Programming Assignment Report

PA1-1 :Linear Regression

Univariate

load_tv_sales function:

implement:

```
import torch
def load_tv_sales(file_name):
    data = []
    with open(file_name, 'r') as f:

        for i, line in enumerate(f.readlines()):
            if i == 0:
                continue # head line, skip it
            values = line.strip().split(',')
            try:
              tv = float(values[1])
              ad_sales = float(values[4])
              data.append([tv,ad_sales])
            except ValueError:
                continue
    data = torch.FloatTensor(data)
    return data
```

2. result

```
[5] data = load_tv_sales('./pa1_data1.csv')
    X = data[:,0].unsqueeze(-1)
    y = data[:,1].unsqueeze(-1)
    m = y.size(0) # number of training examples

    print("X shape : ", X.size()) # X shape must be [200,1]
    print("y shape : ", y.size()) # y shape must be [200,1]
```

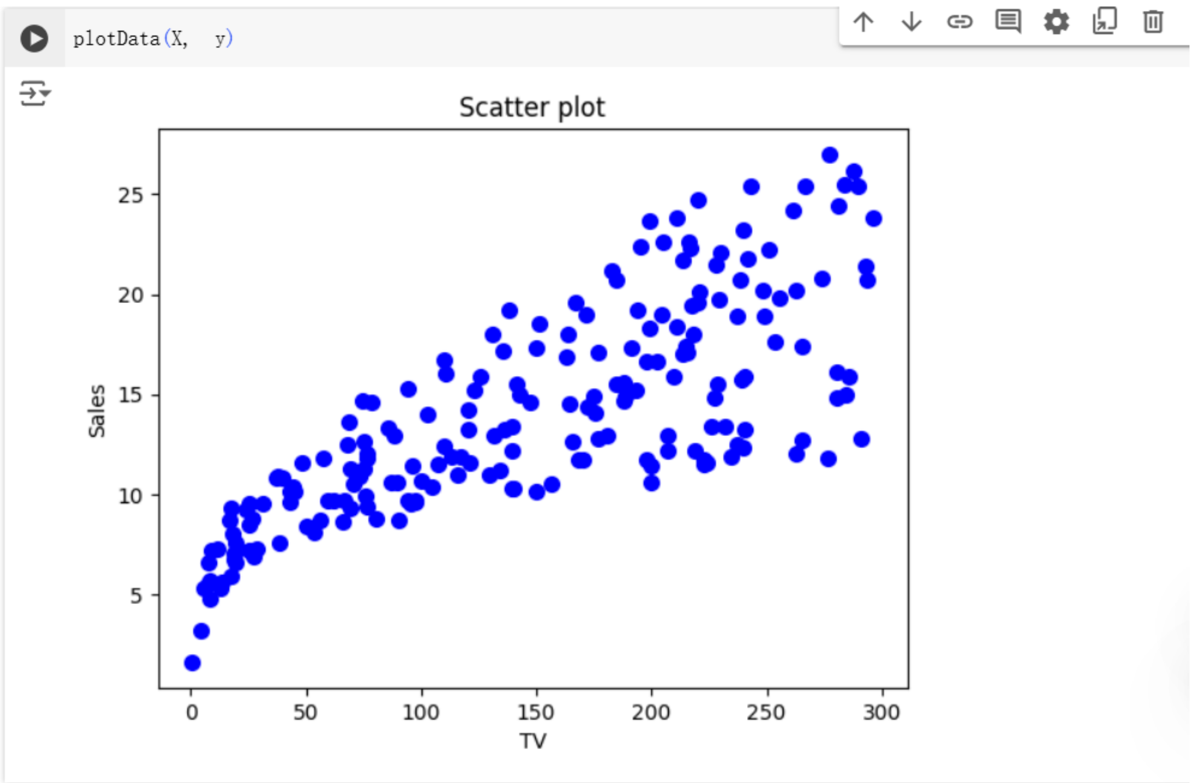
```
X shape :  torch.Size([200, 1])
y shape :  torch.Size([200, 1])
```

plotData function:

1. implement:

```
import matplotlib.pyplot as plt
def plotData(X,y):
    plt.scatter(X, y, color="blue", linewidth=2)
    plt.title("Scatter plot")
    plt.xlabel("TV")
    plt.ylabel("Sales")
```

2. result



computeCost function:

1. implement:

```
import torch
def computeCost(X, y, theta):
    #computeCost compute cost for linear regression
    #   J = computeCost(X, y, theta) computes the cost of using theta as the
    #   parameter for linear regression to fit the data points in X and y

    # Initialize some useful values
    n = y.size(0)
    # You need to return the following variables correctly
    J = 0
    for i in range(n) :
      h = X[i] @ theta
      J += (y[i] - h)**2

    J /= (2*n)

    return J
```

2. result

```
Testing the cost function ...

With theta = [[0],[0]]
Cost computed =  111.85812377929688

Expected cost value (approx) 111.86


With theta = [[-1],[2]]
Cost computed =  31.49065399169922

Expected cost value (approx) 31.49
```

gradientDescent function:

1. implement:

```
def gradientDescent(X, y, theta, alpha, num_iters):
  #gradientDescent performs gradient descent to learn theta
  #   theta, J_history = gradientDescent(X, y, theta, alpha, num_iters) updates theta by
  #   taking num_iters gradient steps with learning rate alpha
```

```
# Initialize some useful values
m = y.size(0)  # number of training examples
J_history = torch.zeros(num_iters,1)

for idx in range(num_iters):
  prediction = X @ theta
  error = prediction - y
  theta = theta - alpha / m * (X.T @ error)

  J_history[idx] = computeCost(X, y, theta)

return theta, J_history
```

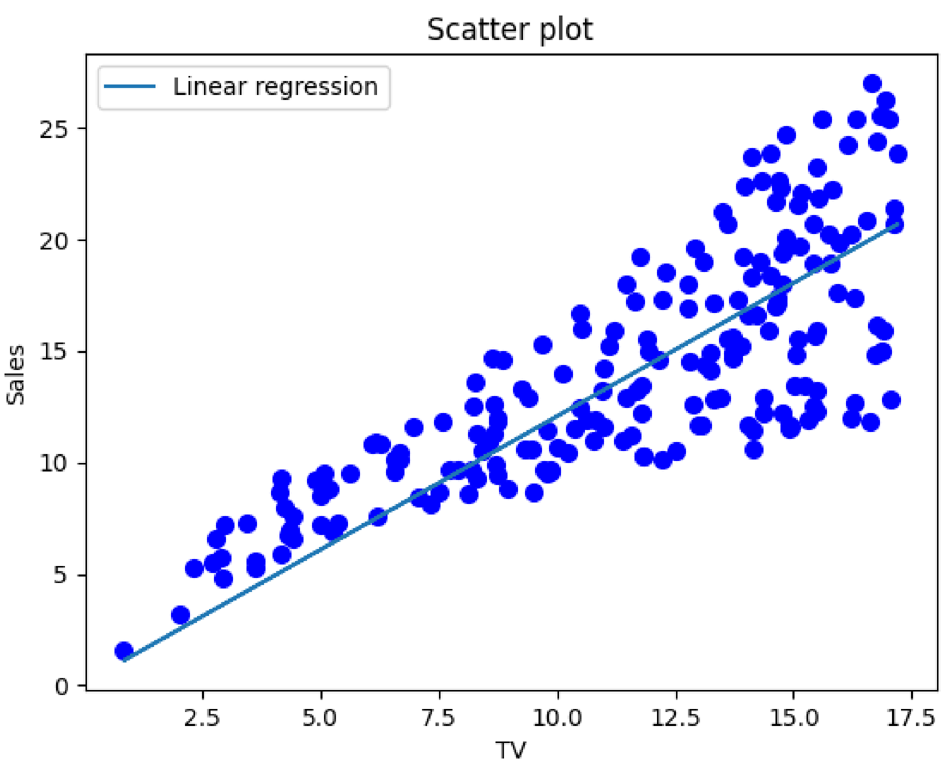
2. result:

```
Running Gradient Descent ...

Theta found by gradient descent:

0.003093911102041602
0.0832342803478241
Expected theta values (approx)

0.0031
0.0832
```

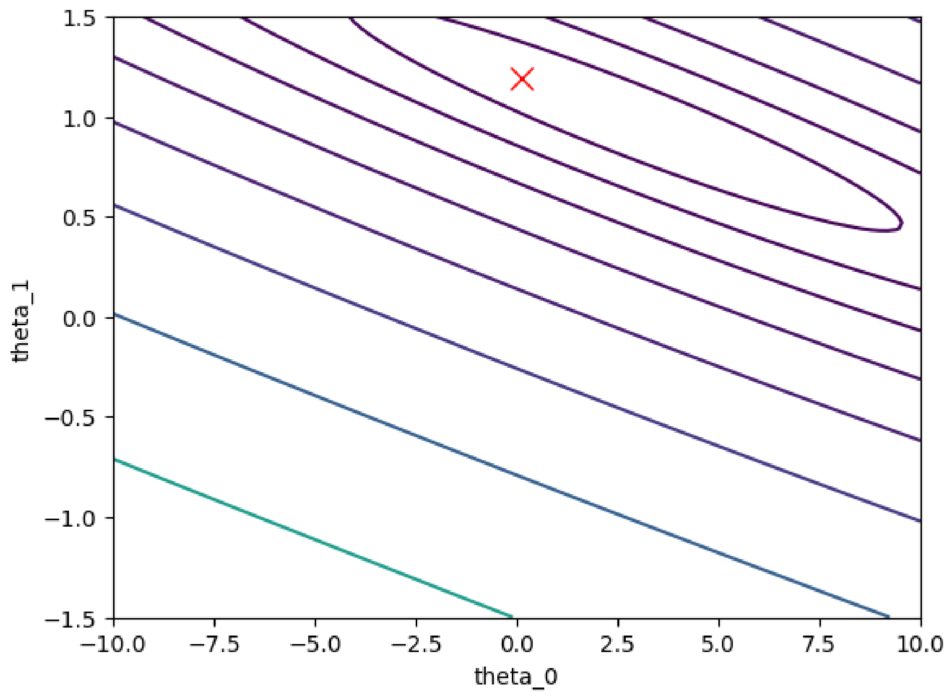
Discussion Answer:

1. Reason why different happens

- The most immediate reason is that the parameters converge too slowly, or the number of iterations is too low, which prevents reaching the optimal solution. Increasing the iteration count improves this issue. i.e. set learning rate is 1e-5
- Looking deeper into the data, we can observe that the relationship is not strictly linear. As TV spending increases, the growth in sales starts to slow down, indicating a non-linear relationship. Therefore, I believe that using a model with $y = \sqrt{x}$ would be more suitable for regression. In practice, we can apply the square root to the TV data before fitting the model to better capture this relationship.



