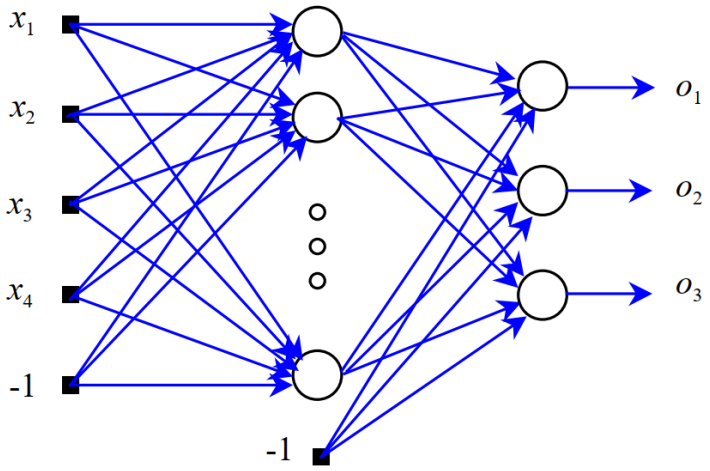
PATTERN RECOGNITION – Homework 3

Solution

Problem 1.

Design a nonlinear classifier using MLPs:



The network has an input layer (input features dim = 4), an output layer (3 neurons ⇔ 3 classes) and one hidden layer (customized number of nodes).

Python code with explanation:

We will build an MLP class from scratch using numpy library.

Use sklearn.preprocessing package to normalize data and create one-hot labels.

Use matplotlib to plot learning curves.

```
import numpy as np
import random
from sklearn.preprocessing import LabelEncoder
from sklearn import preprocessing
import matplotlib.pyplot as plt
```

The __init__ function initializes some parameters:

X_train, y_train, X_test, y_test: our training and testing data

output_size: number of nodes in output layer

hidden_size: number of nodes in hidden layer

momentum: momentum constant with value between 0 and 1. By default, this argument is None, indicating gradient descent without momentum term. To use the algorithm with momentum term, we just basically configure momentum to be a specific value, for example 0.7 => self.mu = 0.7

self.hidden_weight: weights between input and hidden layer. It is initialized randomly.

self.output_weight: weights between hidden layer and output layer.

self.hidden_bias, self.output_bias: biases for hidden layer and output layer respectively.

Some additional params like self.prev_hidden_weight store the previous weights' values for momentum term calculation.

```python
class MLP():
  def __init__(self, X_train, y_train, X_test, y_test, output_size, hidden_size, momentum=None):
    self.X_train = X_train
    self.y_train = y_train
    self.X_test = X_test
    self.y_test = y_test
    self.output_size = output_size
    self.hidden_size = hidden_size
    self.sample_size, self.input_size = X_train.shape

    if momentum and (momentum != 0):
      print("USING MOMENTUM TERM!")
      self.mu = momentum
    else:
      print("NOT USING MOMENTUM TERM!")
      self.mu = 0.0

    # Intialize weights
    # Hidden layers
    self.hidden_weight = 0.1 * np.random.uniform(-1, 1, (self.input_size, self.hidden_size))
    self.hidden_bias = np.zeros((1, self.hidden_size))
    # Output layers
    self.output_weight = 0.1 * np.random.uniform(-1, 1, (self.hidden_size, self.output_size))
    self.output_bias = np.zeros((1, self.output_size))

    # Initialize for momentum
    self.prev_hidden_weight = self.hidden_weight
    self.prev_hidden_bias = self.hidden_bias
    self.prev_output_weight = self.output_weight
    self.prev_output_bias = self.output_bias
```

The following functions define activation functions and their derivative formulas (for calculating gradient later). ReLU is used in hidden layer and Softmax (since problem 1 is classification problem) is used in output layer.

