**Project:**

The project was to build a neural network using TensorFlow. This neural network would read hand signs for numbers.

```
# Loading the dataset

X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()

# Flatten the training and test images

X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T

X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T

# Normalize image vectors

X_train = X_train_flatten/255.

X_test = X_test_flatten/255.

# Convert training and test labels to one hot matrices

Y_train = convert_to_one_hot(Y_train_orig, 6)

Y_test = convert_to_one_hot(Y_test_orig, 6)

def create_placeholders(n_x, n_y):

    """

    Creates the placeholders for the tensorflow session.

    Arguments:

    n_x -- scalar, size of an image vector (num_px * num_px = 64 * 64 * 3 = 12288)

    n_y -- scalar, number of classes (from 0 to 5, so -> 6)

     Returns:
```

```
    X -- placeholder for the data input, of shape [n_x, None] and dtype "float"
    Y -- placeholder for the input labels, of shape [n_y, None] and dtype "float"


    X = tf.placeholder(tf.float32,[n_x,None],name="X")
    Y = tf.placeholder(tf.float32,[n_y,None],name="Y")
    return X, Y
--
def initialize_parameters():
    """
    Initializes parameters to build a neural network with tensorflow. The shapes
are:
                        W1 : [25, 12288]
                        b1 : [25, 1]
                        W2 : [12, 25]
                        b2 : [12, 1]
                        W3 : [6, 12]
                        b3 : [6, 1]
    Returns:
    parameters -- a dictionary of tensors containing W1, b1, W2, b2, W3, b3
    """
    W1 =
tf.get_variable("W1",[25,12288],initializer=tf.contrib.layers.xavier_initializer(seed
=1))
```

```
    b1 = tf.get_variable("b1",[25,1],initializer=tf.zeros_initializer())
    W2 =
tf.get_variable("W2",[12,25],initializer=tf.contrib.layers.xavier_initializer(seed=1))
    b2 = tf.get_variable("b2",[12,1],initializer=tf.zeros_initializer())
    W3 =
tf.get_variable("W3",[6,12],initializer=tf.contrib.layers.xavier_initializer(seed=1))
    b3 = tf.get_variable("b3",[6,1],initializer=tf.zeros_initializer())
    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2,
                  "W3": W3,
                  "b3": b3}
  return parameters
--
def forward_propagation(X, parameters):
    """
    Implements the forward propagation for the model: LINEAR -> RELU -> LINEAR -
> RELU -> LINEAR -> SOFTMAX
    Arguments:
    X -- input dataset placeholder, of shape (input size, number of examples)
    parameters -- python dictionary containing your parameters "W1", "b1", "W2",
"b2", "W3", "b3"
```

```
            the shapes are given in initialize_parameters

    Returns:

    Z3 -- the output of the last LINEAR unit

    """

    # Retrieve the parameters from the dictionary "parameters"

    W1 = parameters['W1']

    b1 = parameters['b1']

    W2 = parameters['W2']

    b2 = parameters['b2']

    W3 = parameters['W3']

    b3 = parameters['b3']

    Z1 = tf.add(tf.matmul(W1,X),b1)                                   # Z1 = np.dot(W1, X)
+ b1

    A1 = tf.nn.relu(Z1)                              # A1 = relu(Z1)

    Z2 = tf.add(tf.matmul(W2,A1),b2)                              # Z2 = np.dot(W2, a1) +
b2

    A2 = tf.nn.relu(Z2)                              # A2 = relu(Z2)

    Z3 = tf.add(tf.matmul(W3,Z2),b3)                             # Z3 = np.dot(W3,Z2) + b3

     return Z3

--

def compute_cost(Z3, Y):

    """
```

Computes the cost

Arguments:

Z3 -- output of forward propagation (output of the last LINEAR unit), of shape (6, number of examples)

Y -- "true" labels vector placeholder, same shape as Z3

Returns:

cost - Tensor of the cost function

"""

# to fit the tensorflow requirement for tf.nn.softmax_cross_entropy_with_logits(...,...)

logits = tf.transpose(Z3)

labels = tf.transpose(Y)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits =logits,labels=labels))

return cost

--

def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,

num_epochs = 1500, minibatch_size = 32, print_cost = True):

"""

Implements a three-layer tensorflow neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SOFTMAX.

Arguments:

X_train -- training set, of shape (input size = 12288, number of training examples = 1080)

Y_train -- test set, of shape (output size = 6, number of training examples = 1080)

X_test -- training set, of shape (input size = 12288, number of training examples = 120)

Y_test -- test set, of shape (output size = 6, number of test examples = 120)

learning_rate -- learning rate of the optimization

num_epochs -- number of epochs of the optimization loop

minibatch_size -- size of a minibatch

print_cost -- True to print the cost every 100 epochs

Returns:

parameters -- parameters learnt by the model. They can then be used to predict.

"""

ops.reset_default_graph()                    # to be able to rerun the model without overwriting tf variables

(n_x, m) = X_train.shape                     # (n_x: input size, m : number of examples in the train set)

n_y = Y_train.shape[0]                       # n_y : output size

costs = []                                   # To keep track of the cost

# Create Placeholders of shape (n_x, n_y)

X, Y = create_placeholders(n_x,n_y)

```
    # Initialize parameters
        parameters = initialize_parameters()
    # Forward propagation: Build the forward propagation in the tensorflow graph
    Z3 = forward_propagation(X,parameters)
   # Cost function: Add cost function to tensorflow graph
    cost = compute_cost(Z3,Y)
    # Backpropagation: Define the tensorflow optimizer. Use an AdamOptimizer.
    optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cost)
    # Initialize all the variables
    init = tf.global_variables_initializer()
    # Start the session to compute the tensorflow graph
    with tf.Session() as sess:
        # Run the initialization
        sess.run(init)
        # Do the training loop
        for epoch in range(num_epochs):
            epoch_cost = 0.                     # Defines a cost related to an epoch
            num_minibatches = int(m / minibatch_size) # number of minibatches of
size minibatch_size in the train set
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train, minibatch_size,
seed)
```

```
        for minibatch in minibatches:

            # Select a minibatch

            (minibatch_X, minibatch_Y) = minibatch

            # The line that runs the graph on a minibatch.

            # Run the session to execute the "optimizer" and the "cost", the feedict
should contain a minibatch for (X,Y).

             _ , minibatch_cost =
sess.run([optimizer,cost],feed_dict={X:minibatch_X,Y:minibatch_Y})

            epoch_cost += minibatch_cost / num_minibatches

        # Print the cost every epoch

        if print_cost == True and epoch % 100 == 0:

            print ("Cost after epoch %i: %f" % (epoch, epoch_cost))

        if print_cost == True and epoch % 5 == 0:

            costs.append(epoch_cost)

    # plot the cost

    plt.plot(np.squeeze(costs))

    plt.ylabel('cost')

    plt.xlabel('iterations (per tens)')

    plt.title("Learning rate =" + str(learning_rate))

    plt.show()

    # lets save the parameters in a variable

    parameters = sess.run(parameters)
```

```
print ("Parameters have been trained!")

# Calculate the correct predictions

correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))

# Calculate accuracy on the test set

accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

print ("Train Accuracy:", accuracy.eval({X: X_train, Y: Y_train}))

print ("Test Accuracy:", accuracy.eval({X: X_test, Y: Y_test}))

return parameters
```