Single Layer Perceptron in TensorFlow:-

In Single Layer Perceptron only contains i/p_layer and o/p_layer and there is no hidden_layer present in b/w them.it is very basic form of ANN and easy to use.It makes dot.product of weighted sum of i/p_layer and i/p_values and after we add i/p_layer_bias then we fed in to the o/p containing activation function. It gives o/p in terms of -1 to 1 & 0 to 1.

First read those files 1)"https://www.edureka.co/blog/deep-learning-with-python/" ,
2)"https://www.geeksforgeeks.org/introduction-to-tensorflow/" ,
3)"https://www.javatpoint.com/cost-function-in-machine-learning" ,
4)"https://chromium.googlesource.com/external/github.com/tensorflow/tensorflow/+/r0.10/tensorflow/g3doc/tutorials/mnist/pros/index.md"<-----this is very important,
5)"https://shreyasshetty.github.io/2016/10/30/Reduction-indices-in-tensorflow/"
6)"https://github.com/jalammar/simpleTensorFlowClassificationExample/blob/master/Basic%20Classification%20Example%20with%20TensorFlow.ipynb"

note:-
*Learning rate:
There are multiple ways to select a good starting point for the learning rate. A naive approach is to try a few different values and see which one gives you the best loss without sacrificing speed of training. We might start with a large value like 0.1, then try exponentially lower values: 0.01, 0.001, etc. When we start training with a large learning rate, the loss doesn't improve and probably even grows while we run the first few iterations of training. When training with a smaller learning rate, at some point the value of the loss function starts decreasing in the first few iterations. This learning rate is the maximum we can use, any higher value doesn't let the training converge. Even this value is too high: it won't be good enough to train for multiple epochs because over time the network will require more fine-grained weight updates. Therefore, a reasonable learning rate to start training from will be probably 1–2 orders of magnitude lower.

*Training data:
Maybe the number of training data is very less in volume for the model to learn, try including more data by sampling, augmentation techniques.

*Epochs:
In your code, decrease the display_step to 10, so for every 10 steps you will print the loss, if the loss is not changing much for continuous steps, you can bring the epochs number to that range where the loss is not changing. Else if you keep large number of epochs, your model will overfit.

*Test Data:
Test data should be unseen data from the train instances. Try to give different varients of test data. Including train_test_split to 80/20 % and do data shuffling before splitting train and test data.

SINGLE LAYER PERCEPTRON PROGRAMME:-

```
# Import the MINST dataset to see this datasets read
"https://www.tensorflow.org/tutorials/keras/classification"
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_ ("/tmp/data/", one_hot=True)

import tensorflow as tf
import matplotlib.pyplot as plt

step.1:-
# Parameters :-First of all, we define some parameters for training our model, like:

learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1        ##display_step to 1, so for every 1 steps you will print the
loss.
Step.2:-
# tf Graph Input :-Then we define placeholder nodes for feature and target vector.
x = tf.placeholder("float", [none, 784])                 # MNIST data image of shape
28*28 = 784
y = tf.placeholder("float", [none, 10])                  # 0-9 digits recognition =>
10 classes BCZ o/p has only 1D.

MODEL CREATION:-
Step.3:-
# Set model weights:-Then, we define variable nodes for weight and bias.
W = tf.Variable(tf.zeros([784, 10]))    ##Here 784 rows bcz i/p layer has 784
neurons is there,and then it fed to the o/p_layer of containing 10 neurons(.'.
[784,10])
b = tf.Variable(tf.zeros([10]))         ##Here bias is adding after dot product of
i/p and i/p_layer weights,.'.it has 10_neuron o/p.So we need to add bias for only
                                  shape of 10.i,e,,,tf.zeros([10]

Step.4:-
# Constructing the model
        ##linear_model is an operational node which calculates the hypothesis for
the linear regression model.
        ##linear_model = W*X + b
        ##output = tf.nn.activation_fun(tf.matmul(x, w)+b)
activation=tf.nn.softmaxx(tf.matmul (x, W)+b) # Softmax of function


Step.5:-
# Minimizing error using cross entropy
        ####(One of the commonly used loss functions for classification is
cross-entropy loss.for linear rigression we can use(1)Means_Errors, 2)MSE, 3)MAE)
        The binary Cost function is a special case of Categorical cross-entropy,
where there is only one output class. For example, classification between red
andblue.
```

        To better understand it, let's suppose there is only a single output
variable Y
        Cross-entropy(D) = - y*log(p) when y = 1
        Cross-entropy(D) = - (1-y)*log(1-p) when y = 0)

Step.6:-
##We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function is the cross-entropy between the target and the model's prediction:

```
cross_entropy = y*tf.log(activation) ##or## cross_entropy =
tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

For linear ligression for finding loss we use Mean Square Error(MSE)