Sum of squared error is used as loss function:

$$\varepsilon(n) = \frac{1}{2}\sum_{k=1}^{K} e_k^2(n),$$

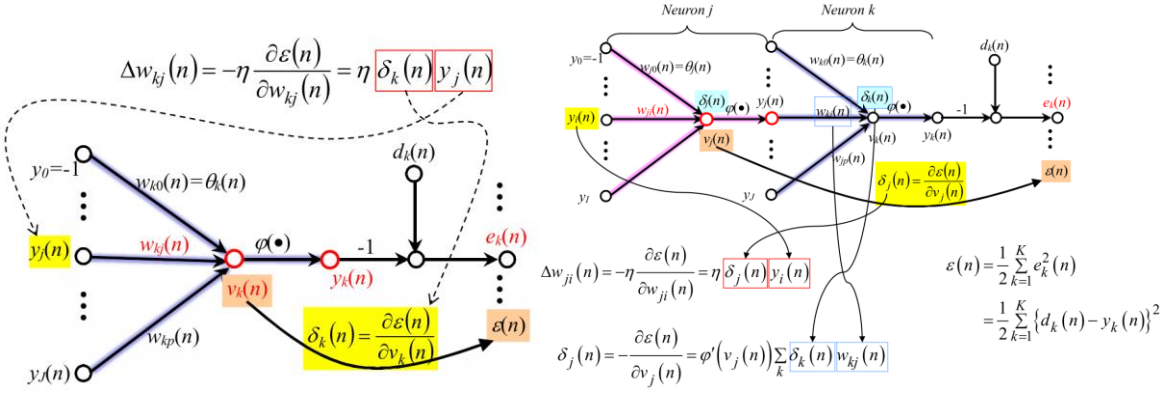
K : # of output nodes

```
def relu(self, x):
  return np.where(x >= 0, x, 0)

def relu_derivative(self, x):
  return np.where(x >= 0, 1, 0)

def softmax(self, x):
  e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
  return e_x / np.sum(e_x, axis=-1, keepdims=True)

def softmax_derivative(self, x):
  s = self.softmax(x)
  return s * (1 - s)

def loss(self, y_pred, y): # Sum of Square Error
  self.error = y - y_pred # ...
  return 0.5 * np.sum(np.power(self.error, 2), axis=1) # => derivative return = - self.error
  # return -(y * np.log(y_pred)).sum(axis=1).mean() # CrossEntropy
```

We define a function to perform the forward pass or predict the output given input feature. The final prediction is the output of softmax function representing the probability distribution over multiple classes (3 in this problem).

```
def forward(self, X): # Forward pass
  self.hidden_in = np.dot(X, self.hidden_weight) + self.hidden_bias
  self.hidden_out = self.relu(self.hidden_in) # hidden layer
  self.output_in = np.dot(self.hidden_out, self.output_weight) + self.output_bias
  y_pred = self.softmax(self.output_in) # output layer
  return y_pred
```

The backward function is implemented to update weight parameters following the backpropagation rule from the lecture 6 (from slide 35 to 39)



self.lr: learning rate. output_local_grad, hidden_local_grad: Local gradients δ with respect to output layer and hidden layer *(in this code, there are some minor modifications regarding the sign -/+ compared to the above figure but the final result is expected to be the same).*

```
def backward(self, X): # Update params
  output_local_grad = self.error * -1 * self.softmax_derivative(self.output_in)
  output_weight_grad = np.dot(self.hidden_out.T, output_local_grad)
  output_bias_grad = np.sum(output_local_grad, axis=0, keepdims=True)

  hidden_local_grad = np.dot(output_local_grad, self.output_weight.T) * self.relu_derivative(self.hidden_in)
  hidden_weight_grad = np.dot(X.T, hidden_local_grad)
  hidden_bias_grad = np.sum(hidden_local_grad, axis=0, keepdims=True)

  tmp = list([self.output_weight, self.output_bias, self.hidden_weight, self.hidden_bias])

  self.output_weight = self.output_weight - self.lr * output_weight_grad + self.mu * (self.output_weight - self.prev_output_weight)
  self.output_bias = self.output_bias - self.lr * output_bias_grad + self.mu * (self.output_bias - self.prev_output_bias)
  self.hidden_weight = self.hidden_weight - self.lr * hidden_weight_grad + self.mu * (self.hidden_weight - self.prev_hidden_weight)
  self.hidden_bias = self.hidden_bias - self.lr * hidden_bias_grad + self.mu * (self.hidden_bias - self.prev_hidden_bias)

  self.prev_output_weight, self.prev_output_bias, self.prev_hidden_weight, self.prev_hidden_bias = tmp
```

