CS/DS 541: Deep Learning

Homework 3

Tom Arnold & Nate Hindman

Due: 5:59pm ET Monday September 29

This problem can be done in teams of up 2 students.

Problem 1: Deriving He's initialization [10 points + 5 bonus pts]

In Lecture 7, we learned about a widely used technique for setting the initial values of the weight parameters in a neural network: the **He initialization**. Specifically, it samples each weight value from $\mathcal{N}(0, 2/n_l)$—i.e., a 0-mean Gaussian distribution with variance $2/n_l$ where n_l is the number of columns of the weight matrix $W^{[l]}$ layer l (or, equivalently the dimension of the inputs fed to layer l).

Here you are asked to derive some of the steps we skipped in class. We will start from:

$$Var[z^{[l]}] = Var[w^{[l]}x^{[l]}] = Var[\sum_{j=1}^{n_l} w_j^{[l]}x_j^{[l]}]$$

$$Var[\sum_{j=1}^{n_l} w_j^{[l]}x_j^{[l]}] = n_l Var[w_l x_l],$$

1. (2 points) Explain what are the assumptions that allow us to write:

$$Var[\sum_{j=1}^{n_l} w_j^{[l]}x_j^{[l]}] = n_l Var[w_l x_l],$$

where w_l represents any of the $w_j^{[l]}$ weights and x_l represents any of the $x_j^{[l]}$ inputs.

Answer

We are given that He initialization samples weights from $\mathcal{N}(0, 2/n_l)$ which means that each weight has expectation and variance:

$$E[w_j^{[l]}] = 0 \text{ and } \text{Var}[w_j^{[l]}] = 2/n_l \quad \text{(definition of } \mathcal{N}(\mu, \sigma^2))$$

$$\text{Var}\left[\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right] = E\left[\left(\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right)^2\right] - \left(E\left[\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right]\right)^2$$

$$\text{(Definition of Variance: } \text{Var}[Y] = E[Y^2] - (E[Y])^2)$$

Assumption 1: $w_j^{[l]}$ independent of all $x_k^{[l]}$

To move forward, we need to assume that the weights $w_j^{[l]}$ are independent of the inputs $x_j^{[l]}$. This assumption allows us to factor terms of the form $E[w_j^{[l]} x_j^{[l]}] = E[w_j^{[l]}] \cdot E[x_j^{[l]}]$. Because we know from initialization that $E[w_j^{[l]}] = 0$, the full product is 0 and drops out of the equation.

We are justified in this assumption because the random initialization of weights at layer l is done before the network ever sees data, so the particular input values flowing through the network cannot influence the weights. Without this assumption, we would have to deal with the full joint distribution of $(w_j^{[l]}, x_j^{[l]})$.

$$E\left[\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right] = \sum_{j=1}^{n_l} E[w_j^{[l]} x_j^{[l]}] \quad \text{(Linearity of Expectation)}$$

$$= \sum_{j=1}^{n_l} E[w_j^{[l]}] E[x_j^{[l]}] \quad \text{(Independence Theorem: } X \perp Y \Rightarrow E[XY] = E[X]E[Y])$$

$$= \sum_{j=1}^{n_l} 0 \cdot E[x_j^{[l]}] = 0$$

$$\text{Var}\left[\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right] = E\left[\left(\sum_{j=1}^{n_l} w_j^{[l]} x_j^{[l]}\right)^2\right] - 0^2$$

$$= E\left[\sum_{j=1}^{n_l} \sum_{k=1}^{n_l} w_j^{[l]} x_j^{[l]} w_k^{[l]} x_k^{[l]}\right] \quad \text{(Expanding } (a_1 + \cdots + a_n)^2 = \sum_j \sum_k a_j a_k)$$

$$= \sum_{j=1}^{n_l} \sum_{k=1}^{n_l} E[w_j^{[l]} x_j^{[l]} w_k^{[l]} x_k^{[l]}] \quad \text{(Linearity of Expectation)}$$

Assumption 2: $w_j^{[l]}$ independent of $w_k^{[l]}$ for $j \neq k$

When we expand the squared sum $(\sum_j w_j x_j)^2$, cross-terms of the form $E[w_j x_j w_k x_k]$ appear for $j \neq k$.

$$\text{Var}\left[\sum_{j=1}^{n_l} w_j x_j\right] = E\left[\begin{bmatrix} w_1 x_1 & \cdots & w_{n_l} x_{n_l} \end{bmatrix} \begin{bmatrix} w_1 x_1 \\ \vdots \\ w_{n_l} x_{n_l} \end{bmatrix}\right] = \begin{bmatrix} \text{Var}[w_1 x_1] & \cdots & E[w_j x_j w_k x_k] & \cdots \\ \vdots & & \ddots & \vdots \\ E[w_k x_k w_j x_j] & \cdots & \text{Var}[w_{n_l} x_{n_l}] & \end{bmatrix}$$

We can't move forward if these terms exist. Therefore we assume that the weights themselves are independent, meaning knowledge of $w_1^{[l]}$ tells us nothing about $w_2^{[l]}$, and so on. This independence causes these cross-terms to be 0, leaving only the diagonal contributions.

Thus, for $j \neq k$:

$$E[w_j^{[l]} x_j^{[l]} w_k^{[l]} x_k^{[l]}] = E[w_j^{[l]}] E[x_j^{[l]}] E[w_k^{[l]}] E[x_k^{[l]}] = 0 \cdot E[x_j^{[l]}] \cdot 0 \cdot E[x_k^{[l]}] = 0$$

(Independence of all four random variables)

$$= \sum_{j=1}^{n_l} E[(w_j^{[l]})^2 (x_j^{[l]})^2] \quad \text{(only diagonal terms } j = k \text{ survive)}$$

$$= \sum_{j=1}^{n_l} E[(w_j^{[l]} x_j^{[l]})^2]$$

$$= \sum_{j=1}^{n_l} \left(\text{Var}[w_j^{[l]} x_j^{[l]}] + (E[w_j^{[l]} x_j^{[l]}])^2\right) \quad \text{(Variance-Mean Relation: } E[Y^2] = \text{Var}[Y] + (E[Y])^2)$$

$$= \sum_{j=1}^{n_l} \text{Var}[w_j^{[l]} x_j^{[l]}] \quad \text{(since } E[w_j^{[l]} x_j^{[l]}] = E[w_j^{[l]}] E[x_j^{[l]}] = 0)$$

Assumption 3: All $w_j^{[l]}$ identically distributed, all $x_j^{[l]}$ identically distributed

To simplify further we must assume that all these weight–input pairs are identically distributed. This assumption is justified because under He initialization all weights are drawn from the same distribution, and the architecture of the network ensures that the inputs $x_j^{[l]}$ at a given layer have the same statistical properties. Without assuming identical distributions, each term in the summation could be different, forcing us to track them one by one rather than collapsing them as we do below.

$$= n_l \cdot \text{Var}[w^{[l]}x^{[l]}] \quad \text{(Identical Distribution: all terms equal)}$$

---

2. (4 points) Show that
$$Var[w_l x_l] = Var[w_l]\mathbb{E}[(x_l)^2]$$
using the following equality regarding the variance of the product of two mutually independent random variables A and B:
$$Var[AB] = Var[A]Var[B] + Var[A]\mathbb{E}[B]^2 + Var[B]\mathbb{E}[A]^2 \quad (0.0.1)$$
You will need to: (i) review and use the assumptions made about the distribution of w_l and (ii) use the relationship between variance and second moment for a random variable B:
$$Var[B] = \mathbb{E}[B^2] - \mathbb{E}[B]^2. \quad (0.0.2)$$

Answer

$$\text{Var}[w^{[l]}x^{[l]}] = E[(w^{[l]}x^{[l]})^2] - (E[w^{[l]}x^{[l]}])^2 \quad \text{(Definition of Variance)}$$

$$E[w^{[l]}x^{[l]}] = E[w^{[l]}] \cdot E[x^{[l]}] \quad \text{(Independence Theorem)}$$

$$= 0 \cdot E[x^{[l]}] = 0 \quad \text{(using } E[w^{[l]}] = 0 \text{ from He initialization)}$$

$$\text{Var}[w^{[l]}x^{[l]}] = E[(w^{[l]}x^{[l]})^2] - 0^2 = E[(w^{[l]})^2(x^{[l]})^2]$$

$$= E[(w^{[l]})^2] \cdot E[(x^{[l]})^2] \quad \text{(Independence Theorem for products)}$$

$$E[(w^{[l]})^2] = \text{Var}[w^{[l]}] + (E[w^{[l]}])^2 \quad \text{(Variance-Mean Relation)}$$

$$= \text{Var}[w^{[l]}] + 0^2 = \text{Var}[w^{[l]}]$$

$$\therefore \text{Var}[w^{[l]}x^{[l]}] = \text{Var}[w^{[l]}] \cdot E[(x^{[l]})^2]$$

---

3. Last, you need to relate the variance of the input x_l with the variance of the pre-activation value z_{l-1} when the activation function is **ReLU**, i.e. when $x_l = ReLU(z_{l-1}) = ReLU(w_{l-1}x_{l-1} + b_{l-1})$. Specifically, you need to prove the relationship:

$$Var[z_{l-1}] = 2 \times \mathbb{E}[x_l^2].$$

To do so, you will need to use these assumptions:

- w_{l-1} has a **symmetric distribution around 0**, i.e., the density $f_{w_{l-1}}(w) = f_{w_{l-1}}(-w)$.
- $b_{l-1} = 0$

As a guide, try to answer these questions in order:

(a) (2 points) What is the expected value of z_{l-1}? Does it depend on the value of x_{l-1}?