```
loss = tf.reduce_sum(tf.square(output - y))/2*n_samples
```

Step.7:-
#Note that tf.reduce_sum sums across all classes and tf.reduce_mean takes the average over these sums.reduction_indices=[1] means indexing of row(0) or column(1) ,here 1 is specified bcz o/p or y is in the form of column.Therefore we need to assign reduction_indices=[1]

```
cost = tf.reduce_mean(-tf.reduce_sum(cross_entropy, reduction_indice = 1))
        ##is same as explained above

# t1 is a tensor [[1,  2,  3],
#                 [4,  5,  6],
#                 [7,  8,  9],
#                 [10, 11, 12]]
t2 = tf.reduce_max(t1, reduction_indices=[0])
# t2 is the tensor [10, 11, 12]
t3 = tf.reduce_max(t1, reduction_indices=[1])
# t3 is the tensor [3, 6, 9, 12]
```

Step.8:-
##Now that we have defined our model and training loss function, it is straightforward to train using TensorFlow. Because TensorFlow knows the entire computation graph, it can use automatic differentiation to find the gradients of the loss with respect to each of the variables.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

learning_rate:-if we want to apply gradient decent the slope will calculate to measure low cost(loss).it will flow down untill minimum cost(loss) is reached.learning_rate gives steps to reach low_cost quickly. it will ecicutes in less time.

Step.9:-

```
#Plot settings
avg_set = []
epoch_set = []

step.10:-
# Initializing the variables where
init = tf.initialize_all_variables()

step.11:-
# Launching the graph
with tf.Session() as sess:
   sess.run(init)


Step.12:-
# Training of the cycle in  the dataset
   for epoch in range(training_epochs):
      avg_cost = 0.
      total_batch = int(mnist.train.num_example/batch_size)
##num_example is equal to the number of test images you are feeding into the API.

Step.13:-
# Creating loops at all the batches in the code
      for i in range(total_batch):
batch_xs, batch_ys = mnist.train.next_batch(batch_size)


Step.14:-
# Fitting the training by the batch data
sess.run(optimizr,  feed_dict = {x: batch_xs, y: batch_ys})


Step.15:-
 # Compute all the average of loss
avg_cost += sess.run(cost, feed_dict = {x: batch_xs, y: batch_ys})      ##total
batch  ##+= Add AND assignment operator. It adds the right operand to the left
operand
        and assign the result to the left operand. C += A is equivalent to C = C +
A.


Step.16:-
# Display the logs at each epoch steps ##in order to display logs(values like
accuracy,loss,,,ect are displayed in the tensorboard) at each epoch steps,we need to
make display_step is equal to epoch.Therefore here we declared it as remaider of
epoch & display_step is equal to 0.Then only it displays logs at each epoch steps.

      if epoch % display_step==0:
      print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format (avg_cost))
            avg_set.append(avg_cost) epoch_set.append(epoch+1)
```

```
##Here we assign or we add the value of avg_cost and epoch+1 into set
   print ("Training phase finished")

   plt.plot(epoch_set,avg_set, 'o', label = 'Logistics Regression Training')
   plt.ylabel('cost')
   plt.xlabel('epoch')
   plt.legend()
   plt.show()
```

Step.17:-

```
# Test the model
   correct_prediction = tf.equal (tf.argmax (activation, 1),tf.argmax(y,1))
```

Parameters used in Test the model:-

1)argmax:-function returns indices of the max element of the array in a particular axis.

tf.argmax(array, axis = None, out = None)

Parameters :

```
array : Input array to work on
axis  : [int, optional]Along a specified axis like 0 or 1
out   : [array optional]Provides a feature to insert output to the out
          array and it should be of appropriate shape and dtype
```

2)tf.equal:-The tf. equal() operator is an elementwise operator. Assuming x and y are the same shape (as they are in your example) tf. equal(x, y) will produce a tensor with the same shape, where each element indicates whether the corresponding elements in x and y are equal.

Examplea = tf.equal([1., 2.], [1., 3.])----->it gives o/p in the form of boolean i,e,, [True False]

Actual explanation:- correct_prediction = tf.equal (tf.argmax (activation, 1),tf.argmax(y,1)):-

*tf.argmax(activation, 1) means:-First it will gives the max index of column(axis=1) in activation i,e,, our predicted value.example[1,2,3,4,5]

*tf.argmax(y,1) means:-First it will gives the max index of column(axis=1) in y i,e,, our actual value.example[1,2,3,4,5]

*tf.equal (tf.argmax (activation, 1),tf.argmax(y,1)) means:-It will compair both pedicted and actual values if both are same it gives True or it gives False i,e,,[True,True,True,True,True]

Conclusion:-

From the above example it gives [True,True,True,True,True] it maens actual valu is equal to predicted value.i,e,, 100% accuracy(example is given by me)

Step.18:-Last step:-