In case of adding momentum term to the gradient descent, the following formula is used. β parameter in the formula is equivalent to **self.mu** while α is equivalent to **self.lr** in the python code. The intuition is that if the current gradient is in the same direction as the previous step, move a little further in the same direction. If it's in the opposite direction, move a little less far.

$$w_{t+1} = w_t - \alpha \nabla f(w_t) + \beta(w_t - w_{t-1}).$$

Finally, define training function as below:

epoch_num: the maximum number of epochs for training.

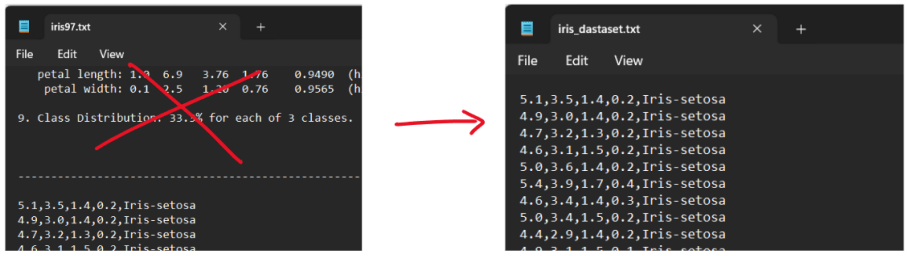
For each epoch, we iterate through the training data. Given each input training sample (feature), we predict the output class probability distribution using self.forward function, then we compare y_pred with the ground truth (label) to obtain training loss. As discussed previously, the self.backward function means to update all model's parameters based on the output error of the network.

After each epoch, we calculate the classification accuracy (train_acc), error rate (train_error_rate) and finally visualize the learning curves after finishing all training epochs.

```
def train(self, epoch_num, lr):
  self.lr = lr
  train_loss_arr = []
  for epoch in range(epoch_num):
    running_loss = 0.0
    total_correct = 0
    for i in range(0, self.sample_size):
      feature = np.array([self.X_train[i]])
      label = np.array([self.y_train[i]])
      y_pred = self.forward(feature)
      loss = self.loss(y_pred, label)
      running_loss += loss.item()
      total_correct += (np.argmax(y_pred, axis=1) == np.argmax(label, axis=1)).item()
      self.backward(feature)

    train_loss = running_loss/self.sample_size
    train_loss_arr.append(train_loss)
    train_acc = total_correct/self.sample_size
    train_error_rate = 1.0 - train_acc
    print('Epoch #{:2}: train_loss = {:2.2f} | train_acc = {:2.2f} | train_error_rate = {:2.2f}'.
          format(epoch+1, train_loss, train_acc, train_error_rate))

  # Plot learning curve
  epochs = np.arange(1,epoch_num+1)
  print(len(train_loss_arr))
  plt.plot(epochs, train_loss_arr)
  plt.xlabel('Epoch')
  plt.ylabel('Sum of Squared Error')
  plt.title('Learning curve: MLP w/ {:1} hidden nodes'.format(self.hidden_size))
  plt.grid(True)
```

So, given the defined MLP class above, we now can implement training and testing on iris dataset.

Firstly, we need to take some steps to prepare the data. Any other information but the dataset is removed so that we can read data more easily.



