Anil Poonai

Problem 1: Part A

$F(x) = ||X||_2^2$

$F(x) = ((\sum_{i=1}^{n} x_i^2)^{1/2})2$

$F(x) = \sum_{i=1}^{n} x_i^2$

Can ignore the summation as n is 1.

$\frac{\partial}{\partial x} = 2x$

Problem 1: Part B

$\sum_{i=1}^{n} ||x_i - \mu||_2^2$

$\frac{\partial}{\partial x} = 2(x - \mu)$ Got the derivative, could just put it in the answer for Part A but accounting for μ. This works because this is the derivative at each point, and we have the summation of all of the points already in the expression.

Now we set the derivative with summation $= 0$

$\sum_{i=1}^{n} 2(x_i - \mu) = 0$

Do some algebra and we end up with:

$\mu = (\sum_{i=1}^{n} x_i)/n$

Problem 2: Part A

$L(w) = ||x||_1$

$L(w) = ||Xw - y||_1$

Sizes: X(data) -> n*c, w(weights) -> c*1, y(labels) -> n*1

Problem 2: Part B

No, because L1 norm isn't differentiable at zero therefore gradient optimization cannot be used. The explanation is partly explained in Part C as well.

Problem 2: Part C

Part A was straightforward since we just need the difference from the estimate and actual value. Since L1 is the summation of $||x||$, we can just replace Xw-y for x. For part B, it wouldn't have a value

at zero if we took the derivative as it's non-convex there, there would be no unique global minimum so we can't minimize the loss function. This would mostly be concerned with the loss function part of the 3-step recipe as we are calculating the losses from the prediction regarding the label.

Problem 3: Part A

The code for this question is below, the answers I got for the parameters in this order W_{h1}, W_{h2}, W_{h3}, W_{h4}, b_{h1}, b_{h2}, b_{h3}, b_{h4}, W_{o1}, W_{o2}, W_{o3}, W_{o4}, b_0.

For Dataset 1:
([-7.82455998e-01, -5.16665284e-01, 1.32120087e+00, -3.55189485e+00, 6.25964798e+00, 3.09863545e+00, -1.05696070e+01, 1.42169031e+01, 2.55605428e+00, -7.76233237e+00, 1.51377436e+00, 1.12912342e+00, 2.69958647e-11]

For Dataset 2:
[-0.35614338, 0.05176055, 0.04679073, 4.26739128, 1.06854914, 0.69438611, 0.33182292, -21.3355306 , -5.61759265, -20.45828289, 1.24519253, 0.46882705, 19.79543786]

I also made an excel solver to calculate these values out for a more insightful look into the process.

For Dataset 1:

Parameters	Input	Z1	Z2	Z3	Z4	ReLu1	ReLu2	ReLu3	ReLu4	Prediction		Value for D1	Value for D2
w1 = -7.82E-01	0	6.25964798	3.09863545	-10.569607	14.2169031	6.25964798	3.09863545	0	14.2169031	8.000000004 =		8	0
w2 = -5.17E-01	1	5.477191982	2.581970166	-9.24840613	10.66500825	5.477191982	2.581970166	0	10.66500825	6 =		6	1
w3 = 1.32E+00	2	4.694735984	2.065304882	-7.92720526	7.1131134	4.694735984	2.065304882	0	7.1131134	3.999999995 =		4	2
w4 = -3.55E+00	3	3.912279986	1.548639598	-6.60600439	3.56121855	3.912279986	1.548639598	0	3.56121855	1.99999999 =		2	3
b1 = 6.26E+00	4	3.129823988	1.031974314	-5.28480352	0.0093237	3.129823988	1.031974314	0	0.0093237	-1.4339E-08 =		0	2
b2 = 3.10E+00	5	2.34736799	0.51530903	-3.96360265	-3.54257115	2.34736799	0.51530903	0	0	2.000000033 =		2	1
b3 = -1.06E+01	6	1.564911992	-0.00135625	-2.64240178	-7.094466	1.564911992	0	0	0	3.999999995 =		4	2
b4 = 1.42E+01	7	0.782455994	-0.51802154	-1.32120091	-10.6463609	0.782455994	0	0	0	1.999999992 =		2	3
w5 = 2.56E+00	8	-4E-09	-1.03468682	-4E-08	-14.1982557	0	0	0	0	2.69959E-11 =		0	4
w6 = -7.76E+00	9	-0.782456	-1.55135211	1.32120083	-17.7501506	0	0	1.32120083	0	1.999999941 =		2	5
w7 = 1.51E+00	10	-1.564912	-2.06801739	2.6424017	-21.3020454	0	0	2.6424017	0	3.999999942 =		4	6
w8 = 1.13E+00	11	-2.347368	-2.58468267	3.96360257	-24.8539403	0	0	3.96360257	0	5.999999944 =		6	
b0 = 2.70E-11	12	-3.129824	-3.10134796	5.28480344	-28.4058351	0	0	5.28480344	0	7.999999945 =		8	

For Dataset 2:

Parameters	Input	Z1	Z2	Z3	Z4	ReLu1	ReLu2	ReLu3	ReLu4	Prediction		Value for D1	Value for D2
w1 = -0.35614338	0	1.06854914	0.69438611	0.33182292	-21.3355306	1.06854914	0.69438611	0.33182292	0	1.29723E-08 =		8	0
w2 = 0.05176055	1	0.71240576	0.74614666	0.37861365	-17.0681393	0.71240576	0.74614666	0.37861365	0	0.99999994 =		6	1
w3 = 0.04679073	2	0.35626238	0.79790721	0.42540438	-12.800748	0.35626238	0.79790721	0.42540438	0	1.999999867 =		4	2
w4 = 4.26739128	3	0.000119	0.84966776	0.47219511	-8.53335676	0.000119	0.84966776	0.47219511	0	2.999999794 =		2	3
b1 = 1.06854914	4	-0.35602438	0.90142831	0.51898584	-4.26596548	0	0.90142831	0.51898584	0	1.99999978 =		0	2
b2 = 0.69438611	5	-0.71216776	0.95318886	0.56577657	0.0014258	0	0.95318886	0.56577657	0.0014258	0.999999727 =		2	1
b3 = 0.33182292	6	-1.06831114	1.00494941	0.6125673	4.26881708	0	1.00494941	0.6125673	4.26881708	1.999999685 =		4	2
b4 = -21.3355306	7	-1.42445452	1.05670996	0.65935803	8.53620836	0	1.05670996	0.65935803	8.53620836	2.999999643 =		2	3
w5 = -5.61759265	8	-1.7805979	1.10847051	0.70614876	12.80359964	0	1.10847051	0.70614876	12.80359964	3.999999601 =		0	4
w6 = -20.4582829	9	-2.13674128	1.16023106	0.75293949	17.07099092	0	1.16023106	0.75293949	17.07099092	4.999999559 =		2	5
w7 = 1.24519253	10	-2.49288466	1.21199161	0.79973022	21.3383822	0	1.21199161	0.79973022	21.3383822	5.999999517 =		4	6
w8 = 0.46882705	11	-2.84902804	1.26375216	0.84652095	25.60577348	0	1.26375216	0.84652095	25.60577348	6.999999475 =		6	
b0 = 19.79543786	12	-3.20517142	1.31551271	0.89331168	29.87316476	0	1.31551271	0.89331168	29.87316476	7.999999433 =		8	

The predicted values are really close to the actual values on all accounts.

Problem 3: Part B

This is just mean squares, so the derivative is straightforward:

$$L(\vec{\theta}) = \sum_{i=1}^{n} (y_i - f(x_i, \vec{\Theta}))^2$$

$$\nabla L(\vec{\theta}) = -2 \sum_{i=1}^{n} \quad ((y_i - f(x_i, \vec{\Theta})) * \nabla f(x_i, \vec{\Theta}))$$

I left it in terms of $\nabla f(x_i, \vec{\Theta})$ since that's what the question asked for.

Problem 3: Part C

First layer values after plugging in the input(x) and parameters.

Z_1 = -1

Z_2 = 3

Z_3 = 1

Z_4 = -1

After ReLu

Z_1 = 0

Z_2 = 3

Z_3 = 1

Z_4 = 0

After summation in output layer

-4

After adding final bias

-3

Did this both manually and on the excel calculator I made:

P3. C:

$$Z_i = w_{hi} x + b_{hi}$$

relu

$$Y = b_0 + \sum_{i=1}^{4} w_{0i} z_i$$

$$x = 2$$

$Z_1 = -1 \to 0 \to 0$

$Z_2 = 3 \to 3 \to$ ->

$Z_3 = 1 \to 1 \to -1$

$Z_4 = -1 \to 0 \to 0$

-4

-3

	Parameters				Input		Z1	Z2	Z3	Z4		ReLu1	ReLu2	ReLu3	ReLu4		Prediction
w1 =	-1				0		1	1	-1	1		1	1	0	1		0
w2 =	1				1		0	2	0	0		0	2	0	0		-1
w3 =	1				2		-1	3	1	-1		0	3	1	0		-3
w4 =	-1				3		-2	4	2	-2		0	4	2	0		-5

Problem 3: Part D

I have all the formulas written out but am not sure what the parameters are supposed to be. But once I'm given those, I can figure out what the derivative at x=2 is with no problem.

$$\frac{\partial L}{\partial w0} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial w0}$$

$$\frac{\partial y}{\partial w0} = \sum_{i=1}^{4} z_i$$

$$\frac{\partial L}{\partial y} = 2 \sum_{i=1}^{n} (y0 - f(x, \theta))$$

$$\frac{\partial L}{\partial w0} = 2 * \sum_{i=1}^{4} z * \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial b0} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial b0}$$

$$\frac{\partial y}{\partial b0} = 1$$

$$\frac{\partial L}{\partial y} = 2 \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial b0} = 2 \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial w1} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial z} * max(0,z) * \frac{\partial z}{\partial wo}$$

$$\frac{\partial z}{\partial wo} = x -> max(0,z) = max(0,x)$$

$$\frac{\partial y}{\partial z} = \sum_{i=1}^{4} wo_i$$

$$\frac{\partial L}{\partial y} = 2 \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial w1} 2 * \sum_{i=1}^{4} wo * max(0,x) * \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial b1} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial z} * max(0,z) * \frac{\partial z}{\partial bo}$$

$$\frac{\partial z}{\partial bo} = 1 -> max(0,z) = 1$$

$$\frac{\partial y}{\partial z} = \sum_{i=1}^{4} wo_i$$

$$\frac{\partial L}{\partial y} = 2 \sum_{i=1}^{n}(y0 - f(x,\theta))$$

$$\frac{\partial L}{\partial w1} 2 * \sum_{i=1}^{4} wo * \sum_{i=1}^{n}(y0 - f(x,\theta))$$

hw1p3

September 26, 2023

```
[ ]: import numpy as np
     from scipy.optimize import fsolve
```

Possible to have different answers for these datasets, but they will all be pretty close to making the dataset work

