = tf.Variable(tf.random_normal([784,10], stddev=0.1))
      # # 784 inputs to 10 final neurons to the output
      # b = tf.Variable(tf.random_normal([10], stddev=0.1))
      # # Every neuron will have its seperate bias
      # U and bU are weights and bias vector leading from input to hidden layer

      nodes_in_first_hidden_layer = 20
      n1 = nodes_in_first_hidden_layer
      U = tf.Variable (tf.random_normal([784, n1], stddev=0.1))
      bU = tf.Variable (tf.random_normal([n1], stddev=0.1))
      # V and bV are weights and bias vector leading from hidden to output layer
      V = tf.Variable (tf.random_normal([n1, 10], stddev=0.1))
      bV = tf.Variable (tf.random_normal([10], stddev=0.1))
```

```
[169] layer1_output = tf.matmul(img, U) + bU
      layer1_output = tf.nn.relu(layer1_output)
```

```
[170] prbs = tf.nn.softmax (tf.matmul(layer1_output, V)+bV) # Affine transformation
      # Get logit values and on those logits we apply softmax and resulting probablities
      xEnt = tf.reduce_mean(-tf.reduce_sum(ans*tf.log(prbs),reduction_indices=[1]))
      # Computing per image corss entropy, summing up all values by reduce sum
      # Reducing is done by summing up the values
      # reduction indices will sum the rows of the matrix
      # Tells tensor has to be reduced in which direction
```

```
[171] img.shape
```

```
TensorShape([Dimension(100), Dimension(784)])
```

```
[172] W.shape
```

```
TensorShape([Dimension(784), Dimension(10)])
```

```
[173] b.shape
```

```
TensorShape([Dimension(10)])
```

```
[174] ans.shape
```

```
TensorShape([Dimension(100), Dimension(10)])
```

Compute the maximum probablity along the row. we get the index of the highest value 1 is the axis along which the answer is present. Answer vector is one hot vector which tells the index where the actual answer is present. So, tf.equal will try to match probablities vector and answer vector. Such that the index of the highest probability value will match the index of the answer label from answer vector since answer vector is one hot vector. If matches then you get num correct vactor of size 100. Which have values as 0 or 1 as per false and true respectively.

```
[175] # Compute gradients and minimize cross entropy
      # Compute gradients wrt the outpt and all the parameters with 0.5 learning rate
      train = tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
      num_correct = tf.equal(tf.argmax(prbs, 1), tf.argmax(ans, 1))
      accuracy = tf.reduce_mean (tf.cast(num_correct, tf.float32))
```

```
[176] sess = tf.Session()
      sess.run(tf.global_variables_initializer())
      # Initialization of global variables
```

Batch generators are loops which go on and which returns the batches from your training set. Purpose of batch generator: to generate your dataset on multiple cores in real time and feed it right away to your deep learning model. This post must be referred to get more insights on batch generators in python : https://www.oreilly.com/library/view/intelligent-projects-using/9781788996921/5997f1dd-1c4b-4694-b001-51eba1bc08d2.xhtml , https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly

```
[177] def batch_generator(X, Y, batch_size):
        indices = np.arange(len(X))
        batch=[]
        while True:
          np.random.shuffle(indices)
          for i in indices:
            batch.append(i)
            if len(batch)==batch_size:
              yield X[batch], Y[batch]
              batch=[]
```

```
[178] train_generator = batch_generator(x_train, y_train, batch_size=batch_size)
```

For 10000 updates Epochs test accuracy = ~97.45%

```
      for i in range(10000):
        xs, ys = next(train_generator)
        print(xs)
        xsn = xs.reshape(100, 784)
        ysn = np.zeros((ys.size, ys.max()+1))
        ysn[np.arange(ys.size), ys] = 1
        xsn = xsn/255
        sess.run(train, feed_dict={
            img: xsn,
            ans: ysn
        })
```

```
[180] sumAcc = 0
      for i in range(1000):
        xs, ys = next(train_generator)
        xsn = xs.reshape(100, 784)
        ysn = np.zeros((ys.size, ys.max()+1))
        ysn[np.arange(ys.size), ys] = 1
        xsn = xsn/255

        sumAcc += sess.run(accuracy, feed_dict={
            img:xsn,
            ans:ysn
        })
```

```
[181] print("For 1000 updates Test accuracy: %r" % (sumAcc/1000))
```

```
      For 1000 updates Test accuracy: 0.9745100095272065
```

```
[ ]
```