To extract data from .txt file, we use np.loadtxt function. sklearn.preprocessing.normalize function is used to normalize input samples to be in between 0 and 1. For the ground truth, we convert string labels to one-hot vector format type. There are totally 150 samples and a half of that would be used for training classifier while another half would be used for testing. As a common practice, the training samples should be shuffled before training.

```
np.random.seed(1)
filename = "/content/iris_dastaset.txt"
dataset = np.loadtxt(filename, delimiter=',', dtype=str)
X = dataset[:, :-1].astype(float) # convert to float
labels = dataset[:, -1]

# Normalize data
X = preprocessing.normalize(X)

# Convert string labels to one-hot vector format
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(labels)
y = np.squeeze(np.eye(3)[labels.reshape(-1)]) # 3 = numclass, one-hot labels

# Split half data for each class for training & another half for testing
X_train = np.concatenate((X[0:25], X[50:75], X[100:125]))
y_train = np.concatenate((y[0:25], y[50:75], y[100:125]))
X_test = np.concatenate((X[25:50], X[75:100], X[125:150]))
y_test = np.concatenate((y[25:50], y[75:100], y[125:150]))

shuffle = True
if shuffle:
  k = np.random.permutation(len(X_train))
  X_train, y_train = X_train[k], y_train[k]
```

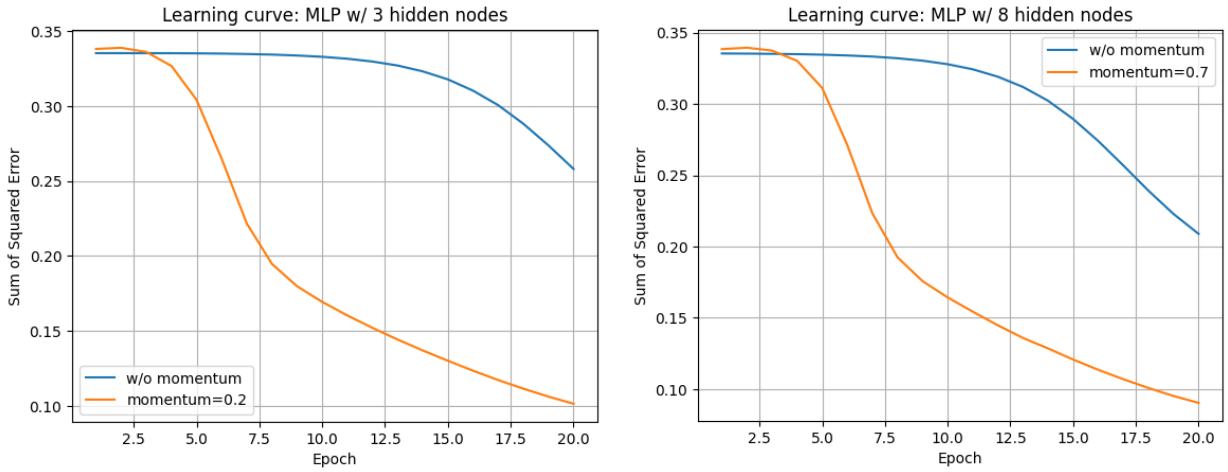
a/

The following code implements the MLP classifier for both cases (gradient descent with and without momentum term) and prints out classification error given the testing data for each case:

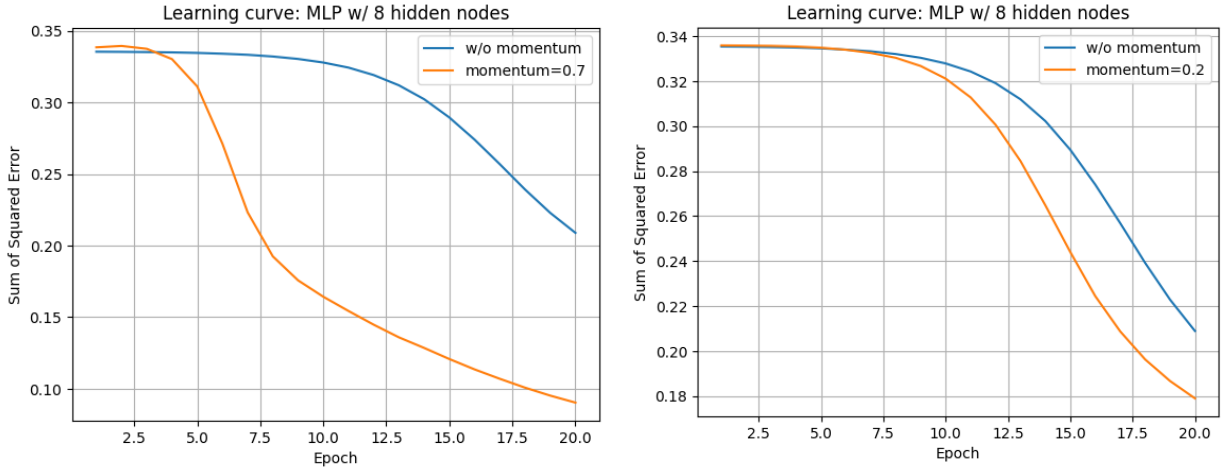
```
################################# NO MOMENTUM ################################
model = MLP(X_train, y_train, X_test, y_test, output_size=3, hidden_size=3)
model.train(epoch_num=20, lr=0.1)
y_pred = np.argmax(model.forward(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
test_acc = np.sum(y_true == y_pred, axis=0) / len(y_true)
test_error_rate = 1.0 - test_acc
print('Test error rate = {:2.2f}'.format(test_error_rate))

################################ WITH MOMENTUM ###############################
model_with_momentum = MLP(X_train, y_train, X_test, y_test, output_size=3, hidden_size=3, momentum=0.7)
model_with_momentum.train(epoch_num=20, lr=0.1)
y_pred = np.argmax(model_with_momentum.forward(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
test_acc = np.sum(y_true == y_pred, axis=0) / len(y_true)
test_error_rate = 1.0 - test_acc
print('Test error rate = {:2.2f}'.format(test_error_rate))
plt.legend(['w/o momentum', 'momentum=0.7'])
```

Learning curves visualization:

With epoch number = 20, learning rate = 0.1, momentum=0.7 (for case 2):



[Figure] 2 hidden nodes vs. 8 hidden nodes



[Figure] momentum=0.7 vs. momentum=0.2

As we can see from the above figures, the training error keeps decreasing as the epoch number increases. Such pattern shows that we have designed an MLP classifier quite well and parameters of neural networks get properly trained.

The model with more nodes in the hidden layer appears to have smaller training loss, demonstrating slightly bigger learning capability of it compared to the one with less hidden neurons.

Especially, when we add momentum term to gradient descent, the model seems to converge much faster with much smaller training error. The bigger momentum (momentum=0.7 > 0.2) results in faster convergence of learning curves.

b/

The classification error rate is computed as the number of misclassified samples per data size. The recorded error rates on both training and testing set are listed as below:

 # 3 nodes hidden layer:

- × Without momentum term:

  Epoch #20: train_loss = 0.26 | train_acc = 0.68 | **train_error_rate = 0.32**

  Test error rate = 0.33

- ✓ With momentum term:

  Epoch #20: train_loss = 0.10 | train_acc = 0.84 | **train_error_rate = 0.16**

  Test error rate = 0.05

 # 8 nodes in hidden layer:

- × Without momentum term:

  Epoch #20: train_loss = 0.21 | train_acc = 0.68 | **train_error_rate = 0.32**

  Test error rate = 0.33

- ✓ With momentum term (momentum = 0.7):

  Epoch #20: train_loss = 0.09 | train_acc = 0.89 | **train_error_rate = 0.11**

  Test error rate = 0.04

- ✓ With momentum term (momentum = 0.2):