(b) (1 point) Think of x_{l-1} as a constant. What is the density $f_{z_{l-1}}(z)$!

(c) (1 point) Is the distribution of z_{l-1} symmetric around 0?

(d) (BONUS: 3 points) The variance of a continuous random variable X is given by $\int_{-\infty}^{\infty}(x - \mathbb{E}[X])^2 f_X(x)dx$. Write an expression for $Var[z_{l-1}]$. Remember what you found in item (a) about $\mathbb{E}[z_{l-1}]$ and try to use the symmetry of the distribution to write $Var[z_{l-1}]$ only in terms of the positive values of z_{l-1}.

(e) (BONUS: 2 points) Last, starting from the definition of the second moment $\mathbb{E}[x_l^2] = \int_{-\infty}^{\infty} x_l^2 f_{x_l}(x)d_x$, find ways to replace x_l by z_{l-1} conditioning on whether z_{l-1} is negative or positive. You should be able to find the expression you derived in item (d) for $Var[z_{l-1}]$.

Answer

(a) Expected value of z_{l-1}:

Given: $b_{l-1} = 0$

$$z_{l-1} = \sum_{k=1}^{n_{l-1}} w_k^{[l-1]} x_k^{[l-1]}$$

$$E[z_{l-1}] = E\left[\sum_{k=1}^{n_{l-1}} w_k^{[l-1]} x_k^{[l-1]}\right] = \sum_{k=1}^{n_{l-1}} E[w_k^{[l-1]} x_k^{[l-1]}] \quad \text{(Linearity of Expectation)}$$

$$= \sum_{k=1}^{n_{l-1}} E[w_k^{[l-1]}] \cdot E[x_k^{[l-1]}] \quad \text{(Independence Theorem)}$$

$$= \sum_{k=1}^{n_{l-1}} 0 \cdot E[x_k^{[l-1]}] = 0 \quad \text{(He initialization: } E[w_k^{[l-1]}] = 0\text{)}$$

(b) Density of z_{l-1} given x_{l-1}:

$$z_{l-1} = w_{l-1} \cdot x_{l-1} \quad \text{(treating } x_{l-1} \text{ as constant scalar)}$$

$$f_{z_{l-1}|x_{l-1}}(z) = \frac{1}{|x_{l-1}|} f_{w_{l-1}}\left(\frac{z}{x_{l-1}}\right)$$

(Change of Variables Formula: if $Y = aX$, then $f_Y(y) = \frac{1}{|a|} f_X(y/a)$)

(c) Symmetry:

Given: $w_{l-1} \sim \mathcal{N}(0, \sigma^2)$

$$\Rightarrow f_{w_{l-1}}(w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-w^2/(2\sigma^2)} = f_{w_{l-1}}(-w)$$

(Gaussian PDF is symmetric about its mean)

$$f_{z_{l-1}}(z) = f_{z_{l-1}}(-z) \quad \text{(Linear transformation preserves symmetry)}$$

(d) Variance calculation:

$$\text{Var}[z_{l-1}] = E[z_{l-1}^2] - (E[z_{l-1}])^2 = E[z_{l-1}^2] - 0 \quad \text{(Definition of Variance)}$$

$$E[z_{l-1}^2] = \int_{-\infty}^{\infty} z^2 f_{z_{l-1}}(z)dz \quad \text{(Definition of Expectation for continuous RV)}$$

$$= \int_{-\infty}^{0} z^2 f_{z_{l-1}}(z)dz + \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz \quad \text{(Additivity of Integrals)}$$

$$= \int_{0}^{\infty} u^2 f_{z_{l-1}}(-u)du + \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz$$

(U-Substitution Rule for Integrals: Let $u = -z$ in first integral, then $du = -dz$)

$$= \int_{0}^{\infty} u^2 f_{z_{l-1}}(u)du + \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz$$

(Using symmetry: $f_{z_{l-1}}(-u) = f_{z_{l-1}}(u)$)

$$= 2\int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz \quad \text{(Both integrals are identical)}$$

(e) Connecting to $E[x_l^2]$:

$$x_l = \text{ReLU}(z_{l-1}) = \begin{cases} z_{l-1} & \text{if } z_{l-1} > 0 \\ 0 & \text{if } z_{l-1} \leq 0 \end{cases} \quad \text{(Definition of ReLU)}$$

$$E[x_l^2] = \int_{-\infty}^{\infty} [\text{ReLU}(z)]^2 f_{z_{l-1}}(z)dz \quad \text{(Definition of Expectation)}$$

$$= \int_{-\infty}^{0} 0^2 \cdot f_{z_{l-1}}(z)dz + \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz$$

($\text{ReLU}(z) = 0$ for $z \leq 0$, $\text{ReLU}(z) = z$ for $z > 0$)

$$= 0 + \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz = \int_{0}^{\infty} z^2 f_{z_{l-1}}(z)dz$$

From part (d): $\text{Var}[z_{l-1}] = 2\int_0^\infty z^2 f_{z_{l-1}}(z)dz$

$$\therefore \text{Var}[z_{l-1}] = 2 \cdot E[x_l^2]$$

Problem 2: Training NNs with pytorch [15 points + 2 bonus pts]

In this problem you will use pytorch to train a multi-layer neural network to classify images of fashion items (10 different classes) from the **Fashion MNIST** dataset. Similarly to Homework 2, the input to the network will be a 28×28 pixel image; the output will be a real number.

Specifically, the starting network you will create should implement a function $f : \mathbb{R}^{784} \to \mathbb{R}^{10}$, where:

$$\begin{aligned} z^{(1)} &= W^{(1)}x + b^{(1)} \\ h^{(1)} &= ReLU(z^{(1)}) \\ z^{(2)} &= W^{(2)}h^{(1)} + b^{(2)} \\ &\vdots \\ z^{(l)} &= W^{(l)}h^{(l-1)} + b^{(l)} \\ \hat{y} &= softmax(z^{(l)}) \end{aligned}$$

The network specified above is shown in the figure below:

As usual, the (unregularized) cross-entropy cost function should be:

$$f_{CE}(W^{(1)}, b^{(1)}, ..., W^{(l)}, b^{(l)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{10} y_k^{(i)} log\hat{y}_k^{(i)}$$

where n is the number of examples.

The Fashion MNIST dataset can be obtained from the following web links: https://s3.amazonaws.com/jrwprojects/fashion_mnist_train_images.npy https://s3.amazonaws.com/jrwprojects/fashion_mnist_train_labels.npy https://s3.amazonaws.com/jrwprojects/fashion_mnist_test_images.npy https://s3.amazonaws.com/jrwprojects/fashion_mnist_test_labels.npy

1. (6 points) Implement the same network using PyTorch or Tensorflow.

Answer

```
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, random_split
import matplotlib.pyplot as plt
import tqdm


#####################################################################################

print("sees CUDA device:", torch.cuda.is_available())
print("CUDA Version:", torch.version.cuda)
if torch.cuda.is_available():
    print("GPU in Use:", torch.cuda.get_device_name(0))

#####################################################################################

def build_mlp(input_dim, n_layers, hidden_units, output_dim):
    """Build an MLP with `n_layers` hidden linear+ReLU layers and a linear
     ↪  output.

    Returns
    -------
    nn.Module
        The constructed MLP.
    """
    layers = []
    # BEGIN YOUR CODE HERE (~5-6 lines)

    for _ in range(n_layers):
        layers += [nn.Linear(input_dim,hidden_units), nn.ReLU()]
        input_dim = hidden_units

    #Y_HAT = nn.Softmax(layers[-1]) I think I need to do this somewhere?
     ↪  like softmax was in the ssignment
    layers += [nn.Linear(input_dim, output_dim)]


    # END YOUR CODE HERE
```

```
    return nn.Sequential(*layers)

def extract_model_params(model):
    """Extract all parameters of a PyTorch model."""
    return torch.cat([p.view(-1) for p in model.parameters() if
     ↪  p.requires_grad])

def load_params_into_model(model, all_params):
    """Loads a flattened array of parameters back into a PyTorch model."""
    current_pos = 0
    for param in model.parameters():
        if param.requires_grad:
            num_params = param.numel()
            # Reshape the flattened parameters to the original shape of the
             ↪  parameter
            param.data.copy_(all_params[current_pos : current_pos +
 ↪  num_params].view(param.size()))
            current_pos += num_params

def train_epoch(model, loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    for X, y in loader:
        # BEGIN YOUR CODE HERE (~5-7 lines)


        optimizer.zero_grad()
        logits = model(X)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()



        # END YOUR CODE HERE
        running_loss += loss.item() * X.size(0)

    # Extract and store the model parameters after the epoch
    all_params = extract_model_params(model)

    return running_loss / len(loader.dataset),
     ↪  all_params.detach().cpu().numpy()
```

```
def evaluate(model, loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for X, y in loader:
            # BEGIN YOUR CODE HERE (~4 lines)

            logits = model(X)
            preds = torch.argmax(logits, dim=1)
            correct += (preds == y).sum().item()
            total += y.size(0)



            # END YOUR CODE HERE

    return correct / total


def compute_loss(model, loader, criterion):
    running_loss = 0.0
    for X, y in loader:
        # BEGIN YOUR CODE HERE (~3 lines)
        with torch.no_grad():
            logits = model(X)
            loss = criterion(logits, y)
            running_loss += loss.item() * X.size(0)

        # END YOUR CODE HERE
    return running_loss / len(loader.dataset)

# Load data (numpy arrays assumed present in workspace)
X_train = np.load("fashion_mnist_train_images.npy").astype(np.float32) /
 ↪  255.0
y_train = np.load("fashion_mnist_train_labels.npy").astype(np.int64)
X_test = np.load("fashion_mnist_test_images.npy").astype(np.float32) / 255.0
y_test = np.load("fashion_mnist_test_labels.npy").astype(np.int64)

# Flatten if images are HxW
```

```
if X_train.ndim == 3:
    X_train = X_train.reshape(X_train.shape[0], -1)
    X_test = X_test.reshape(X_test.shape[0], -1)

# center data similarly to previous versions
X_train = X_train - 0.5
X_test = X_test - 0.5

# Convert to tensors
X_train_t = torch.from_numpy(X_train)
y_train_t = torch.from_numpy(y_train)
X_test_t = torch.from_numpy(X_test)
y_test_t = torch.from_numpy(y_test)

# SEED FIX
torch.manual_seed(541)
np.random.seed(541)

# Train/val split
full_train = TensorDataset(X_train_t, y_train_t)
val_size = int(0.2 * len(full_train))
train_size = len(full_train) - val_size
train_dataset, val_dataset = random_split(full_train, [train_size, val_size])
```

---

2. (5 points) Implement a method called **hyperparam_tuning**. Optimize the hyperparameters by training on the training set and selecting the parameter settings that optimize performance on the validation set. You should systematically (i.e., in code) try at least **10** (in total, not for each hyperparameter) different hyperparameter settings; **Hyperparameter tuning:** In this problem, there are several different hyperparameters that will impact the network's performance:

   - (Required) **Number of hidden layers** (suggestions: $\{3, 4, 5\}$)
   - **Number of units in each hidden layer** (suggestions: $\{30, 40, 50\}$)
   - **Learning rate** (suggestions: $\{0.001, 0.005, 0.01, 0.05, 0.1, 0.5\}$)
   - **Minibatch size** (suggestions: $\{16, 32, 64, 128, 256\}$)
   - **Number of epochs**
   - L_2 **Regularization strength** applied to the weight matrices (but not bias terms)

   In order not to "cheat" and thus overestimate the performance of the network it is crucial to optimize the hyperparameters **only on the validation set**; do **not** use the test set. (The training set would be ok, but typically leads to worse

performance.) Hence, just like in Homework 2, you should fork off part of the training set into a validation portion.

Answer

```
else:
        biases.append(p)
#| #| label: prob2_2

def hyperparam_tuning(train_dataset, val_dataset, seed=541):
    """Systematically search hyperparameters and return the best config and
     ↪  model state.

    Returns
    -------
    dict, state_dict
        Best hyperparameters and the corresponding model state dict.
    """
    torch.manual_seed(seed)
    np.random.seed(seed)

    best_cfg = None
    best_acc = 0.0
    # BEGIN YOUR CODE HERE (~15-20 lines)


    batch_sizes = [8,12,16,20,32,48,64,128,145,256]
    learning_rates= [0.01,0.005,0.001,0.0005,0.0001,0.00005,0.00001]
    layers = [2,3,4,5,6]
    hidden = [5,10,20,30,40]
    alphas = [0,0.0005,0.001,0.0001,0.00001]
    epochs = [15, 20, 25]


    ########################################
    x0, y0 = train_dataset[0]
    n_features = x0.numel()
    n_classes  = 10
```

```
    ##############################################3


    loopcount = 0

    config_attempts = 50



    for _ in range(config_attempts):


        layer_choice =  int(np.random.choice(layers).item())
        batch_size_choice = int(np.random.choice(batch_sizes).item())
        learning_rate_choice =float(np.random.choice(learning_rates).item())
        hidden_choice = int(np.random.choice(hidden).item())
        aplha_choice = float(np.random.choice(alphas).item())
        epoch_choices = int(np.random.choice(epochs).item())

        train_loader = DataLoader(train_dataset,
↪   batch_size=batch_size_choice, shuffle=True)
        val_loader   = DataLoader(val_dataset,
↪   batch_size=batch_size_choice, shuffle=False)

        # SEED FIX
        torch.manual_seed(seed + loopcount)
        model = build_mlp(n_features, layer_choice,hidden_choice, n_classes)
        criterion = nn.CrossEntropyLoss()


         ↪  ##########################################################################
         ↪


        weights =[]
        biases = []

        for name, p in model.named_parameters():

            if "weight" in name:
                weights.append(p)
            else:
```

```
                biases.append(p)

        optimizer = torch.optim.SGD([{'params': weights, 'weight_decay':
↪  aplha_choice},{'params': biases,   'weight_decay':
↪  0.0}],lr=learning_rate_choice)




        ↪  ##############################################################################

        for _e in range(epoch_choices):
            train_epoch(model, train_loader, criterion, optimizer)

        current_accuracy = evaluate(model, val_loader)

        # sanity check to see what loop im on and to ensure its not like
        ↪  frozen
        loopcount = loopcount +1
        #print("on loop ", loopcount)# turned off for final output

        if current_accuracy > best_acc:
            best_acc = current_accuracy
            print("best accuracy ",best_acc)
            #best_configs = [layer_choice , batch_size_choice ,
            ↪  learning_rate_choice, hidden_choice , aplha_choice ,
            ↪  epoch_choices]

            best_cfg = {
                    "layers": layer_choice,
                    "hidden": hidden_choice,
                    "batch": batch_size_choice,
                    "learning_rate": learning_rate_choice,
                    "alpha": aplha_choice,
                    "epochs": epoch_choices,
                    "seed": seed + loopcount - 1  # SEED FIX
                }

            print("New best:", best_cfg, "val_acc=", f"{best_acc:.4f}")
```

```
    # END YOUR CODE HERE

    return best_cfg, best_acc


# Run hyperparameter tuning
best_cfg, best_acc = hyperparam_tuning(train_dataset, val_dataset)
print(f"Best config: {best_cfg} with val_acc={best_acc:.4f}")
```

```
best accuracy  0.2663333333333333
New best: {'layers': 2, 'hidden': 5, 'batch': 20, 'learning_rate': 0.0001,
'alpha': 1e-05, 'epochs': 20, 'seed': 541} val_acc= 0.2663
best accuracy  0.2911666666666667
New best: {'layers': 6, 'hidden': 5, 'batch': 20, 'learning_rate': 0.001,
'alpha': 0.0005, 'epochs': 25, 'seed': 544} val_acc= 0.2912
best accuracy  0.85925
New best: {'layers': 4, 'hidden': 40, 'batch': 8, 'learning_rate': 0.001,
'alpha': 0.0001, 'epochs': 25, 'seed': 548} val_acc= 0.8592
best accuracy  0.8629166666666667
New best: {'layers': 4, 'hidden': 20, 'batch': 16, 'learning_rate': 0.005,
'alpha': 0.0005, 'epochs': 25, 'seed': 564} val_acc= 0.8629
best accuracy  0.87725
New best: {'layers': 3, 'hidden': 40, 'batch': 16, 'learning_rate': 0.01,
'alpha': 0.001, 'epochs': 25, 'seed': 576} val_acc= 0.8772
Best config: {'layers': 3, 'hidden': 40, 'batch': 16, 'learning_rate': 0.01,
'alpha': 0.001, 'epochs': 25, 'seed': 576} with val_acc=0.8772
```

---

3. (4 points) After you have optimized your hyperparameters, then re-train your network again and show a screenshot displaying the training loss evolving over the last **20 iterations** of SGD. (You can pick the last 20 mini-batches or last 20 epochs; it's up to you). In addition, make sure your screenshot also displays the **test accuracy** of the final trained classifier. The accuracy (percentage correctly classified test images) should be **at least 87%**.

Answer

```
# Retrain best model on full training set (train+val) for more epochs
n_features = X_train.shape[1]
n_classes = int(y_train.max() + 1)
# Instantiate the best model
# BEGIN YOUR CODE HERE (~3 lines)


torch.manual_seed(best_cfg['seed']) # SEED FIX
best_model = build_mlp(
    n_features,
    best_cfg['layers'],
    best_cfg['hidden'],
    n_classes
)



# END YOUR CODE HERE

# create full training loader and test loader
batch_size = best_cfg['batch']
full_train_loader = DataLoader(full_train, batch_size=batch_size,
 ↪  shuffle=True)
test_loader = DataLoader(TensorDataset(X_test_t, y_test_t),
 ↪  batch_size=batch_size, shuffle=False)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(best_model.parameters(),
                            lr=best_cfg['learning_rate'],
                            weight_decay=best_cfg['alpha'])
# retrain best model
epochs_final = 50
parameter_history = []
for epoch in range(epochs_final):
    # BEGIN YOUR CODE HERE (~2 lines)

    t0 = __import__('time').time()
    loss, flat_params = train_epoch(best_model, full_train_loader, criterion,
 ↪  optimizer)
```

```
    parameter_history.append(flat_params)
    test_acc = evaluate(best_model, test_loader)
    t1 = __import__('time').time()


    # END YOUR CODE HERE
    print(f"Final Train Epoch {epoch+1}/{epochs_final}: loss={loss:.4f},
     ↪  test_acc={test_acc:.4f}, time={t1-t0:.1f}s")

# Print the length of the parameter_history list
print(f"Length of parameter history: {len(parameter_history)}")
print(f"Final Test Accuracy: {test_acc:.4f}")
```

```
Final Train Epoch 1/50: loss=1.0717, test_acc=0.7680, time=4.7s
Final Train Epoch 2/50: loss=0.5385, test_acc=0.8125, time=4.8s
Final Train Epoch 3/50: loss=0.4607, test_acc=0.8287, time=4.8s
Final Train Epoch 4/50: loss=0.4252, test_acc=0.8376, time=4.8s
Final Train Epoch 5/50: loss=0.4013, test_acc=0.8472, time=4.8s
Final Train Epoch 6/50: loss=0.3842, test_acc=0.8484, time=4.8s
Final Train Epoch 7/50: loss=0.3693, test_acc=0.8541, time=4.7s
Final Train Epoch 8/50: loss=0.3564, test_acc=0.8619, time=4.7s
Final Train Epoch 9/50: loss=0.3480, test_acc=0.8578, time=4.7s
Final Train Epoch 10/50: loss=0.3390, test_acc=0.8604, time=4.7s
Final Train Epoch 11/50: loss=0.3320, test_acc=0.8555, time=4.7s
Final Train Epoch 12/50: loss=0.3253, test_acc=0.8678, time=4.7s
Final Train Epoch 13/50: loss=0.3207, test_acc=0.8588, time=4.7s
Final Train Epoch 14/50: loss=0.3159, test_acc=0.8644, time=4.6s
Final Train Epoch 15/50: loss=0.3112, test_acc=0.8685, time=4.8s
Final Train Epoch 16/50: loss=0.3073, test_acc=0.8704, time=4.7s
Final Train Epoch 17/50: loss=0.3050, test_acc=0.8697, time=4.7s
Final Train Epoch 18/50: loss=0.3017, test_acc=0.8724, time=4.7s
Final Train Epoch 19/50: loss=0.2978, test_acc=0.8691, time=4.7s
Final Train Epoch 20/50: loss=0.2945, test_acc=0.8611, time=4.7s
Final Train Epoch 21/50: loss=0.2922, test_acc=0.8697, time=4.7s
Final Train Epoch 22/50: loss=0.2905, test_acc=0.8685, time=4.8s
Final Train Epoch 23/50: loss=0.2876, test_acc=0.8646, time=4.7s
Final Train Epoch 24/50: loss=0.2843, test_acc=0.8659, time=4.7s
Final Train Epoch 25/50: loss=0.2840, test_acc=0.8721, time=4.7s
Final Train Epoch 26/50: loss=0.2806, test_acc=0.8657, time=4.7s
Final Train Epoch 27/50: loss=0.2793, test_acc=0.8697, time=4.7s
Final Train Epoch 28/50: loss=0.2769, test_acc=0.8676, time=4.8s
```

```
Final Train Epoch 29/50: loss=0.2758, test_acc=0.8686, time=4.7s
Final Train Epoch 30/50: loss=0.2732, test_acc=0.8696, time=4.7s
Final Train Epoch 31/50: loss=0.2720, test_acc=0.8731, time=4.8s
Final Train Epoch 32/50: loss=0.2709, test_acc=0.8744, time=4.7s
Final Train Epoch 33/50: loss=0.2692, test_acc=0.8774, time=4.8s
Final Train Epoch 34/50: loss=0.2670, test_acc=0.8681, time=4.8s
Final Train Epoch 35/50: loss=0.2660, test_acc=0.8691, time=4.8s
Final Train Epoch 36/50: loss=0.2650, test_acc=0.8712, time=4.7s
Final Train Epoch 37/50: loss=0.2637, test_acc=0.8752, time=4.7s
Final Train Epoch 38/50: loss=0.2608, test_acc=0.8677, time=4.7s
Final Train Epoch 39/50: loss=0.2597, test_acc=0.8696, time=4.8s
Final Train Epoch 40/50: loss=0.2588, test_acc=0.8637, time=4.7s
Final Train Epoch 41/50: loss=0.2592, test_acc=0.8771, time=4.8s
Final Train Epoch 42/50: loss=0.2562, test_acc=0.8731, time=4.7s
Final Train Epoch 43/50: loss=0.2552, test_acc=0.8673, time=4.7s
Final Train Epoch 44/50: loss=0.2545, test_acc=0.8758, time=4.7s
Final Train Epoch 45/50: loss=0.2531, test_acc=0.8745, time=4.7s
Final Train Epoch 46/50: loss=0.2518, test_acc=0.8773, time=4.7s
Final Train Epoch 47/50: loss=0.2528, test_acc=0.8745, time=4.8s
Final Train Epoch 48/50: loss=0.2503, test_acc=0.8786, time=4.8s
Final Train Epoch 49/50: loss=0.2505, test_acc=0.8831, time=4.7s
Final Train Epoch 50/50: loss=0.2494, test_acc=0.8771, time=4.7s
Length of parameter history: 50
Final Test Accuracy: 0.8771
```

---

4. (2 bonus points) Now you have to experiment with a variant of the previous network that includes either **BatchNorm, Dropout layers** or **different activation functions** (e.g., LeakyReLU, ELU or PReLU). Find an architecture that outperforms the one you found in item 3 and report the test accuracy.

Answer

```python
def build_enhanced_mlp(input_dim, n_layers, hidden_units, output_dim,
                       use_batchnorm=False, dropout_rate=0.0,
 ↪  activation='relu'):
```

```
    """Build enhanced MLP with optional BatchNorm, Dropout, and different
     ↪  activations."""
    layers = []

    activations = {
        'relu': nn.ReLU(),
        'leaky': nn.LeakyReLU(0.01),
        'elu': nn.ELU(),
        'prelu': nn.PReLU()
    }
    act_fn = activations[activation]

    for i in range(n_layers):
        layers.append(nn.Linear(input_dim, hidden_units))

        if use_batchnorm:
            layers.append(nn.BatchNorm1d(hidden_units))

        layers.append(act_fn if activation != 'prelu' else nn.PReLU())

        if dropout_rate > 0:
            layers.append(nn.Dropout(dropout_rate))

        input_dim = hidden_units

    layers.append(nn.Linear(input_dim, output_dim))

    return nn.Sequential(*layers)


def enhanced_hyperparam_tuning(train_dataset, val_dataset, seed=541):
    """Find best enhanced architecture config."""
    torch.manual_seed(seed)
    np.random.seed(seed)

    best_cfg = None
    best_acc = 0.0

    batch_sizes = [32, 64, 128]
    learning_rates = [0.001, 0.0005, 0.0001]
    layers = [3, 4, 5]
    hidden = [64, 128, 256]
```

```
    alphas = [0, 0.0001, 0.00001]
    epochs = [30, 40]

    use_batchnorm_options = [True, False]
    dropout_rates = [0.0, 0.2, 0.3, 0.5]
    activations = ['relu', 'leaky', 'elu', 'prelu']

    x0, y0 = train_dataset[0]
    n_features = x0.numel()
    n_classes = 10

    loopcount = 0
    config_attempts = 15

    for _ in range(config_attempts):

        layer_choice = int(np.random.choice(layers))
        batch_size_choice = int(np.random.choice(batch_sizes))
        learning_rate_choice = float(np.random.choice(learning_rates))
        hidden_choice = int(np.random.choice(hidden))
        alpha_choice = float(np.random.choice(alphas))
        epoch_choice = int(np.random.choice(epochs))
        use_bn = bool(np.random.choice(use_batchnorm_options))
        dropout = float(np.random.choice(dropout_rates))
        activation = np.random.choice(activations)

        train_loader = DataLoader(train_dataset,
↪   batch_size=batch_size_choice, shuffle=True)
        val_loader = DataLoader(val_dataset, batch_size=batch_size_choice,
↪   shuffle=False)

        torch.manual_seed(seed + loopcount) # SEED FIX
        model = build_enhanced_mlp(n_features, layer_choice, hidden_choice,
↪   n_classes,
                                   use_batchnorm=use_bn, dropout_rate=dropout,
↪   activation=activation)

        weights = []
        biases = []

        for name, p in model.named_parameters():
```

```
            if "weight" in name:
                weights.append(p)
            else:
                biases.append(p)

        optimizer = torch.optim.SGD([
            {'params': weights, 'weight_decay': alpha_choice},
            {'params': biases, 'weight_decay': 0.0}
        ], lr=learning_rate_choice)

        criterion = nn.CrossEntropyLoss()

        for _e in range(epoch_choice):
            train_epoch(model, train_loader, criterion, optimizer)

        current_accuracy = evaluate(model, val_loader)

        loopcount = loopcount + 1
        #print("on enhanced loop ", loopcount)

        if current_accuracy > best_acc:
            best_acc = current_accuracy
            print("best enhanced accuracy ", best_acc)

            best_cfg = {
                "layers": layer_choice,
                "hidden": hidden_choice,
                "batch": batch_size_choice,
                "learning_rate": learning_rate_choice,
                "alpha": alpha_choice,
                "epochs": epoch_choice,
                "batchnorm": use_bn,
                "dropout": dropout,
                "activation": activation,
                "seed": seed + loopcount - 1  # SEED FIX
            }

            print("New best enhanced:", best_cfg, "val_acc=",
             ↪  f"{best_acc:.4f}")

    return best_cfg, best_acc
```

```
enhanced_cfg, enhanced_acc = enhanced_hyperparam_tuning(train_dataset,
 ↪  val_dataset)
print(f"Enhanced config: {enhanced_cfg} with val_acc={enhanced_acc:.4f}")

# Train final enhanced model
torch.manual_seed(enhanced_cfg['seed']) # SEED FIX
enhanced_model = build_enhanced_mlp(
    n_features,
    enhanced_cfg['layers'],
    enhanced_cfg['hidden'],
    n_classes,
    use_batchnorm=enhanced_cfg['batchnorm'],
    dropout_rate=enhanced_cfg['dropout'],
    activation=enhanced_cfg['activation']
)

batch_size = enhanced_cfg['batch']
full_train_loader = DataLoader(full_train, batch_size=batch_size,
 ↪  shuffle=True)
test_loader = DataLoader(TensorDataset(X_test_t, y_test_t),
 ↪  batch_size=batch_size, shuffle=False)
criterion = nn.CrossEntropyLoss()

weights = []
biases = []
for name, p in enhanced_model.named_parameters():
    if "weight" in name:
        weights.append(p)
     else:
         biases.append(p)

optimizer = torch.optim.SGD([
    {'params': weights, 'weight_decay': enhanced_cfg['alpha']},
    {'params': biases, 'weight_decay': 0.0}
], lr=enhanced_cfg['learning_rate'])

# retrain enhanced model
epochs_final = 50
for epoch in range(epochs_final):
    t0 = __import__('time').time()
    loss, _ = train_epoch(enhanced_model, full_train_loader, criterion,
 ↪  optimizer)
```

```
    test_acc = evaluate(enhanced_model, test_loader)
    t1 = __import__('time').time()
    print(f"Enhanced Train Epoch {epoch+1}/{epochs_final}: loss={loss:.4f},
     ↪  test_acc={test_acc:.4f}, time={t1-t0:.1f}s")

print(f"Final Enhanced Test Accuracy: {test_acc:.4f}")
```

```
best enhanced accuracy  0.7305833333333334
New best enhanced: {'layers': 5, 'hidden': 64, 'batch': 32, 'learning_rate':
0.001, 'alpha': 0.0001, 'epochs': 30, 'batchnorm': False, 'dropout': 0.0,
'activation': np.str_('relu'), 'seed': 541} val_acc= 0.7306
best enhanced accuracy  0.8628333333333333
New best enhanced: {'layers': 4, 'hidden': 64, 'batch': 64, 'learning_rate':
0.001, 'alpha': 0.0, 'epochs': 30, 'batchnorm': True, 'dropout': 0.2,
'activation': np.str_('leaky'), 'seed': 542} val_acc= 0.8628
best enhanced accuracy  0.8870833333333333
New best enhanced: {'layers': 4, 'hidden': 256, 'batch': 32, 'learning_rate':
0.001, 'alpha': 0.0001, 'epochs': 30, 'batchnorm': True, 'dropout': 0.2,
'activation': np.str_('relu'), 'seed': 546} val_acc= 0.8871
Enhanced config: {'layers': 4, 'hidden': 256, 'batch': 32, 'learning_rate':
0.001, 'alpha': 0.0001, 'epochs': 30, 'batchnorm': True, 'dropout': 0.2,
'activation': np.str_('relu'), 'seed': 546} with val_acc=0.8871
Enhanced Train Epoch 1/50: loss=1.2671, test_acc=0.7679, time=6.5s
Enhanced Train Epoch 2/50: loss=0.7456, test_acc=0.8083, time=6.6s
Enhanced Train Epoch 3/50: loss=0.6332, test_acc=0.8228, time=6.7s
Enhanced Train Epoch 4/50: loss=0.5770, test_acc=0.8362, time=6.6s
Enhanced Train Epoch 5/50: loss=0.5371, test_acc=0.8431, time=6.6s
Enhanced Train Epoch 6/50: loss=0.5169, test_acc=0.8469, time=6.6s
Enhanced Train Epoch 7/50: loss=0.4968, test_acc=0.8538, time=6.7s
Enhanced Train Epoch 8/50: loss=0.4827, test_acc=0.8537, time=6.7s
Enhanced Train Epoch 9/50: loss=0.4686, test_acc=0.8581, time=6.6s
Enhanced Train Epoch 10/50: loss=0.4560, test_acc=0.8632, time=6.6s
Enhanced Train Epoch 11/50: loss=0.4433, test_acc=0.8652, time=6.6s
Enhanced Train Epoch 12/50: loss=0.4379, test_acc=0.8675, time=6.7s
Enhanced Train Epoch 13/50: loss=0.4283, test_acc=0.8687, time=6.6s
Enhanced Train Epoch 14/50: loss=0.4193, test_acc=0.8693, time=6.6s
Enhanced Train Epoch 15/50: loss=0.4168, test_acc=0.8711, time=6.6s
Enhanced Train Epoch 16/50: loss=0.4093, test_acc=0.8707, time=6.6s
Enhanced Train Epoch 17/50: loss=0.3988, test_acc=0.8725, time=6.7s
Enhanced Train Epoch 18/50: loss=0.3958, test_acc=0.8720, time=6.6s
Enhanced Train Epoch 19/50: loss=0.3882, test_acc=0.8755, time=6.6s
```

```
Enhanced Train Epoch 20/50: loss=0.3888, test_acc=0.8753, time=6.6s
Enhanced Train Epoch 21/50: loss=0.3835, test_acc=0.8776, time=6.7s
Enhanced Train Epoch 22/50: loss=0.3776, test_acc=0.8773, time=6.7s
Enhanced Train Epoch 23/50: loss=0.3708, test_acc=0.8798, time=7.2s
Enhanced Train Epoch 24/50: loss=0.3676, test_acc=0.8800, time=7.9s
Enhanced Train Epoch 25/50: loss=0.3650, test_acc=0.8805, time=8.3s
Enhanced Train Epoch 26/50: loss=0.3622, test_acc=0.8797, time=7.4s
Enhanced Train Epoch 27/50: loss=0.3569, test_acc=0.8812, time=6.6s
Enhanced Train Epoch 28/50: loss=0.3499, test_acc=0.8809, time=6.5s
Enhanced Train Epoch 29/50: loss=0.3502, test_acc=0.8807, time=6.6s
Enhanced Train Epoch 30/50: loss=0.3498, test_acc=0.8821, time=6.7s
Enhanced Train Epoch 31/50: loss=0.3425, test_acc=0.8829, time=6.6s
Enhanced Train Epoch 32/50: loss=0.3408, test_acc=0.8832, time=6.8s
Enhanced Train Epoch 33/50: loss=0.3381, test_acc=0.8846, time=7.9s
Enhanced Train Epoch 34/50: loss=0.3344, test_acc=0.8848, time=7.0s
Enhanced Train Epoch 35/50: loss=0.3333, test_acc=0.8828, time=6.6s
Enhanced Train Epoch 36/50: loss=0.3335, test_acc=0.8841, time=6.9s
Enhanced Train Epoch 37/50: loss=0.3265, test_acc=0.8831, time=7.1s
Enhanced Train Epoch 38/50: loss=0.3285, test_acc=0.8848, time=6.7s
Enhanced Train Epoch 39/50: loss=0.3232, test_acc=0.8856, time=6.7s
Enhanced Train Epoch 40/50: loss=0.3217, test_acc=0.8849, time=6.6s
Enhanced Train Epoch 41/50: loss=0.3184, test_acc=0.8852, time=6.8s
Enhanced Train Epoch 42/50: loss=0.3178, test_acc=0.8864, time=6.8s
Enhanced Train Epoch 43/50: loss=0.3144, test_acc=0.8849, time=6.9s
Enhanced Train Epoch 44/50: loss=0.3100, test_acc=0.8867, time=6.7s
Enhanced Train Epoch 45/50: loss=0.3096, test_acc=0.8876, time=6.7s
Enhanced Train Epoch 46/50: loss=0.3079, test_acc=0.8879, time=6.6s
Enhanced Train Epoch 47/50: loss=0.3060, test_acc=0.8881, time=6.6s
Enhanced Train Epoch 48/50: loss=0.3030, test_acc=0.8856, time=6.7s
Enhanced Train Epoch 49/50: loss=0.3004, test_acc=0.8879, time=6.8s
Enhanced Train Epoch 50/50: loss=0.2979, test_acc=0.8868, time=6.6s
Final Enhanced Test Accuracy: 0.8868
```

Problem 3: Visualizing the loss landscape and optimization trajectories (15 points)

Visualize the SGD trajectory of your network when trained on Fashion MNIST:

1. Plot in 3-D the cross-entropy loss f_{CE} as a function of the neural network parameters (weights and bias terms). Rather than showing the loss as a function of any particular parameters (e.g., the third component of $b^{(2)}$), use the **x and y-axes** to span the two directions along which your parameters vary the most during SGD training i.e., you will need to use **principal component analysis (PCA)**. (In this assignment, you are free to use `sklearn.decomposition.PCA`.) The **z-axis** should represent the (unregularized) f_{CE} on training data. For each point (x, y) on a grid, compute f_{CE} and then interpolate between the points. (The interpolation and rendering are handled for you completely by the `plot_surface` function; see the starter code).
2. Superimpose a scatter plot of the different points (in the neural network's parameter space) that were reached during SGD (just sample a few times per epoch). To accelerate the rendering, train the network and plot the surface using just a small subset of the training data (e.g., 2500 examples) to estimate f_{CE}. See the starter code, in particular the `plotPath` function, for an example of 3-D plotting. Submit your graph in the PDF file and the code in the Python file.

Answer

```
def plotPath(loader, trajectory, model):
    # TODO: change this toy plot to show a 2-d projection of the weight space
    # along with the associated loss (cross-entropy), plus a superimposed
    # trajectory across the landscape that was traversed using SGD. Use
    # sklearn.decomposition.PCA's fit_transform and inverse_transform
     ↪  methods.

    from sklearn.decomposition import PCA

    # Setup PCA and subset loader for efficient computation
    trajectory_array = np.array(trajectory)
    pca = PCA(n_components=2)
    trajectory_2d = pca.fit_transform(trajectory_array)

    # Create subset of training data (2500 examples) as suggested
    subset_indices = np.random.choice(len(loader.dataset), min(2500,
 ↪  len(loader.dataset)), replace=False)
    subset_loader = DataLoader(torch.utils.data.Subset(loader.dataset,
 ↪  subset_indices), batch_size=256)
    criterion = nn.CrossEntropyLoss()
```

```
    def lossFunction(x1, x2):
        point_params = pca.inverse_transform(np.array([[x1, x2]]))[0]
        load_params_into_model(model, torch.from_numpy(point_params).float())
        return compute_loss(model, subset_loader, criterion)

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')

    # Compute the CE loss on a grid of points (corresonding to different w).
    x_min, x_max = trajectory_2d[:, 0].min(), trajectory_2d[:, 0].max()
    y_min, y_max = trajectory_2d[:, 1].min(), trajectory_2d[:, 1].max()
    x_margin = 0.2 * (x_max - x_min)
    y_margin = 0.2 * (y_max - y_min)
    axis1 = np.linspace(x_min - x_margin, x_max + x_margin, 25) #have to cap
↪   this at 25 or i get an error
    axis2 = np.linspace(y_min - y_margin, y_max + y_margin, 25)
    Xaxis, Yaxis = np.meshgrid(axis1, axis2)
    Zaxis = np.zeros((25, 25))
    for i in tqdm.tqdm(range(25)):
        for j in range(25):
            Zaxis[i,j] = lossFunction(Xaxis[i,j], Yaxis[i,j])
    ax.plot_surface(Xaxis, Yaxis, Zaxis, alpha=0.6)

    # Now superimpose a scatter plot showing the weights during SGD.
    Xaxis = trajectory_2d[:, 0]
    Yaxis = trajectory_2d[:, 1]
    Zaxis = np.array([lossFunction(x, y) for x, y in zip(Xaxis, Yaxis)])
    ax.scatter(Xaxis, Yaxis, Zaxis, color='r')

    plt.show()

# Problem 3: Visualizing the loss landscape and optimization trajectories
# Plot the trajectory of parameters during training
# Uncomment to run
plotPath(full_train_loader, parameter_history, best_model)
```

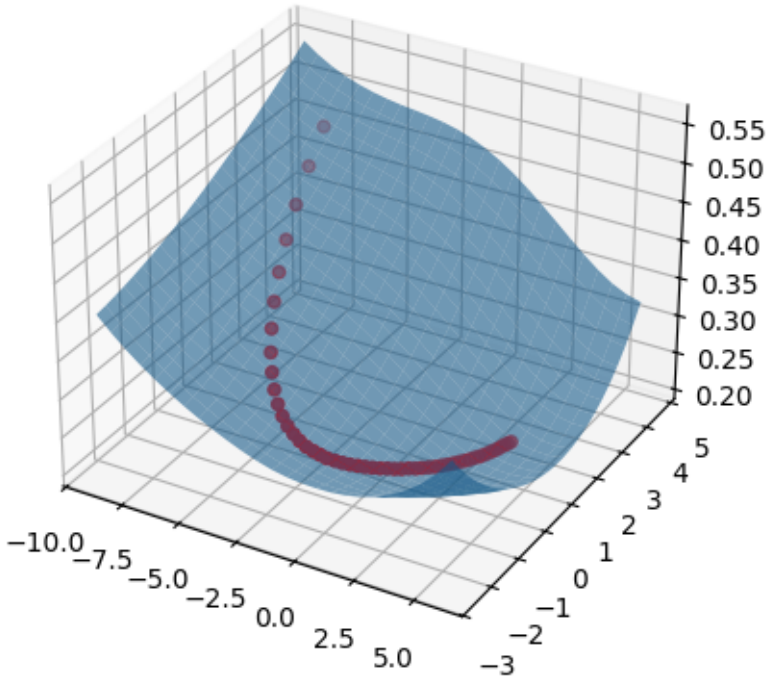

Figure 1: Visualization of the loss landscape and optimization trajectory.

The visualization shows that the neural network optimization follows a clear descent path through the loss landscape. The PCA projection allows us to see the optimization dynamics in a reduced (and therefore visualizable) dimension. It shows how SGD navigates from high-loss regions (initialization) down into a valley or local minimum. The curved shows that SGD doesn't take the shortest path but rather follows the gradient, making consistent but progressivly more mythodical progress toward lower loss. The concentration of trajectory points near the aparent minimum shows successful convergence of the training process.

Problem 4: Autoencoder (20 points)

This assignment explores one powerful concept in deep learning: **unsupervised representation learning with autoencoders**. You will build a neural network that learns to compress and then reconstruct images, forcing it to learn a meaningful representation of the data in the process.

Task 1: Data Preparation (2 point)

1. Load the **Fashion MNIST** dataset using `torchvision.datasets.FashionMNIST`.
2. Use `torchvision.transforms.ToTensor` to convert the images to PyTorch tensors and normalize the pixel values to be in the range $[0.0, 1.0]$.
3. Create a `DataLoader` for both the training and test sets. For this unsupervised task, we only need the image data, not the labels.

Answer

```
# Task 1

def prepare_data(batch_size: int = 128):
    transform = transforms.ToTensor()  # 28x28 grayscale -> float tensor in
 ↪  [0,1]

    train_set = torchvision.datasets.FashionMNIST(
        root="./", train=True, download=True, transform=transform
    )
    test_set = torchvision.datasets.FashionMNIST(
        root="./", train=False, download=True, transform=transform
    )

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True,
 ↪  num_workers=2)
    test_loader  = DataLoader(test_set,  batch_size=batch_size,
 ↪  shuffle=False, num_workers=2)

    # quick checks
    print("n/Data ready")
    print(f"- train samples: {len(train_set)}")
    print(f"- test samples : {len(test_set)}")
    print(f"- image shape  : {train_set[0][0].shape}")  # (1, 28, 28)
    print(f"- train batches: {len(train_loader)}")
    return train_loader, test_loader

# run task 1
train_loader, test_loader = prepare_data(batch_size=128)
```

```
Data:
- train samples: 60000
- test samples : 10000
- image shape  : torch.Size([1, 28, 28])
- train batches: 469
```

Task 2: Autoencoder Architecture (6 points)

Construct a **dense autoencoder model** by creating a class that inherits from `torch.nn.Module`. The model should consist of an **encoder** and a **decoder** with the following symmetric architecture:

- **An Encoder** that takes a flattened 784-dimensional input and maps it to a compressed representation.
  - Input layer (implicitly defined by the input shape).
  - Dense layer (`nn.Linear`) with **128 units** followed by a **ReLU** activation (`nn.ReLU`).
  - Dense layer with **64 units** followed by a **ReLU** activation. This layer's output is the **latent representation**, also known as the "**bottleneck**".
- **A Decoder** that takes the 64-dimensional latent representation and reconstructs the 784-dimensional image.
  - Dense layer with **128 units** followed by a **ReLU** activation.
  - Dense layer with **784 units** followed by a **Sigmoid** activation (`nn.Sigmoid`). The sigmoid activation ensures the output values are in the range $[0.0, 1.0]$, matching the normalized input data.

Answer

```python
# Task 2

class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Linear(784, 128), nn.ReLU(),
            nn.Linear(128, 64),  nn.ReLU(),
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 128),  nn.ReLU(),
            nn.Linear(128, 784), nn.Sigmoid(),  # keep outputs in [0,1]
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.decoder(self.encoder(x))

    def encode(self, x: torch.Tensor) -> torch.Tensor:
        return self.encoder(x)

# build model and show size
model = Autoencoder()
params = sum(p.numel() for p in model.parameters())
print("Model ready")
print(f"- parameters: {params:,}")
```

```
Model ready
- parameters: 218,192
```

Task 3: Model Training (6 points)

1. Instantiate your model, a loss function (`nn.MSELoss` is a good choice), and an optimizer (`torch.optim.Adam`).
2. Write a training loop that iterates for **20 epochs**. In each epoch, iterate through the training `DataLoader`.
3. For each batch of images, you must:
    - Flatten the 28×28 images into 784-dimensional vectors.
    - Perform a forward pass to get the reconstructed images.
    - Calculate the loss between the original and reconstructed images.
    - Zero the gradients, perform a backward pass, and update the weights.

Answer

```
# Task 3

def train_autoencoder(model: nn.Module, loader: DataLoader, epochs: int = 20,
 ↪  lr: float = 1e-3):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)
    opt = optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()

    print(f"Training on {device} for {epochs} epochs")
    losses = []

    for epoch in range(1, epochs + 1):
        model.train()
        running = 0.0

        for i, (imgs, _) in enumerate(loader):
            imgs = imgs.to(device).view(imgs.size(0), -1)   # (B, 1, 28, 28)
 ↪  -> (B, 784)

            opt.zero_grad()
            out = model(imgs)
            loss = loss_fn(out, imgs)
            loss.backward()
            opt.step()

            running += loss.item()
            if i % 100 == 0:
                print(f"  epoch {epoch:2d} | batch {i:4d}/{len(loader)} |
                 ↪  loss {loss.item():.4f}")

        avg = running / len(loader)
        losses.append(avg)
        print(f"  epoch {epoch:2d} complete | avg loss {avg:.4f}")

    return model, losses

# run task 3
model, losses = train_autoencoder(model, train_loader, epochs=20, lr=1e-3)
```

```
Training on cuda for 20 epochs
  epoch  1 | batch    0/469 | loss 0.1683
  epoch  1 | batch  100/469 | loss 0.0449
  epoch  1 | batch  200/469 | loss 0.0267
  epoch  1 | batch  300/469 | loss 0.0231
  epoch  1 | batch  400/469 | loss 0.0232
  epoch  1 complete | avg loss 0.0375
  epoch  2 | batch    0/469 | loss 0.0220
  epoch  2 | batch  100/469 | loss 0.0206
  epoch  2 | batch  200/469 | loss 0.0188
  epoch  2 | batch  300/469 | loss 0.0195
  epoch  2 | batch  400/469 | loss 0.0180
  epoch  2 complete | avg loss 0.0196
  epoch  3 | batch    0/469 | loss 0.0184
  epoch  3 | batch  100/469 | loss 0.0178
  epoch  3 | batch  200/469 | loss 0.0167
  epoch  3 | batch  300/469 | loss 0.0169
  epoch  3 | batch  400/469 | loss 0.0160
  epoch  3 complete | avg loss 0.0169
  epoch  4 | batch    0/469 | loss 0.0156
  epoch  4 | batch  100/469 | loss 0.0157
  epoch  4 | batch  200/469 | loss 0.0143
  epoch  4 | batch  300/469 | loss 0.0141
  epoch  4 | batch  400/469 | loss 0.0146
  epoch  4 complete | avg loss 0.0151
  epoch  5 | batch    0/469 | loss 0.0140
  epoch  5 | batch  100/469 | loss 0.0139
  epoch  5 | batch  200/469 | loss 0.0136
  epoch  5 | batch  300/469 | loss 0.0138
  epoch  5 | batch  400/469 | loss 0.0134
  epoch  5 complete | avg loss 0.0140
  epoch  6 | batch    0/469 | loss 0.0136
  epoch  6 | batch  100/469 | loss 0.0132
  epoch  6 | batch  200/469 | loss 0.0129
  epoch  6 | batch  300/469 | loss 0.0132
  epoch  6 | batch  400/469 | loss 0.0142
  epoch  6 complete | avg loss 0.0133
  epoch  7 | batch    0/469 | loss 0.0138
  epoch  7 | batch  100/469 | loss 0.0121
  epoch  7 | batch  200/469 | loss 0.0117
  epoch  7 | batch  300/469 | loss 0.0123
  epoch  7 | batch  400/469 | loss 0.0141
  epoch  7 complete | avg loss 0.0127
```

```
epoch  8 | batch    0/469 | loss 0.0128
epoch  8 | batch  100/469 | loss 0.0120
epoch  8 | batch  200/469 | loss 0.0111
epoch  8 | batch  300/469 | loss 0.0115
epoch  8 | batch  400/469 | loss 0.0118
epoch  8 complete | avg loss 0.0121
epoch  9 | batch    0/469 | loss 0.0130
epoch  9 | batch  100/469 | loss 0.0115
epoch  9 | batch  200/469 | loss 0.0125
epoch  9 | batch  300/469 | loss 0.0117
epoch  9 | batch  400/469 | loss 0.0111
epoch  9 complete | avg loss 0.0117
epoch 10 | batch    0/469 | loss 0.0109
epoch 10 | batch  100/469 | loss 0.0125
epoch 10 | batch  200/469 | loss 0.0115
epoch 10 | batch  300/469 | loss 0.0104
epoch 10 | batch  400/469 | loss 0.0114
epoch 10 complete | avg loss 0.0114
epoch 11 | batch    0/469 | loss 0.0104
epoch 11 | batch  100/469 | loss 0.0113
epoch 11 | batch  200/469 | loss 0.0106
epoch 11 | batch  300/469 | loss 0.0120
epoch 11 | batch  400/469 | loss 0.0100
epoch 11 complete | avg loss 0.0111
epoch 12 | batch    0/469 | loss 0.0115
epoch 12 | batch  100/469 | loss 0.0099
epoch 12 | batch  200/469 | loss 0.0107
epoch 12 | batch  300/469 | loss 0.0101
epoch 12 | batch  400/469 | loss 0.0112
epoch 12 complete | avg loss 0.0108
epoch 13 | batch    0/469 | loss 0.0099
epoch 13 | batch  100/469 | loss 0.0104
epoch 13 | batch  200/469 | loss 0.0103
epoch 13 | batch  300/469 | loss 0.0107
epoch 13 | batch  400/469 | loss 0.0105
epoch 13 complete | avg loss 0.0106
epoch 14 | batch    0/469 | loss 0.0097
epoch 14 | batch  100/469 | loss 0.0105
epoch 14 | batch  200/469 | loss 0.0115
epoch 14 | batch  300/469 | loss 0.0106
epoch 14 | batch  400/469 | loss 0.0103
epoch 14 complete | avg loss 0.0104
epoch 15 | batch    0/469 | loss 0.0101
```

```
epoch 15 | batch  100/469 | loss 0.0106
epoch 15 | batch  200/469 | loss 0.0115
epoch 15 | batch  300/469 | loss 0.0099
epoch 15 | batch  400/469 | loss 0.0107
epoch 15 complete | avg loss 0.0102
epoch 16 | batch    0/469 | loss 0.0102
epoch 16 | batch  100/469 | loss 0.0102
epoch 16 | batch  200/469 | loss 0.0104
epoch 16 | batch  300/469 | loss 0.0106
epoch 16 | batch  400/469 | loss 0.0097
epoch 16 complete | avg loss 0.0101
epoch 17 | batch    0/469 | loss 0.0092
epoch 17 | batch  100/469 | loss 0.0108
epoch 17 | batch  200/469 | loss 0.0097
epoch 17 | batch  300/469 | loss 0.0097
epoch 17 | batch  400/469 | loss 0.0099
epoch 17 complete | avg loss 0.0099
epoch 18 | batch    0/469 | loss 0.0102
epoch 18 | batch  100/469 | loss 0.0113
epoch 18 | batch  200/469 | loss 0.0100
epoch 18 | batch  300/469 | loss 0.0104
epoch 18 | batch  400/469 | loss 0.0089
epoch 18 complete | avg loss 0.0098
epoch 19 | batch    0/469 | loss 0.0108
epoch 19 | batch  100/469 | loss 0.0100
epoch 19 | batch  200/469 | loss 0.0099
epoch 19 | batch  300/469 | loss 0.0093
epoch 19 | batch  400/469 | loss 0.0092
epoch 19 complete | avg loss 0.0096
epoch 20 | batch    0/469 | loss 0.0085
epoch 20 | batch  100/469 | loss 0.0095
epoch 20 | batch  200/469 | loss 0.0088
epoch 20 | batch  300/469 | loss 0.0106
epoch 20 | batch  400/469 | loss 0.0097
epoch 20 complete | avg loss 0.0095
```

Task 4: Visualizing Reconstructions (6 points)

1. Set your model to **evaluation mode**.

2. Get a batch of images from the test set and pass them through your trained autoencoder to get the reconstructions.
3. Create a plot showing **two different original images** from the test set and their corresponding reconstructions directly below them.
4. **Deliverable:** A single figure containing 4 images (2 original test images, 2 reconstructed images), clearly labeled.

Answer

```
# Task 4

def visualize_reconstructions(model: nn.Module, test_loader: DataLoader,
 ↪  num_images: int = 2):
    model.eval()
    device = next(model.parameters()).device

    images, _ = next(iter(test_loader))
    images = images.to(device)
    with torch.no_grad():
        recon = model(images.view(images.size(0), -1)).view(-1, 1, 28, 28)

    images = images.cpu()
    recon = recon.cpu()

    fig, axes = plt.subplots(2, 2, figsize=(8, 4))
    axes[0, 0].imshow(images[0].squeeze(), cmap="gray"); axes[0,
 ↪  0].set_title("Original 1"); axes[0, 0].axis("off")
    axes[1, 0].imshow(recon[0].squeeze(),   cmap="gray"); axes[1,
 ↪  0].set_title("Reconstructed 1"); axes[1, 0].axis("off")
    axes[0, 1].imshow(images[1].squeeze(), cmap="gray"); axes[0,
 ↪  1].set_title("Original 2"); axes[0, 1].axis("off")
    axes[1, 1].imshow(recon[1].squeeze(),   cmap="gray"); axes[1,
 ↪  1].set_title("Reconstructed 2"); axes[1, 1].axis("off")
    fig.suptitle("Autoencoder: Test Originals vs Reconstructions")
    plt.tight_layout()
    plt.show()

# run task 4
visualize_reconstructions(model, test_loader, num_images=2)
```

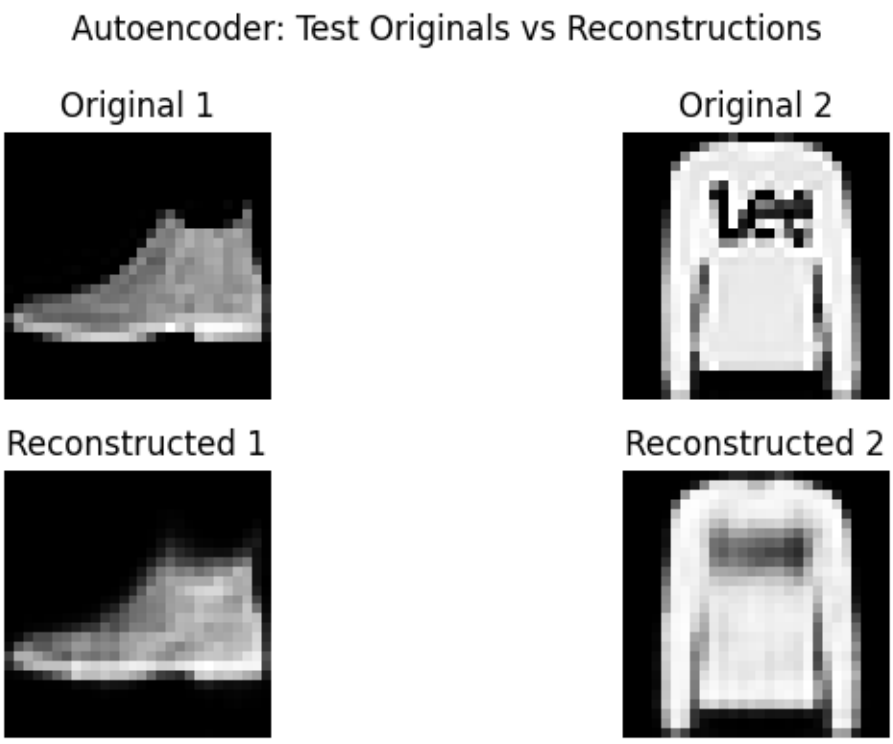

Figure 2: Autoencoder reconstructions on test set: Originals (top) vs Reconstructions (bottom).

Submission

Submit one PDF file that includes your notes for the theoretical problems (scanned or typed) and screenshots of your code for the programming problems. All material in the submitted PDF must be presented in a clear and readable format.

If you are working as part of a group, then indicate the members on canvas: https://canvas.wpi.edu/courses/7677
14853. Once you do that, be aware that any submission from a team member will overwrite an existing one.

Teamwork

You may complete this homework assignment either individually or in teams up to 2 people.