

# DATA 609 - Homework 7: Nonconvex Optimization and Deep Learning

Eddie Xu

## Instructions

For this problem I recommend submitting a python notebook or a quarto file. It might be easier with a python notebook. You can use google colab if you do not have enough computational power in your personal computer.

## Problem 1: Comparing Optimization Algorithms

Consider the function,  $f(x, y) = (1 - x^2) + 100(y - x^2)^2$  which has a global minimum at,  $x = 1$ . For this problem, you are going to explore using different optimization algorithms to find the global minimum, to gain an intuitive understanding of their different strengths and weaknesses.

- (a) Make a contour plot of this function. You should observe a that the contour lines are “banana-shaped” around the global minimum point, which lies in a deep valley. In technical terms, we would say that the gradient of this function is strongly anisotropic, a fact that can cause slow or no convergence for optimization algorithms.

## Problem 1(a) Solution

```
# load dependencies
import numpy as np
import matplotlib.pyplot as plt

# define function
def prob_func(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2
```

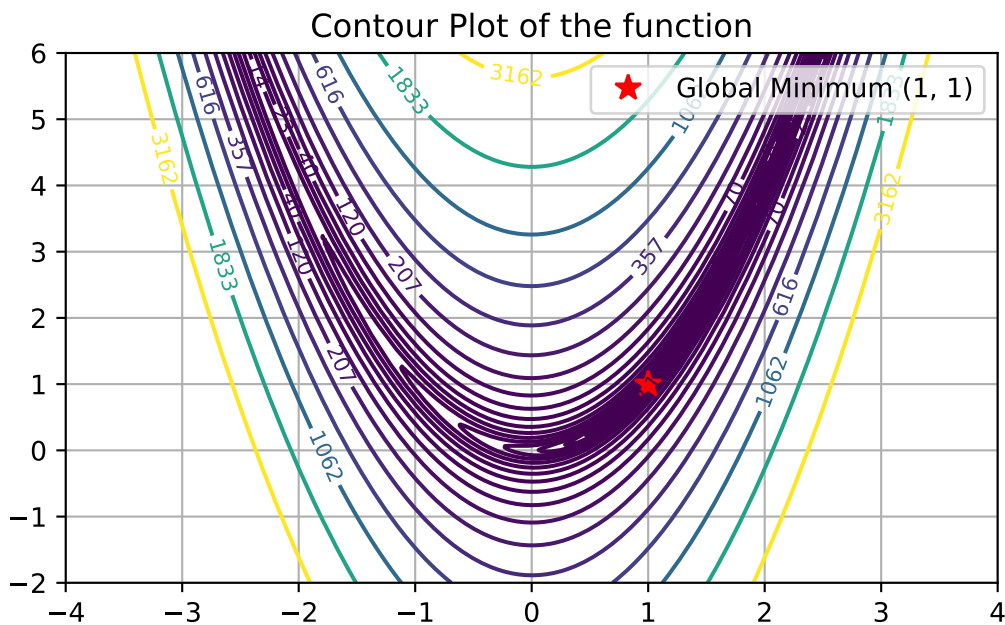
```

# Create a grid of (x, y) values
x = np.linspace(-4, 4, 400)
y = np.linspace(-2, 6, 400)
X, Y = np.meshgrid(x, y)
Z = prob_func(X, Y)

# Plot the contour
contours = plt.contour(X, Y, Z, levels=np.logspace(-1, 3.5, 20), cmap='viridis')
plt.clabel(contours, inline=True, fontsize=8)
plt.plot(1, 1, 'r*', markersize=10, label='Global Minimum (1, 1)')
plt.title('Contour Plot of the function')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

```



- (b) In the code chunk below I have python code for three different optimization algorithms, (1) stochastic gradient descent; (2) stochastic gradient descent with momentum, and (3) ADAM (ADaptive Moment Estimation). Starting at the initial point  $x = -4, y = -2$ , use each algorithm to find the minimum of the function  $f$ . Start with a learning rate of  $\kappa = 10^{-4}$  for all three algorithms, and run the algorithm for  $10^5$  timesteps. Plot the trajectories of each algorithm and the log base 10 of the error rate as

a function of the time step. What do you notice about the performance of the difference algorithms, both in terms of convergence speed and ultimate accuracy?

```
# available defined function
def gd(grad, init, n_epochs=1000, eta=10**-4):

    params=np.array(init)
    param_traj=np.zeros([n_epochs+1,2])
    param_traj[0,]=init
    v=0;
    for j in range(n_epochs):
        v=eta*(np.array(grad(params)))
        params=params-v
        param_traj[j+1,]=params
    return param_traj

def gd_with_mom(grad, init, n_epochs=5000, eta=10**-4, beta=0.9,gamma=0.9):
    params=np.array(init) # Start with initial condition
    param_traj=np.zeros([n_epochs+1,2]) # Save the entire trajecotry
    param_traj[0,]=init # Also save the initial condition to the trajectory

    v=0 # Starting with 0 momentum

    # Epochs is borrowing term from machine learning
    # Here it means timestep

    for j in range(n_epochs):
        v=gamma*v+(np.array(grad(params))) # Compute v
        params=params-eta*v # Update the location
        param_traj[j+1,]=params # Save the trajectory
    return param_traj

def adams(grad, init, n_epochs=5000, eta=10**-4, gamma=0.9, beta=0.99,epsilon=10**-8):
    params=np.array(init)
    param_traj=np.zeros([n_epochs+1,2])
    param_traj[0,]=init
    v=0;
    grad_sq=0;
    for j in range(n_epochs):
        g=np.array(grad(params))
        v=gamma*v+(1-gamma)*g
        grad_sq=beta*grad_sq+(1-beta)*g*g
        v_hat=v/(1-gamma**(j+1))
```

```

        grad_sq_hat=grad_sq/(1-beta**(j+1))
        params=params-eta*np.divide(v_hat,np.sqrt(grad_sq_hat+epsilon))
        param_traj[j+1,]=params
    return param_traj

# load dependencies
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# define function
def prob_func(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

def prob_func_grad(xy):
    x, y = xy
    dfdx = -2 * (1 - x) - 400 * x * (y - x**2)
    dfdy = 200 * (y - x**2)
    return np.array([dfdx, dfdy])

# initialize the optimizer
init_point = [-4.0, -2.0]
eta = 1e-4
n_epochs = 100000

# run the optimizer
gd_traj = gd(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta)
mom_traj = gd_with_mom(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta)
adam_traj = adams(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta)

# compute log10 error
true_min = np.array([1, 1])
gd_error = np.log10(np.sum((gd_traj - true_min)**2, axis=1))
mom_error = np.log10(np.sum((mom_traj - true_min)**2, axis=1))
adam_error = np.log10(np.sum((adam_traj - true_min)**2, axis=1))

# create a contour plot of the function
x = np.linspace(-4.5, 4.5, 400)
y = np.linspace(-3, 3, 400)
X, Y = np.meshgrid(x, y)
Z = prob_func(X, Y)