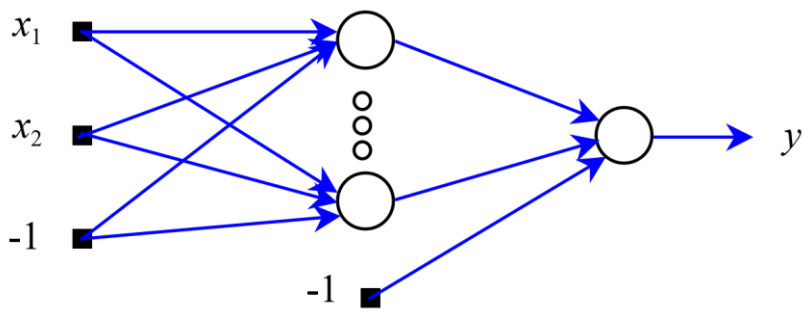
  Epoch #20: train_loss = 0.18 | train_acc = 0.79 | **train_error_rate = 0.21**

  Test error rate = 0.19

It is clear that the models with momentum term always give better performance on both training and testing set and the one with larger momentum trained with the same number of epochs proved lower error rate. There is not big difference of the error rates between of the model with 3 hidden nodes and the model with 8 hidden nodes.

Problem 2.

Design a function approximator (regressor) using MLPs.



Generally, the approach is similar to problem #1, the only different is that we have only one output for the output layer this time and we do not need softmax function for the output layer since this is the regression problem.

Hence, we can leverage the same MLP class we designed in previous problem with some minor modifications in self.forward(), self.backward() and self.training_function().

We simply remove the softmax function part in the forward and backward pass:

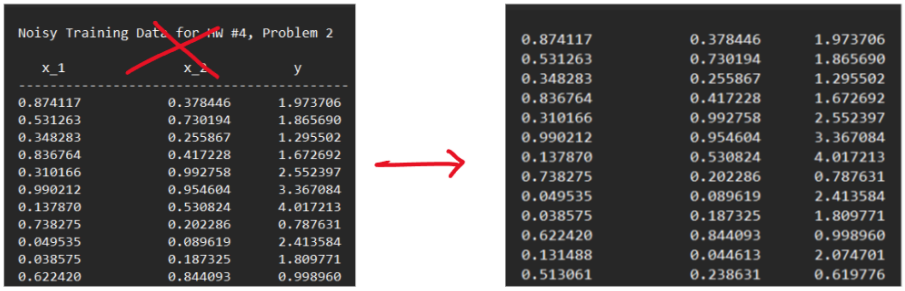
```
def forward(self, X): # forward pass
  self.hidden_in = np.dot(X, self.hidden_weight) + self.hidden_bias
  self.hidden_out = self.relu(self.hidden_in) # hidden layer
  self.output_in = np.dot(self.hidden_out, self.output_weight) + self.output_bias
  y_pred = self.output_in # output layer
  return y_pred
```

```
def backward(self, X):
  # Update params
  output_local_grad = self.error * -1 * 1
  output_weight_grad = np.dot(self.hidden_out.T, output_local_grad)
  output_bias_grad = np.sum(output_local_grad, axis=0, keepdims=True)

  hidden_local_grad = np.dot(output_local_grad, self.output_weight.T) * self.relu_derivative(self.hidden_in)
  hidden_weight_grad = np.dot(X.T, hidden_local_grad)
  hidden_bias_grad = np.sum(hidden_local_grad, axis=0, keepdims=True)

  tmp = list([self.output_weight, self.output_bias, self.hidden_weight, self.hidden_bias])

  self.output_weight = self.output_weight - self.lr * output_weight_grad + self.mu * (self.output_weight - self.prev_output_weight)
  self.output_bias = self.output_bias - self.lr * output_bias_grad + self.mu * (self.output_bias - self.prev_output_bias)
  self.hidden_weight = self.hidden_weight - self.lr * hidden_weight_grad + self.mu * (self.hidden_weight - self.prev_hidden_weight)
  self.hidden_bias = self.hidden_bias - self.lr * hidden_bias_grad + self.mu * (self.hidden_bias - self.prev_hidden_bias)

  self.prev_output_weight, self.prev_output_bias, self.prev_hidden_weight, self.prev_hidden_bias = tmp
```