Dataset 1

```
[2]: def f(p): #Formula's followed in the neural network plus the addition of␣
     ↪subtracting the outout value, as fsolve revolves around making the equation␣
     ↪equal to 0
         input = [0,1,2,3,4,5,6,7,8,9,10,11,12]
         output = [8, 6, 4, 2, 0, 2, 4, 2, 0, 2, 4, 6, 8]
         return [p[12] + p[8]*max(0,p[0]*input[0] + p[4]) + p[9]*max(0,p[1]*input[0]␣
     ↪+ p[5]) + p[10]*max(0,p[2]*input[0] + p[6]) + p[11]*max(0,p[3]*input[0] +␣
     ↪p[7]) - output[0],
             p[12] + p[8]*max(0,p[0]*input[1] + p[4]) + p[9]*max(0,p[1]*input[1] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[1] + p[6]) + p[11]*max(0,p[3]*input[1] + p[7]) -␣
     ↪output[1],
             p[12] + p[8]*max(0,p[0]*input[2] + p[4]) + p[9]*max(0,p[1]*input[2] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[2] + p[6]) + p[11]*max(0,p[3]*input[2] + p[7]) -␣
     ↪output[2],
             p[12] + p[8]*max(0,p[0]*input[3] + p[4]) + p[9]*max(0,p[1]*input[3] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[3] + p[6]) + p[11]*max(0,p[3]*input[3] + p[7]) -␣
     ↪output[3],
             p[12] + p[8]*max(0,p[0]*input[4] + p[4]) + p[9]*max(0,p[1]*input[4] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[4] + p[6]) + p[11]*max(0,p[3]*input[4] + p[7]) -␣
     ↪output[4],
             p[12] + p[8]*max(0,p[0]*input[5] + p[4]) + p[9]*max(0,p[1]*input[5] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[5] + p[6]) + p[11]*max(0,p[3]*input[5] + p[7]) -␣
     ↪output[5],
             p[12] + p[8]*max(0,p[0]*input[6] + p[4]) + p[9]*max(0,p[1]*input[6] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[6] + p[6]) + p[11]*max(0,p[3]*input[6] + p[7]) -␣
     ↪output[6],
             p[12] + p[8]*max(0,p[0]*input[7] + p[4]) + p[9]*max(0,p[1]*input[7] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[7] + p[6]) + p[11]*max(0,p[3]*input[7] + p[7]) -␣
     ↪output[7],
```

```
    p[12] + p[8]*max(0,p[0]*input[8] + p[4]) + p[9]*max(0,p[1]*input[8] + p[5])
↪+ p[10]*max(0,p[2]*input[8] + p[6]) + p[11]*max(0,p[3]*input[8] + p[7]) -
↪output[8],
    p[12] + p[8]*max(0,p[0]*input[9] + p[4]) + p[9]*max(0,p[1]*input[9] + p[5])
↪+ p[10]*max(0,p[2]*input[9] + p[6]) + p[11]*max(0,p[3]*input[9] + p[7]) -
↪output[9],
    p[12] + p[8]*max(0,p[0]*input[10] + p[4]) + p[9]*max(0,p[1]*input[10] +
↪p[5]) + p[10]*max(0,p[2]*input[10] + p[6]) + p[11]*max(0,p[3]*input[10] +
↪p[7]) - output[10],
    p[12] + p[8]*max(0,p[0]*input[11] + p[4]) + p[9]*max(0,p[1]*input[11] +
↪p[5]) + p[10]*max(0,p[2]*input[11] + p[6]) + p[11]*max(0,p[3]*input[11] +
↪p[7]) - output[11],
    p[12] + p[8]*max(0,p[0]*input[12] + p[4]) + p[9]*max(0,p[1]*input[12] +
↪p[5]) + p[10]*max(0,p[2]*input[12] + p[6]) + p[11]*max(0,p[3]*input[12] +
↪p[7]) - output[12]]
```

[3]:
```python
initial_guess = np.random.normal(size =13)
solution = fsolve(f, initial_guess)
while not(np.all(np.isclose(f(solution),[0]*13)==True)):
  new_guess = np.random.normal(size =13)

  solution = fsolve(f, new_guess)
solution
```

```
C:\Users\Anil\AppData\Local\Programs\Python\Python310\lib\site-
packages\scipy\optimize\_minpack_py.py:177: RuntimeWarning: The iteration is not
making good progress, as measured by the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
C:\Users\Anil\AppData\Local\Programs\Python\Python310\lib\site-
packages\scipy\optimize\_minpack_py.py:177: RuntimeWarning: The iteration is not
making good progress, as measured by the
  improvement from the last five Jacobian evaluations.
  warnings.warn(msg, RuntimeWarning)
```

[3]:
```
array([-7.82455998e-01, -5.16665284e-01,  1.32120087e+00, -3.55189485e+00,
        6.25964798e+00,  3.09863545e+00, -1.05696070e+01,  1.42169031e+01,
        2.55605428e+00, -7.76233237e+00,  1.51377436e+00,  1.12912342e+00,
        2.69958647e-11])
```

[4]:
```python
f(solution) #Closer to 0 the better
```

[4]:
```
[-1.751665479332587e-11,
 -2.9530156098189764e-11,
 -4.154543375989306e-11,
 -5.356071142159635e-11,
 -6.557229412229582e-11,
```

```
     -1.637445734559151e-11,
     -6.759481863127803e-12,
     -1.2145839889399213e-12,
     2.6995864717007652e-11,
     3.0686564400639327e-13,
     2.851052727237402e-13,
     2.6290081223123707e-13,
     2.39808173319038e-13]
```