As we can see , after square the TV data, our image is more likely liner, and loss function become lower and more close to the optimum solution.



Multivariate

load_ad_sales function

1. implement

```
import torch
def load_ad_sales(file_name):
    data = []
    with open(file_name, 'r') as f:

        for i, line in enumerate(f.readlines()):
            if i == 0:
                continue
            values = line.strip().split(',')
            try:
              tv = float(values[1])
              radio = float(values[2])
              newspaper = float(values[3])
              sales = float(values[4])
              data.append([tv,radio,newspaper,sales])
            except ValueError:
              continue
    data = torch.FloatTensor(data)
    return data
```

2. result

```
[120] data = load_ad_sales('./pa1_data1.csv')
      X = data[:, 0:3]
      y = torch.Tensor(data[:, 3]).unsqueeze(-1)
      m = y.size(0)

      print("X shape : ", X.size())
      print("y shape : ", y.size())
```

```
X shape :  torch.Size([200, 3])
y shape :  torch.Size([200, 1])
```

feasureNormalize function

1. implement

```
def featureNormalize(X):

    X_norm = X
    mu = torch.zeros(X.size(1))
    sigma = torch.zeros(X.size(1))

    mu = X.mean(dim = 0)
    sigma = X.std(dim = 0)
    X_norm = (X_norm - mu) / sigma

    return X_norm, mu, sigma
```

2. result

```
First 10 examples from the dataset:

x =  tensor([230.1000,   37.8000,   69.2000])  y =  tensor([22.1000])
x =  tensor([44.5000, 39.3000, 45.1000])  y =  tensor([10.4000])
x =  tensor([17.2000, 45.9000, 69.3000])  y =  tensor([9.3000])
x =  tensor([151.5000,   41.3000,   58.5000])  y =  tensor([18.5000])
x =  tensor([180.8000,   10.8000,   58.4000])  y =  tensor([12.9000])
x =  tensor([ 8.7000, 48.9000, 75.0000])  y =  tensor([7.2000])
x =  tensor([57.5000, 32.8000, 23.5000])  y =  tensor([11.8000])
x =  tensor([120.2000,   19.6000,   11.6000])  y =  tensor([13.2000])
x =  tensor([8.6000, 2.1000, 1.0000])  y =  tensor([4.8000])
x =  tensor([199.8000,    2.6000,   21.2000])  y =  tensor([10.6000])

Normalizing Features ...

x =  tensor([0.9674, 0.9791, 1.7745])  y =  tensor([22.1000])
x =  tensor([-1.1944,  1.0801,  0.6679])  y =  tensor([10.4000])
x =  tensor([-1.5124,  1.5246,  1.7791])  y =  tensor([9.3000])
x =  tensor([0.0519, 1.2148, 1.2832])  y =  tensor([18.5000])
x =  tensor([ 0.3932, -0.8395,  1.2786])  y =  tensor([12.9000])
x =  tensor([-1.6114,  1.7267,  2.0408])  y =  tensor([7.2000])
x =  tensor([-1.0430,  0.6423, -0.3239])  y =  tensor([11.8000])
x =  tensor([-0.3127, -0.2468, -0.8703])  y =  tensor([13.2000])
x =  tensor([-1.6125, -1.4255, -1.3570])  y =  tensor([4.8000])
x =  tensor([ 0.6145, -1.3918, -0.4295])  y =  tensor([10.6000])
```

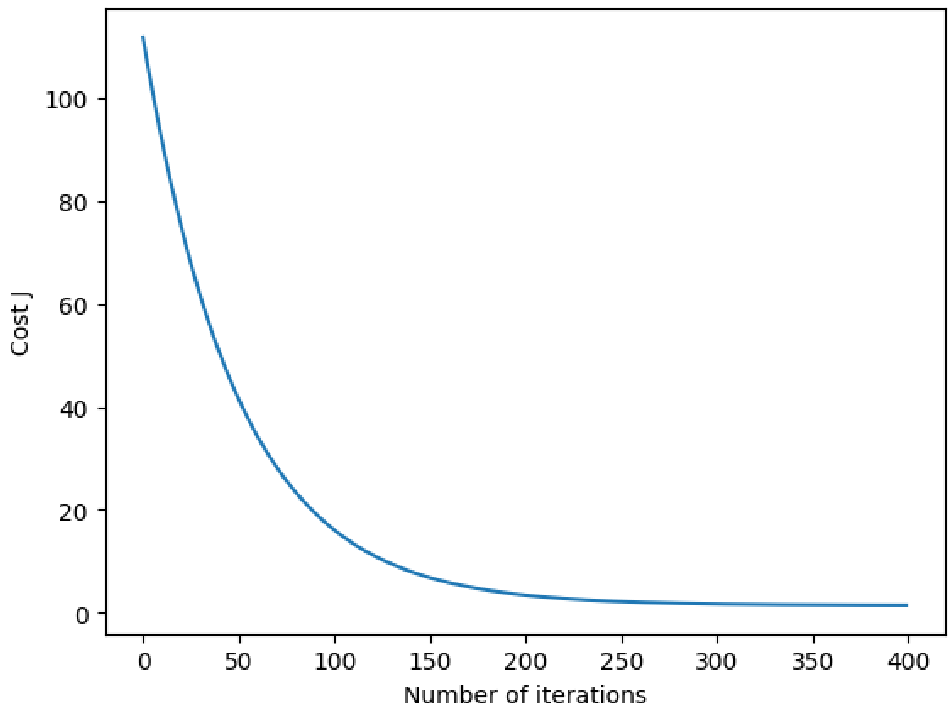
computeCostMulti and gradientDescentMulti function

1. implement:

```
import torch
def computeCostMulti(X, y, theta):

    m = y.size(0)  # number of training examples

    # You need to return the following variables correctly
    J = 0

    diff = torch.matmul(X , theta) - y
    J = (diff.T @ diff) / (2 * m)
    return J

def gradientDescentMulti(X, y, theta, alpha, num_iters):
    # Initialize some useful values
    m = y.size(0) # number of training examples
    J_history = torch.zeros(num_iters, 1)

    for i in range(num_iters):
```

```
        # Save the cost J in every iteration
        J_history[i, 0] = computeCostMulti(X, y, theta)

        diff = X @ theta - y
        theta -= alpha * (X.T @ diff) / m

    return theta, J_history
```

2. result



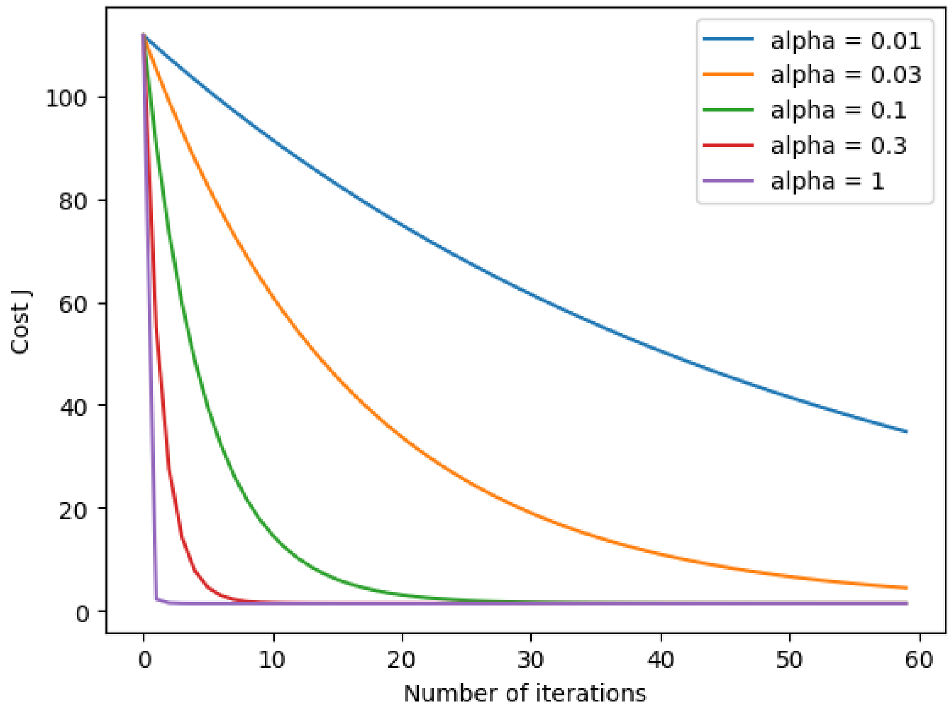

```
Theta found by gradient descent:

tensor([[13.7708],
        [ 3.8605],
        [ 2.6936],
        [ 0.0871]])

Expected theta values (approx)

 13.7708
 3.8605
 2.6936
 0.0871
```