```

```

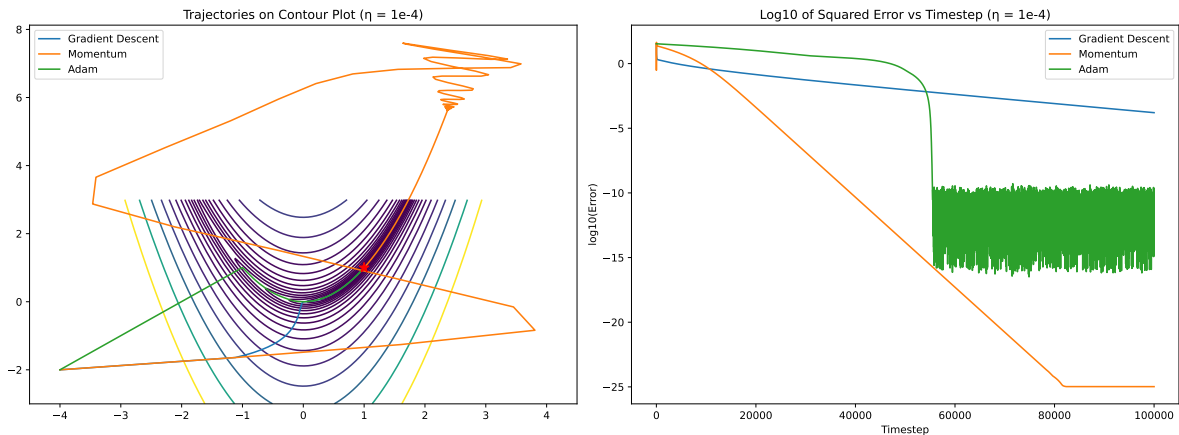
# plot results
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# plot the contour with trajectories
axs[0].contour(X, Y, Z, levels=np.logspace(-1, 3.5, 20), cmap='viridis')
axs[0].plot(gd_traj[:, 0], gd_traj[:, 1], label='Gradient Descent')
axs[0].plot(mom_traj[:, 0], mom_traj[:, 1], label='Momentum')
axs[0].plot(adam_traj[:, 0], adam_traj[:, 1], label='Adam')
axs[0].plot(1, 1, 'r*', markersize=10)
axs[0].set_title("Trajectories on Contour Plot (  $\eta = 1e-4$ )")
axs[0].legend()

# plot the log error
axs[1].plot(gd_error, label='Gradient Descent')
axs[1].plot(mom_error, label='Momentum')
axs[1].plot(adam_error, label='Adam')
axs[1].set_title("Log10 of Squared Error vs Timestep (  $\eta = 1e-4$ )")
axs[1].set_xlabel("Timestep")
axs[1].set_ylabel("log10(Error)")
axs[1].legend()

plt.tight_layout()
plt.show()

```



- (c) Perform the same experiment for the learning rate  $\kappa = 10^{-3}$ , only comparing ADAM and gradient descent with momentum. You will likely observe that one of the methods does not converge, keep the same range of values for your trajectory/contour plot as you did in (b). Which method worked better with  $\kappa = 10^3$ ?

- (d) Now perform a comparison between ADAM with  $\kappa = 10^{-2}$  against gradient descent with momentum using . What are the trade-offs between the two methods for these values of the learning rate?

### Problem 1(c) and 1(d) Solution

```
# load dependencies
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# define function
def prob_func(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

def prob_func_grad(xy):
    x, y = xy
    dfdx = -2 * (1 - x) - 400 * x * (y - x**2)
    dfdy = 200 * (y - x**2)
    return np.array([dfdx, dfdy])

# set the applied the function
def adams(grad, init, n_epochs=5000, eta=10**-4, gamma=0.9, beta=0.99, epsilon=10**-8):
    params=np.array(init)
    param_traj=np.zeros([n_epochs+1,2])
    param_traj[0,]=init
    v=0;
    grad_sq=0;
    for j in range(n_epochs):
        g=np.array(grad(params))
        v=gamma*v+(1-gamma)*g
        grad_sq=beta*grad_sq+(1-beta)*g*g
        v_hat=v/(1-gamma**(j+1))
        grad_sq_hat=grad_sq/(1-beta**(j+1))
        params=params-eta*np.divide(v_hat,np.sqrt(grad_sq_hat+epsilon))
        param_traj[j+1,]=params
    return param_traj

def gd_with_mom(grad, init, n_epochs=5000, eta=10**-4, beta=0.9, gamma=0.9):
    params=np.array(init) # Start with initial condition
    param_traj=np.zeros([n_epochs+1,2]) # Save the entire trajecotry
```

```

param_traj[0,]=init # Also save the initial condition to the trajectory

v=0 # Starting with 0 momentum

# Epochs is borrowing term from machine learning
# Here it means timestep

for j in range(n_epochs):
    v=gamma*v+(np.array(grad(params))) # Compute v
    params=params-eta*v # Update the location
    param_traj[j+1,]=params # Save the trajectory
return param_traj

# initialize the optimizer
init_point = [-4.0, -2.0]
n_epochs = 100000

# define the eta for part c
eta_c = 1e-3

# run the optimizer for the momentum and ADAM with higher learning rate
mom_traj_c = gd_with_mom(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta_c)
adam_traj_c = adams(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta_c)

# compute the log 10 error
true_min = np.array([1, 1])
mom_error_c = np.log10(np.sum((mom_traj_c - true_min)**2, axis=1))
adam_error_c = np.log10(np.sum((adam_traj_c - true_min)**2, axis=1))

# plot
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# plot the contour with trajectories
axs[0].contour(X, Y, Z, levels=np.logspace(-1, 3.5, 20), cmap='viridis')
axs[0].plot(mom_traj_c[:, 0], mom_traj_c[:, 1], label='Momentum (=1e-3)')
axs[0].plot(adam_traj_c[:, 0], adam_traj_c[:, 1], label='Adam (=1e-3)')
axs[0].plot(1, 1, 'r*', markersize=10)
axs[0].set_title("Trajectories at = 1e-3")
axs[0].legend()

# plot the log error
axs[1].plot(mom_error_c, label='Momentum (=1e-3)')

