Homework 1

Graded

Student

Rohan Gore

Total Points

15 / 15 pts

Question 1

Expressivity of neural nets **4** / 4 pts

1.1 **Box function** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Partially Correct

– 1 pt Incorrect

Click here to replace this description.

– 0 pts Click here to replace this description.

1.2 **Smooth bounded functions** **2** / 2 pts

✔ **– 0 pts** Correct and clear explanation

– 0.5 pts Explanation with missing detail

– 1 pt Only partially correct explanation

– 2 pts Incorrect or No clear explanation

– 2 pts Not attempted

1.3 **Multiple dimensions** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Partially correct comment on practical issues.

> The number of trainable parameters –
> becomes exponentially large.

– 1 pt Missing solution

Question 2

Initialization **4** / 4 pts

2.1 **Linear layer** **1** / 1 pt

✔ **– 0 pts** Correct final answer with a clear step-by-step derivation. Well done!

– 0.5 pts Final answer is correct, but missing steps in the derivation.

– 1 pt Incorrect final answer; key conceptual errors in the derivation.

– 0.5 pts Partially correct

– 1 pt Not attempted

2.2 **Scale of variance** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.5 pts Incorrect final answer

– 0.5 pts Reasoning incorrect or missing

– 0.5 pts Handwritten submission

2.3 **ReLU** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.5 pts Incorrect final answer

– 0.5 pts Explanation for value of factor missing or incorrect.

– 0.5 pts Handwritten submission

2.4 **Backward passes** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts No explanation provided

– 1 pt Incorrect

– 0.5 pts n should represent the output neurons only

– 1 pt No submission

2.5 **Xavier/Glorot init** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Missing explanation

– 1 pt Incorrect explanation and final answer $\sigma^2 = \frac{2}{n_{in}+n_{out}}$

– 1 pt Handwritten submission

– 0.5 pts Incorrect expression. $\sigma^2 = \frac{2}{n_{in}+n_{out}}$

Question 3

FashionMNIST **3** / 3 pts

3.1 **Training** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** Train Loss not decreasing
- **– 0.5 pts** Incorrect model size
- **– 0.5 pts** Important code blocks truncated due to incorrect conversion to pdf
- **– 1 pt** Incorrect model type
- **– 1 pt** Training code missing
- **– 0.5 pts** Training code partially missing
- **– 0.5 pts** Validation(test) losses not reported during training
- **– 0.5 pts** Train and test losses not reported during training
- **– 0.5 pts** Test loss not decreasing

3.2 **Curves, accuracies, explanation** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** No or incorrect train and test loss curves
- **– 0.5 pts** No good plots and no comments
- **– 0.5 pts** No explanation
- **– 0.5 pts** No test accuracies reported

3.3 **Three instances of model prediction** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** Missing observations/comments
- **– 0.5 pts** Missing Probabilities
- **– 0.5 pts** Incorrect Observations/comments
- **– 0.5 pts** Incorrect answer
- **– 1 pt** Not attempted

Question 4

Backpropagation **4** / 4 pts

4.1 **Part 1** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Incorrect initialization

– 1 pt No answer given

4.2 **Part 2** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Incorrect

– 1 pt No answer given

4.3 **Part 3** **1** / 1 pt

– 0.5 pts incorrect feed forward code

– 0.5 pts incorrect backward propagation code

✔ **– 0 pts** Correct

– 0.5 pts insufficient comments/explanation for code

need better comments/explanation for code

4.4 **Part 4** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Accuracy is not within the specified range

– 1 pt Missing/incorrect output

– 0.5 pts instructions were export notebook to PDF and include, not placing external links in the pdf.

Question 5

Late Submission **0** / 0 pts

– 1 pt Late Submission

✔ **– 0 pts** Submission On Time

Question assigned to the following page: 1.1

NYU: Deep Learning CS 6953 Spring 2025

Homework 1

Name: Rohan Mahesh Gore (N19332535)

Problem 1: Expressivity of Neural Networks

Collaborators: Perplexity - Sonar Huge & DeepSeek-R1 , ChatGPT 4o.

a. Realizing a Box Function with a Neural Network

We are given a mathematical function called a box function and need to construct a simple neural network with only two hidden neurons that can represent it. The box function is ON (i.e., has a value h) for a specific interval $0 < x < \delta$, and OFF (i.e., has a value 0) outside that interval.

Mathematically, it can be defined as:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & \text{otherwise} \end{cases}$$

Therefore, the neurons should have an activation function, which looks like this:

$$\sigma(u) = \begin{cases} 1, & \text{if } u > 0 \\ 0, & \text{otherwise} \end{cases}$$

Also the output neuron should not have an activation function.

Step 1: Constructing hidden layer neurons

Since we only have two hidden neurons, and their rough working should be like $\rightarrow$ ON - inside the interval $0 < x < \delta$, and, OFF - outside this interval.

Neuron 1 should activate when $x > 0$ and remain off otherwise. Therfore we have: $\sigma(w_1 x + b_1) = 1 \quad \text{for } x > 0$, and, $\sigma(w_1 x + b_1) = 0 \quad \text{for } x \leq 0$ by setting:

$$w_1 = 1, \quad b_1 = 0$$

Therefore, our first neuron computes:$\sigma(x)$

Neuron 2 should activate when $x > \delta$ and remain off otherwise. Therfore we have: $\sigma(w_2 x + b_2) = 1 \quad \text{for } x > \delta$, and, $\sigma(w_2 x + b_2) = 0 \quad \text{for } x \leq delta$ by setting

$$w_2 = 1, \quad b_2 = -\delta$$

Therefore, our second neuron computes:$\sigma(x - \delta)$

Question assigned to the following page: 1.1

Step 2: Constructing output layer neurons

Now, our two neurons produce:

$$\sigma(x) \quad \text{and} \quad \sigma(x-\delta)$$

- The first neuron turns on when $x > 0$. - The second neuron turns on when $x > \delta$.