Dataset 2

```
[5]: def f(p): #Added 2 repetitive lines at the bottom of this one so that the shape␣
     ↪would line up but it doesn't affect the actual results
         input = [0,1,2,3,4,5,6,7,8,9,10]
         output = [0,1,2,3,2,1,2,3,4,5,6]
         return [p[12] + p[8]*max(0,p[0]*input[0] + p[4]) + p[9]*max(0,p[1]*input[0]␣
     ↪+ p[5]) + p[10]*max(0,p[2]*input[0] + p[6]) + p[11]*max(0,p[3]*input[0] +␣
     ↪p[7]) - output[0],
         p[12] + p[8]*max(0,p[0]*input[1] + p[4]) + p[9]*max(0,p[1]*input[1] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[1] + p[6]) + p[11]*max(0,p[3]*input[1] + p[7]) -␣
     ↪output[1],
         p[12] + p[8]*max(0,p[0]*input[2] + p[4]) + p[9]*max(0,p[1]*input[2] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[2] + p[6]) + p[11]*max(0,p[3]*input[2] + p[7]) -␣
     ↪output[2],
         p[12] + p[8]*max(0,p[0]*input[3] + p[4]) + p[9]*max(0,p[1]*input[3] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[3] + p[6]) + p[11]*max(0,p[3]*input[3] + p[7]) -␣
     ↪output[3],
         p[12] + p[8]*max(0,p[0]*input[4] + p[4]) + p[9]*max(0,p[1]*input[4] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[4] + p[6]) + p[11]*max(0,p[3]*input[4] + p[7]) -␣
     ↪output[4],
         p[12] + p[8]*max(0,p[0]*input[5] + p[4]) + p[9]*max(0,p[1]*input[5] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[5] + p[6]) + p[11]*max(0,p[3]*input[5] + p[7]) -␣
     ↪output[5],
         p[12] + p[8]*max(0,p[0]*input[6] + p[4]) + p[9]*max(0,p[1]*input[6] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[6] + p[6]) + p[11]*max(0,p[3]*input[6] + p[7]) -␣
     ↪output[6],
         p[12] + p[8]*max(0,p[0]*input[7] + p[4]) + p[9]*max(0,p[1]*input[7] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[7] + p[6]) + p[11]*max(0,p[3]*input[7] + p[7]) -␣
     ↪output[7],
         p[12] + p[8]*max(0,p[0]*input[8] + p[4]) + p[9]*max(0,p[1]*input[8] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[8] + p[6]) + p[11]*max(0,p[3]*input[8] + p[7]) -␣
     ↪output[8],
         p[12] + p[8]*max(0,p[0]*input[9] + p[4]) + p[9]*max(0,p[1]*input[9] + p[5])␣
     ↪+ p[10]*max(0,p[2]*input[9] + p[6]) + p[11]*max(0,p[3]*input[9] + p[7]) -␣
     ↪output[9],
```

```
    p[12] + p[8]*max(0,p[0]*input[10] + p[4]) + p[9]*max(0,p[1]*input[10] +↵
↪p[5]) + p[10]*max(0,p[2]*input[10] + p[6]) + p[11]*max(0,p[3]*input[10] +↵
↪p[7]) - output[10],
    p[12] + p[8]*max(0,p[0]*input[10] + p[4]) + p[9]*max(0,p[1]*input[10] +↵
↪p[5]) + p[10]*max(0,p[2]*input[10] + p[6]) + p[11]*max(0,p[3]*input[10] +↵
↪p[7]) - output[10],
    p[12] + p[8]*max(0,p[0]*input[10] + p[4]) + p[9]*max(0,p[1]*input[10] +↵
↪p[5]) + p[10]*max(0,p[2]*input[10] + p[6]) + p[11]*max(0,p[3]*input[10] +↵
↪p[7]) - output[10]]
```

[6]:
```
initial_guess = np.random.normal(size =13)
solution = fsolve(f, initial_guess)
while not(np.all(np.isclose(f(solution),[0]*13)==True)):
  new_guess = np.random.normal(size =13)

  solution = fsolve(f, new_guess)
solution
```

[6]:
```
array([ -0.35614338,    0.05176055,    0.04679073,    4.26739128,
          1.06854914,    0.69438611,    0.33182292, -21.3355306 ,
         -5.61759265, -20.45828289,    1.24519253,    0.46882705,
         19.79543786])
```

[7]: `f(solution)`

[7]:
```
[4.218847493575595e-15,
 7.105427357601002e-15,
 6.217248937900877e-15,
 3.552713678800501e-15,
 8.881784197001252e-16,
 4.440892098500626e-16,
 0.0,
 1.7763568394002505e-15,
 2.6645352591003757e-15,
 -8.881784197001252e-16,
 1.7763568394002505e-15,
 1.7763568394002505e-15,
 1.7763568394002505e-15]
```

[ ]:

hw1p4

September 21, 2023

```
[1]: import numpy as np
     import torch
     import torchvision
     import matplotlib.pyplot as plt
```

```
[2]: 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 ())
```

```
[3]: print(len(trainingdata)) #Training data length
     print(len(testdata)) #Testing data length
```

```
60000
10000
```

```
[4]: trainDataLoader = torch.utils.data.
      ↪DataLoader(trainingdata,batch_size=64,shuffle=True) #Making it iterable
     testDataLoader = torch.utils.data.
      ↪DataLoader(testdata,batch_size=64,shuffle=False)
     images, labels = next(iter(trainDataLoader))
     print(images.shape)
     print(labels.shape)
```

```
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

```
[5]: class Model(torch.nn.Module):
       def __init__(self):
         super(Model, self).__init__()
         self.linear = torch.nn.Linear(784, 256) #764 inputs and 256 outputs
         self.activation = torch.nn.ReLU() #Activation function
         self.linear2 = torch.nn.Linear(256, 128)
         self.activation2 = torch.nn.ReLU()
         self.linear3 = torch.nn.Linear(128, 64)
         self.activation3 = torch.nn.ReLU()
         self.linear4 = torch.nn.Linear(64, 10) #10 categories
```

```python
    def forward(self, x):
        x = x.view(-1, 28*28) #Vectorize
        x = self.linear(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.activation2(x)
        x = self.linear3(x)
        x = self.activation3(x)
        x = self.linear4(x)
        return x
```