```

```

axs[1].plot(adam_error_c, label='Adam ( =1e-3)')
axs[1].set_title("Log10 Error vs Time Step ( = 1e-3)")
axs[1].set_xlabel("Timestep")
axs[1].set_ylabel("log10(Error)")
axs[1].legend()

# define the eta for part d
eta_d = 1e-2

# Run with even higher learning rate
mom_traj_d = gd_with_mom(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta_d)
adam_traj_d = adams(prob_func_grad, init_point, n_epochs=n_epochs, eta=eta_d)

# calculate errors
mom_error_d = np.log10(np.sum((mom_traj_d - true_min)**2, axis=1))
adam_error_d = np.log10(np.sum((adam_traj_d - true_min)**2, axis=1))

# plot
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# plot the contour with trajectories
axs[0].contour(X, Y, Z, levels=np.logspace(-1, 3.5, 20), cmap='viridis')
axs[0].plot(mom_traj_d[:, 0], mom_traj_d[:, 1], label='Momentum ( =1e-2)')
axs[0].plot(adam_traj_d[:, 0], adam_traj_d[:, 1], label='Adam ( =1e-2)')
axs[0].plot(1, 1, 'r*', markersize=10)
axs[0].set_title("Trajectories at = 1e-2")
axs[0].legend()

# plot the log error
axs[1].plot(mom_error_d, label='Momentum ( =1e-2)')
axs[1].plot(adam_error_d, label='Adam ( =1e-2)')
axs[1].set_title("Log10 Error vs Time Step ( = 1e-2)")
axs[1].set_xlabel("Timestep")
axs[1].set_ylabel("log10(Error)")
axs[1].legend()

plt.tight_layout()
plt.show()

```

/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:12: RuntimeWarning



overflow encountered in scalar power

/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:13: RuntimeWarning

overflow encountered in scalar power

/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:12: RuntimeWarning

invalid value encountered in scalar subtract

/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:13: RuntimeWarning

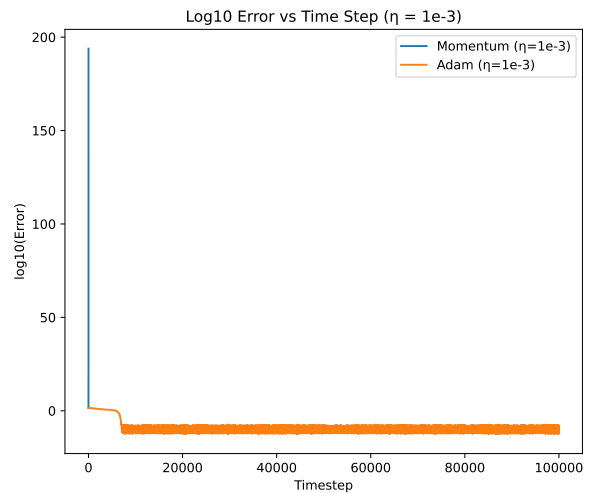
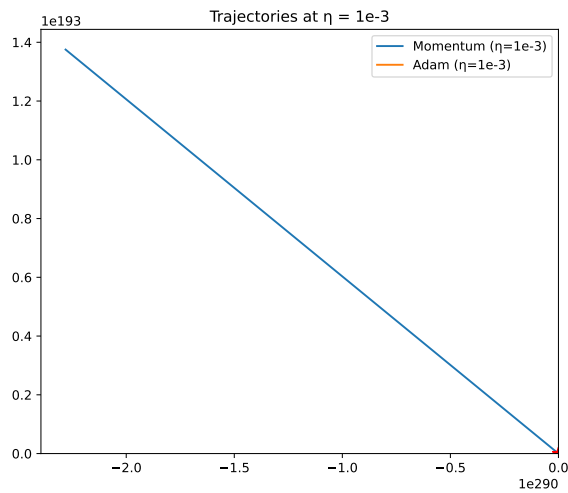
invalid value encountered in scalar subtract

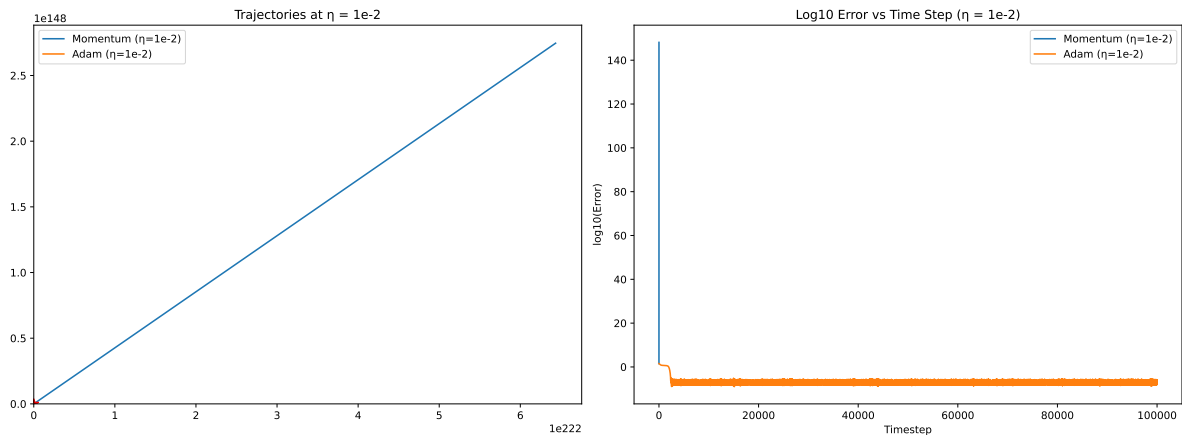
/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:62: RuntimeWarning