Selecting learning rates

1. implement:

```
import torch
rate = [0.01, 0.03, 0.1, 0.3, 1]
num_iters = 60
plt.figure()
for i in range(0,5):
  theta = torch.zeros(4,1)
  alpha = rate[i]
  theta, J_history = gradientDescentMulti(X, y, theta, alpha, num_iters)
  plt.plot(range(num_iters), J_history ,label = f'alpha = {alpha}')
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
plt.legend()
plt.show()
```

2. result:



Ordinary Least Square (OLS)

1. implement:

```
import torch
def OLS(X, y):
    optimal_theta = torch.zeros(4,1)

    optimal_theta = torch.linalg.inv(X.T @ X) @ X.T @ y
    return optimal_theta
```

2. result:

```
Solving with OLS...

Theta computed from the OLS:

tensor([[ 2.9389e+00],
        [ 4.5765e-02],
        [ 1.8853e-01],
        [-1.0377e-03]])

Expected theta values (approx)

 2.9389e+00
 4.5765e-02
 1.8853e-01
 -1.0373e-03
```

Discuss

1. Gradient Descent. Because when the number of dimension is large, compute matrix inverse is more complex than Gradient Descent
2. feature normalization prevents features with larger values from dominating the loss function. Plus, smaller feature leads to faster training.

PA1-2: Logistic Regression & Clustering

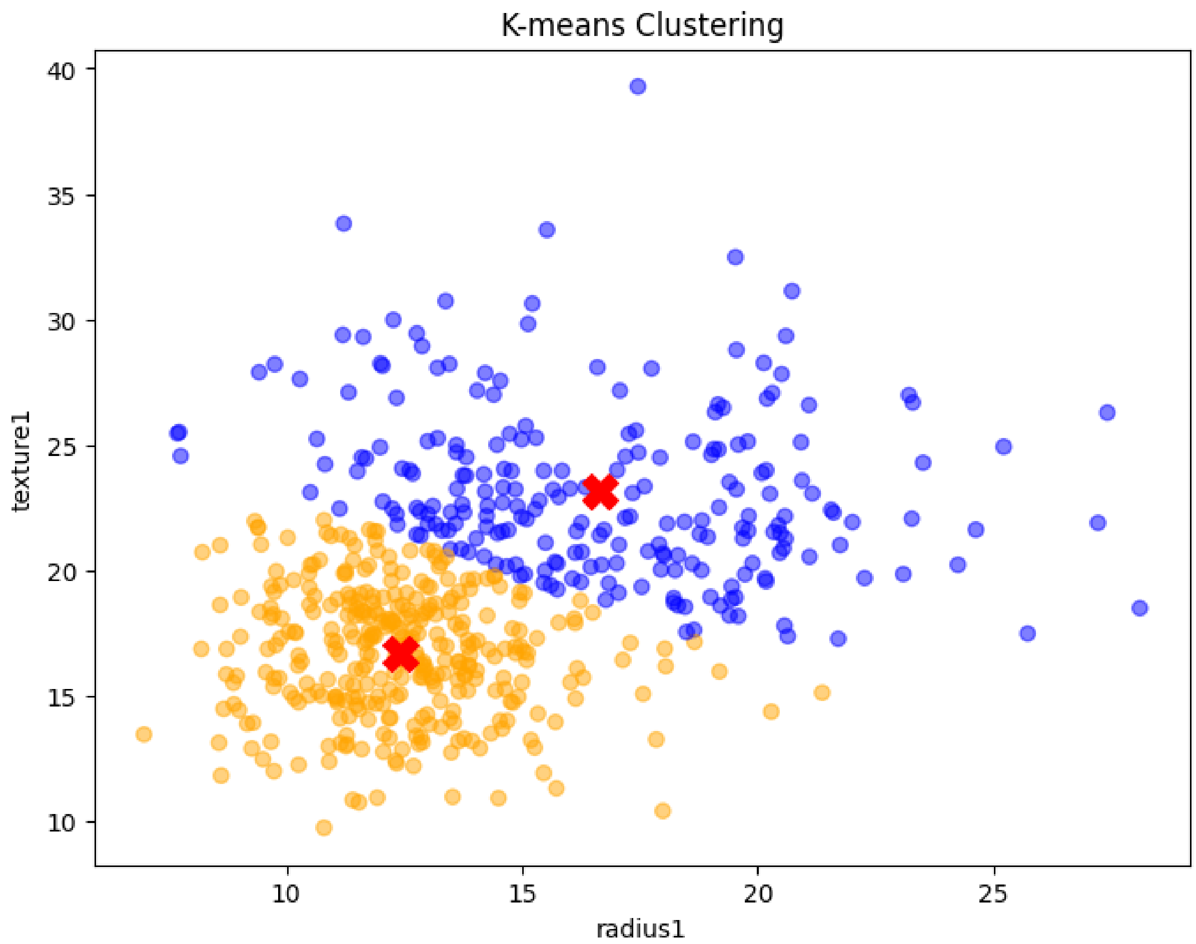
K-means clustering

1. implement

```
import torch
import numpy as np
import matplotlib.pyplot as plt

# origin_X shape :  torch.Size([569, 2])
def k_means(X, K, num_iters=100):
    N = X.size(0)
    # randomly choose the initial center points of cluster
    torch.manual_seed(0)
    centroids = X[torch.randperm(N)[:K]]

    for _ in range(num_iters):
      distances = torch.cdist(X,centroids)
      cluster_assignments = torch.argmin(distances,dim=1)
      new_centroids = torch.zeros(K,X.size(1))
      for i in range(K):
        sum = torch.zeros(1,X.size(1))
        counter = 0
        for m in range(N):
          if i == cluster_assignments[m]:
            sum += X[m,:]
            counter += 1
        new_centroids[i] = sum / counter
      has_converged = torch.allclose(centroids,new_centroids, atol=1e-6)

      centroids = new_centroids
      if(has_converged): break
    return centroids, cluster_assignments
```

2. result

```
Final centroids:
tensor([[12.4232, 16.6911],
        [16.6573, 23.1477]])
Expected centroids (approx):
 [12.4232, 16.6911]
 [16.6573, 23.1477]
```



