"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

x

I went to office hours and helped/got help from many different people.

4.1.1:

4.1)

$$\frac{\partial Y}{\partial z} = \begin{cases} 0 & z < 0 \\ 1 & \text{otherwise} \end{cases}$$

$$Y = \sigma_{RELU}(z)$$

$$\frac{\partial Y}{\partial z} = \sigma'_{Relu}(z)$$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial z} = \frac{\partial L}{\partial Y} \cdot \sigma'_{RELU}(z)$$

$$\boxed{\frac{\partial L}{\partial z} = \frac{\partial L}{\partial Y} \cdot \sigma'_{RELU}(z)}$$

4.1.2:

```python
#GOOD
def forward(self, Z: np.ndarray) -> np.ndarray:
    """Forward pass for relu activation:
    f(z) = z if z >= 0
           0 otherwise

    Parameters
    ----------
    Z  input pre-activations (any shape)

    Returns
    -------
    f(z) as described above applied elementwise to `Z`
    """
    ### YOUR CODE HERE ###
    return np.maximum(0,Z)

#GOOD
def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
    """Backward pass for relu activation.

    Parameters
    ----------
    Z   input to `forward` method
    dY  gradient of loss w.r.t. the output of this layer
        same shape as `Z`

    Returns
    -------
    gradient of loss w.r.t. input of this layer
    """
    ### YOUR CODE HERE ###
    a = Z > 0
    b = dY * a
    return b
```

4.2.1:

4.2)

$$\frac{\partial L}{\partial W} = \frac{\partial Z}{\partial W} \cdot \frac{\partial L}{\partial Z} = \boxed{X^T \frac{\partial L}{\partial Z}}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial b} = \frac{\partial L}{\partial Z} = \boxed{\frac{\partial \ell}{\partial Z}}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial X} = \boxed{\frac{\partial L}{\partial Z} W^T}$$

4.2.2:

```python
#GOOD
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: multiply by a weight matrix, add a bias, apply activation.
    Also, store all necessary intermediate results in the `cache` dictionary
    to be able to compute the backward pass.

    Parameters
    ----------
    X  input matrix of shape (batch_size, input_dim)

    Returns
    -------
    a matrix of shape (batch_size, output_dim)
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

    ### BEGIN YOUR CODE ###
    Z = np.matmul(X, self.parameters["W"])
    Z = Z + + self.parameters["b"]
    a = self.activation(Z)
    self.cache = {"X" : X, "Z" : Z}
    ### END YOUR CODE ###

    return a

#GOOD
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for fully connected layer.
    Compute the gradients of the loss with respect to:
        1. the weights of this layer (mutate the `gradients` dictionary)
        2. the bias of this layer (mutate the `gradients` dictionary)
        3. the input of this layer (return this)

    Parameters
    ----------
    dLdY  gradient of the loss with respect to the output of this layer
          shape (batch_size, output_dim)

    Returns
    -------
    gradient of the loss with respect to the input of this layer
    shape (batch_size, input_dim)
    """
    ### BEGIN YOUR CODE ###
    X, Z = self.cache["X"], self.cache["Z"]
    XT = X.T
    a = self.activation.backward(Z, dLdY)
    b = a.dot(self.parameters["W"].T)
    c = np.sum(a, axis=0, keepdims=True)
    d = XT.dot(a)
    self.gradients = {"W" : d, "b" : c}
    ### END YOUR CODE ###

    return b
```

4.3.1:

$$4.3)\quad \sigma_i = \frac{e^{s_i}}{\sum_{j=1}^{K} e^{s_j}}$$

<u>On diagonal</u>

$$\frac{\partial \sigma_i}{\partial s_j} = \frac{\partial}{\partial s_i}\left(\frac{e^{s_i - m}}{\sum_{i=1}^{K} e^{s_i - m}} \right)$$

$$= \sigma_i (1 - \sigma_i)$$

<u>Off diagonal</u>

$$\frac{e^{s_i - m} \sum_{j=1}^{K} e^{s_j - m} - e^{s_i - m} e^{s_j - m}}{\left(\sum_{j=1}^{K} e^{s_j - m}\right)^2}$$

$$\boxed{-\sigma_i \sigma_j}$$

since off diagonal cancels terms from numerator

4.3.2:

```python
class SoftMax(Activation):
    def __init__(self):
        super().__init__()

    #GOOD
    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for softmax activation.
        Hint: The naive implementation might not be numerically stable.

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        ### YOUR CODE HERE ###
        a = np.max(Z, axis=1, keepdims=True)
        b = Z - a
        c = np.exp(b)
        d = c / np.sum(c, axis=1, keepdims=True)
        return d

    #GOOD
    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for softmax activation.

        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        gradient of loss w.r.t. input of this layer
        """
        ### YOUR CODE HERE ###
        A = self.forward(Z)
        b = dY * A
        c = np.sum(b, axis=1, keepdims=True)
        d = dY - c
        dZ = A * (d)
        return dZ
```

4.4.1:

4.4) $L = -Y \cdot \ln(\hat{Y})$

$$L = -\frac{1}{m}\left(\sum_{i=1}^{m} Y_i \ln(\hat{Y}_i)\right)$$

$$\boxed{\frac{\partial L}{\partial Y} = -\frac{1}{m}\left(\sum_{i=1}^{m} \frac{Y_i}{\hat{Y}_i}\right)}$$

4.4.2:

```
#GOOD
def forward(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
    """Computes the loss for predictions `Y_hat` given one-hot encoded labels
    `Y`.

    Parameters
    ----------
    Y      one-hot encoded labels of shape (batch_size, num_classes)
    Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

    Returns
    -------
    a single float representing the loss
    """
    ### YOUR CODE HERE ###
    a = np.log(Y_hat)
    b = np.multiply(Y, a)
    c = np.sum(b)
    c = (-c) / Y.shape[0]
    return c

#GOOD
def backward(self, Y: np.ndarray, Y_hat: np.ndarray) -> np.ndarray:
    """Backward pass of cross-entropy loss.
    NOTE: This is correct ONLY when the loss function is SoftMax.

    Parameters
    ----------
    Y      one-hot encoded labels of shape (batch_size, num_classes)
    Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

    Returns
    -------
    the gradient of the cross-entropy loss with respect to the vector of
    predictions, `Y_hat`
    """
    ### YOUR CODE HERE ###
    a = np.divide(Y, Y_hat)
    b = (-1)/(Y.shape[0])
    c = np.multiply(a, b)
    return c
```

5.1:

```python
#GOOD
def forward(self, X: np.ndarray) -> np.ndarray:
    """One forward pass through all the layers of the neural network.

    Parameters
    ----------
    X  design matrix whose must match the input shape required by the
       first layer

    Returns
    -------
    forward pass output, matches the shape of the output of the last layer
    """
    ### YOUR CODE HERE ###
    # Iterate through the network's layers.
    a = X
    b = []
    for i in self.layers:
        b.append(i.forward(a))
    return b[len(b) - 1]

#GOOD
def backward(self, target: np.ndarray, out: np.ndarray) -> float:
    """One backward pass through all the layers of the neural network.
    During this phase we calculate the gradients of the loss with respect to
    each of the parameters of the entire neural network. Most of the heavy
    lifting is done by the `backward` methods of the layers, so this method
    should be relatively simple. Also make sure to compute the loss in this
    method and NOT in `self.forward`.

    Note: Both input arrays have the same shape.

    Parameters
    ----------
    target  the targets we are trying to fit to (e.g., training labels)
    out     the predictions of the model on training data

    Returns
    -------
    the loss of the model given the training inputs and targets
    """
    ### YOUR CODE HERE ###
    # Compute the loss.
    a = self.loss(target, out)
    # Backpropagate through the network's layers.
    c = 0
    for i in self.layers[::-1]:
        c = i.backward(a)
    c = c
    return a
```

```python
#GOOD
def predict(self, X: np.ndarray, Y: np.ndarray) -> Tuple[np.ndarray, float]:
    """Make a forward and backward pass to calculate the predictions and
    loss of the neural network on the given data.

    Parameters
    ----------
    X  input features
    Y  targets (same length as `X`)

    Returns
    -------
    a tuple of the prediction and loss
    """
    ### YOUR CODE HERE ###
    # Do a forward pass. Maybe use a function you already wrote?
    # Get the loss. Remember that the `backward` function returns the loss.
    a = self.forward(X)
    b = self.backward(Y, a)
    return a, b
```

5.2:

Pass 1:
Learning Rate: 0.7
Hidden Layer Size: 20

Test Loss: 5.5331 Test Accuracy: 0.3667

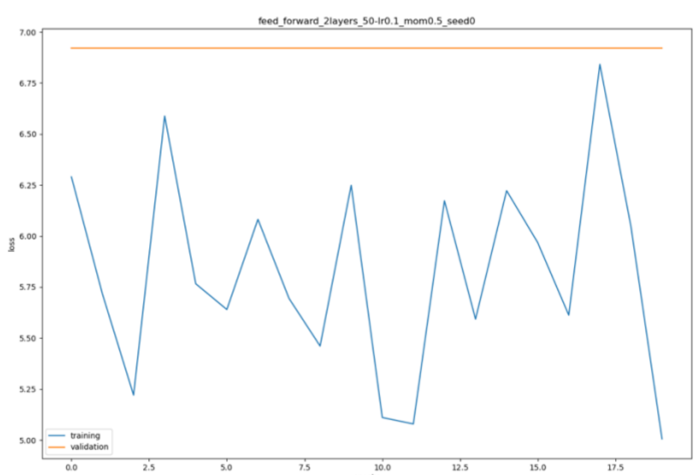
Pass 2:
Learning Rate: 0.5
Hidden Layer Size: 25

Test Loss: 5.5331 Test Accuracy: 0.3667

Pass 3:
Learning Rate: 0.1
Hidden Layer Size: 50

Test Loss: 5.5331 Test Accuracy: 0.3667

Loss vs Iterations for best model:



6.1:

```
import numpy as np

#1
A = np.random.rand(5, 5)
A_einsum = np.einsum('ii->', A)
A_numpy = np.trace(A)
diff1 = np.linalg.norm(A_einsum - A_numpy)

#2
B = np.random.rand(5, 5)
AB_einsum = np.einsum('ij,jk->ik', A, B)
AB_numpy = np.dot(A, B)
diff2 = np.linalg.norm(AB_einsum - AB_numpy)

#3
first = np.random.rand(3, 4, 5)
second = np.random.rand(3, 5, 6)
einsum = np.einsum('ijk,ikl->ijl', first, second)
numpy = np.matmul(first, second)
diff3 = np.linalg.norm(einsum - numpy)

print("A_einsum: ", A_einsum, "\n")
print("A_numpy: ", A_numpy, "\n")
print("Diff: ", diff1, "\n\n")

print("A_einsum: ", AB_einsum, "\n")
print("A_numpy: ", AB_numpy, "\n")
print("Diff: ", diff2, "\n\n")

print("einsum: ", einsum, "\n")
print("numpy: ", numpy, "\n")
print("Diff: ", diff3)
```

(variable) A_numpy: Any

```
A_einsum:  3.1140889727808325

A_numpy:  3.1140889727808325

Diff:  0.0


A_einsum:  [[0.50761207 0.81302028 0.87979608 0.59148936 0.84560596]
 [0.60483    1.05424765 0.34118848 0.40960392 0.52280119]
 [1.25844341 1.72994241 1.55304393 1.2198013  1.68738648]
 [1.36458669 1.98345353 1.95771633 1.61914343 2.09835346]
 [0.64491508 0.96707925 0.4940578  0.55631997 0.72117949]]

A_numpy:  [[0.50761207 0.81302028 0.87979608 0.59148936 0.84560596]
 [0.60483    1.05424765 0.34118848 0.40960392 0.52280119]
 [1.25844341 1.72994241 1.55304393 1.2198013  1.68738648]
 [1.36458669 1.98345353 1.95771633 1.61914343 2.09835346]
 [0.64491508 0.96707925 0.4940578  0.55631997 0.72117949]]

Diff:  0.0


einsum:  [[[0.81254004 1.0455103  1.28609626 1.13885994 1.7266988  0.89643037]
  [0.26781489 0.32063534 0.3084309  0.513428   0.61786485 0.63043506]
  [0.74578147 0.70112508 0.92505916 0.7646015  1.49263762 0.72193682]
...
  [1.76009685 1.20244624 1.76286245 0.94101872 1.2898272  0.47554543]
  [1.71081851 1.03322907 1.54627589 1.32496937 1.52144324 0.9780944 ]]]

Diff:  0.0
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

6.2.1:

6.2)

a) $$\sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial z[d_1, d_2, f]}$$

b) $$\sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial z[d_1, d_2, f]} \cdot X[d_1+i, d_2+k, c]$$

c) $$\sum_{f} \sum_{i} \sum_{k} \left(\frac{\partial L}{\partial z[x-i, y-k, f]} \cdot W[i, k, c, f] \right)$$

6.2.2:

```python
### BEGIN YOUR CODE ###
if self.n_in is None:
    self._init_parameters(X.shape)

W = self.parameters["W"]
b = self.parameters["b"]

kernel_height, kernel_width, in_channels, out_channels = W.shape
n_examples, in_rows, in_cols, in_channels = X.shape
kernel_shape = (kernel_height, kernel_width)

a1 = 2 * self.pad[0]
a = in_rows - kernel_height + a1
b1 = self.stride
o_r = ((a) // b1)
o_r = o_r + 1

a2 = 2 * self.pad[1]
a3 = in_cols - kernel_width + a2
o_c = (a3) // self.stride
o_c = o_c + 1

xp1 = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
xp = np.pad(X, xp1, mode='constant')

o = np.zeros((n_examples, o_r, o_c, out_channels))

for i in range(0, o_r):
    for j in range(0, o_c):
        h_s = j * self.stride
        h_e = h_s + kernel_width

        v_s = i * self.stride
        v_e = v_s + kernel_height


        X = xp[:, v_s:v_e, h_s:h_e, :]

        o1 = np.tensordot(X, W, axes=([1, 2, 3], [0, 1, 2]))
        o[:, i, j, :] = o1 + b
        Z = o

self.cache = {"X": X, "Z": Z}

return self.activation(o)
### END YOUR CODE ###
```

6.3.1:

6.3) For Max Pooling, when doing a forward pass only the max value gets recorded and passed through during back prop. However for average pooling, since we record the average of all feature values, when doing backprop we need to remember all values that average came from while for max we only need to remember the max value.

6.3.2:

```python
### BEGIN YOUR CODE ###
k_h, k_w = self.kernel_shape
s_h = self.stride
s_w = self.stride
p_h, p_r = self.pad

b_s, i_c, i_r, i_chan = X.shape

c1 = ((0, 0), (p_h, p_h), (p_r, p_r), (0, 0))
xp = np.pad(X, c1, mode='constant')

a1 = i_c + (2 * p_h) - k_h
o_c = (a1) // s_h + 1

b1 = i_r + (2 * p_r) - k_w
o_r = (b1) // s_w + 1

p = np.zeros((b_s, o_c, o_r, i_chan))

for i in range(0, b_s):
    for j in range(0, o_c):
        for k in range(0, o_r):
            for l in range(0, i_chan):
                h_s = j * s_h
                h_e = h_s + k_h

                w_s = k * s_w
                w_e = w_s + k_w

                x_s = xp[i, h_s:h_e, w_s:w_e, l]
                p[i, j, k, l] = self.pool_fn(x_s)

self.cache['out_rows'] = o_c
self.cache['out_cols'] = o_r
self.cache['X_pad'] = xp
self.cache['pool_shape'] = self.kernel_shape
return p
### END YOUR CODE ###
```

7.1:

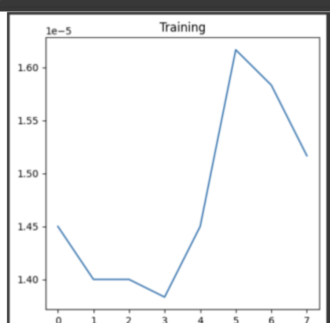
```python
gpu = torch.device("cuda" if torch.cuda.is_available() else "cpu")

testi_loss = []
test_acc = []
def run_nn(nn_model, test_load, a):
    nn_model.eval()
    test_loss = 0
    right_ones = 0
    with torch.no_grad():
        for i, j in test_load:
            i, j = i.to(gpu), j.to(gpu)
            i = matrix_compat(i)
            result = nn_model(i)
            result_loss = a(result, j)
            test_loss += result_loss.item()
            useless, guess = torch.max(result, dim=1)
            right_ones += (guess == j).sum().item()
    test_acc.append(right_ones / len(mnist_test))
    acc = right_ones / len(mnist_test)
    print("Test Loss: ", test_loss , "Accuracy: ", acc)
    return acc
```

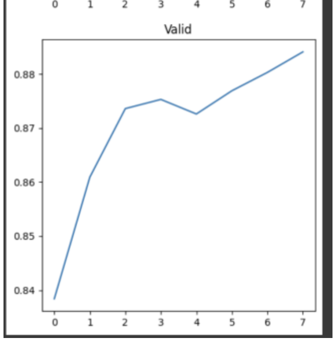
```python
class MLP(nn.Module):
    def __init__(self, i, h, o):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(i, h)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(h, o)

    def forward(self, x):
        x = self.fc1(x)
        y = self.relu(x)
        z = self.fc2(y)
        return z
```

```python
size = 100
lr = 0.001
epochs = 8

mnist_train = datasets.FashionMNIST(root='data', train=True, download=True, transform=transforms.ToTensor())
mnist_test = datasets.FashionMNIST(root='data', train=False, download=True, transform=transforms.ToTensor())

train_load = DataLoader(mnist_train, batch_size=size, shuffle=True)
test_load = DataLoader(mnist_test, batch_size=size, shuffle=False)

nn_model = MLP(1000, 1000, 1000).to(gpu)
a = nn.CrossEntropyLoss()
b = optim.Adam(nn_model.parameters(), lr=lr)

def matrix_compat(img):
    return img.view(img.size(0), -1)

e_train_acc = []
e_test_acc = []
train_loss = []
train_acc = []
for i in range(0, epochs):
    for j, k in train_load:
        j, k = j.to(gpu), k.to(gpu)
        j = matrix_compat(j)

        nn_result = nn_model(j)
        nn_loss = a(nn_result, k)

        b.zero_grad()
        nn_loss.backward()
        b.step()

        useless, predict = torch.max(nn_result, dim=1)
        accuracy = (predict == k).sum().item() / k.size(0)

        train_loss.append(nn_loss.item())
        train_acc.append(accuracy)

    train_acc_e = accuracy / len(train_load.dataset)
    e_train_acc.append(e_train_acc)

    test_acc_e = run_nn(nn_model, test_load, a)
    e_test_acc.append(e_test_acc)

print("Le training done")

#run_nn(nn_model, test_load, a)
```

```python
plt.figure(figsize=(5, 5))
plt.plot(e_train_acc, label='Training acc')
plt.title('Training')
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(e_test_acc, label='Valid Accuracy')
plt.title('Valid')
plt.show()
```

```
Test Loss:  44.81322491168976 Accuracy:  0.8384
Test Loss:  38.74152335524559 Accuracy:  0.8609
Test Loss:  35.517285123467445 Accuracy:  0.8736
Test Loss:  35.068400263786316 Accuracy:  0.8753
Test Loss:  34.52844831216335 Accuracy:  0.8726
Test Loss:  33.288587003946304 Accuracy:  0.8769
Test Loss:  33.01851260662079 Accuracy:  0.8803
Test Loss:  32.15085527300835 Accuracy:  0.8841
Le training done
```





7.2:

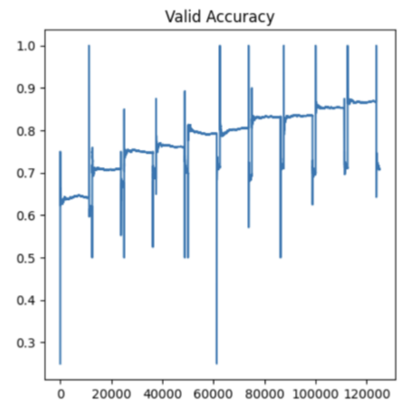
```python
### YOUR CODE HERE ###
#FIX
#torch.multiprocessing.set_sharing_strategy('file_system')

class cifarnet(nn.Module):
    def __init__(self):
        super(cifarnet, self).__init__()
        self.a = nn.Conv2d(3, 32, 3, padding=1)
        self.b = nn.MaxPool2d(2, 2)
        self.c = nn.Conv2d(32, 64, 3, padding=1)
        self.d = nn.Linear(64 * 8 * 8, 10)
    def forward(self, x):
        x = self.b(F.relu(self.a(x)))
        x = self.b(F.relu(self.c(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = self.d(x)
        return x

net_class = cifarnet()

normalization = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

training_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=normalization)
trainging_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True, num_workers=0)

testing_data_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=normalization)
testing_data_loader = torch.utils.data.DataLoader(testing_data_set, batch_size=4, shuffle=False, num_workers=0)

e = nn.CrossEntropyLoss()
optimSGD = optim.SGD(net_class.parameters(), lr=0.001, momentum=0.9)

len_of_training_set = len(training_set)
f = list(range(len_of_training_set))
index = int(np.floor(0.1 * len_of_training_set))

np.random.seed(0)
np.random.shuffle(f)

traini, validi = f[index:], f[:index]
train_data_sampler = torch.utils.data.sampler.SubsetRandomSampler(traini)
valid_data_sampler = torch.utils.data.sampler.SubsetRandomSampler(validi)

trainging_loader = torch.utils.data.DataLoader(training_set, batch_size=4, sampler=train_data_sampler)
validation_set_dataloader = torch.utils.data.DataLoader(training_set, batch_size=4, sampler=valid_data_sampler)

acc  = []
def acc_function(data):
    number_of_correct_guesses = 0
    number_of_guesses = 0
    with torch.no_grad():
        for i in data:
            data, label = i
            result = net_class(data)
            useless, the_guess = torch.max(result.data, 1)
            number_of_guesses += label.size(0)
            number_of_correct_guesses += (the_guess == label).sum().item()
            acc.append(number_of_correct_guesses / number_of_guesses)
    return number_of_correct_guesses / number_of_guesses

for epochs in range(0,10):
    so_far_loss = 0.0
    for i, j in enumerate(trainging_loader, 0):
        data, label = j
        optimSGD.zero_grad()
        result = net_class(data)
        result_loss = e(result, label)
        result_loss.backward()
        optimSGD.step()
        so_far_loss += result_loss.item()

    net_class.eval()
    acc_train = acc_function(trainging_loader)
    acc_validation = acc_function(validation_set_dataloader)
    acc.append(acc_validation)
    print("Training accuracy: ", acc_train, "Validation accuracy: ", acc_validation)

print("Done") #65.2% was without so_far_loss = 0.0
```

```
...
```

```python
plt.figure(figsize=(5, 5))
plt.plot(acc, label='Valid Accuracy')
plt.title('Valid Accuracy')
plt.show()
```

7.2:
Kaggle Username: Christopher Avakian
Kaggle Score: 0.704

7.3:



7.4:

A lot of my design choices were mainly just normalizing the data, which helped improve the overall speed, and running (and waiting) on a large number of epochs, which allowed me to gain approximately 5% on my Kaggle submission score. Otherwise, implementation wise, it has a lot fundamentally in common with the neural net we had to code up for questions 4, 5, and 6, only instead it was done with PyTorch and some of the functions already implemented (because of PyTorch).

References:

- https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
- https://en.wikipedia.org/wiki/Convolutional_neural_network
- https://builtin.com/machine-learning/fully-connected-layer
- https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528
- https://en.wikipedia.org/wiki/Softmax_function
- https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78
- https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax
- https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant
- https://en.wikipedia.org/wiki/Cross-entropy
- https://machinelearningmastery.com/cross-entropy-for-machine-learning/
- https://numpy.org/doc/stable/reference/generated/numpy.einsum.html
- https://rockt.github.io/2018/04/30/einsum
- https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns
- https://pytorch.org/docs/stable/index.html
- https://en.wikipedia.org/wiki/Multilayer_perceptron

Code Appendix:

```
#GOOD
def forward(self, Z: np.ndarray) -> np.ndarray:
    """Forward pass for softmax activation.
    Hint: The naive implementation might not be numerically stable.

    Parameters
    ----------
    Z  input pre-activations (any shape)

    Returns
    -------
    f(z) as described above applied elementwise to `Z`
    """
    ### YOUR CODE HERE ###
    a = np.max(Z, axis=1, keepdims=True)
    b = Z - a
    c = np.exp(b)
    d = c / np.sum(c, axis=1, keepdims=True)
    return d

#GOOD
def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
    """Backward pass for softmax activation.

    Parameters
    ----------
    Z   input to `forward` method
    dY  gradient of loss w.r.t. the output of this layer
        same shape as `Z`

    Returns
    -------
    gradient of loss w.r.t. input of this layer
    """
    ### YOUR CODE HERE ###
    A = self.forward(Z)
    b = dY * A
    c = np.sum(b, axis=1, keepdims=True)
    d = dY - c
    dZ = A * (d)
    return dZ
```

```python
#GOOD
def forward(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
    """Computes the loss for predictions `Y_hat` given one-hot encoded labels
    `Y`.

    Parameters
    ----------
    Y      one-hot encoded labels of shape (batch_size, num_classes)
    Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

    Returns
    -------
    a single float representing the loss
    """
    ### YOUR CODE HERE ###
    a = np.log(Y_hat)
    b = np.multiply(Y, a)
    c = np.sum(b)
    c = (-c) / Y.shape[0]
    return c

#GOOD
def backward(self, Y: np.ndarray, Y_hat: np.ndarray) -> np.ndarray:
    """Backward pass of cross-entropy loss.
    NOTE: This is correct ONLY when the loss function is SoftMax.

    Parameters
    ----------
    Y      one-hot encoded labels of shape (batch_size, num_classes)
    Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

    Returns
    -------
    the gradient of the cross-entropy loss with respect to the vector of
    predictions, `Y_hat`
    """
    ### YOUR CODE HERE ###
    a = np.divide(Y, Y_hat)
    b = (-1)/(Y.shape[0])
    c = np.multiply(a, b)
    return c
```

```python
#GOOD
def forward(self, X: np.ndarray) -> np.ndarray:
    """One forward pass through all the layers of the neural network.

    Parameters
    ----------
    X  design matrix whose must match the input shape required by the
       first layer

    Returns
    -------
    forward pass output, matches the shape of the output of the last layer
    """
    ### YOUR CODE HERE ###
    # Iterate through the network's layers.
    a = X
    b = []
    for i in self.layers:
        b.append(i.forward(a))
    return b[len(b) - 1]

#GOOD
def backward(self, target: np.ndarray, out: np.ndarray) -> float:
    """One backward pass through all the layers of the neural network.
    During this phase we calculate the gradients of the loss with respect to
    each of the parameters of the entire neural network. Most of the heavy
    lifting is done by the `backward` methods of the layers, so this method
    should be relatively simple. Also make sure to compute the loss in this
    method and NOT in `self.forward`.

    Note: Both input arrays have the same shape.

    Parameters
    ----------
    target  the targets we are trying to fit to (e.g., training labels)
    out     the predictions of the model on training data

    Returns
    -------
    the loss of the model given the training inputs and targets
    """
    ### YOUR CODE HERE ###
    # Compute the loss.
    a = self.loss(target, out)
    # Backpropagate through the network's layers.
    c = 0
    for i in self.layers[::-1]:
        c = i.backward(a)
    c = c
    return a
```

```python
#GOOD
def predict(self, X: np.ndarray, Y: np.ndarray) -> Tuple[np.ndarray, float]:
    """Make a forward and backward pass to calculate the predictions and
    loss of the neural network on the given data.

    Parameters
    ----------
    X  input features
    Y  targets (same length as `X`)

    Returns
    -------
    a tuple of the prediction and loss
    """
    ### YOUR CODE HERE ###
    # Do a forward pass. Maybe use a function you already wrote?
    # Get the loss. Remember that the `backward` function returns the loss.
    a = self.forward(X)
    b = self.backward(Y, a)
    return a, b
```

```python
#GOOD
def _init_parameters(self, X_shape: Tuple[int, int]) -> None:
    """Initialize all layer parameters (weights, biases)."""
    self.n_in = X_shape[1]

    ### BEGIN YOUR CODE ###

    W_shape = (self.n_in,) + (self.n_out,)
    W = self.init_weights(W_shape)
    b = np.zeros((1, self.n_out))

    self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
    self.cache = OrderedDict({"Z": [], "X": []}) # cache for backprop
    self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)}) # parameter gradients initialized to zero
                                                                                  # MUST HAVE THE SAME KEYS AS `self.parameters`
    ### END YOUR CODE ###

#GOOD
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: multiply by a weight matrix, add a bias, apply activation.
    Also, store all necessary intermediate results in the `cache` dictionary
    to be able to compute the backward pass.

    Parameters
    ----------
    X  input matrix of shape (batch_size, input_dim)

    Returns
    -------
    a matrix of shape (batch_size, output_dim)
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

    ### BEGIN YOUR CODE ###
    Z = np.matmul(X, self.parameters["W"])
    Z = Z + + self.parameters["b"]
    a = self.activation(Z)
    self.cache = {"X" : X, "Z" : Z}
    ### END YOUR CODE ###

    return a

#GOOD
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for fully connected layer.
    Compute the gradients of the loss with respect to:
        1. the weights of this layer (mutate the `gradients` dictionary)
        2. the bias of this layer (mutate the `gradients` dictionary)
        3. the input of this layer (return this)

    Parameters
    ----------
    dLdY  gradient of the loss with respect to the output of this layer
          shape (batch_size, output_dim)

    Returns
    -------
    gradient of the loss with respect to the input of this layer
    shape (batch_size, input_dim)
    """
    ### BEGIN YOUR CODE ###
    X, Z = self.cache["X"], self.cache["Z"]
    XT = X.T
    a = self.activation.backward(Z, dLdY)
    b = a.dot(self.parameters["W"].T)
    c = np.sum(a, axis=0, keepdims=True)
    d = XT.dot(a)
    self.gradients = {"W" : d, "b" : c}
    ### END YOUR CODE ###

    return b
```

```python
### BEGIN YOUR CODE ###
if self.n_in is None:
    self._init_parameters(X.shape)

W = self.parameters["W"]
b = self.parameters["b"]

kernel_height, kernel_width, in_channels, out_channels = W.shape
n_examples, in_rows, in_cols, in_channels = X.shape
kernel_shape = (kernel_height, kernel_width)

a1 = 2 * self.pad[0]
a = in_rows - kernel_height + a1
b1 = self.stride
o_r = ((a) // b1)
o_r = o_r + 1

a2 = 2 * self.pad[1]
a3 = in_cols - kernel_width + a2
o_c = (a3) // self.stride
o_c = o_c + 1

xp1 = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
xp = np.pad(X, xp1, mode='constant')

o = np.zeros((n_examples, o_r, o_c, out_channels))

for i in range(0, o_r):
    for j in range(0, o_c):
        h_s = j * self.stride
        h_e = h_s + kernel_width

        v_s = i * self.stride
        v_e = v_s + kernel_height


        X = xp[:, v_s:v_e, h_s:h_e, :]

        o1 = np.tensordot(X, W, axes=([1, 2, 3], [0, 1, 2]))
        o[:, i, j, :] = o1 + b
        Z = o

self.cache = {"X": X, "Z": Z}

return self.activation(o)
### END YOUR CODE ###
```

```python
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: use the pooling function to aggregate local information
    in the input. This layer typically reduces the spatial dimensionality of
    the input while keeping the number of feature maps the same.

    As with all other layers, please make sure to cache the appropriate
    information for the backward pass.

    Parameters
    ----------
    X  input array of shape (batch_size, in_rows, in_cols, channels)

    Returns
    -------
    pooled array of shape (batch_size, out_rows, out_cols, channels)
    """
    ### BEGIN YOUR CODE ###
    k_h, k_w = self.kernel_shape
    s_h = self.stride
    s_w = self.stride
    p_h, p_r = self.pad

    b_s, i_c, i_r, i_chan = X.shape

    c1 = ((0, 0), (p_h, p_h), (p_r, p_r), (0, 0))
    xp = np.pad(X, c1, mode='constant')

    a1 = i_c + (2 * p_h) - k_h
    o_c = (a1) // s_h + 1

    b1 = i_r + (2 * p_r) - k_w
    o_r = (b1) // s_w + 1

    p = np.zeros((b_s, o_c, o_r, i_chan))

    for i in range(0, b_s):
        for j in range(0, o_c):
            for k in range(0, o_r):
                for l in range(0, i_chan):
                    h_s = j * s_h
                    h_e = h_s + k_h

                    w_s = k * s_w
                    w_e = w_s + k_w

                    x_s = xp[i, h_s:h_e, w_s:w_e, l]
                    p[i, j, k, l] = self.pool_fn(x_s)

    self.cache['out_rows'] = o_c
    self.cache['out_cols'] = o_r
    self.cache['X_pad'] = xp
    self.cache['pool_shape'] = self.kernel_shape
    return p
    ### END YOUR CODE ###
```

```python
import numpy as np

#1
A = np.random.rand(5, 5)
A_einsum = np.einsum('ii->', A)
A_numpy = np.trace(A)
diff1 = np.linalg.norm(A_einsum - A_numpy)

#2
B = np.random.rand(5, 5)
AB_einsum = np.einsum('ij,jk->ik', A, B)
AB_numpy = np.dot(A, B)
diff2 = np.linalg.norm(AB_einsum - AB_numpy)

#3
first = np.random.rand(3, 4, 5)
second = np.random.rand(3, 5, 6)
einsum = np.einsum('ijk,ikl->ijl', first, second)
numpy = np.matmul(first, second)
diff3 = np.linalg.norm(einsum - numpy)

print("A_einsum: ", A_einsum, "\n")
print("A_numpy: ", A_numpy, "\n")
print("Diff: ", diff1, "\n\n")

print("A_einsum: ", AB_einsum, "\n")
print("A_numpy: ", AB_numpy, "\n")
print("Diff: ", diff2, "\n\n")

print("einsum: ", einsum, "\n")
print("numpy: ", numpy, "\n")
print("Diff: ", diff3)
```

```python
gpu = torch.device("cuda" if torch.cuda.is_available() else "cpu")

testi_loss = []
test_acc = []
def run_nn(nn_model, test_load, a):
    nn_model.eval()
    test_loss = 0
    right_ones = 0
    with torch.no_grad():
        for i, j in test_load:
            i, j = i.to(gpu), j.to(gpu)
            i = matrix_compat(i)
            result = nn_model(i)
            result_loss = a(result, j)
            test_loss += result_loss.item()
            useless, guess = torch.max(result, dim=1)
            right_ones += (guess == j).sum().item()
    test_acc.append(right_ones / len(mnist_test))
    acc = right_ones / len(mnist_test)
    print("Test Loss: ", test_loss , "Accuracy: ", acc)
    return acc
```

```python
class MLP(nn.Module):
    def __init__(self, i, h, o):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(i, h)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(h, o)

    def forward(self, x):
        x = self.fc1(x)
        y = self.relu(x)
        z = self.fc2(y)
        return z
```

```python
size = 100
lr = 0.001
epochs = 8

mnist_train = datasets.FashionMNIST(root='data', train=True, download=True, transform=transforms.ToTensor())
mnist_test = datasets.FashionMNIST(root='data', train=False, download=True, transform=transforms.ToTensor())

train_load = DataLoader(mnist_train, batch_size=size, shuffle=True)
test_load = DataLoader(mnist_test, batch_size=size, shuffle=False)

nn_model = MLP(1000, 1000, 1000).to(gpu)
a = nn.CrossEntropyLoss()
b = optim.Adam(nn_model.parameters(), lr=lr)

def matrix_compat(img):
    return img.view(img.size(0), -1)

e_train_acc = []
e_test_acc = []
train_loss = []
train_acc = []
for i in range(0, epochs):
    for j, k in train_load:
        j, k = j.to(gpu), k.to(gpu)
        j = matrix_compat(j)

        nn_result = nn_model(j)
        nn_loss = a(nn_result, k)

        b.zero_grad()
        nn_loss.backward()
        b.step()

        useless, predict = torch.max(nn_result, dim=1)
        accuracy = (predict == k).sum().item() / k.size(0)

        train_loss.append(nn_loss.item())
        train_acc.append(accuracy)

    train_acc_e = accuracy / len(train_load.dataset)
    e_train_acc.append(e_train_acc)

    test_acc_e = run_nn(nn_model, test_load, a)
    e_test_acc.append(e_test_acc)

print("Le training done")

#run_nn(nn_model, test_load, a)
```

```python
plt.figure(figsize=(5, 5))
plt.plot(e_train_acc, label='Training acc')
plt.title('Training')
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(e_test_acc, label='Valid Accuracy')
plt.title('Valid')
plt.show()
```