```
# Calculating the accuracy of dataset
accuracy = tf.reduce_mean(tf.cast (correct_prediction, "float"))
print("Model accuracy:", accuracy.eval({x:mnist.test.images, y: mnist.test.labels}))
```

Parameters:-
*tf.cast:-syntax:-tf.cast(x, dtype, name=None)
The "tf. cast" function casts a tensor to new type.Therefore due to this it will convert boolean data_type into folat_data_type. in order to find mean of accuracy.

##last step is very very imp bcz we use paceholder .Therefore in the last we need to assign the datapoints.
here in our datasets we Import the Fashion MNIST dataset
This guide uses the Fashion MNIST dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:
Loading the dataset returns four NumPy arrays:

The train_images and train_labels arrays are the training set—the data the model uses to learn.
The model is tested against the test set, the test_images, and test_labels arrays.
The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The labels are an array of integers, ranging from 0 to 9. These correspond to the class of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the class names are not included with the dataset, store them here to use later when plotting the images:

read this "https://www.tensorflow.org/tutorials/keras/classification"
also refer:-
1)"https://stackoverflow.com/questions/59646219/test-set-accuracy-of-1-how-to-debug"
2)"https://www.codementor.io/blog/tensorboard-integration-5fh168wqvi"
3)"https://github.com/jalammar/simpleTensorFlowClassificationExample/blob/master/Basic%20Classification%20Example%20with%20TensorFlow.ipynb"
4)"https://stackoverflow.com/questions/41708572/tensorflow-questions-regarding-tf-argmax-and-tf-equal"

Hidden Layer Perceptron in TensorFlow:-

The only difference we need to observe b/w Simple Layer Perceptron and Hidden Layer Perceptron is:-

1)In Simple Layer Perceptron there is no hidden layer present. it includes accept i/p and by wighted sum of i/p and bias it will directedly gives o/p.

2)In Hidden Layer Perceptron there is an hidden layer in b/w i/p and o/p.bcz of incresing efficiency,imp is it prvide backpropagation.

*Backpropagation:-The hidden neural network is set up in some techniques. In many cases, weighted inputs are randomly assigned. On the other hand, they are fine-tuned and calibrated through a process called backpropagation.

*o/p of Hidden Layer Perceptron is equal to hidlayer_1 or hidlayer_n weighted sum and bias of Simple Layer Perceptron o/p.

Formula:-

O/P=((O/P of Simple Layer Perceptron)*W_hidlayer1)+B_hidlayer1)<------This is very imp

The code for the hidden layers of the perceptron is shown below:

```
#Importing the essential modules in the hidden layer
import tensorflow as tf
import numpy as np
import mat
plotlib.pyplot as plt
import math, random



np.random.seed(1000)                                                          ##it
is used to gives same valuses in all iteration
function_to_learn = lambda x: np.cos(x) + 0.1*np.random.randn(*x.shape)         ##it
is the function used in this programm
layer_1_neurons = 10                                                          ##10
neurons  contained in layer 1 or in hidden_layer
NUM_points = 1000
##Total number of points

#Train the parameters of hidden layer
batch_size = 100
NUM_EPOCHS = 1500

all_x = np.float32(np.random.uniform(-2*math.pi, 2*math.pi, (1, NUM_points))).T
####random.uniform(low=0.0(-2pi), high=1.0(2pi), size=None(1,1000))we use 2pi means

radian of 360_degree.bcz we use cos() fun.Therefore here we selected the random

range b/w (-360deg to +360deg)

  Draw samples from a uniform distribution.Samples are uniformly distributed over
```

the half-open interval [low, high) (includes low, but excludes high). In other words,
any value within the given interval is equally likely to be drawn by uniform.
vvvimp point:-
Here we use shape is (1,NUM_points) or (1,1000).bcz in our i/p_layer consists of only one neurons so it will accecpt only one value but we have i/p_value for i/p_layer has 1000_or_NUM_points. .'.we need to give shape of (1,1000)shape for i/p_layer's i/p.

```
   np.random.shuffle(all_x)  ###please
read"https://valueml.com/shuffle-the-training-data-in-tensorflow/#:~:text=shuffle()%20will%20randomly%20shuffle,the%20data%20of
%20our%20datasets.&text=As%20we%20can%20see%2C%20The,only%20one%20output%5By%5D."
```

###Data shuffling satisfies the purpose of variance reduction. It’s goal is to keep the model general and makes sure that it doesn’t over fit a lot.Training, testing and validation are the phases that our presented dataset will be further splitting into, in our machine learning model. We need to shuffle these datasets well, avoiding any possible elements in the split datasets before training the ML model.