```python
[6]: model = Model()
     loss = torch.nn.CrossEntropyLoss()
     optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```python
[7]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') #Mine␣
      ↪isn't setup as I have a new computer
     #Note to follow this guide for setup: https://youtu.be/hHWkvEcDBOO?
      ↪si=MnJQsm7fIhbEes74
     model = model.to(device)
```

```python
[8]: train_losses = []
     test_losses = []
```

```python
[9]: for epoch in range(10): # We'll train for 10 "epochs"
         train_loss = 0
         test_loss = 0

         for data in testDataLoader: #Predicts test data
             images, labels = data
             images, labels = images.to(device), labels.to(device)
             predicted_output = model(images)
             fit = loss(predicted_output, labels)
             test_loss += fit.item()

         for data in trainDataLoader: #Predicts train data
             images, labels = data
             images, labels = images.to(device), labels.to(device)
             optimizer.zero_grad() # Zero out the gradient values
             predicted_output = model(images)
             fit = loss(predicted_output, labels) # Measure how well the predicted␣
      ↪output matches the labels
             fit.backward() # Compute the gradient of the fit with respect to the model␣
      ↪parameters
             optimizer.step() # Update the weights in the model using gradient descent
             train_loss += fit.item()
```

```
    train_losses += [train_loss/len(trainDataLoader)]
    test_losses += [test_loss/len(testDataLoader)]

    print(f'Epoch {epoch}, Train loss {train_loss}, Test loss {test_loss}')
```

```
Epoch 0, Train loss 1787.3244709968567, Test loss 362.3181862831116
Epoch 1, Train loss 875.131316781044, Test loss 184.43580251932144
Epoch 2, Train loss 657.0507292747498, Test loss 127.7506094276905
Epoch 3, Train loss 570.6527071595192, Test loss 104.46469113230705
Epoch 4, Train loss 523.0395578444004, Test loss 94.95994547009468
Epoch 5, Train loss 489.00073251128197, Test loss 112.74039062857628
Epoch 6, Train loss 464.0913973748684, Test loss 85.38151663541794
Epoch 7, Train loss 446.2029498517513, Test loss 81.49660468101501
Epoch 8, Train loss 429.4090850651264, Test loss 90.68424804508686
Epoch 9, Train loss 416.1134061217308, Test loss 79.3811206817627
```

```python
[10]: plt.plot(range(10),train_losses, label='Train error')
      plt.plot(range(10),test_losses, label='Test error')
      plt.xlabel('Epoch')
      plt.ylabel('Loss')
      plt.legend()
      plt.show()
```
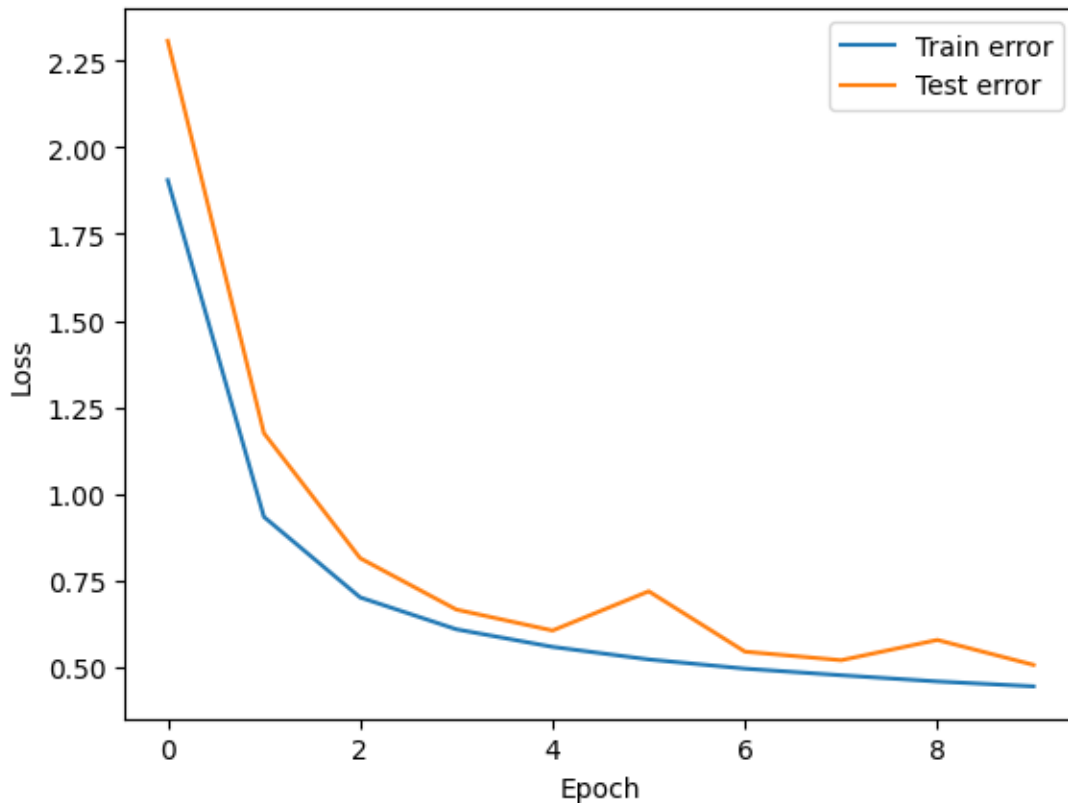
```
[11]: predicted_classes = torch.max(predicted_output, 1)[1] #Isn't perfect
      print('Predicted:', predicted_classes)
      print('Labels:', labels)
```