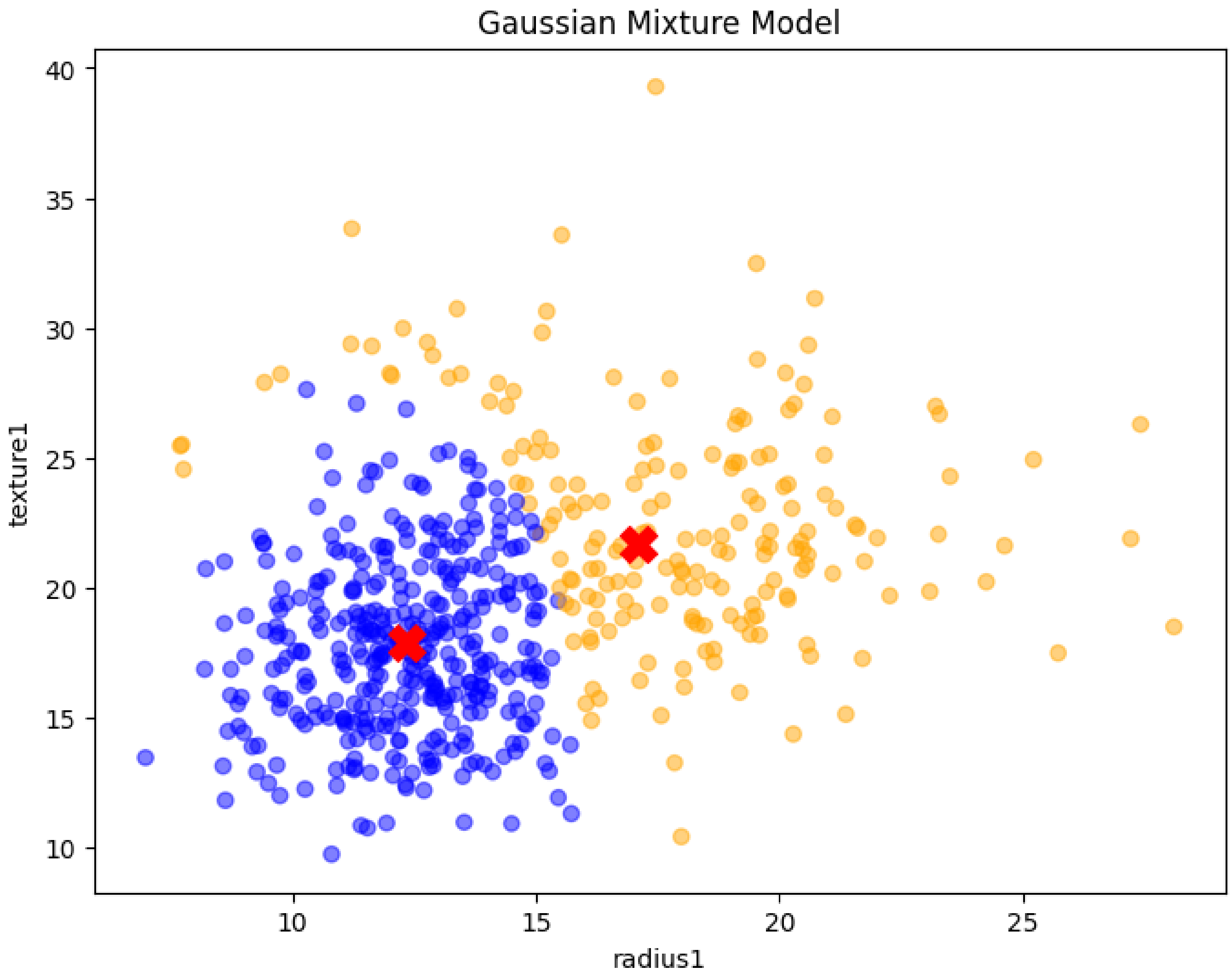
gmm_em function

1. implement:

```
import torch
import numpy as np
import matplotlib.pyplot as plt

def gmm_em(X, K, num_iters=100):
    N, D = X.size()

    # initialization
    torch.manual_seed(0)
    means = X[torch.randperm(N)[:K]]  # (K, D)
    covariances = torch.stack([torch.eye(D) for _ in range(K)])  # (K, D, D)
    weights = torch.ones(K) / K  # (K,)

    eps = 1e-6
    tolerance = 1e-6

    for _ in range(num_iters):
      prev_means = means.clone()
      prev_covariances = covariances.clone()
      # E-step
      resp = torch.zeros(N,K)
      for n in range(N):
        for k in range(K):
          mvn = torch.distributions.MultivariateNormal(means[k],covariances[k])
          log_probs = mvn.log_prob(X[n])
          resp[n,k] = weights[k] * torch.exp(log_probs)

        resp = resp / (resp.sum(dim=1,keepdim=True) + eps)

      cluster_assignments = torch.argmax(resp, dim=1)

      # M-step update
      weights = torch.sum(resp, dim=0) / N
      for k in range(K):
        numerator = 0
        denominator = 0
        for n in range(N):
          numerator += resp[n,k] * X[n]
          denominator += resp[n,k]
        means[k] = numerator / (denominator + eps)

      for k in range(K):
        diff = X - means[k]  # [N, D]
        weighted_diff = resp[:, k].unsqueeze(-1) * diff  # [N, D]
        covariances[k] = (weighted_diff.T @ diff) / resp[:, k].sum()  # [D, D]
        covariances[k] += torch.eye(D) * eps

      mean_change = torch.norm(means - prev_means)
      cov_change = torch.norm(covariances - prev_covariances)
      if mean_change < tolerance and cov_change < tolerance:
        break

    return means, covariances, weights, cluster_assignments
```

2. result

```
Final means:
tensor([[17.0908, 21.6380],
        [12.3380, 17.8716]])
Final covariances:
tensor([[[13.4008,  0.6658],
         [ 0.6658, 19.4277]],

        [[ 3.2861,  0.7161],
         [ 0.7161, 12.5450]]])
Final weights:
tensor([0.3765, 0.6235])
```

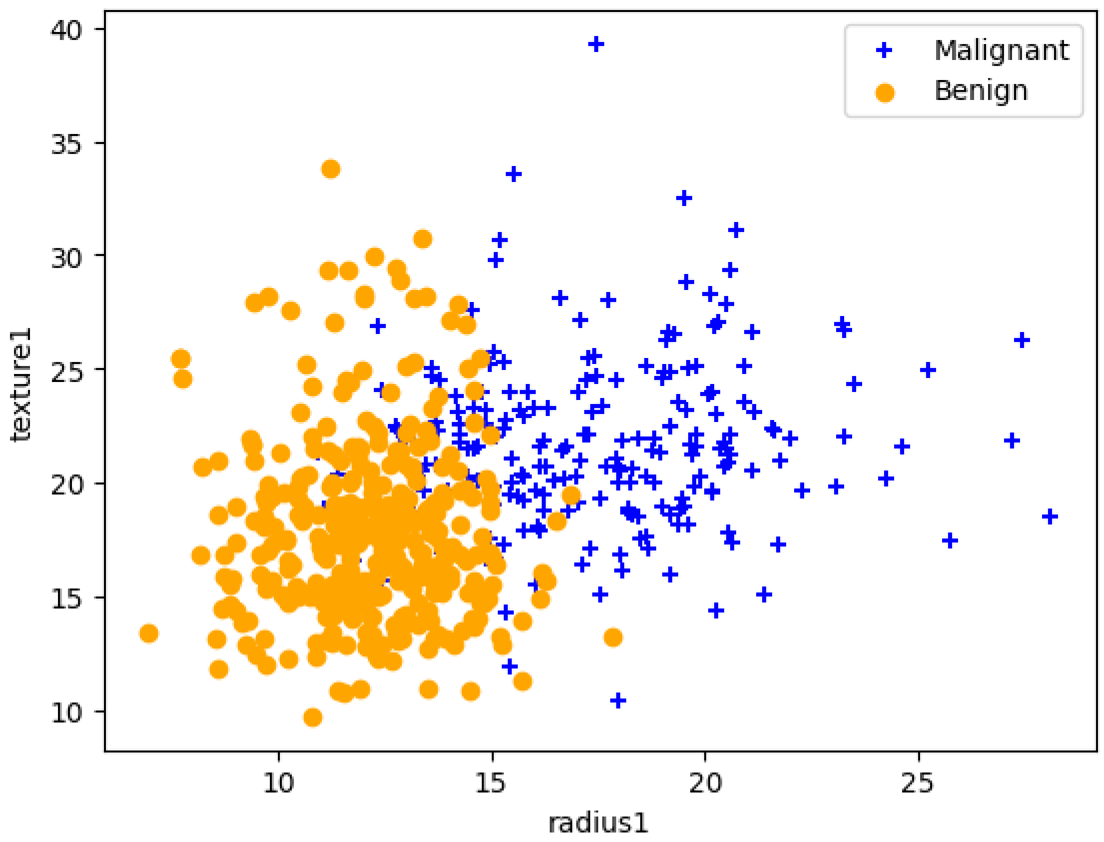


plotData function:

1. implement:

```
import matplotlib.pyplot as plt
def plotData(X, y):

    X = X.numpy()
    y = y.numpy()

    plt.figure()
    pos = np.where(y[:,0] == 1)
    plt.scatter(X[pos,0],X[pos,1],marker='+',color = 'blue', label='Malignant')
    neg = np.where(y[:,0] == 0)
    plt.scatter(X[neg,0],X[neg,1],marker='o',color = 'orange',label='Benign')
    plt.xlabel('radius1')
    plt.ylabel('texture1')
    plt.legend()
```

2. result



sigmoid and costFunction:

1. implement:

```
import torch
def sigmoid(X):

  if not torch.is_tensor(X):
    X = torch.tensor(X)
  input_exp = torch.exp(-X)
  output = 1 / (1 + input_exp)
  return output

def costFunction(theta, X, y):
  m = X.shape[0] # number of training examples

  # You need to return the following variables correctly
  J = 0
  grad = torch.zeros_like(theta)

  predict = sigmoid(X @ theta)
  log_p = torch.log(predict.clamp(min=1e-8))
  log_n = torch.log((1 - predict).clamp(min=1e-8))
  vector = ( y.T @ log_p ) + ( (1 - y).T @ log_n )
  J = - vector.sum() / m

  diff = predict - y
  grad = (X.T @ diff) / m
  return J.item(), grad
```

2. result:

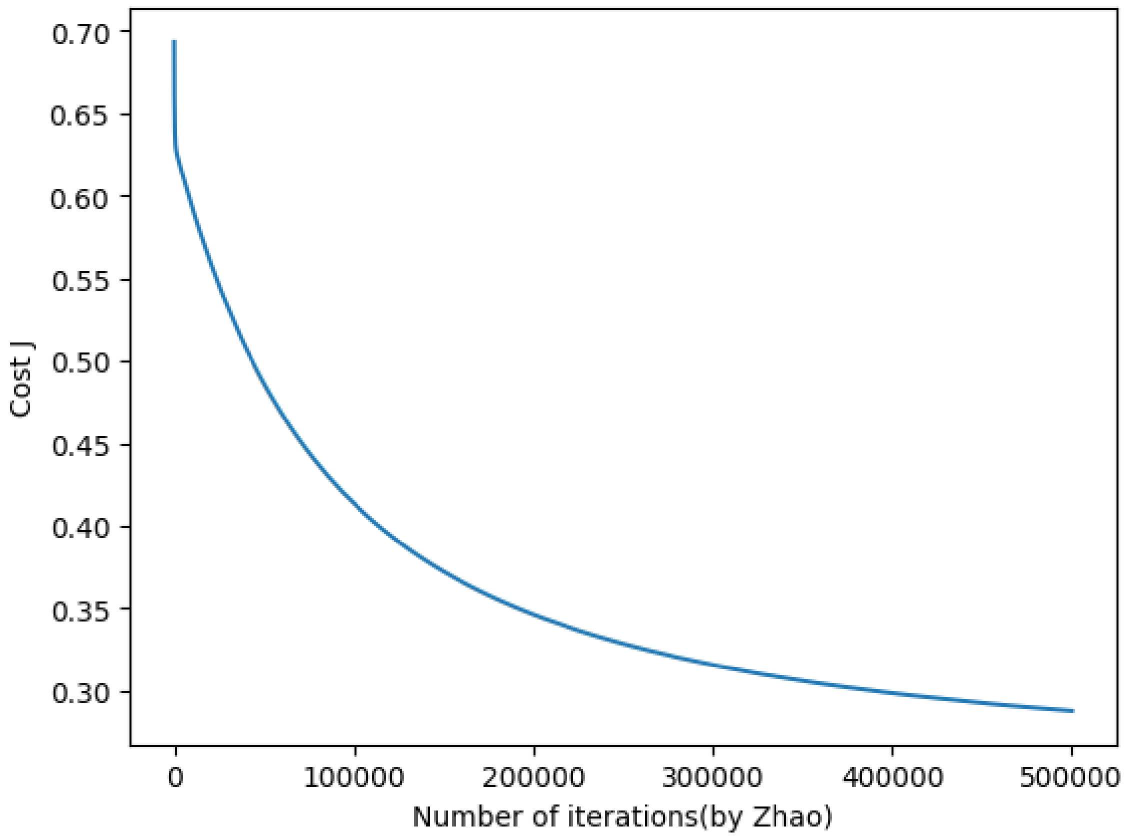
```
Cost at initial theta (zeros):  0.6931460499763489

Expected cost (approx): 0.693

Gradient at initial theta (zeros):

tensor([[0.1274],
        [0.5573],
        [1.5952]])

Expected gradients (approx):
 0.1274
 0.5573
 1.5952

Cost at test theta:  6.030610084533691

Expected cost (approx): 6.0308

Gradient at test theta:

tensor([[-0.3726],
        [-6.5064],
        [-8.0496]])

Expected gradients (approx):
 -0.3726
 -6.5064
 -8.0496
```