```
train_size = int(900)
#Train the first 700 points in the set
x_training = all_x[:train_size]
y_training = function_to_learn(x_training)

#Training the last 300 points in the given set
x_validation = all_x[train_size:]
y_validation = function_to_learn(x_validation)

plt.figure(1)
plt.scatter(x_training, y_training, c = 'blue', label = 'train')
plt.scatter(x_validation, y_validation, c = 'pink', label = 'validation')
plt.legend()
plt.show()

X = tf.placeholder(tf.float32, [None, 1], name = "X")
Y = tf.placeholder(tf.float32, [None, 1], name = "Y")

#first layer
#Number of neurons = 10
w_h = tf.Variable(tf.random_uniform([1, layer_1_neurons], minval = -1, maxval = 1,
dtype = tf.float32))


##tf.random.uniform(shape, minval=0, maxval=None,

dtype=tf.dtypes.float32, seed=None, name=None) in this case we  use
                                                                  sigmoid
```

activation fun.Therefore the o/p is given b/w -1 to +1(minval =
-1, maxval =
1)
Here why we took [1, layer_1_neurons] or [1,10] is given for w_h is bcz it has only "1" i/p_neuron_layer and "10" neuron in the o/p or i/p_hidden_layer.

```
b_h = tf.Variable(tf.zeros([1, layer_1_neurons], dtype = tf.float32))  ##Bias of
hidden layer.Why (1,10)shape is same as above it will add bias for 1 neuron in i/p

h = tf.nn.sigmoid(tf.matmul(X, w_h) + b_h)                               ##O/P of
hidden layer(next it will fed as i/p for o/p layer)

#output layer
#Number of neurons = 10
w_o = tf.Variable(tf.random_uniform([layer_1_neurons, 1],\ minval = -1, maxval = 1,
dtype = tf.float32))  ##Weight of o/p layer
```

*Here we have shape for hidden_layer's o/p is [layer_1_neurons, 1] or [10,1].bcz in the hidden layer we have 10 neurons. .'. it will produce 10 column o/p,and this hidden_o/p fed to o/p_layer consist of only one neurons. so it will deduce to 1. .'.we have shape [10,1]

```
b_o = tf.Variable(tf.zeros([1, 1], dtype = tf.float32))   ##bias of o/p layer ##it
was shape of [1,1].bcz bias  is added after dot.product. .'. after dot. we have only
                                                                 one value. .'. it
has only 1 row, 1 column o/p.[1,1]

#building the model
model = tf.matmul(h, w_o) + b_o                                            ##o/p of
model(from o/p_layer)=(dot.product of(o/p of hidden_layer(h)

            and weighted sum of o/p_layer(w_o))+Bias of o/p_layer)

#minimize the cost function (model - Y)
train_op = tf.train.AdamOptimizer().minimize(tf.nn.l2_loss(model - Y))    ##Adam is
an alternative of gradient_decent ##(model -y)--->Mean loss for regression

                                        & cross entropy for classification

#Starting the Learning phase
sess = tf.Session() sess.run(tf.initialize_all_variables())

errors = []
for i in range(NUM_EPOCHS):
   for start, end in zip(range(0, len(x_training), batch_size),range(batch_size,
len(x_training), batch_size)):
      sess.run(train_op, feed_dict = {X: x_training[start:end],\ Y:
y_training[start:end]})

                ###Zip: Creates a Dataset by zipping together datasets. Useful
```

```
        in scenarios where you have features and labels and you

need to provide the pair of feature and label for
                                                                                training the
model.
   cost = sess.run(tf.nn.l2_loss(model - y_validation),\ feed_dict =
{X:x_validation})
   errors.append(cost)

   if i%100 == 0:
      ###i%100==0 means, Display loss for every epoch(nothing but display_steps)
      print("epoch %d, cost = %g" % (i, cost))
   ##it prints epoch and cost values in tensor_board

plt.plot(errors,label='MLP Function Approximation') plt.xlabel('epochs')
plt.ylabel('cost')
plt.legend()
plt.show()

--------------
Read for reference
1)"https://www.tensorflow.org/api_docs/python/tf/random/uniform"
2)"https://towardsdatascience.com/building-efficient-data-pipelines-using-tensorflow
-8f647f03b4ce"
3)"https://valueml.com/shuffle-the-training-data-in-tensorflow/#:~:text=shuffle()%20
will%20randomly%20shuffle,the%20data%20of%20our%20datasets.&text=As%20we%20can%20see
%2C%20The,only%20one%20output%5By%5D."


-------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
```