```
Predicted: tensor([0, 3, 7, 1, 9, 8, 4, 3, 0, 2, 5, 6, 9, 0, 9, 2, 1, 1, 9, 7,
6, 8, 2, 2,
        0, 3, 2, 0, 7, 4, 7, 0])
Labels: tensor([6, 3, 7, 1, 9, 8, 4, 3, 0, 2, 5, 0, 9, 0, 9, 6, 1, 1, 7, 7, 6,
8, 2, 2,
        0, 3, 4, 0, 7, 4, 7, 0])
```

```
[12]: i = 30
      j = 20
      k = 10
```

```
[13]: print('Predicted:', predicted_classes[i].item())
      print('Labels:', labels[i].item())
      plt.imshow(images[i].squeeze().cpu()) # Visualize iamge
      plt.show()
```
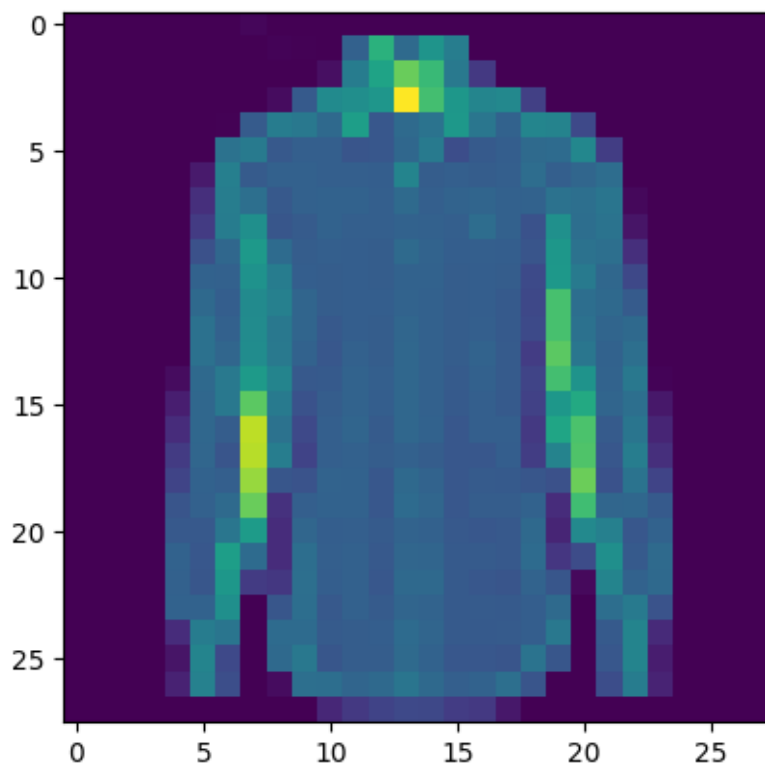
```
Predicted: 7
Labels: 7
```
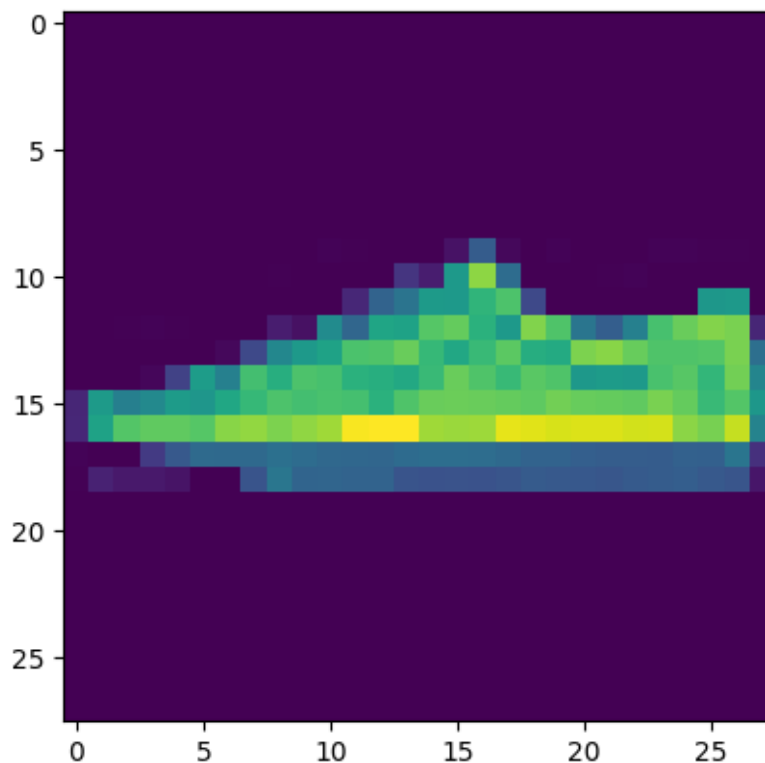
The prediction for item 30 is correct since it was categorized as a shoe and it is predicted to be a shoe.