Remove classification error rate:

```
def train(self, epoch_num, lr):
  self.lr = lr
  train_loss_arr = []
  for epoch in range(epoch_num):
    running_loss = 0.0

    for i in range(0, self.sample_size):
      feature = np.array([self.X_train[i]])
      label = np.array([self.y_train[i]])
      y_pred = self.forward(feature)
      loss = self.loss(y_pred, label)
      running_loss += loss.item()
      self.backward(feature)

    train_loss = running_loss/self.sample_size
    train_loss_arr.append(train_loss)
    # print('Epoch #{:2}: train_loss = {:2.2f}'.format(epoch+1, train_loss))

  # Plot learning curve
  epochs = np.arange(1,epoch_num+1)
  plt.plot(epochs, train_loss_arr)
  plt.xlabel('Epoch')
  plt.ylabel('Sum of Squared Error')
  plt.title('Learning curve: MLP w/ {:1} hidden nodes'.format(self.hidden_size))
  plt.grid(True)
```

Prepare new dataset:



Noisy Training Data for HW #4, Problem 2

x_1	x_2	y
0.874117	0.378446	1.973706
0.531263	0.730194	1.865690
0.348283	0.255867	1.295502
0.836764	0.417228	1.672692
0.310166	0.992758	2.552397
0.990212	0.954604	3.367084
0.137870	0.530824	4.017213
0.738275	0.202286	0.787631
0.049535	0.089619	2.413584
0.038575	0.187325	1.809771
0.622420	0.844093	0.998960

0.874117	0.378446	1.973706
0.531263	0.730194	1.865690
0.348283	0.255867	1.295502
0.836764	0.417228	1.672692
0.310166	0.992758	2.552397
0.990212	0.954604	3.367084
0.137870	0.530824	4.017213
0.738275	0.202286	0.787631
0.049535	0.089619	2.413584
0.038575	0.187325	1.809771
0.622420	0.844093	0.998960
0.131488	0.044613	2.074701
0.513061	0.238631	0.619776

```
np.random.seed(1)
filename = "/content/regressor_trn.txt"
dataset = np.loadtxt(filename, dtype=float)
X_train = dataset[:, :-1]
y_train = np.expand_dims(dataset[:, -1], axis=1)

filename = "/content/regressor_tst.txt"
dataset = np.loadtxt(filename, dtype=float)
X_test = dataset[:, :-1]
y_test = np.expand_dims(dataset[:, -1], axis=1)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)

shuffle = True
if shuffle:
  k = np.random.permutation(len(X_train))
  X_train, y_train = X_train[k], y_train[k]
```

Train the regression model and evaluate with testing data:

Epoch num = 100, learning rate = 0.03

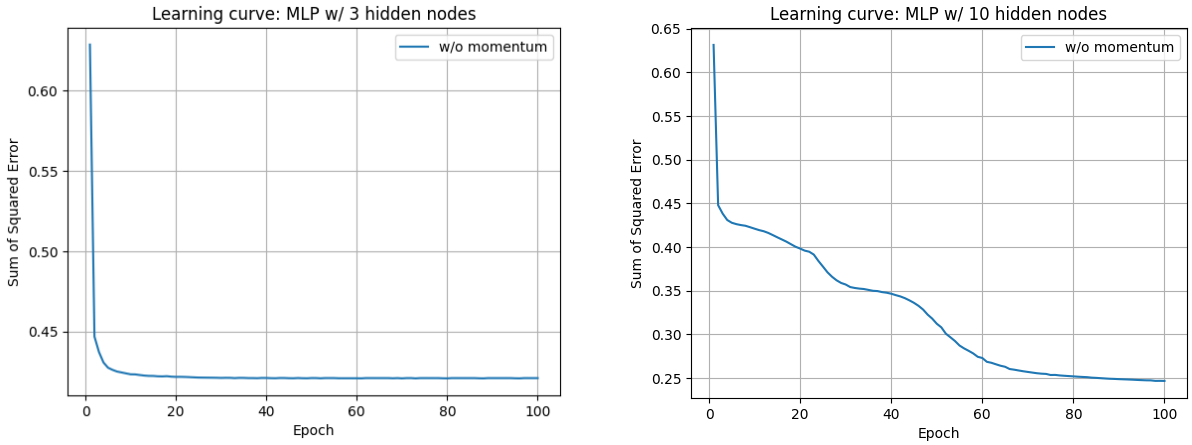
- ✖ <u>Without momentum term</u>:

```
model = MLP(X_train, y_train, X_test, y_test, output_size=1, hidden_size=3)
model.train(epoch_num=100, lr=0.03)
y_pred = model.forward(X_test)
print("Sum of Squared Error: ", model.loss(y_pred, y_test).mean())
```

- ✓ <u>With momentum term</u>:;

```
model_with_momentum = MLP(X_train, y_train, X_test, y_test, output_size=1, hidden_size=3, momentum=0.5)
model_with_momentum.train(epoch_num=100, lr=0.03)
y_pred = model_with_momentum.forward(X_test)
print("Sum of Squared Error: ", model_with_momentum.loss(y_pred, y_test).mean())

plt.legend(['w/o momentum', 'momentum=0.5'])
```

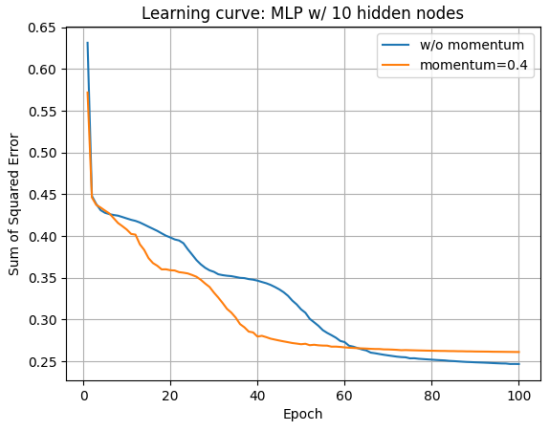
Learning curves visualization:



[Figure] 3 hidden neurons (train_loss = 0.42 at epoch #100)

vs. 10 hidden neurons (train_loss = 0.25 at epoch #100)

⇨ The model with less hidden neurons suffers overfitting earlier with higher training error. The one with more hidden neurons, with better learning capacity can converge to smaller training error. This can explain the results on test set of these 2 models later, where the networks with 10 hidden neurons produce much lower error.



[Figure] with vs. without momentum term

⇨ During first 60 epochs, the model with momentum term seems to converge faster. At the end of training process, the two models have comparative training errors (0.26 vs. 0.25), where training error of the model without momentum tends to be slightly lower. However, when comparing the performance on test set (below), the model with momentum term produces smaller sum of squared error.

Results on **Test set** given the model:

 # with 3 hidden neurons:

```
NOT USING MOMENTUM TERM!
Epoch #100: train_loss = 0.42
Sum of Squared Error:  0.4373934982306922
```

 # with 10 hidden neurons for both cases: (momentum=0.4)

```
NOT USING MOMENTUM TERM!
Epoch #100: train_loss = 0.25
Sum of Squared Error:  0.23255742605657792
USING MOMENTUM TERM!
Epoch #100: train_loss = 0.26
Sum of Squared Error:  0.21116337592094162
```