Gradient Descent:

1. implement:

```
from tqdm import tqdm
def gradientDescent(X, y, theta, alpha, num_iters):

  J_history = torch.zeros(num_iters,1)

  for idx in tqdm(range(num_iters)):
    grad = torch.zeros_like(theta)
    J_history[idx],grad = costFunction(theta, X, y)
    theta -= alpha * grad
  return theta, J_history
```

2. result

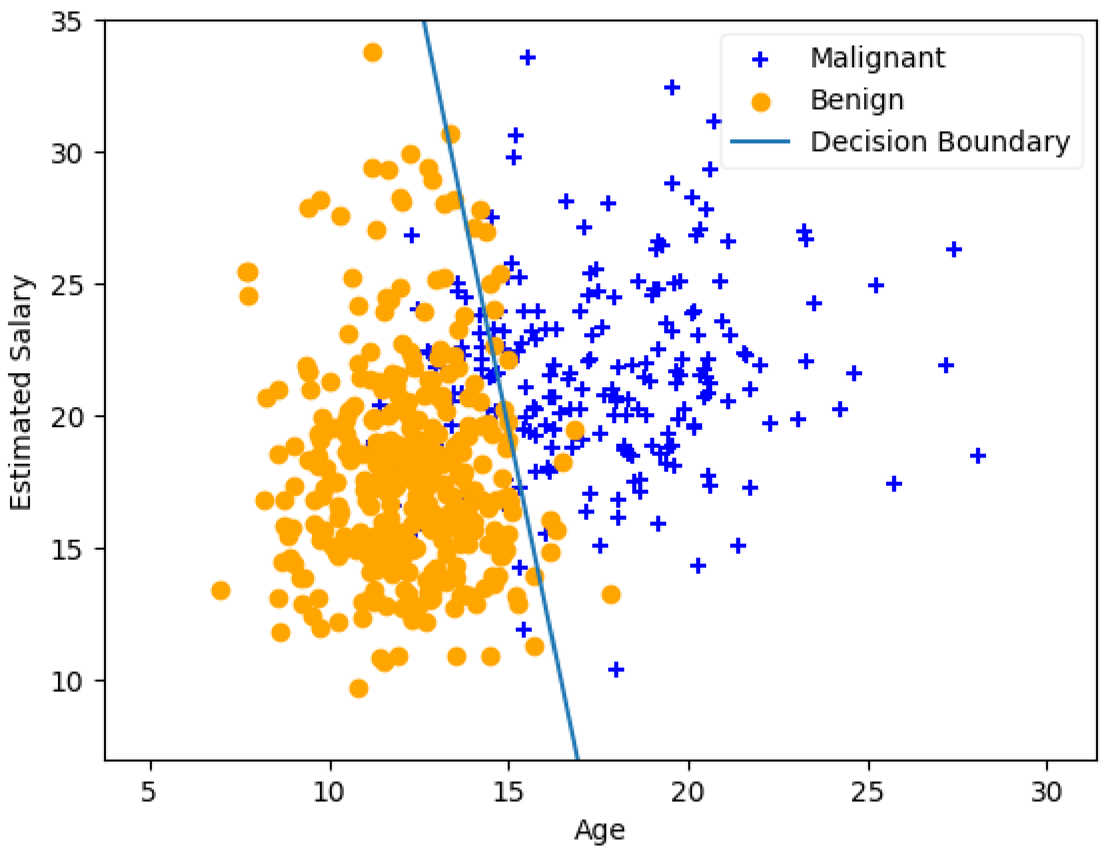


Linear Decision Boundary

1. implement:

```
def plotLinearDecisionBoundary(theta, X, y):

    if X.shape[1] <= 2:
      X = torch.cat((torch.ones(m, 1), X), dim=1)

    # Plot Data
    plt.figure()
    plotData(origin_X, y)
    # Only need 2 points to define a line, so choose two endpoints
    plot_x = torch.Tensor([X[:,1].min() - 2, X[:,1].max() + 2])
    # Calculate the decision boundary line
    plot_y = torch.Tensor((-1/theta[2])*(theta[1]*plot_x + theta[0]))
    # Plot, and adjust axes for better viewing
    plt.plot(plot_x, plot_y, color='C0',label = 'Decision Boundary')
    plt.ylim(7,35)
    plt.legend(frameon=True, facecolor='white', framealpha=0.3)
```

2. result



Check the loss when the model get unseen data as input

1. implement:

```
def gradientDescent_ver2(X_train, y_train, X_unseen, y_unseen, theta, alpha, iterations) :
  # Initialize some useful values
  m = y_train.size(0)  # number of training examples
  m_unseen = y_unseen.size(0)
  J_history = torch.zeros(iterations,1)

  for idx in tqdm(range(iterations)):
    grad = torch.zeros_like(theta)
    # ====================== YOUR CODE HERE ======================
    _,grad = costFunction(theta, X_train, y_train)
    theta -= alpha * grad
    _,grad = costFunction(theta, X_unseen, y_unseen)
```

```
  return theta, J_history
# ====================== Fill the blank parts ======================
X_train = X[:512]
y_train = y[:512]
X_unseen = X[513:569]
y_unseen = y[513:569]

print(X_train.size(), y_train.size())
print(X_unseen.size(), y_unseen.size())
# =============================================================
```

2. result

```
torch.Size([569, 3]) torch.Size([569, 1])
torch.Size([512, 3]) torch.Size([512, 1])
torch.Size([56, 3]) torch.Size([56, 1])
100%|█████████████| 500000/500000 [03:49<00:00, 2183.24it/s]tensor([[-11.4006],
        [  0.6057],
        [  0.1219]])

Expected theta value :
[-11.3874   0.6073   0.1204]
```

Optimizer

1. implement:

```
import numpy as np
def sigmoid_for_optimize(X):
  # ====================== YOUR CODE HERE ======================
  input_exp = np.exp(-X)
  output = 1 / (1 + input_exp)
  # ==============================================================
  return output

def costFunction_for_optimize(theta, X, y) :
    # ====================== YOUR CODE HERE ======================
    # Instructions: Re-implement the costFunction that avoids the TypeError
    # =============================================================
    m = X.shape[0] # number of training examples
    theta = theta.reshape(-1, 1)
    y = y.reshape(-1, 1)
    # You need to return the following variables correctly
    J = 0
    grad = np.zeros_like(theta)

    predict = sigmoid_for_optimize(np.matmul(X, theta))
    log_p = np.log(predict + 1e-8)
    log_n = np.log((1 - predict)+ 1e-8)
    J = -1 / m * np.sum(y * log_p + (1 - y) * log_n)

    diff = predict - y
    grad = (np.matmul(X.T, diff) / m).flatten()
    return J, grad
```

2. result:

```
Executing minimize function...

Cost at theta found :
 0.25582011345964734

Expected cost (approx): 0.0.256

theta:
 [-19.84941693   1.05710182   0.21814105]

Expected theta (approx):

 -19.85
 1.06
 0.22
```

predict function:

1. implement:

```
def predict(theta, X):


    threshold = 0.5
    m = X.shape[0] # Number of training examples

    # You need to return the following variables correctly
    p = torch.zeros(m, 1)

    h_x = sigmoid(X @ theta)
    p = (h_x >= threshold).float()
    return p

import torch
theta = torch.tensor([[-19.84941693 ,  1.05710182 ,  0.21814105] ])
x_sample = torch.tensor([[1, 15, 20]], dtype=torch.float32)
theta = theta.reshape(-1, 1)
prob = sigmoid(x_sample @ theta)
```

2. result:

```
For a cancer with radius 15 and texture 20, we predict an Malignancy probability of  0.59144258499
Expected value: 0.591


Train Accuracy:  tensor(89.1037)

Expected accuracy (approx): 89.1
```

Think about

Your answer:

Q1:

- sigmoid function can map the output to a probability between 0 and 1.
- we can use other function, such as tanh or Softmax.

Q2:

- adding polynomial features transforms the original feature space into a higher-dimensional space.This allow model to fit non-linear relationships in the data.

Q3:

- λ is a punishment item , large λ punish the model by increase the loss value.
- Plus,large λ cause model becomes constrained , limiting its ability to capture the underlyting pattens in the data, This leads to **underftting**.
- small λ makes model more flexible, potentially fitting the noise in the trainning data. leading to **overfitting**.

PA1-3:Fully-Connected Neural Nets & Convolution Neural Nets

Implementing a Neural Network

class TwoLayerNet :

1. implement:

```
def ReLU(self, X):
        ###################################################
        # TODO: Implement the ReLU activation function    #
        ###################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #return torch.max(X,torch.tensor(0.0))
        return torch.clamp(X,min=0)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def SoftMax(self, X):
        #########################################################################
        # TODO: Implement the SoftMax activation function                       #
        #########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        data_withoutMax = X - X.max(dim=1,keepdim=True).values
        return torch.exp(data_withoutMax) / torch.exp(data_withoutMax).sum(dim=1,keepdim=True)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def forward(self, X):

        out = None
        #########################################################################
        # TODO: Implement the code that gets ouptut by using input data and     #
        # the parameter of models(i.e. weight, bias) and ReLU                   #
        #########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #layer1
        a1 = X.mm(self.params['W1']) + self.params['b1']
        #use ReLU
        z1 = self.ReLU(a1)
```

```
        #layer2
        out = z1.mm(self.params['W2']) + self.params['b1']
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        return out
```

2. result:

```
X: tensor([[0., 0.],
        [0., 1.],
        [1., 0.],
        [1., 1.]])
y: tensor([0, 1, 1, 0])

y_hat: tensor([[ 0.0000,  0.0000],
        [ 0.0025, -0.0019],
        [-0.0009, -0.0014],
        [ 0.0015, -0.0034]])
prediction: tensor([[0.5000, 0.5000],
        [0.5011, 0.4989],
        [0.5001, 0.4999],
        [0.5012, 0.4988]])
sum of pred prob. for each input: tensor([1., 1., 1., 1.])
```

loss_funciton

1. implement:

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
one_hot = torch.nn.functional.one_hot(y, num_classes = n_cls) #[y_indexMax,n_cls]
print(one_hot)
y_hat = net.SoftMax(y_hat)
log_y_hat = torch.log(y_hat + 1e-6)
print(log_y_hat)
#each_class_loss = - log_y_hat.T @ one_hot.float() - torch.log(1 - y_hat).T @ (1 - one_hot.fl
loss_per_sample = -torch.sum(one_hot.float() * log_y_hat, dim=1)
print(loss_per_sample)
loss = torch.mean(loss_per_sample)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result

```
tensor([[1, 0],
        [0, 1],
        [0, 1],
        [1, 0]])
tensor([[-0.6931, -0.6931],
        [-0.6910, -0.6953],
        [-0.6929, -0.6934],
        [-0.6907, -0.6956]])
tensor([0.6931, 0.6953, 0.6934, 0.6907])
loss_from_implemented_method: 0.6931464672088623, loss_from_pytorch_module: 0.6931484937667847
distance: 2.0265579223632812e-06
```

get_gradient

1. implement:

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #forward prosess
    #layer1
    a1 = X.mm(net.params['W1']) + net.params['b1']
    #use ReLU
    z1 = net.ReLU(a1)
    #layer2
    out = z1.mm(net.params['W2']) + net.params['b1']
```

```
    y_hat = net.SoftMax(net.forward(X))
    y_one_hot = torch.nn.functional.one_hot(y, num_classes = n_cls)
    #probability map activate function derivative
    dy_dz = y_hat - y_one_hot
    #layers 2 backpropagation
    gradient['W2'] = a1.T @ dy_dz
    gradient['b2'] = dy_dz.sum(0)
    #layer activate function ReLU
    da1_dz1 = dy_dz @ net.params['W2'].T * (a1 > 0).float()
    gradient['W1'] = X.T @ da1_dz1
    gradient['b1'] = da1_dz1.sum(0)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result

```
{'W1': tensor([[0.0337, 0.0129],
        [0.0234, 0.0230]]), 'b1': tensor([0., 0.]), 'W2': tensor([[-0.1123, -0.0186],
        [ 0.2208, -0.0638]]), 'b2': tensor([0., 0.])}
Epoch [100/1000], Loss: 0.6824, Accuracy: 75.00%
Epoch [200/1000], Loss: 0.6888, Accuracy: 75.00%
Epoch [300/1000], Loss: 0.6915, Accuracy: 100.00%
Epoch [400/1000], Loss: 0.6918, Accuracy: 100.00%
Epoch [500/1000], Loss: 0.6918, Accuracy: 100.00%
Epoch [600/1000], Loss: 0.6919, Accuracy: 100.00%
Epoch [700/1000], Loss: 0.6919, Accuracy: 100.00%
Epoch [800/1000], Loss: 0.6919, Accuracy: 100.00%
Epoch [900/1000], Loss: 0.6919, Accuracy: 100.00%
Epoch [1000/1000], Loss: 0.6919, Accuracy: 100.00%
```

```
=== Outputs Before Training ===
Predictions: tensor([0, 0, 0, 0])
Actual Labels: tensor([0, 1, 1, 0])
Accuracy Before Training: 50.00%

=== Outputs After Training ===
Predictions: tensor([0, 1, 1, 0])
Actual Labels: tensor([0, 1, 1, 0])
Accuracy After Training: 100.00%
```

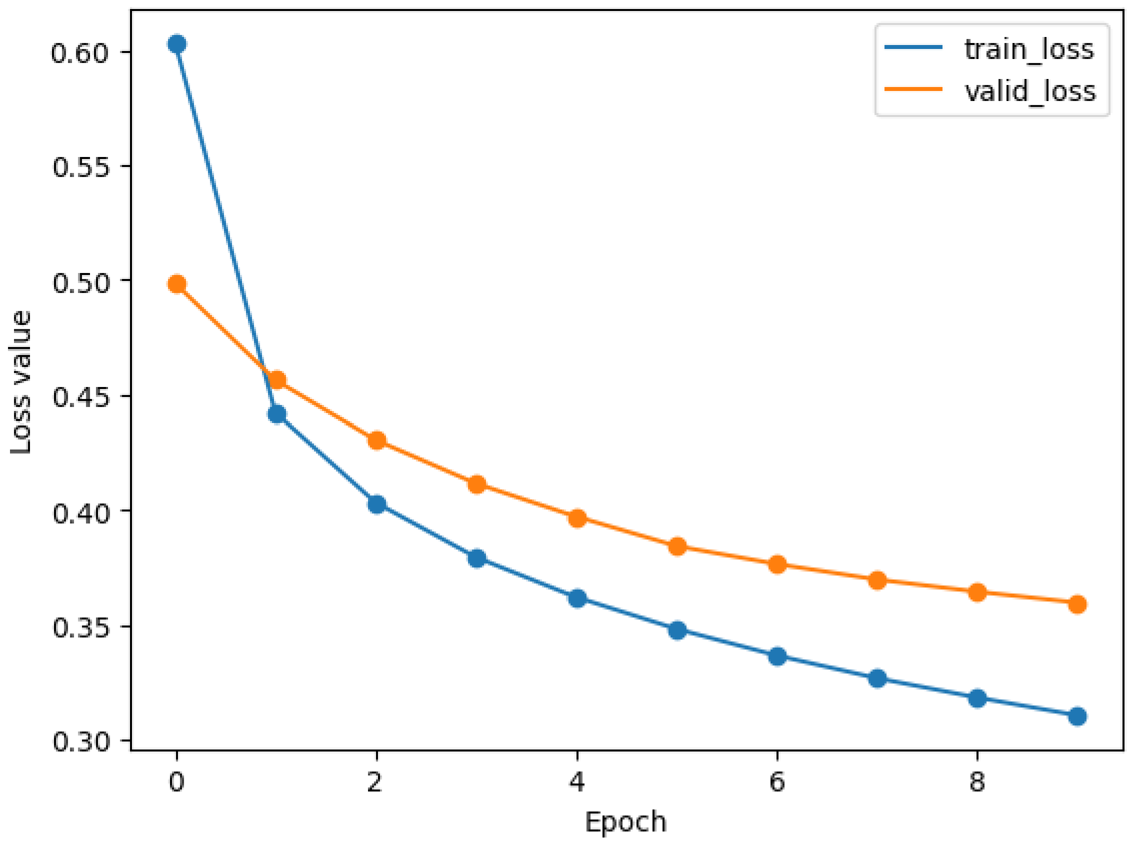
Train the model using Fashion-MNIST

1. implement:

```
##########################################################################################
# TODO: Implement the training part of train() method.
# In this part, you need to add the loss to mean_train_loss that is appended to 'train_loss_hist
##########################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
batch_start = i * batch_size
batch_end = (i + 1) * batch_size
loss = loss_function(net, train_X[batch_start:batch_end], train_y[batch_start:batch_end], n_cls)
mean_train_loss += loss.item()
grads = get_gradient(net, train_X[batch_start:batch_end], train_y[batch_start:batch_end], n_cls)
net = update_network(net, grads, learning_rate)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
##########################################################################################
# TODO: Implement the validation part of train() method.
# In this part, you need to add the loss to mean_valid_loss that is appended to 'valid_loss_hist
##########################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
loss = loss_function(net,valid_X[i*valid_batch_size:(i+1)*valid_batch_size],valid_y[i*valid_batc
mean_valid_loss += loss
```

```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
##################################################################################################
# TODO: Implement the test()
# Hint: If you get accuracy of one input data, you should reshape the size of input as [1, 3072]
##################################################################################################
answer_count = 0
total_count = test_X.size(0)
for idx in range(total_count):
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    pred = net.prediction(test_X[idx].reshape(1, -1))
    predicted = torch.max(pred, 1)[1]
    if predicted.item() == test_y[idx]:
        answer_count += 1
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result



accuracy: 85.84%, expected value (app.) : 78.04%

Train Convolution Neural Network on Fashion-MNIST

reshape and normalization

1. implement

```
mean = 0.1307
std = 0.3081
def reshape_and_Normalization(X):
  X = X.view(-1,1,28,28)
  X = (X - mean) / std
  return X

train_X = reshape_and_Normalization(train_X)
valid_X = reshape_and_Normalization(valid_X)
test_X = reshape_and_Normalization(test_X)
```

2. result

```
train_X shape: torch.Size([54000, 1, 28, 28])
train_y shape: torch.Size([54000])
valid_X shape: torch.Size([6000, 1, 28, 28])
valid_y shape: torch.Size([6000])
test_X shape: torch.Size([10000, 1, 28, 28])
test_y shape: torch.Size([10000])
```

CNNNet

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv = nn.Conv2d(in_channels=1,out_channels=32,kernel_size=3,stride=1,padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2,stride=2)
        self.fc1 = nn.Linear(in_features=32 * 14 * 14,out_features=10)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
def forward(self, x):
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    x = self.conv(x)
    x = torch.relu(x)
    x = self.pool(x)
    x = x.view(x.size(0),-1)
    x = self.fc1(x)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return x
```

train_model

```
def train_model(model, train_X, train_y, valid_X, valid_y, criterion, optimizer, num_epochs):
    train_dataset = torch.utils.data.TensorDataset(train_X, train_y)
    valid_dataset = torch.utils.data.TensorDataset(valid_X, valid_y)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False)

    train_loss_history = []
    valid_loss_history = []
    train_acc_history = []
    valid_acc_history = []
    epoch = 0

    for i in tqdm(range(num_epochs), desc=f"train, epoch: {epoch + 1}"):
        ################################################################################################
        # TODO: Design train procedure, calculate train_loss, train_acc, valid_loss, and valid_a
        # Hint: Use train() for training procedure and eval() for validation, torch.nn.CrossEnt
        #       torch.optim.Adam.step() automatically updates parameters by using gradients. Use
        #
        ################################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        epoch += 1
        # Training Phase
        model.train()
        running_loss = 0.0
        correct_predictions = 0
        total_samples = 0

        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
```

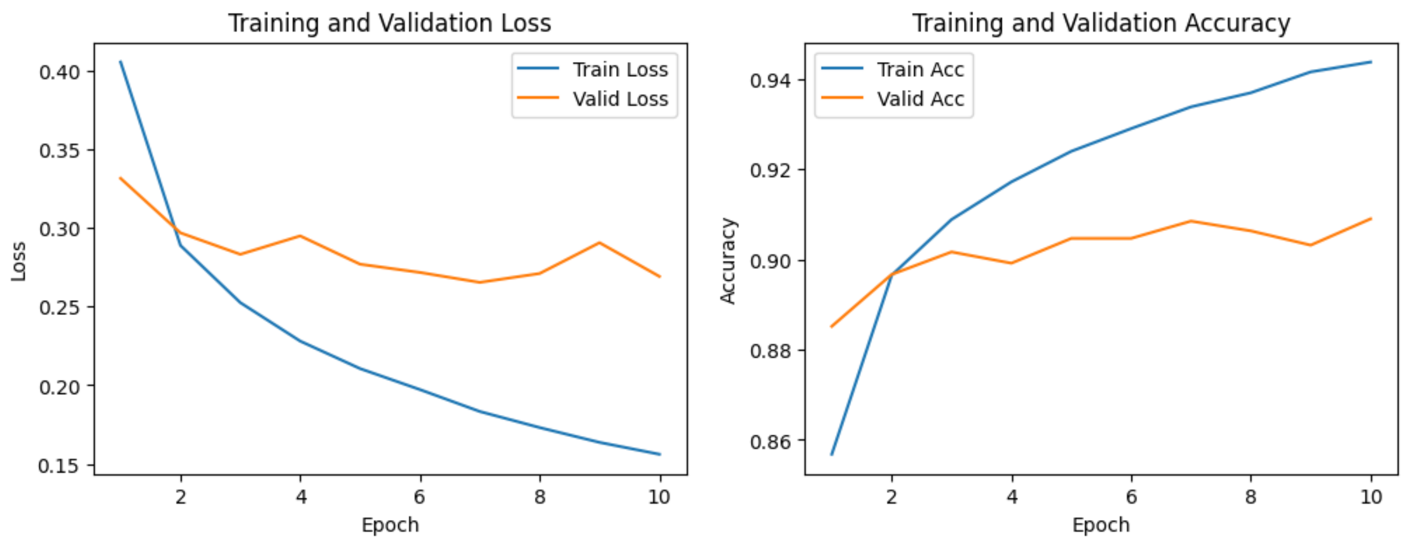
```
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * batch_X.size(0)
            _, preds = torch.max(outputs, 1)
            correct_predictions += torch.sum(preds == batch_y).item()
            total_samples += batch_X.size(0)

        train_loss = running_loss / total_samples
        train_acc = correct_predictions / total_samples
        train_loss_history.append(train_loss)
        train_acc_history.append(train_acc)

        # Validation Phase
        model.eval()
        val_running_loss = 0.0
        val_correct_predictions = 0
        val_total_samples = 0

        with torch.no_grad():
            for batch_X, batch_y in valid_loader:
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                val_running_loss += loss.item() * batch_X.size(0)
                _, preds = torch.max(outputs, 1)
                val_correct_predictions += torch.sum(preds == batch_y).item()
                val_total_samples += batch_X.size(0)

        valid_loss = val_running_loss / val_total_samples
        valid_acc = val_correct_predictions / val_total_samples
        valid_loss_history.append(valid_loss)
        valid_acc_history.append(valid_acc)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        print(f'Epoch [{epoch+1}/{num_epochs}] '
              f'Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} '
              f'Valid Loss: {valid_loss.item():.4f} Acc: {valid_acc:.4f}')

    return model, train_loss_history, valid_loss_history, train_acc_history, valid_acc_history
```

test_model

```
def test_model(model, test_X, test_y):
    ###########################################################################################
    # TODO: Design test procedure
    # Hint: If you properly designed validation process, it is not actually different process a
    ###########################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    test_dataset = TensorDataset(test_X, test_y)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

    model.eval()
    test_correct_pred = 0
    test_total_samples = 0

    with torch.no_grad():
```

```
        for batch_X, batch_y in test_loader:
            outputs = model(batch_X)
            _, preds = torch.max(outputs, 1)
            test_correct_pred += torch.sum(preds == batch_y).item()
            test_total_samples += batch_X.size(0)

    test_acc = test_correct_pred / test_total_samples
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    print(f'Test Accuracy: {test_acc * 100:.2f}%')
    return test_acc

test_accuracy = test_model(model, test_X, test_y)
```

result




```
Epoch [1/10] Train Loss: 0.4056 Acc: 0.8568 Valid Loss: 0.3316 Acc: 0.8852
Epoch [2/10] Train Loss: 0.2890 Acc: 0.8965 Valid Loss: 0.2970 Acc: 0.8967
Epoch [3/10] Train Loss: 0.2526 Acc: 0.9089 Valid Loss: 0.2833 Acc: 0.9017
Epoch [4/10] Train Loss: 0.2281 Acc: 0.9172 Valid Loss: 0.2950 Acc: 0.8992
Epoch [5/10] Train Loss: 0.2106 Acc: 0.9240 Valid Loss: 0.2770 Acc: 0.9047
Epoch [6/10] Train Loss: 0.1973 Acc: 0.9290 Valid Loss: 0.2718 Acc: 0.9047
Epoch [7/10] Train Loss: 0.1833 Acc: 0.9338 Valid Loss: 0.2654 Acc: 0.9085
Epoch [8/10] Train Loss: 0.1731 Acc: 0.9370 Valid Loss: 0.2711 Acc: 0.9063
Epoch [9/10] Train Loss: 0.1637 Acc: 0.9416 Valid Loss: 0.2907 Acc: 0.9032
Epoch [10/10] Train Loss: 0.1561 Acc: 0.9438 Valid Loss: 0.2692 Acc: 0.9090
```

```
Test Accuracy: 90.74%
```

PA1-4 Optimizer

SGD_with_momeentum

1. implement

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    for key in params.keys():
        params[key] = params[key] - learning_rate * grads[key] + momentum * previous_grads[ke
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result

```
params_before_update['b2']: tensor([0., 0., 0.])
===Comparison===
params_after_update_sgd['b2']: tensor([-1.3875e-04,  6.3491e-05,  7.5257e-05])
params_after_update_sgd_momentum['b2']: tensor([ 0.6252, -0.2872, -0.3380])
```

AdaGrad

1. implement

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    eps = 1e-8
    for keys in params.keys():
      acc_grads[keys] = acc_grads[keys] + grads[keys]**2
      adjusted_learning_rate = learning_rate / (acc_grads[keys] + eps)
      params[keys] = params[keys] - adjusted_learning_rate * grads[keys]
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result

```
params_before_update['b2']: tensor([0., 0., 0.])
===Comparison===
params_after_update_sgd['b2']: tensor([-1.3875e-04,  6.3491e-05,  7.5257e-05])
params_after_update_sgd_momentum['b2']: tensor([ 0.6252, -0.2872, -0.3380])
params_after_update_ada_grad['b2']: tensor([-0.0002,  0.0005,  0.0004])
```

Adam

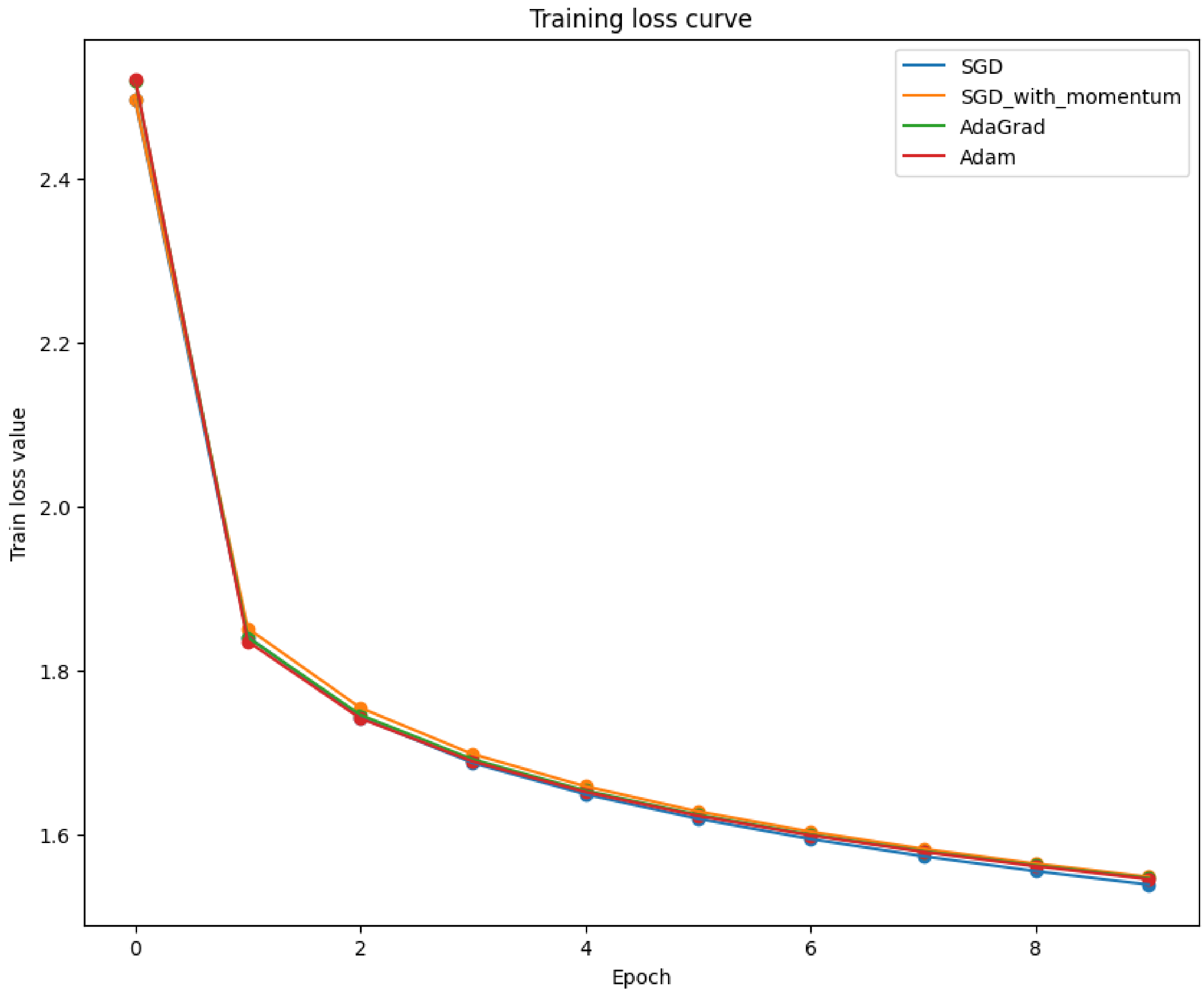
1. implement

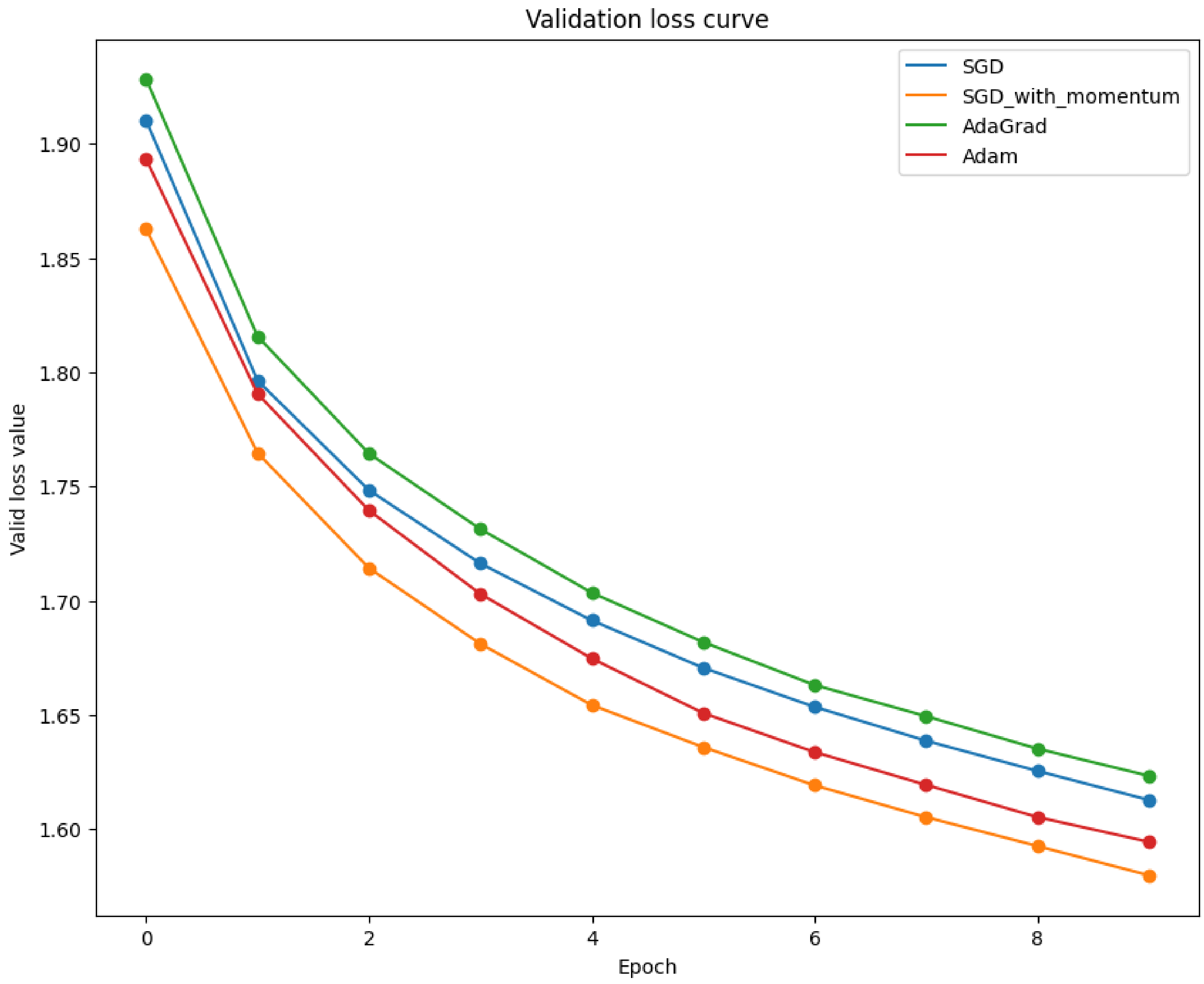
```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    for key in params.keys() :
      first_moment[key] = beta1 * first_moment[key] + (1 - beta1) * grads[key]
      first_moment[key] = first_moment[key] / (1 - beta1)
      second_moment[key] = beta2 * second_moment[key] + (1 - beta2) * grads[key]**2
      second_moment[key] = second_moment[key] / (1 - beta2)
      params[key] = params[key] - (learning_rate * first_moment[key]) / (torch.sqrt(second_mo

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2. result

```
params_before_update['b2']: tensor([0., 0., 0.])
===Comparison===
params_after_update_sgd['b2']: tensor([-1.3875e-04,  6.3491e-05,  7.5257e-05])
params_after_update_sgd_momentum['b2']: tensor([ 0.6252, -0.2872, -0.3380])
params_after_update_ada_grad['b2']: tensor([-0.0002,  0.0005,  0.0004])
params_after_update_adam['b2']: tensor([-0.0001,  0.0001,  0.0001])
```

Result


Training loss curve
SGD
SGD_with_momentum
AdaGrad
Adam
Train loss value
2.4
2.2
2.0
1.8
1.6
0
2
4
6
8
Epoch


Validation loss curve
SGD
SGD_with_momentum
AdaGrad
Adam
Valid loss value
1.90
1.85
1.80
1.75
1.70
1.65
1.60
0
2
4
6
8
Epoch