```
[17]: print('Predicted:', predicted_classes[j].item())
      print('Labels:', labels[j].item())
      plt.imshow(images[j].squeeze().cpu()) # Visualize iamge
      plt.show()
```
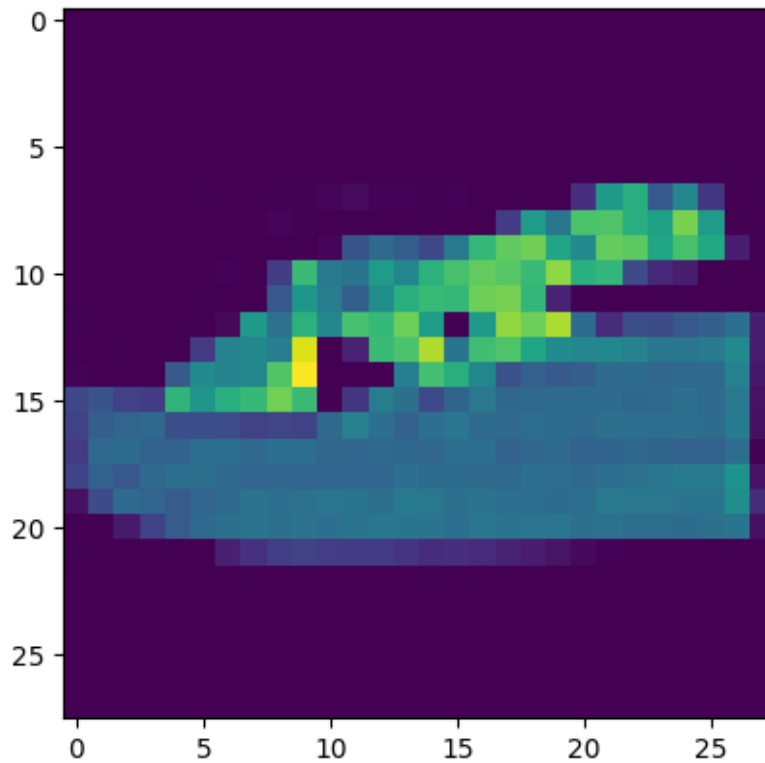
```
Predicted: 6
Labels: 6
```


The prediction for item 20 is correct since it was categorized as a shirt and it is predicted to be a shirt.

```
[18]: print('Predicted:', predicted_classes[k].item())
      print('Labels:', labels[k].item())
      plt.imshow(images[k].squeeze().cpu()) # Visualize iamge
      plt.show()
```
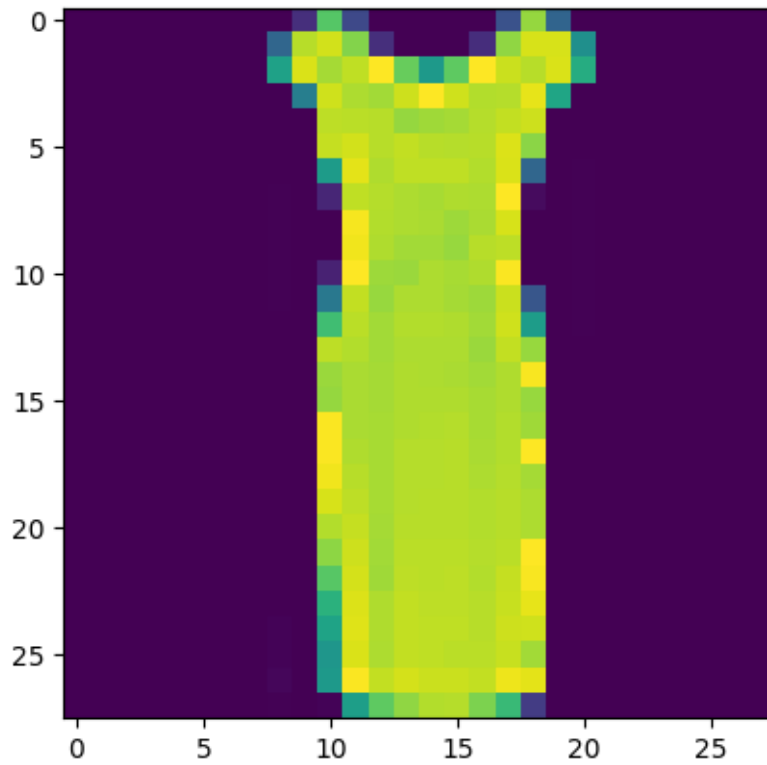
```
Predicted: 5
Labels: 5
```

The prediction for item 10 is correct since although I do not know what it is.

```
[20]: x = 1
      print('Predicted:', predicted_classes[x].item())
      print('Labels:', labels[x].item())
      plt.imshow(images[x].squeeze().cpu()) # Visualize iamge
      plt.show()
```

```
Predicted: 3
Labels: 3
```

The prediction for item 1 is correct since it was categorized as a dress and it is predicted to be a dress.

[ ]:

dlf23-hw1-prob5

September 21, 2023

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.

```python
[7]: 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") #Loading dataset

plt.imshow(x_train[0],cmap='gray'); #One of the data points
```
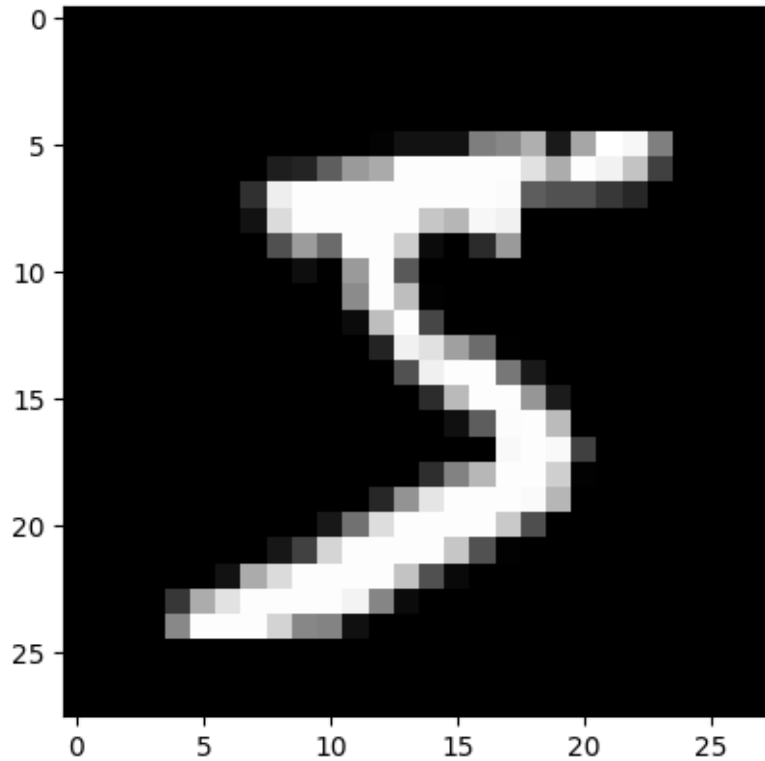
Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

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.

I understand what is happening here but I found this demo helped a lot: https://www.youtube.com/watch?v=w8yWXqWQYmU

Everything I referenced for help outside of this course is from that video.