We know what outputs we will be accumulating at the output layer, and for us to get the the correct box function, we setup the output neuron as a subtractive neuron, which basically calculates the difference of both the received inputs from the hidden layers: $\sigma(x) - \sigma(x-\delta)$

Step 3: Simluation for verifying correct ON and OFF of neurons

- For $x \leq 0$: $\sigma(x) = 0$, $\sigma(x-\delta) = 0$, so output $= 0 - 0 = 0$.
- For $0 < x < \delta$: $\sigma(x) = 1$, $\sigma(x-\delta) = 0$, so output $= 1 - 0 = 1$.
- For $x \geq \delta$: $\sigma(x) = 1$, $\sigma(x-\delta) = 1$, so output $= 1 - 1 = 0$.

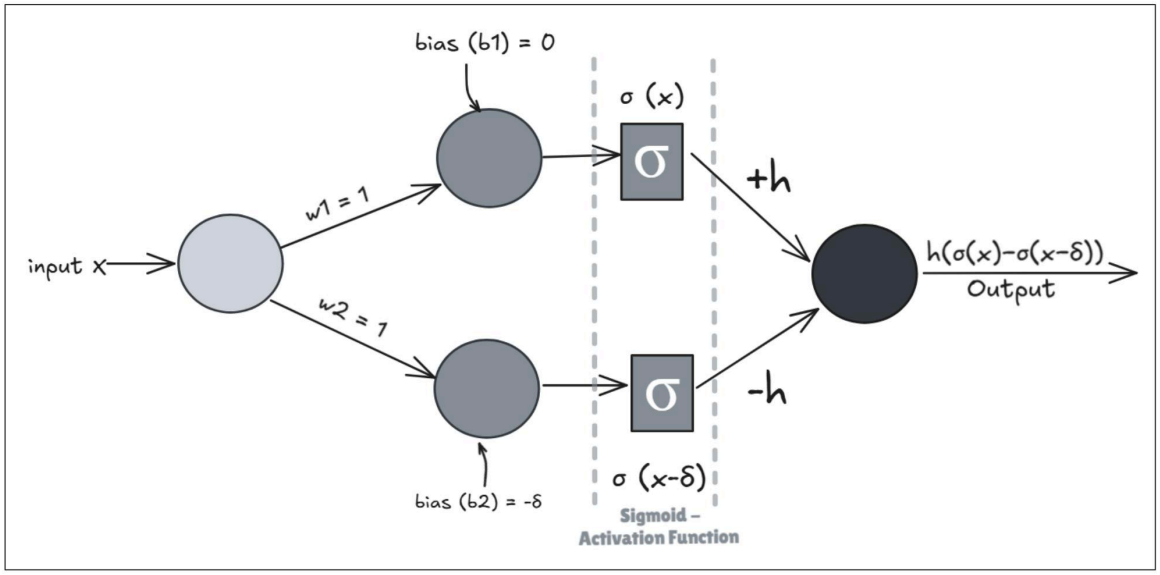
Step 4: Final Function

We multiply the above derived subtractive equation by h to get the final form of the output function. Therfore we have,

$$h \cdot (\sigma(x) - \sigma(x-\delta))$$

which, exactly matches the box function.

Diagram for Demonstrating Working



Hence, Solved.

Question assigned to the following page: 1.2

b. Approximation of Any Smooth Function Using a Neural Network

We are given an **arbitrary smooth bounded function** $f(x)$ defined over the interval $[-B, B]$. which we need to show, can be closely realized by a neural network with a hidden layer of step-activated neurons.

From Problem 1(a), we know that we can approximate a box function as follows:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & \text{otherwise} \end{cases}$$

Since a box function is a simple function that turns "ON" in a small range, we can combine multiple box functions to approximate any smooth function.

Therfore the approach I want to take is:

1. Divide the interval $[-B, B]$ into small subintervals of width δ.
2. In each small interval, use a box function to approximate $f(x)$.
3. Sum up multiple such box functions with appropriate heights to match the function $f(x)$.

Simplifying Mathematically

We approximate $f(x)$ using a sum of small box functions centered at different points x_i:

$$f(x) \approx \sum_i h_i(\sigma(x - x_i) - \sigma(x - (x_i + \delta)))$$

where,
- x_i is the starting point of each box function.
- δ is the width of each box function.
- h_i is the height of each box function, chosen to match $f(x)$.

NOTE:
- If δ is small, the sum of box functions will closely resemble $f(x)$.
- The smaller we make δ, the better the approximation.

Constructing Neural Network

1. Each hidden neuron represents one box function which activates in a small region of width δ and can be declared by the same structure from Problem 1(a).
2. The output neuron sums these box functions to approximate $f(x)$.

Therefore we have the final form as,

$$y = \sum_i h_i(\sigma(x - x_i) - \sigma(x - (x_i + \delta)))$$

Hence, Proved.

Question assigned to the following page: 1.3

c. Extending to Higher Dimensions

In 1D space, we divided the interval $[-B, B]$ into small intervals (boxes) and used neurons to approximate $f(x)$ by summing these boxes. Each neuron corresponded to a box in 1D.

For d-dimensional inputs $x = (x_1, x_2, \ldots, x_d)$, we would need to cover the hypercube $[-B, B]^d$. Each "box" becomes a **hyper-rectangle** in d-dimensional space.

Step 1: Box Function Definition

As we know in 1D, we used small box functions of the form: $h_i(\sigma(x - x_i) - \sigma(x - (x_i + \delta)))$

In d-dimensions, we can extend this idea by defining a multi-dimensional box function:

$$B(x) = \begin{cases} h, & x_1 \in [a_1, b_1], x_2 \in [a_2, b_2], \ldots, x_d \in [a_d, b_d] \\ 0, & \text{otherwise} \end{cases}$$

This means the function is **ON inside the hyper-rectangle** and OFF elsewhere.

Step 2: Using Neurons to Construct a 2D Box Function

Each dimension now needs two step-activation neurons:

- One to detect when x_i enters the box ($x_i > a_i$).
- One to detect when x_i leaves the box ($x_i > b_i$).

Therefore, for each dimension, we can define activation functions as: $\sigma(x_i - a_i) - \sigma(x_i - b_i)$

Hence Generalizing to d dimensions,

$$B(x) = h \cdot \prod_{i=1}^{d} (\sigma(x_i - a_i) - \sigma(x_i - b_i))$$

Step 3: Constructing the Full Neural Network

Therfore, to approximate a full form function $f(x_1, x_2, \ldots, x_d)$, we sum all the such box functions:

$$f(x_1, x_2, \ldots, x_d) \approx \sum_j h_j \prod_{i=1}^{d} (\sigma(x_i - a_{ij}) - \sigma(x_i - b_{ij}))$$

where:
- j indexes different box functions.
- a_{ij} and b_{ij} define the boundaries of each box.

This follows the same idea as in 1D, but now each neuron detects activation in multiple dimensions.

Question assigned to the following page: 1.3

Practical Challenges:

1. **Curse of Dimensionality**: In 1D, we needed around n neurons for n small intervals. But in d-dimensions, we need a grid of box functions to cover the space. Therfore, if we use n intervals per dimension, the total number of required neurons is: $O(n^d)$

   For Example:
   - In 1D, if we use 10 neurons, we get 10 intervals.
   - In 2D, we need a grid of $10 \times 10 = 100$ neurons.
   - In 3D, we need $10^3 = 1,000$ neurons.
   - In 100D, we would need 10^{100} neurons, which is too much.

2. **Data Sparsity:** Most hyperrectangles will be empty (no training data) in high dimensions. This would act as a problem as the network cannot learn meaningful patterns without dense coverage.

Hence, Solved.

Question assigned to the following page: 2.1

Problem 2: Choosing the Right Scale of Initialization

Collaborators: Perplexity - DeepSeek-R1 , ChatGPT 4o.

a. Variance Calculation in a Fully-Connected Layer

We have to derive the variance of the activation $a_j$in a fully-connected linear layer.

We are given:
- Weights $w_{ij} \sim \mathcal{N}(0, \sigma^2)$ (Gaussian with mean 0, variance σ^2).
- Inputs x_i are independent, zero-mean, with variance $\text{var}(x_i) = v^2$.
- Biases are zero.

Goal: Express $\text{var}(a_j)$ in terms of σ^2, v^2, and n_{in}.

Step 1: Write the Expression for a_j

We know that the activation a_j of the j-th output neuron is:

$$a_j = \sum_{i=1}^{n_{\text{in}}} w_{ij} x_i$$

Step 2: BY using variance properties

For independent random variables X and Y: $\text{var}(X + Y) = \text{var}(X) + \text{var}(Y)$

For a scalar c: $\text{var}(cX) = c^2 \,\text{var}(X)$

Step 3: Expanding $\text{var}(a_j)$

Since x_i and w_{ij} are independent:

$$\text{var}(a_j) = \sum_{i=1}^{n_{\text{in}}} \text{var}(w_{ij} x_i)$$

Step 4: Computing $\text{var}(w_{ij} x_i)$

For independent w_{ij} and x_i:

$$\text{var}(w_{ij} x_i) = \mathbb{E}[w_{ij}^2 x_i^2] - (\mathbb{E}[w_{ij}]\mathbb{E}[x_i])^2$$

Since $\mathbb{E}[w_{ij}] = 0$ and $\mathbb{E}[x_i] = 0$, Therefore: $\text{var}(w_{ij} x_i) = \mathbb{E}[w_{ij}^2]\mathbb{E}[x_i^2] = \text{var}(w_{ij})\,\text{var}(x_i)$

Step 5: Substituting $\text{var}(w_{ij}) = \sigma^2$ **and** $\text{var}(x_i) = v^2$

Therefore we will get,

$$\text{var}(w_{ij} x_i) = \sigma^2 v^2$$

Question assigned to the following page: 2.1

Step 6: Summing Over All Inputs

There are n_{in} terms in the sum:

$$\text{var}(a_j) = \sum_{i=1}^{n_{\text{in}}} \sigma^2 v^2 = n_{\text{in}} \cdot \sigma^2 \cdot v^2$$

Therefore the final result is:

$$\text{var}(a_j) = n_{\text{in}} \cdot \sigma^2 \cdot v^2$$

Hence, Derived.

Question assigned to the following page: 2.2

b. Preventing Vanishing or Exploding Variance

From Problem 2(a), we found the variance of the activation a_j for the j-th neuron in a fully-connected layer is $\text{Var}(a_j) = n_{\text{in}} v^2 \sigma^2$

1. If $\text{Var}(a_j)$ is too small, the outputs of neurons get very close to zero. This leads to saturated activations (e.g., Sigmoid outputs stuck at 0 or 1) and hence leads to vanishing gradient.
    - This happens if: σ^2 is too small n_{in} is too large
2. If $\text{Var}(a_j)$ is too large, neuron outputs blow up which leads to numerical instability (e.g., ReLU outputs becoming very large) and hecne leads to exploding gradients.
    - This happens if: - σ^2 is too large - n_{in} is too large

Formulating Thumb Rule

Therfore we now know that we need to exactly balance variance such that it is neither too small nor too large. A good balance is to maintain the same variance through each layer for which we would need:

$$\text{Var}(a_j) \approx v^2$$

Substituting from Problem 2(a) we have:

$$n_{\text{in}} v^2 \sigma^2 \approx v^2$$

Simplifying:

$$\sigma^2 \approx \frac{1}{n_{\text{in}}}$$

Therefore the final thumb rule for weight initialization can be written as

$$\sigma^2 = \frac{1}{n_{\text{in}}}$$

OR (since weights are normally distributed):

$$w_{ji} \sim N\left(0, \frac{1}{n_{\text{in}}}\right)$$

Question assigned to the following page: 2.3

c. Impact of ReLU on Variance Scaling

From Problem 2(b) we know that for a linear layer without ReLU, we derived:

$$\sigma^2 = \frac{1}{n_{\text{in}}}$$

to ensure stable variance: $\text{var}(a_j) = v^2$.

Step 1: Analysing Effect of ReLU on Variance

Since ReLU zeroes out all negative inputs, the probability is 0.5 (half the Gaussian distribution is negative. Therefore, by using properties of half-Gaussian distributions, the variance becomes:

$$\text{var}(\text{ReLU}(a_j)) = \frac{1}{2}\,\text{var}(a_j)$$

Step 2: Deriving the Scaling Factor

AS done before , for stabilizing variance after ReLU, we need: $\text{var}(\text{ReLU}(a_j)) = v^2$

From Problem 2(a) we can substitute: $\text{var}(a_j) = n_{\text{in}} \cdot \sigma^2 \cdot v^2$ Therfore we have:

$$\frac{1}{2} \cdot n_{\text{in}} \cdot \sigma^2 \cdot v^2 = v^2$$

Step 3: Solving for σ^2

Therefore we have,

$$\sigma^2 = \frac{2}{n_{\text{in}}}$$

This can also be verified by the fact that: since ReLU halves the variance, we need to **double** the initialization variance to compensate. Hence our **scaling factor is 2**.

Hence, Solved.

Question assigned to the following page: 2.4

d. Variance of Gradients During Backpropagation

We already know that gradients at the output neurons have variance g^2 & Gradients flow backward through the transposed weight matrix W^T.

Step 1: Backward Pass Gradient Flow

During backpropagation, the gradient of the loss $\mathcal{L}$ with respect to the input x_i is:

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_{j=1}^{n_{\text{out}}} w_{ij} \cdot \frac{\partial \mathcal{L}}{\partial a_j}$$

where, $\frac{\partial \mathcal{L}}{\partial a_j}$ (output gradient) has variance g^2.

Step 2: Variance of Input Gradients

Assume that output gradients $\frac{\partial \mathcal{L}}{\partial a_j}$ are independent. and Weights $w_{ij} \sim \mathcal{N}(0, \sigma^2)$. Therefore, variance of $\frac{\partial \mathcal{L}}{\partial x_i}$ is:

$$\text{var}\left(\frac{\partial \mathcal{L}}{\partial x_i}\right) = \sum_{j=1}^{n_{\text{out}}} \text{var}\left(w_{ij} \cdot \frac{\partial \mathcal{L}}{\partial a_j}\right)$$

Step 3: Expand the Variance Term

For independent w_{ij} and $\frac{\partial \mathcal{L}}{\partial a_j}$:

$$\text{var}\left(w_{ij} \cdot \frac{\partial \mathcal{L}}{\partial a_j}\right) = \text{var}(w_{ij}) \cdot \text{var}\left(\frac{\partial \mathcal{L}}{\partial a_j}\right) = \sigma^2 \cdot g^2$$

Since there are n_{out} terms in the sum, we have:

$$\text{var}\left(\frac{\partial \mathcal{L}}{\partial x_i}\right) = n_{\text{out}} \cdot \sigma^2 \cdot g^2$$

Step 4: Stabilizing Variance

To prevent vanishing/exploding gradients we would need:

$$\text{var}\left(\frac{\partial \mathcal{L}}{\partial x_i}\right) = g^2$$

Therefore substituting in Step 3's equation we get:

$$n_{\text{out}} \cdot \sigma^2 \cdot g^2 = g^2 \implies \sigma^2 = \frac{1}{n_{\text{out}}}$$

Therfore to prevent vanishing or exploding gradients during backpropagation we can use the thumb rule as:

$$\sigma^2 = \frac{1}{n_{\text{out}}}$$

Or equivalently:

$$w_{ji} \sim N\left(0, \frac{1}{n_{\text{out}}}\right)$$

Hence, Solved.

Question assigned to the following page: 2.5

e. Extending to Higher Dimensions

From Problems 2b and 2d we know that to have stabilized activations we require $\sigma^2 = \frac{1}{n_{\text{in}}}$ and $\sigma^2 = \frac{1}{n_{\text{out}}}$ for forward and backward pass respectively.

Therefore for both the directions we need to make a compromise, which is the harmonic mean of both the constraints. This gives us:

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

This is the Xavier/Glorot initialization rule.

In practice, the weights are drawn from a uniform distribution:

$$w_{ji} \sim U\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$$

The factor of 6 comes from the variance of a uniform distribution:

$$\text{Var}(U[-a, a]) = \frac{a^2}{3}$$

We want the variance to be $\frac{2}{n_{\text{in}}+n_{\text{out}}}$, so we solve:

$$\frac{a^2}{3} = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

$$a^2 = \frac{6}{n_{\text{in}} + n_{\text{out}}}$$

$$a = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$$

Since the uniform distribution is symmetric around 0, the range becomes:

$$U\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$$

Hence, Demonstrated.

Questions assigned to the following page: 4.1 and 3.1

Problem 3: Improving the FashionMNIST classifier

Collaborators: Perplexity - DeepSeek-R1 , ChatGPT 4o.

Attached ahead.

Problem 4: Implementing back-propagation in Python from scratch

Collaborators: Perplexity - DeepSeek-R1 , ChatGPT 4o.

Attached ahead.

Question assigned to the following page: 3.1

DL --> Assignment 1 --> Question 3

```
# We start by importing the libraries we'll use today
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
```

```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,transform=torchvision.transforms.ToTensor())
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,transform=torchvision.transforms.ToTensor())
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|| 26.4M/26.4M [00:01<00:00, 18.7MB/s]
Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|| 29.5k/29.5k [00:00<00:00, 301kB/s]
Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|| 4.42M/4.42M [00:00<00:00, 5.60MB/s]
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|| 5.15k/5.15k [00:00<00:00, 8.26MB/s]
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw
```

```
"""
just checking the shape of the train and test datasets
Also, i came back here to confirm the sizes during adjusting the batch size for
the train and test datasets
"""


print(len(trainingdata))
print(len(testdata))
```

```
60000
10000
```

```
trainDataLoader = torch.utils.data.DataLoader(trainingdata,batch_size=128,shuffle=True)
testDataLoader = torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)
```

Network Setup

```
class DenseNN(torch.nn.Module):
    def __init__(self):
        super(DenseNN, self).__init__()


# hidden layer mapping


        # Input layer to H1 (256)
        self.layer1 = torch.nn.Linear(28*28, 256)


        # H1 to H2 (256 -> 128)
        self.layer2 = torch.nn.Linear(256, 128)


        # H2 to H3 (128 -> 64)
        self.layer3 = torch.nn.Linear(128, 64)
        # Output layer (64 -> 10 classes)


        self.layer4 = torch.nn.Linear(64, 10)

        self.relu = torch.nn.ReLU()

    def forward(self, x):
        # Flatten the input image
        x = x.view(-1, 28*28)


        # RELu for all hidden layers

        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.relu(self.layer3(x))

        # Final layer (no activation)
        x = self.layer4(x)
        return x
```

Question assigned to the following page: 3.1

```
"""
Initializing the model now with the DenseNN function
The 2 main items that we have are yhe loss, which over here is the Cross Entropy Loss
and the optimizer, which is the Adam optimizer.
"""

model = DenseNN()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Complete Model's Epoch Execution

```
"""
train_loss_history, test_loss_history, train_acc_history, test_acc_history are
used to store the training and testing loss and accuracy values to keep track
of the model's performance over time. They are crucial during model's
evaluation and analysis, exspecially for the loss and accuracy curves.
"""
train_loss_history = []
test_loss_history = []
train_acc_history = []
test_acc_history = []

print("Starting neural network training...")

'''

--------------------------------

prefering training on device - on CPU since it is not that compute intensive

- Training for 30 epochs in order to attempt to get the accuracy to cross 90%,
but so far it has not worked out

- Also tried changing the learning rate, train and test batch size, and even number of epochs
but I am not able to break the 90% resistance.

-AS of now, the best config. is:
-->train_batch = 128, test_batch = 64, lr = 0.001, epochs = 30, accuracy = 89.42

--------------------------------

'''

for epoch in range(30):
    train_loss = 0.0
    test_loss = 0.0
    train_correct = 0
    test_correct = 0
    total_samples = 0



    # Training phase
    model.train()
    for i, (images, labels) in enumerate(trainDataLoader):
        images, labels = images, labels
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Calculate training metrics
        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_correct += (predicted == labels).sum().item()
        total_samples += labels.size(0)


    """
    At the end of each epoch, the average training loss and training accuracy
    are computed. These acts as models performance indicators.
    """

    train_loss = train_loss / len(trainDataLoader)
    train_acc = 100 * train_correct / total_samples
    train_loss_history.append(train_loss)
    train_acc_history.append(train_acc)

    # Testing phase
    model.eval()
    test_total = 0
    with torch.no_grad(): # disabled for efficiency
        for i, (images, labels) in enumerate(testDataLoader):
            images, labels = images, labels
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Calculate testing metrics
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            test_correct += (predicted == labels).sum().item()
            test_total += labels.size(0)
```

```
            test_total += labels.size(0)
```

Questions assigned to the following page: 3.2 and 3.1

```
    """
    As one epoch is been completed, that is, training and testing ohase as well,
    we calculate and display the test loss and test accuracy.
    These act as models performance indicators.
    """

    test_loss = test_loss / len(testDataLoader)
    test_acc = 100 * test_correct / test_total
    test_loss_history.append(test_loss)
    test_acc_history.append(test_acc)

    print(f'Epoch {epoch+1}:')
    print(f'Train Loss: {train_loss:.4f} | Test Loss: {test_loss:.4f}')
    print(f'Train Acc: {train_acc:.2f}% | Test Acc: {test_acc:.2f}%')
```

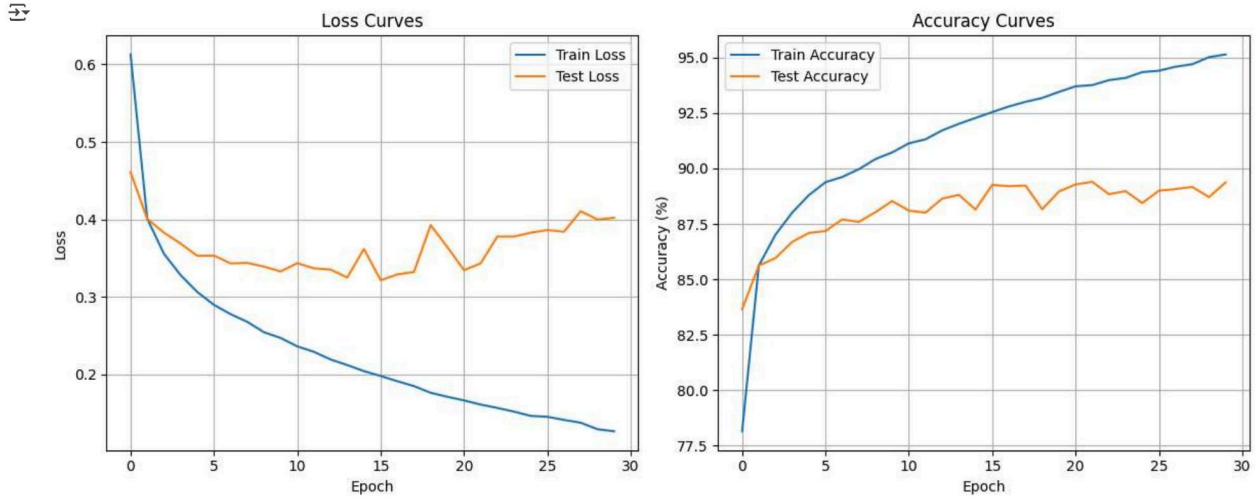
```
Train Acc: 91.13% | Test Acc: 88.10%
Epoch 12:
Train Loss: 0.2291 | Test Loss: 0.3369
Train Acc: 91.32% | Test Acc: 88.01%
Epoch 13:
Train Loss: 0.2193 | Test Loss: 0.3354
Train Acc: 91.72% | Test Acc: 88.64%
Epoch 14:
Train Loss: 0.2121 | Test Loss: 0.3251
Train Acc: 92.02% | Test Acc: 88.81%
Epoch 15:
Train Loss: 0.2042 | Test Loss: 0.3619
Train Acc: 92.28% | Test Acc: 88.15%
Epoch 16:
Train Loss: 0.1979 | Test Loss: 0.3216
Train Acc: 92.54% | Test Acc: 89.26%
Epoch 17:
Train Loss: 0.1912 | Test Loss: 0.3290
Train Acc: 92.80% | Test Acc: 89.20%
Epoch 18:
Train Loss: 0.1849 | Test Loss: 0.3322
Train Acc: 93.01% | Test Acc: 89.23%
Epoch 19:
Train Loss: 0.1763 | Test Loss: 0.3926
Train Acc: 93.18% | Test Acc: 88.16%
Epoch 20:
Train Loss: 0.1711 | Test Loss: 0.3644
Train Acc: 93.45% | Test Acc: 88.96%
Epoch 21:
Train Loss: 0.1665 | Test Loss: 0.3346
Train Acc: 93.71% | Test Acc: 89.28%
Epoch 22:
Train Loss: 0.1611 | Test Loss: 0.3433
Train Acc: 93.76% | Test Acc: 89.40%
Epoch 23:
Train Loss: 0.1568 | Test Loss: 0.3779
Train Acc: 93.98% | Test Acc: 88.84%
Epoch 24:
Train Loss: 0.1520 | Test Loss: 0.3779
Train Acc: 94.08% | Test Acc: 88.98%
Epoch 25:
Train Loss: 0.1465 | Test Loss: 0.3829
Train Acc: 94.35% | Test Acc: 88.44%
Epoch 26:
Train Loss: 0.1453 | Test Loss: 0.3863
Train Acc: 94.41% | Test Acc: 89.00%
Epoch 27:
Train Loss: 0.1413 | Test Loss: 0.3841
Train Acc: 94.59% | Test Acc: 89.07%
Epoch 28:
Train Loss: 0.1377 | Test Loss: 0.4106
Train Acc: 94.70% | Test Acc: 89.17%
Epoch 29:
Train Loss: 0.1293 | Test Loss: 0.4000
Train Acc: 95.02% | Test Acc: 88.71%
Epoch 30:
Train Loss: 0.1266 | Test Loss: 0.4022
Train Acc: 95.14% | Test Acc: 89.37%
```

```
"""
As given in the question:  we are plotting two plots to help visualize
model performance over training epochs.
1. The first plot shows how training and test loss.
2. The second plot shows how training and test accuracy.
"""

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)

plt.plot(train_loss_history, label='Train Loss')
plt.plot(test_loss_history, label='Test Loss')

plt.title('Loss Curves')
plt.xlabel('Epoch')
plt.ylabel('Loss')

plt.legend()
plt.grid(True)

# --------------------

plt.subplot(1, 2, 2)

plt.plot(train_acc_history, label='Train Accuracy')
plt.plot(test_acc_history, label='Test Accuracy')
```

```
plt.title('Accuracy Curves')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
```

Questions assigned to the following page: 3.3 and 3.2

```
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Inference drawn from Curves

- The training loss keeps on decreaseing steadily but test loss starts to fluctuate after a few epochs, **showing signs of potential overfitting**.
- IN case of accuracy curves, training accuracy improves consistently but **test accuracy plateaus around < 90%** after a few epochs and does not improve further –> This tells us that model's limit of generalization towards unseen data has been reached.
- The widening gap between training and test loss curves & divergence between training and test accuracy curves –> also gives us more hint on the overfitting of the model, since it starts to memorize training data rather than generalizing for unseeen data.

Final evaluation of the trained model on the test dataset.

Procedure:

1. The model iterates through the test dataset, predicting class labels.
2. The total number of correct predictions is accumulated.

Finally, the overall test accuracy is printed.

This step is crucial as itgives the final accuracy of the model for the test dataset>

```
model.eval()

correct = 0
total = 0

with torch.no_grad():

    for images, labels in testDataLoader:
        images, labels = images, labels
        outputs = model(images)

        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)

        correct += (predicted == labels).sum().item()

print(f'\nFinal Test Accuracy: {100 * correct / total:.2f}%')
```

```
Final Test Accuracy: 89.37%
```

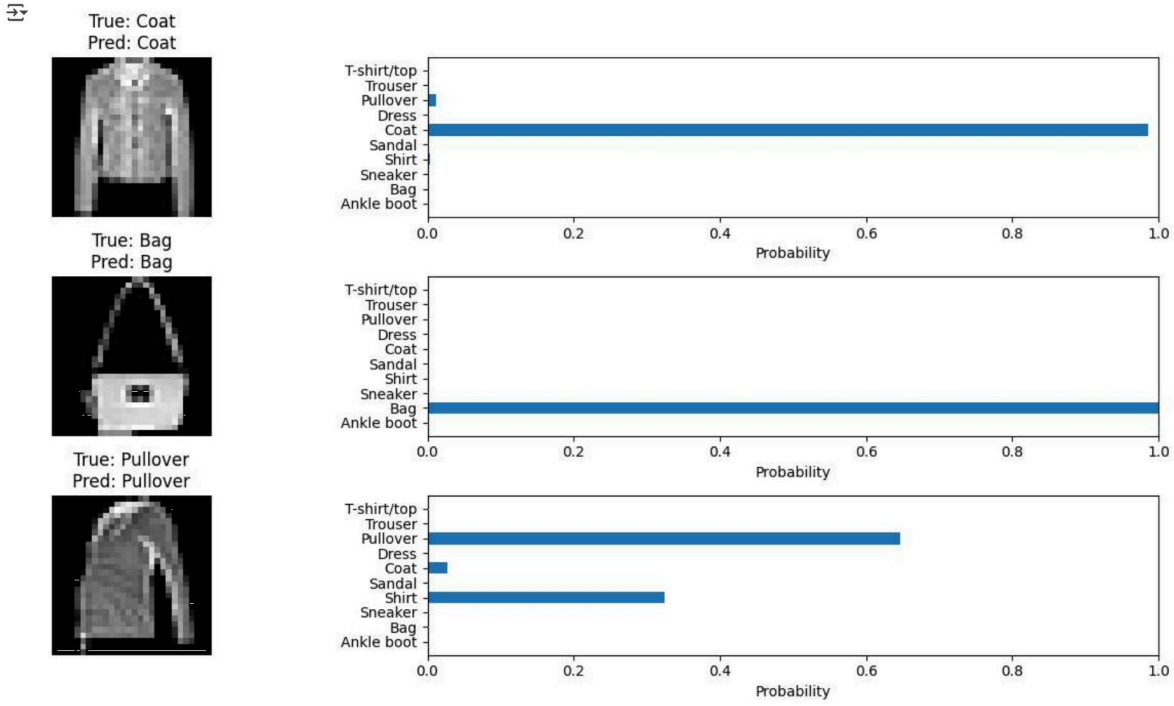
This following block of code selects three random test samples, predicts their classes using the trained model, and visualizes the predictions along with their class probabilities.

```
'''
A batch of test images is fetched and for that batch the predictions are
generated using the model, --> The model's predictions clubbed with the applied
softmax function give us the final probabilistic class probabilities.

These are then displyed using a horizontal bar chart along with the
original images and the actual and predicted class values.
```

```
'''

model.eval()
```

Question assigned to the following page: 3.3

```
test_images, test_labels = next(iter(testDataLoader))
with torch.no_grad():
    test_outputs = model(test_images)
    probabilities = torch.nn.functional.softmax(test_outputs, dim=1)
    _, predicted = torch.max(test_outputs.data, 1)

# Select 3 random samples
random_indices = torch.randperm(len(test_images))[:3]
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure(figsize=(14, 7))
for idx, sample_idx in enumerate(random_indices):

    # Get data
    image = test_images[sample_idx].squeeze().numpy()

    true_label = test_labels[sample_idx].item()

    probs = probabilities[sample_idx].numpy()

    # Plot image
    plt.subplot(3, 2, 2*idx+1)
    plt.imshow(image, cmap='gray')

    plt.title(f'True: {class_names[true_label]}\nPred: {class_names[predicted[sample_idx]]}')
    plt.axis('off')

    # Plot probabilities
    plt.subplot(3, 2, 2*idx+2)
    plt.barh(class_names, probs)

    plt.xlabel('Probability')
    plt.xlim([0, 1])

    plt.gca().invert_yaxis()  # Show highest probability at top



plt.tight_layout()
plt.show()
```



Comments

- **Most of the selected test samples** are **correctly classified**, showing that the model has effectively learned the distinguishing features of categories like **Bag, Sneaker, Sandal, Coat, and Shirt** and does not overfit and has good generalization capabilites.
- Some probability distributions reveal **minor likelihoods for alternative classes**, such as **Shirt vs. Pullover** or **Coat vs. Dress**, which are inherent as they are not mutually exclusive sets if objects, but overlapp features and shapes.
- Hence, even if there are deviations in the probabilty distribution for predictions, the model mostly manages to put out the correctly classified label.

```
print("GG")
```

GG

Question assigned to the following page: 4.1

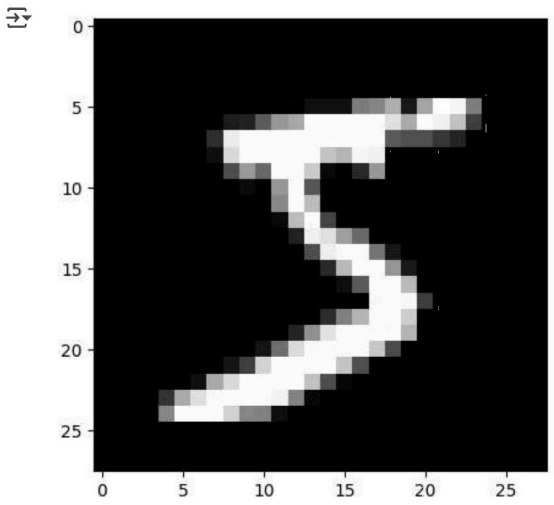
DL --> Assignment 1 --> Question 4

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))
```

```
def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
```

Questions assigned to the following page: 4.2 and 4.1

```
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance 1/ $\max(n_{in}, n_{out})$.

```
import math

# Initialize weights of each layer with a normal distribution of mean 0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with mean
# 0 and standard deviation close to 1 (if biases are initialized as 0, standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# ---------- Q1. Fill initialization code here. ----------

layer_dims = [784, 32, 32, 10]


weights = []
biases = []

for i in range(len(layer_dims)-1):
    n_in, n_out = layer_dims[i], layer_dims[i+1]
    std_dev = 1/np.sqrt(max(n_in, n_out))

    # He initialization as given in the task list
    # it is also a variant for sigmoid

    weights.append(rng.normal(
        scale=std_dev,
        size=(n_in, n_out)
    ))
    biases.append(np.zeros(n_out))
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

Important Note: Along with flattening, I am normalizing the pixel value and bringing them between 0 and 1.

```
def feed_forward_sample(sample, y):

  activations = [sample.flatten().astype(np.float32)/255]



  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """

  # ---------- Q2. Fill code here. ----------


  # OPerations consisiting of Hidden layers
  # We use sigmoid as the activation function
```

```
for i in range(len(weights)-1):
  z = activations[-1] @ weights[i] + biases[i]
```

Questions assigned to the following page: 4.2 and 4.3

```
    activations.append(sigmoid(z))

  # OPerations consisiting of Output Layer
  # We use SOFTMAX as the activation function

  z = activations[-1] @ weights[-1] + biases[-1]
  y_hat = softmax(z)


  """

  We convert the true label 'y' into a one-hot encoded vector --> Then calculate
  cross-entropy loss  --> and also extract the predicted class
  as the index of the HIGHEST probability.

  """

  loss = cross_entropy_loss(integer_to_one_hot(y, 10), y_hat)
  prediction = np.argmax(y_hat)

  return loss, integer_to_one_hot(prediction, 10)


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))


  # ---------- Q2. Fill code here to calculate losses, one_hot_guesses ----------

  for i in range(x.shape[0]):
    losses[i], one_hot_guesses[i] = feed_forward_sample(x[i], y[i])




  # ---------- rest of original code for metrics ----------

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1



  '''
    Step 1: Element-wise multiplication between:
    true labels (y_one_hot) and predicted labels (one_hot_guesses).

    Step 2: Sum the resulting values to count correctly classified samples.

  '''
  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")




  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0], "(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
Feeding forward all test data...

    Average loss: 2.36
    Accuracy (# of correct guesses): 1028.0 / 10000 ( 10.28 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

Questions assigned to the following page: 4.3 and 4.4

```
def train_one_sample(sample, y, learning_rate=0.003):

  # Normalizing the input by scaling pixel values between 0 and 1
  a = sample.flatten().astype(np.float32)/255

  activations = [a]
  zs = []

  # Q3. Forward Pass: This should be the same as what you did in feed_forward_sample above.

  for i in range(len(weights)):
      z = activations[-1] @ weights[i] + biases[i]
      zs.append(z)
      activations.append(sigmoid(z) if i < len(weights)-1 else softmax(z))




  # Q3. Backward pass: Implement backpropagation by backward-stepping gradients through each layer.
  # You may need to be careful to make sure your Jacobian matrices are the right shape.
  # At the end, you should get two vectors: weight_gradients and bias_gradients.

  delta = (activations[-1] - integer_to_one_hot(y, 10)).reshape(-1, 1)
  weight_grads = []
  bias_grads = []



  """

  Iterating in backwards direction by computing gradients.
  We compute them using the derivation chain rule.

  """
  for i in reversed(range(len(weights))):
      # Compute gradients
      weight_grad = activations[i].reshape(-1, 1) @ delta.T
      bias_grad = delta.flatten()

      # Store gradients
      weight_grads.insert(0, weight_grad)
      bias_grads.insert(0, bias_grad)

      # Update delta for next layer --> until its not input layer
      if i > 0:
          delta = (weights[i] @ delta) * dsigmoid(zs[i-1]).reshape(-1, 1)


  # Updating weights and biases

  for i in range(len(weights)):
      weights[i] -= learning_rate * weight_grads[i]
      biases[i] -= learning_rate * bias_grads[i]
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):

  print("Training for one epoch over the training dataset...")

  # ---------- Q4. Write the training loop over the epoch here. ----------


  '''

  Shuffling training data so that we get randomness

  '''

  permutation = rng.permutation(len(x_train))
  x_shuffled = x_train[permutation]
  y_shuffled = y_train[permutation]

  # Iterate through entire dataset
  for i in range(x_train.shape[0]):
      if i % 10000 == 0:
```

```
            print(f"Processed {i}/{x_train.shape[0]} samples")
        train_one_sample(x_shuffled[i], y_shuffled[i], learning_rate)
```

Question assigned to the following page: 4.4

```
  print("Finished training.\n")



feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data...

    Average loss: 2.36
    Accuracy (# of correct guesses): 1028.0 / 10000 ( 10.28 %)

    Training for one epoch over the training dataset...
    Processed 0/60000 samples
    Processed 10000/60000 samples
    Processed 20000/60000 samples
    Processed 30000/60000 samples
    Processed 40000/60000 samples
    Processed 50000/60000 samples
    Finished training.

    Feeding forward all test data...

    Average loss: 0.47
    Accuracy (# of correct guesses): 8718.0 / 10000 ( 87.18 %)

    Training for one epoch over the training dataset...
    Processed 0/60000 samples
    Processed 10000/60000 samples
    Processed 20000/60000 samples
    Processed 30000/60000 samples
    Processed 40000/60000 samples
    Processed 50000/60000 samples
    Finished training.

    Feeding forward all test data...

    Average loss: 0.32
    Accuracy (# of correct guesses): 9118.0 / 10000 ( 91.18 %)

    Training for one epoch over the training dataset...
    Processed 0/60000 samples
    Processed 10000/60000 samples
    Processed 20000/60000 samples
    Processed 30000/60000 samples
    Processed 40000/60000 samples
    Processed 50000/60000 samples
    Finished training.

    Feeding forward all test data...

    Average loss: 0.25
    Accuracy (# of correct guesses): 9292.0 / 10000 ( 92.92 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.

Attributes & Values for Satyrium spini (Spine Hairstreak):

- **Name**: Satyrium spini (Spine Hairstreak)
- **Wingspan**: 1.5 to 2 inches (3.8 to 5.1 cm)
- **Color**: Dark brown or black with blue and orange markings
- **Underside of Wings**: Mottled brown with orange and white markings
- **Distinctive Features**: Spine-like projection at the end of the hind wing
- **Habitat**: Forests, fields, and gardens
- **Nectar Sources**: Variety of flowers
- **Attracted to**: Bright, open areas

```
print ("GG")
```