overflow encountered in square

/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel\_66181/3856364375.py:92: RuntimeWarning

overflow encountered in square





## Problem 2: Shallow Nets and MNIST

For this exercise, we will work on one of the standard model problems in Machine Learning, classifying handwritten digits. We will use an adaptation of the neural network code from your reading assignment to pytorch, which is one of the leading frameworks for training neural networks. pytorch is fairly flexible, you can use it with the CPU on your personal computer, with GPUs, and even on computing clusters.

If you have trouble getting pytorch to work on your own computer, I recommend trying in on google colab, or alternatively you are welcome to develop your own implementation. This and the next assignment have helper code in the provided ipython notebook Lab 7 Helper Notebook

First, you should acquire the MNIST dataset. This can be downloaded automatically using pytorch via the following code chunk:

```
# # Here is some code that automatically downloads the MNIST data. Technically it will also
# # data in if you have already downloaded and the path points to the folder where you have
# # There will be 4 binary files which together contain the testing and training examples and
# # for the testing and training examples.

# from torchvision import datasets, transforms

# # Load MNIST

# # transform defines a function which takes an image file, converts the analog bits into float
# transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(
```

```

# # The first line downloads the entire MNIST dataset to the data directory (or wherever you
# # If the data is already there, this won't download it. This downloads both the training and
# # the transform keyword applies the transform defined above, the train dataset has 60,000 examples
# # the test dataset has 10,000 examples. The train and test data is loaded in the variables train_dataset and test_dataset

# train_dataset = datasets.MNIST('data/', train=True, download=True, transform=transform)
# test_dataset = datasets.MNIST('data/', train=False, transform=transform)

```

We are going to train a simple neural network to classify the MNIST images. The neural network has an input layer of 784 neurons (one for each pixel), a 30 neuron hidden layer, and a 10 neuron output layer, which provided a weight that corresponds to the predictions of the neural network for each class. Initially we will use sigmoidal neurons to process the inputs.

Throughout the rest of the assignment, you will use and improve the code in this file to study the performance of different combinations of network structure, optimization algorithm choice, hyperparameters, and activation functions.

The initial configuration of the neural network uses stochastic gradient descent without momentum. The following code sets several key hyperparameters and trains the neural network:

```

# # batch_size determines the minibatch size
# import torch
# import torch.nn as nn
# import torch.nn.functional as F
# import torch.optim as optim
# from torch.utils.data import DataLoader

# train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
# test_loader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# # Initialize network
# # net = Network([784, 30, 10])

# # Train
# # sol = train(net, train_loader, epochs=30, eta=0.001, test_data=test_loader)

```

Additional parameters can be changed within the code of the function itself (and you can organize your code in any way you see fit), in the train function:

```

# optimizer = optim.SGD(network.parameters(),momentum=0.8,nesterov=True, lr=eta,weight_decay=0.0001)

# This uncomment this (and comment the above line) if you want to use ADAM. The betas are

```

```

# parameters, you can experiment with these hyperparameters if you like:
#optimizer = optim.Adam(network.parameters(),betas = (0.9,0.999), lr=eta,weight_decay=1e-4)

# Here is code for using learning rate scheduling. You might find this helpful
#step_size = 2
#gamma = 0.7
#scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

```

and also in the class function:

```

# class Network(nn.Module):
#     def __init__(self, sizes):
#         super(Network, self).__init__()
#         self.sizes = sizes
#         self.num_layers = len(sizes)

#         self.layers = nn.ModuleList()
#         for i in range(self.num_layers - 1):
#             layer = nn.Linear(sizes[i], sizes[i+1])
#             nn.init.xavier_normal_(layer.weight) # Good initialization for shallow/sigmoid
#             #nn.init.kaiming_normal_(layer.weight, mode='fan_out', nonlinearity='relu') in
#             #nn.init.kaiming_uniform_(layer.weight, mode='fan_out', nonlinearity='relu') in

#             nn.init.zeros_(layer.bias) # initialize the bias to 0
#             self.layers.append(layer)

#     # Forward is the method that calculates the value of the neural network. Basically we
#     # layer

#     def forward(self, x):
#         for layer in self.layers[:-1]:
#             x = F.sigmoid(layer(x)) # sigmoid layers
#             # x = F.relu(layer(x)) # You will try the relu layer in the last problem
#         x = self.layers[-1](x)
#         return x

```

The default set of hyperparameters lead to a neural network that successfully trains, and has an ultimate out of sample accuracy of just below 0.95 after 30 epochs of training on my computer.

Validate this result by executing the train command with the parameters described above. Next, the learning rate is one of the most important hyperparameters in machine learning.

Train the same structure of neural network with a range of different learning rates both higher and lower.

Make sure you find a learning rate high enough so that the neural network performance is poor. What was the learning rate that led to the best accuracy at the end of 30 epochs? Plot the test accuracy as a function of epochs (it is one of the outputs of the train function) for all of the learning rates that you tested.

How does the ADAM optimizer handle this problem? Modify the neural network code so that the optimizer is ADAM optimizer (you will see commented code in the train function). Then train a neural network using ADAM again with a range of different learning rates (including the same starting point as in part(a)). Compare the behavior of the learning curves, the final accuracy, and the values of the learning rates that were most successful with part (a).

How good can you make your 3-layer network? You don't need to do an exhaustive search of all possible options (which would take forever) but experiment with the optimization algorithm, the learning rate and the other hyper-parameters. For example, you could include more or fewer neurons in the hidden layer, change the values of the betas in ADAM or the momentum in SGD, alter the batch size in the data loader, change the learning rate, increase or decrease the weight decay (which is L2 regularization), or change the number of training epochs.

Some tips: - For this problem I don't think it is likely to have an accuracy above 0.99 without expanding the data - Larger batch sizes have less "noise" so might need more regularization. They also sometimes require larger learning rates - The learning rate scheduler will decrease the learning rate by a factor of  $\gamma$  every steps number of epochs

## Problem 2 Solution

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# run the already defined functions
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1))
])

train_dataset = datasets.MNIST('data/', train=True, download=True, transform=transform)
```

```

test_dataset = datasets.MNIST('data/', train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=10, shuffle=False)

class Network(nn.Module):
    def __init__(self, sizes, activation='sigmoid'):
        super(Network, self).__init__()
        self.activation = activation
        self.layers = nn.ModuleList()
        for i in range(len(sizes) - 1):
            layer = nn.Linear(sizes[i], sizes[i+1])
            nn.init.xavier_normal_(layer.weight) if activation == 'sigmoid' else nn.init.kaiming_normal_(layer.weight)
            nn.init.zeros_(layer.bias)
            self.layers.append(layer)

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = F.sigmoid(layer(x)) if self.activation == 'sigmoid' else F.relu(layer(x))
        return self.layers[-1](x)

# define the train function
def train(network, train_loader, test_loader, epochs=30, eta=0.001, optimizer_type='SGD'):
    criterion = nn.CrossEntropyLoss()
    if optimizer_type == 'SGD':
        optimizer = optim.SGD(network.parameters(), lr=eta, momentum=0.8, nesterov=True, weight_decay=0.0001)
    elif optimizer_type == 'Adam':
        optimizer = optim.Adam(network.parameters(), betas=(0.9, 0.999), lr=eta, weight_decay=0.0001)
    else:
        raise ValueError("Unsupported optimizer type.")

    test_acc_list = []

    for epoch in range(epochs):
        network.train()
        for images, labels in train_loader:
            outputs = network(images)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

```

```

        # Evaluate on test set
        network.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                outputs = network(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        acc = correct / total
        test_acc_list.append(acc)
        print(f"Epoch {epoch+1}/{epochs}, Accuracy: {acc:.4f}")

    return test_acc_list

# initiate the optimizers and run it
learning_rates = [0.0001, 0.001, 0.01, 0.05]
optimizers = ['SGD', 'Adam']
results = {}

for opt in optimizers:
    for lr in learning_rates:
        print(f"\nTraining with {opt}, Learning Rate = {lr}")
        model = Network([784, 30, 10], activation='sigmoid')
        acc = train(model, train_loader, test_loader, epochs=30, eta=lr, optimizer_type=opt)
        results[(opt, lr)] = acc

# plot
plt.figure(figsize=(12, 7))
for key, acc in results.items():
    label = f"{key[0]} (lr={key[1]})"
    plt.plot(acc, label=label)

plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.title("Test Accuracy vs Epochs for Different Optimizers and Learning Rates")
plt.legend()
plt.grid(True)
plt.show()

```

Training with SGD, Learning Rate = 0.0001

Epoch 1/30, Accuracy: 0.5658  
Epoch 2/30, Accuracy: 0.6448  
Epoch 3/30, Accuracy: 0.7038  
Epoch 4/30, Accuracy: 0.7441  
Epoch 5/30, Accuracy: 0.7703  
Epoch 6/30, Accuracy: 0.7880  
Epoch 7/30, Accuracy: 0.8079  
Epoch 8/30, Accuracy: 0.8162  
Epoch 9/30, Accuracy: 0.8280  
Epoch 10/30, Accuracy: 0.8375  
Epoch 11/30, Accuracy: 0.8422  
Epoch 12/30, Accuracy: 0.8487  
Epoch 13/30, Accuracy: 0.8553  
Epoch 14/30, Accuracy: 0.8602  
Epoch 15/30, Accuracy: 0.8653  
Epoch 16/30, Accuracy: 0.8688  
Epoch 17/30, Accuracy: 0.8716  
Epoch 18/30, Accuracy: 0.8739  
Epoch 19/30, Accuracy: 0.8765  
Epoch 20/30, Accuracy: 0.8787  
Epoch 21/30, Accuracy: 0.8818  
Epoch 22/30, Accuracy: 0.8846  
Epoch 23/30, Accuracy: 0.8858  
Epoch 24/30, Accuracy: 0.8884  
Epoch 25/30, Accuracy: 0.8898  
Epoch 26/30, Accuracy: 0.8911  
Epoch 27/30, Accuracy: 0.8920  
Epoch 28/30, Accuracy: 0.8938  
Epoch 29/30, Accuracy: 0.8948  
Epoch 30/30, Accuracy: 0.8965

Training with SGD, Learning Rate = 0.001

Epoch 1/30, Accuracy: 0.8258  
Epoch 2/30, Accuracy: 0.8739  
Epoch 3/30, Accuracy: 0.8919  
Epoch 4/30, Accuracy: 0.9014  
Epoch 5/30, Accuracy: 0.9071  
Epoch 6/30, Accuracy: 0.9117  
Epoch 7/30, Accuracy: 0.9145  
Epoch 8/30, Accuracy: 0.9181  
Epoch 9/30, Accuracy: 0.9200



Epoch 10/30, Accuracy: 0.9229  
Epoch 11/30, Accuracy: 0.9235  
Epoch 12/30, Accuracy: 0.9257  
Epoch 13/30, Accuracy: 0.9289  
Epoch 14/30, Accuracy: 0.9289  
Epoch 15/30, Accuracy: 0.9307  
Epoch 16/30, Accuracy: 0.9321  
Epoch 17/30, Accuracy: 0.9329  
Epoch 18/30, Accuracy: 0.9345  
Epoch 19/30, Accuracy: 0.9350  
Epoch 20/30, Accuracy: 0.9367  
Epoch 21/30, Accuracy: 0.9376  
Epoch 22/30, Accuracy: 0.9392  
Epoch 23/30, Accuracy: 0.9386  
Epoch 24/30, Accuracy: 0.9405  
Epoch 25/30, Accuracy: 0.9394  
Epoch 26/30, Accuracy: 0.9404  
Epoch 27/30, Accuracy: 0.9418  
Epoch 28/30, Accuracy: 0.9427  
Epoch 29/30, Accuracy: 0.9430  
Epoch 30/30, Accuracy: 0.9445

Training with SGD, Learning Rate = 0.01

Epoch 1/30, Accuracy: 0.9209  
Epoch 2/30, Accuracy: 0.9368  
Epoch 3/30, Accuracy: 0.9436  
Epoch 4/30, Accuracy: 0.9483  
Epoch 5/30, Accuracy: 0.9517  
Epoch 6/30, Accuracy: 0.9546  
Epoch 7/30, Accuracy: 0.9564  
Epoch 8/30, Accuracy: 0.9572  
Epoch 9/30, Accuracy: 0.9603  
Epoch 10/30, Accuracy: 0.9601  
Epoch 11/30, Accuracy: 0.9612  
Epoch 12/30, Accuracy: 0.9624  
Epoch 13/30, Accuracy: 0.9634  
Epoch 14/30, Accuracy: 0.9646  
Epoch 15/30, Accuracy: 0.9644  
Epoch 16/30, Accuracy: 0.9638  
Epoch 17/30, Accuracy: 0.9657  
Epoch 18/30, Accuracy: 0.9654  
Epoch 19/30, Accuracy: 0.9651  
Epoch 20/30, Accuracy: 0.9658

Epoch 21/30, Accuracy: 0.9645  
Epoch 22/30, Accuracy: 0.9652  
Epoch 23/30, Accuracy: 0.9642  
Epoch 24/30, Accuracy: 0.9653  
Epoch 25/30, Accuracy: 0.9664  
Epoch 26/30, Accuracy: 0.9652  
Epoch 27/30, Accuracy: 0.9665  
Epoch 28/30, Accuracy: 0.9639  
Epoch 29/30, Accuracy: 0.9672  
Epoch 30/30, Accuracy: 0.9667

Training with SGD, Learning Rate = 0.05

Epoch 1/30, Accuracy: 0.9447  
Epoch 2/30, Accuracy: 0.9536  
Epoch 3/30, Accuracy: 0.9594  
Epoch 4/30, Accuracy: 0.9608  
Epoch 5/30, Accuracy: 0.9601  
Epoch 6/30, Accuracy: 0.9635  
Epoch 7/30, Accuracy: 0.9628  
Epoch 8/30, Accuracy: 0.9623  
Epoch 9/30, Accuracy: 0.9635  
Epoch 10/30, Accuracy: 0.9645  
Epoch 11/30, Accuracy: 0.9649  
Epoch 12/30, Accuracy: 0.9623  
Epoch 13/30, Accuracy: 0.9659  
Epoch 14/30, Accuracy: 0.9638  
Epoch 15/30, Accuracy: 0.9651  
Epoch 16/30, Accuracy: 0.9631  
Epoch 17/30, Accuracy: 0.9634  
Epoch 18/30, Accuracy: 0.9666  
Epoch 19/30, Accuracy: 0.9654  
Epoch 20/30, Accuracy: 0.9666  
Epoch 21/30, Accuracy: 0.9626  
Epoch 22/30, Accuracy: 0.9656  
Epoch 23/30, Accuracy: 0.9619  
Epoch 24/30, Accuracy: 0.9652  
Epoch 25/30, Accuracy: 0.9669  
Epoch 26/30, Accuracy: 0.9599  
Epoch 27/30, Accuracy: 0.9655  
Epoch 28/30, Accuracy: 0.9647  
Epoch 29/30, Accuracy: 0.9621  
Epoch 30/30, Accuracy: 0.9662

Training with Adam, Learning Rate = 0.0001

Epoch 1/30, Accuracy: 0.8726  
Epoch 2/30, Accuracy: 0.9030  
Epoch 3/30, Accuracy: 0.9143  
Epoch 4/30, Accuracy: 0.9194  
Epoch 5/30, Accuracy: 0.9246  
Epoch 6/30, Accuracy: 0.9278  
Epoch 7/30, Accuracy: 0.9310  
Epoch 8/30, Accuracy: 0.9339  
Epoch 9/30, Accuracy: 0.9368  
Epoch 10/30, Accuracy: 0.9389  
Epoch 11/30, Accuracy: 0.9393  
Epoch 12/30, Accuracy: 0.9419  
Epoch 13/30, Accuracy: 0.9432  
Epoch 14/30, Accuracy: 0.9446  
Epoch 15/30, Accuracy: 0.9454  
Epoch 16/30, Accuracy: 0.9461  
Epoch 17/30, Accuracy: 0.9469  
Epoch 18/30, Accuracy: 0.9481  
Epoch 19/30, Accuracy: 0.9499  
Epoch 20/30, Accuracy: 0.9500  
Epoch 21/30, Accuracy: 0.9509  
Epoch 22/30, Accuracy: 0.9516  
Epoch 23/30, Accuracy: 0.9526  
Epoch 24/30, Accuracy: 0.9540  
Epoch 25/30, Accuracy: 0.9553  
Epoch 26/30, Accuracy: 0.9554  
Epoch 27/30, Accuracy: 0.9552  
Epoch 28/30, Accuracy: 0.9564  
Epoch 29/30, Accuracy: 0.9564  
Epoch 30/30, Accuracy: 0.9566

Training with Adam, Learning Rate = 0.001

Epoch 1/30, Accuracy: 0.9324  
Epoch 2/30, Accuracy: 0.9451  
Epoch 3/30, Accuracy: 0.9512  
Epoch 4/30, Accuracy: 0.9563  
Epoch 5/30, Accuracy: 0.9589  
Epoch 6/30, Accuracy: 0.9593  
Epoch 7/30, Accuracy: 0.9603  
Epoch 8/30, Accuracy: 0.9612  
Epoch 9/30, Accuracy: 0.9612  
Epoch 10/30, Accuracy: 0.9612

Epoch 11/30, Accuracy: 0.9629  
Epoch 12/30, Accuracy: 0.9635  
Epoch 13/30, Accuracy: 0.9629  
Epoch 14/30, Accuracy: 0.9632  
Epoch 15/30, Accuracy: 0.9635  
Epoch 16/30, Accuracy: 0.9650  
Epoch 17/30, Accuracy: 0.9634  
Epoch 18/30, Accuracy: 0.9632  
Epoch 19/30, Accuracy: 0.9647  
Epoch 20/30, Accuracy: 0.9623  
Epoch 21/30, Accuracy: 0.9649  
Epoch 22/30, Accuracy: 0.9644  
Epoch 23/30, Accuracy: 0.9626  
Epoch 24/30, Accuracy: 0.9646  
Epoch 25/30, Accuracy: 0.9628  
Epoch 26/30, Accuracy: 0.9631  
Epoch 27/30, Accuracy: 0.9652  
Epoch 28/30, Accuracy: 0.9663  
Epoch 29/30, Accuracy: 0.9647  
Epoch 30/30, Accuracy: 0.9624

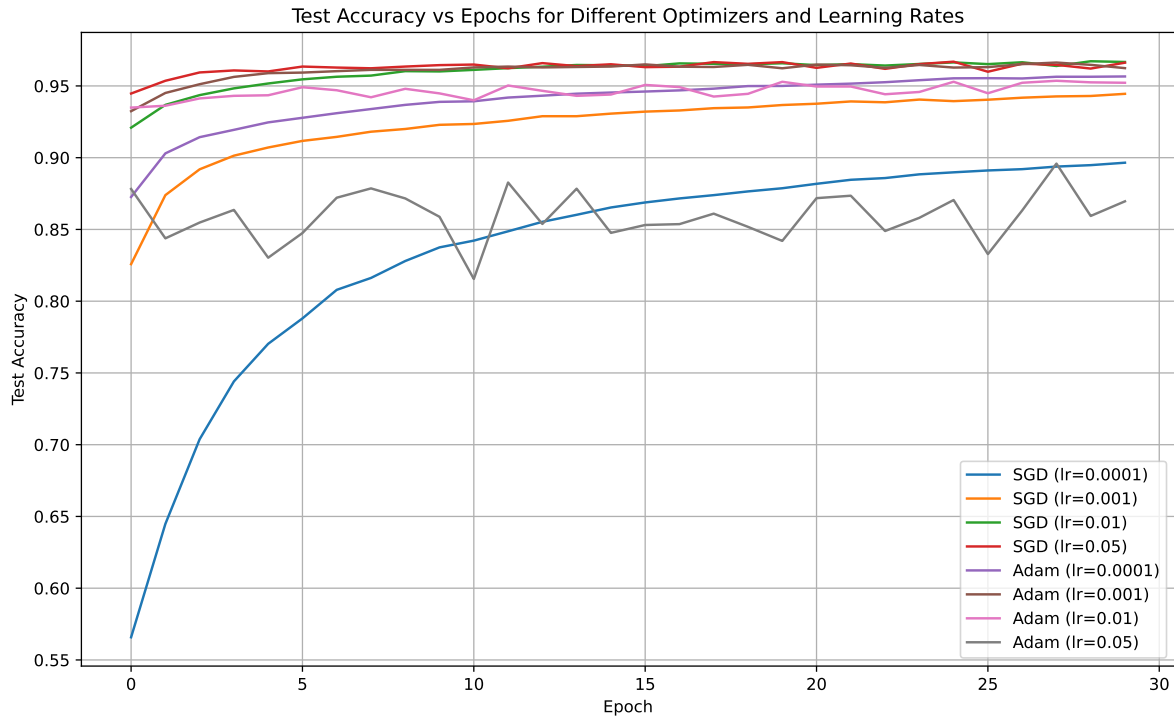
Training with Adam, Learning Rate = 0.01

Epoch 1/30, Accuracy: 0.9349  
Epoch 2/30, Accuracy: 0.9362  
Epoch 3/30, Accuracy: 0.9413  
Epoch 4/30, Accuracy: 0.9431  
Epoch 5/30, Accuracy: 0.9435  
Epoch 6/30, Accuracy: 0.9491  
Epoch 7/30, Accuracy: 0.9470  
Epoch 8/30, Accuracy: 0.9421  
Epoch 9/30, Accuracy: 0.9480  
Epoch 10/30, Accuracy: 0.9448  
Epoch 11/30, Accuracy: 0.9400  
Epoch 12/30, Accuracy: 0.9503  
Epoch 13/30, Accuracy: 0.9466  
Epoch 14/30, Accuracy: 0.9431  
Epoch 15/30, Accuracy: 0.9440  
Epoch 16/30, Accuracy: 0.9507  
Epoch 17/30, Accuracy: 0.9492  
Epoch 18/30, Accuracy: 0.9426  
Epoch 19/30, Accuracy: 0.9445  
Epoch 20/30, Accuracy: 0.9529  
Epoch 21/30, Accuracy: 0.9496

Epoch 22/30, Accuracy: 0.9496  
Epoch 23/30, Accuracy: 0.9442  
Epoch 24/30, Accuracy: 0.9458  
Epoch 25/30, Accuracy: 0.9529  
Epoch 26/30, Accuracy: 0.9449  
Epoch 27/30, Accuracy: 0.9522  
Epoch 28/30, Accuracy: 0.9536  
Epoch 29/30, Accuracy: 0.9525  
Epoch 30/30, Accuracy: 0.9522

Training with Adam, Learning Rate = 0.05

Epoch 1/30, Accuracy: 0.8782  
Epoch 2/30, Accuracy: 0.8438  
Epoch 3/30, Accuracy: 0.8548  
Epoch 4/30, Accuracy: 0.8636  
Epoch 5/30, Accuracy: 0.8303  
Epoch 6/30, Accuracy: 0.8475  
Epoch 7/30, Accuracy: 0.8721  
Epoch 8/30, Accuracy: 0.8786  
Epoch 9/30, Accuracy: 0.8716  
Epoch 10/30, Accuracy: 0.8588  
Epoch 11/30, Accuracy: 0.8155  
Epoch 12/30, Accuracy: 0.8827  
Epoch 13/30, Accuracy: 0.8538  
Epoch 14/30, Accuracy: 0.8784  
Epoch 15/30, Accuracy: 0.8476  
Epoch 16/30, Accuracy: 0.8531  
Epoch 17/30, Accuracy: 0.8537  
Epoch 18/30, Accuracy: 0.8610  
Epoch 19/30, Accuracy: 0.8518  
Epoch 20/30, Accuracy: 0.8420  
Epoch 21/30, Accuracy: 0.8718  
Epoch 22/30, Accuracy: 0.8735  
Epoch 23/30, Accuracy: 0.8489  
Epoch 24/30, Accuracy: 0.8581  
Epoch 25/30, Accuracy: 0.8705  
Epoch 26/30, Accuracy: 0.8328  
Epoch 27/30, Accuracy: 0.8632  
Epoch 28/30, Accuracy: 0.8959  
Epoch 29/30, Accuracy: 0.8594  
Epoch 30/30, Accuracy: 0.8696



### Problem 3: Deep Nets: Overcoming Gradients

In deep networks, the gradients for the neural network weights can either vanish or explode due to the compositional nature of the network. This typically happens for weights near the input layer. In order to observe vanishing gradients we need to calculate the size of the gradients of the loss function with respect to each weight.

The neural network training routine has commented code that computes the norm of the gradients of the input layer and the norm of the gradients of the output layer and divides and averages them across all the batches in an epoch. Uncomment the code (and modify the print and return statement) so that the gradients are computed and the gradient ratio output and saved. Then try training a deep neural network.

I recommend beginning with the architecture `net = Network([784,30,30,30,30,30,30,30,30,10])`. Train this neural network. You will almost certainly find that the training does not succeed. What are the gradient ratios that you observe during the training?

If your computer cannot train this network, you can try this problem with a shallower network, or my recommendation is to use only a single epoch of training which should still demonstrate the vanishing gradient problem.

There are several techniques to deal with vanishing (or exploding) gradients. These include using neurons with different activation functions or using different normalization schemes. The forward method in the Network class defines the activation functions. Change the activation function from sigmoid to relu and change the initialization from Xavier.normal\_ to kaiming.uniform\_ (which is more optimal for deep ReLU neurons) and train the deep neural network. How does the gradient ratio change? How does the test accuracy compare to the shallow net accuracy you achieved in problem 2?

Deep neural networks are a superior architecture of image classification problems than shallow networks, however typically the dense structure that we have implemented here is not used. Instead the neural networks usually have several convolutionary layers at the beginning. With some effort and experimentation, it should still be possible to achieve a very high accuracy with a dense neural network. Experiment with the network architecture and the hyperparameters and see how good you can make your deep/dense network.

You can try a combination of increasing the number of neurons in the hidden layers or increasing the depth of the network. A structure that often works is one which decreases the number of neurons per hidden layer steadily from the input layer to the final layer (i.e. starting at 784 and ending at 10). Can you improve upon the best shallow network that you constructed for problem 3?

```
# load dependencies
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# load dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(-1, 1, 1, 1))])
train_dataset = datasets.MNIST('data/', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('data/', train=False, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# use the applied function
class Network(nn.Module):
    def __init__(self, sizes, activation='sigmoid'):
        super(Network, self).__init__()
        self.activation = activation
        self.layers = nn.ModuleList()
        for i in range(len(sizes) - 1):
```

```

        layer = nn.Linear(sizes[i], sizes[i+1])
        nn.init.xavier_normal_(layer.weight) if activation == 'sigmoid' else nn.init.kaiming_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
        self.layers.append(layer)

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = F.sigmoid(layer(x)) if self.activation == 'sigmoid' else F.relu(layer(x))
        return self.layers[-1](x)

# Train function with gradient monitoring
def train(network, train_loader, test_loader, epochs=1, eta=0.001):
    optimizer = optim.SGD(network.parameters(), lr=eta)
    criterion = nn.CrossEntropyLoss()

    grad_ratios = []
    test_acc = []

    for epoch in range(epochs):
        network.train()
        input_grads, output_grads = 0.0, 0.0
        n_batches = 0

        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = network(images)
            loss = criterion(outputs, labels)
            loss.backward()

            # gradient norm
            input_grad_norm = network.layers[0].weight.grad.norm().item()
            output_grad_norm = network.layers[-1].weight.grad.norm().item()
            input_grads += input_grad_norm
            output_grads += output_grad_norm
            n_batches += 1

        optimizer.step()

        grad_ratio = (input_grads / n_batches) / (output_grads / n_batches + 1e-8)
        grad_ratios.append(grad_ratio)

    # review test data

```



```

network.eval()
correct, total = 0, 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = network(images)
        predicted = torch.argmax(outputs, dim=1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = correct / total
test_acc.append(accuracy)

print(f"Epoch {epoch+1}: Gradient Ratio = {grad_ratio:.6f}, Test Accuracy = {accuracy:.6f}")

return grad_ratios, test_acc

# define the deep learning network
net = Network([784,30,30,30,30,30,30,30,30,10])
ratios_sigmoid, acc_sigmoid = train(net, train_loader, test_loader, epochs=5)

# plot
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(ratios_sigmoid)
plt.title("Gradient Ratio (Input/Output) - Sigmoid")
plt.subplot(1, 2, 2)
plt.plot(acc_sigmoid)
plt.title("Test Accuracy - Sigmoid")
plt.show()

```

```

Epoch 1: Gradient Ratio = 0.000046, Test Accuracy = 0.1135
Epoch 2: Gradient Ratio = 0.000046, Test Accuracy = 0.1135
Epoch 3: Gradient Ratio = 0.000046, Test Accuracy = 0.1135
Epoch 4: Gradient Ratio = 0.000046, Test Accuracy = 0.1135
Epoch 5: Gradient Ratio = 0.000046, Test Accuracy = 0.1135

```