```python
import numpy as np

def sigmoid(x): #For hidden layers
    # 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))
    )
```

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

def softmax(x): #For output layer
  # 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): #Loss function
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max): #Categorical to numerical
  # x: integer to convert to one hot encoding
  # 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 $1/\max(n_{in}, n_{out})$.

```python
[9]: 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.
     # ...

     weights = []
     biases = []
```

```
l1w=rng.normal(loc=0,scale=1/math.sqrt(784),size=(784,32)) #784 input and 32␣
  ↪output
l1b=np.zeros(32) #A bunch of zeroes
l2w=rng.normal(loc=0,scale=1/math.sqrt(32),size=(32,32))
l2b=np.zeros(32)
l3w=rng.normal(loc=0,scale=1/math.sqrt(32),size=(32,10)) #10 outputs
l3b=np.zeros(10)


weights.append(l1w)
weights.append(l2w)
weights.append(l3w)
biases.append(l1b)
biases.append(l2b)
biases.append(l3b)
```

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).

[10]:
```python
def feed_forward_sample(sample, y):
  """ 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.
  # ...
  dummy=sample
  for i in range(len(weights)): #Each weight set
    z=np.matmul(dummy,weights[i])+biases[i] #dataset times weights plus biases
    if i==(len(weights)-1): #last weight set, output layer
        activation=softmax(z)
        output=activation
    else:
        activation=sigmoid(z)
    dummy=activation #Updates input for loop

  one_hot=integer_to_one_hot(y,10) #Make categorical data numerical
  loss=cross_entropy_loss(one_hot,output) #Loss function
  two_hot=integer_to_one_hot(np.argmax(output),10)
```

4

```python
    return loss, two_hot


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]): #Makes images into vectors
      dummy=x[i].reshape(-1)
      label=y[i]
      loss,one_hot=feed_forward_sample(dummy,label)
      losses[i]=loss
      one_hot_guesses[i]=one_hot

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

  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: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 (8.80 %)

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.

```python
[11]: def train_one_sample(sample, y, learning_rate=0.003):
        a = sample.flatten()

        # We will store each layer's activations to calculate gradient
        activations = []

        # Forward pass

        # Q3. This should be the same as what you did in feed_forward_sample above.
        # ...
        weight_gradients=[None]*3 #3 of each
        bias_gradients=[None]*3
        # Forward pass
        dummy=a
        for i in range(len(weights)):
          dummy2=np.matmul(dummy,weights[i])+biases[i] #Same as before just with
        ↪activations
          if i==(len(weights)-1):
              activation=softmax(dummy2)
              output=activation
          else:
              activation=sigmoid(dummy2)
          dummy=activation
          activations.append(dummy)

      # Backward pass
        one_hot=integer_to_one_hot(y,10)

        for index in range(len(weights)): #Getting the gradients, derivatives are
        ↪written out
            i=len(weights)-index-1
            if index==0:
                dl_dz=activations[i]-one_hot #For output layer
            else:
                dl_dac=np.matmul(weights[i+1],dl_dz.T).reshape(-1) #Hidden layers
                dl_dz=dl_dac*(activations[i]*(1-activations[i]))


            if i==0: #To the end
                dl_dw=np.matmul(a[:,np.newaxis],dl_dz[np.newaxis,:]) #First layer
            else:
                dl_dw=np.matmul(activations[i-1][:,np.newaxis],dl_dz[np.newaxis,:])
        ↪#Others
```

```
        dl_db=dl_dz
        weight_gradients[i]=dl_dw
        bias_gradients[i]=dl_db


    # Q3. 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.
    # ...


    # Update weights & biases based on your calculated gradient
    for i in range(len(weights)):
        weights[i] -= weight_gradients[i] * learning_rate
        biases[i] -= bias_gradients[i].flatten() * learning_rate
```

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

Q4. Train your model for 3 epochs.

```
[12]: 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.
        # ...
        for j in range(x_train.shape[0]): #3 epochs
            train_one_sample(x_train[j],y_train[j],learning_rate=0.003)
        feed_forward_training_data()

        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: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)

Training for one epoch over the training dataset…
```

```
Feeding forward all training data…

Average loss: 1.04
Accuracy (# of correct guesses): 39068.0 / 60000 ( 65.11 %)

Finished training.

Feeding forward all test data…

Average loss: 1.04
Accuracy (# of correct guesses): 6505.0 / 10000 ( 65.05 %)

Training for one epoch over the training dataset…
Feeding forward all training data…

Average loss: 0.84
Accuracy (# of correct guesses): 43749.0 / 60000 ( 72.91 %)

Finished training.

Feeding forward all test data…

Average loss: 0.83
Accuracy (# of correct guesses): 7330.0 / 10000 ( 73.30 %)

Training for one epoch over the training dataset…
Feeding forward all training data…

Average loss: 0.9
Accuracy (# of correct guesses): 41467.0 / 60000 ( 69.11 %)

Finished training.

Feeding forward all test data…

Average loss: 0.88
Accuracy (# of correct guesses): 6994.0 / 10000 ( 69.94 %)
```

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.

[ ]: