1 Honor Code

Declare and sign the following statement:

"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."

Signature : _____

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

No collaborators

4.1.1

$$\nabla_z L = \frac{\partial L}{\partial y} \nabla_z y =$$

$$\sigma_{ReLU}'(y) = \begin{cases} 0, & y < 0 \\ 1, & y \geq 0 \end{cases}$$

$$(\nabla_z y)_{ij} = \nabla_{z_{ij}} (\sigma_{ReLU}(z_{ij})) = \begin{cases} 0, & z_{ij} < 0 \\ 1, & z_{ij} \geq 0 \end{cases}$$

so

$$(\nabla_z L)_{ij} = \left(\frac{\partial L}{\partial y}\right)_{ij} \cdot \begin{cases} 0, & y_{ij} < 0 \\ 1, & y_{ij} \geq 0 \end{cases}$$

so $\nabla_z L = \frac{\partial L}{\partial y} * Y'$ where $(Y')_{ij} = \begin{cases} 0, & y_{ij} < 0 \\ 1, & y_{ij} \geq 0 \end{cases}$

$*$ ← elementwise

4.1.2

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

    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)
```

```python
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 ###

    Z_batch = self.forward(Z)
    mask = Z_batch > 0

    return dY * mask
```

4.2.1

$$Z = XW + 1b \in \mathbb{R}^{m \times n^{[l+1]}}$$

$$\frac{\partial L}{\partial Z} \in \mathbb{R}^{m \times n^{[l+1]}}$$

$$\nabla_X Z = W^T$$

$$\nabla_b Z = 1_{m \times 1}$$

$$\nabla_W Z = X_{m \times n^{[l]}}$$

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

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Z} \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$$

$$\frac{\partial L}{\partial b} = 1^T \frac{\partial L}{\partial Z} = \sum_{i=1}^{m} \left(\frac{\partial L}{\partial Z}\right)_i$$

4.2.2

```python
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 = self.init_weights([self.n_in, self.n_out])
    b = np.zeros((1, self.n_out))

    self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
    self.cache: OrderedDict = OrderedDict()  # cache for backprop
    self.gradients: OrderedDict = OrderedDict({"W": np.zeros((self.n_in, self.n_out)), "b": np.zeros((1, self.n_out))})  # parame
                                                # MUST HAVE THE SAME KEYS AS `self.parameters`

    ### END YOUR CODE ###
```

```python
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 = X.dot(self.parameters["W"]) + self.parameters["b"]
    out = self.activation.forward(Z)

    self.cache["X"] = X
    self.cache["Z"] = Z
    ### END YOUR CODE ###

    return out
```

```python
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 = self.cache["X"]
    Z = self.cache["Z"]
    dLdZ = self.activation.backward(Z, dLdY)
    # unpack the cache

    # compute the gradients of the loss w.r.t. all parameters as well as the
    # input of the layer

    dX = dLdZ @ self.parameters["W"].T

    # store the gradients in `self.gradients`
    # the gradient for self.parameters["W"] should be stored in
    # self.gradients["W"], etc.

    self.gradients["W"] = X.T @ dLdZ
    self.gradients["b"] = np.sum(dLdZ, axis=0, keepdims=True)

    ### END YOUR CODE ###

    return dX
```

4.3.1

$i \neq j$

$$\frac{\partial \sigma_i}{\partial s_j} = \frac{\partial}{\partial s_j}\left(\frac{e^{s_i}}{\sum_{g=1}^{K} e^{s_g}}\right) = e^{s_i} \cdot \frac{\partial}{\partial s_j}\left(\left(\sum e^{s_g}\right)^{-1}\right)$$

$$= e^{s_i} \cdot (-1) \cdot \left(\sum_{g=1}^{K} e^{s_g}\right)^{-2} \cdot e^{s_j}$$

$i = j$

$$\frac{\partial \sigma_i}{\partial s_j} = \frac{e^{s_i}}{\sum_{g=1}^{K} e^{s_g}} + e^{s_i} \cdot (-1) \cdot \left(\sum_{g=1}^{K} e^{s_g}\right)^{-2} \cdot e^{s_j}$$

$$[\sigma_1 \quad \sigma_2 \quad \ldots \quad \sigma_n]$$

4.3.2

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

    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 ###

        E = []
        for z in Z:
            m = np.max(z)
            elems = [np.exp(s - m) for s in z]
            elems = elems / np.sum(elems)
            E.append(elems)


        return np.array(E)
```

```python
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 ###


    dLdZ = np.zeros(Z.shape)
    Y = self.forward(Z)
    i = 0
    for y in Y:
        J = -np.outer(y, y)  # Outer product for all off-diagonal elements
        np.fill_diagonal(J, y * (1 - y))  # Diagonal elements
        dLdZ[i] = dY[i] @ J
        i +=1

    return dLdZ
```

4.4.1

$$L(y, \hat{y}) = -\frac{1}{m}\left(\sum_{i=1}^{m} y_i \cdot \ln(\hat{y}_i)\right)$$

$$\left(\frac{\partial L}{\partial \hat{y}}\right)_{ij} = -\frac{1}{m}\frac{\partial\, y_i \cdot \ln(\hat{y}_i)}{\partial \hat{y}_{ij}} = -\frac{1}{m}\frac{y_{ij}}{\hat{y}_{ij}}$$

thus $\frac{\partial L}{\partial \hat{y}} = -\frac{1}{m}\, y \,\%\, \hat{y}$

4.4.2

```python
class CrossEntropy(Loss):
    """Cross entropy loss function."""

    def __init__(self, name: str) -> None:
        self.name = name

    def __call__(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
        return self.forward(Y, Y_hat)

    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
        """
        sum = np.sum(Y * np.log(Y_hat), axis=1)
        return -1/len(Y) * np.sum(sum)

    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`
        """

        return -1/len(Y) * Y / Y_hat
```

5.1

```python
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.
    for i in range(self.n_layers):
        X = self.layers[i].forward(X)
    return X
```

```python
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.
    # Backpropagate through the network's layers.
    loss = self.loss.forward(target, out)
    dy = self.loss.backward(target, out)
    for i in range(self.n_layers - 1, -1, -1):
        self.layers[i].backward(dy)
    return loss
```

```python
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.
    Y_hat = self.forward(X)
    return (Y_hat, self.backward(Y, Y_hat))
```

Learning rate : 0.01
Hidden layer size : 25

```
Example prediction: [0.0, 0.0271, 0.9729]
Epoch 99 Training Loss: 0.0846 Training Accuracy: 0.952 Val Loss: 0.0201 Val Accuracy: 1.0
Test Loss: 0.3021 Test Accuracy: 0.88
```

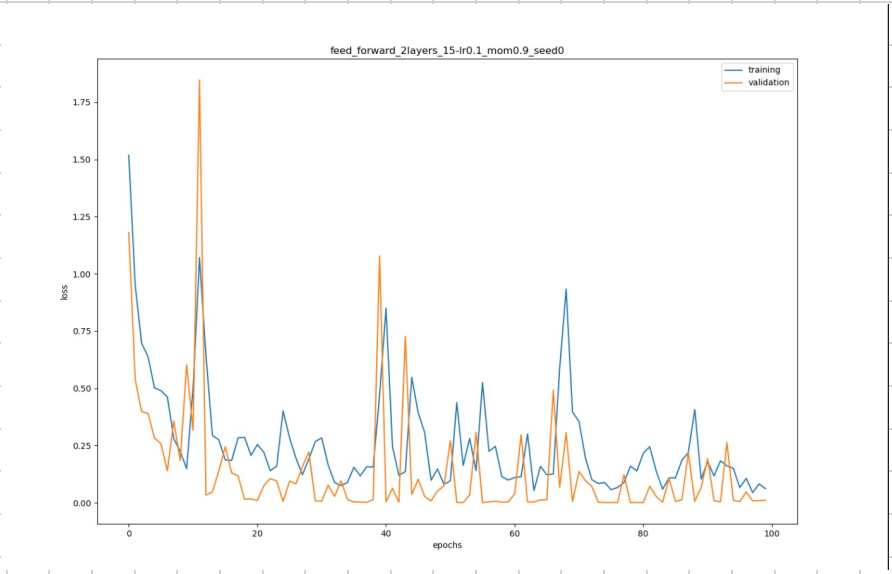
Learning rate : 0.1
Hidden layer size : 25

```
Epoch 99 Training Loss: 0.0457 Training Accuracy: 0.984 Val Loss: 0.0298 Val Accuracy: 1.0
Test Loss: 0.2985 Test Accuracy: 0.88
```

Learning rate : 0.01
Hidden layer size : 15

```
Epoch 99 Training Loss: 0.0616 Training Accuracy: 0.968 Val Loss: 0.0108 Val Accuracy: 1.0
Test Loss: 0.129 Test Accuracy: 0.96
```

seems 15 layers is better



6.1.

```python
import numpy as np

np.random.seed(4444)

A = np.random.rand(5, 5)

trace_np = np.trace(A)
trance_einsum = np.einsum("ii", A)

print("Trace comparison, norm is: ", np.linalg.norm(trace_np - trance_einsum))

B = np.random.rand(5, 5)

mult_np = A.dot(B)
mult_einsum = np.einsum("ij,jk->ik", A, B)
print("Multiplication comparison, norm is: ", np.linalg.norm(mult_np - mult_einsum))


A_batch = np.random.rand(3,4,5)
B_batch = np.random.rand(3,5,6)

mult_batch_np = np.matmul(A_batch,B_batch)
mult_batch_einsum = np.einsum("ijk,ikm->ijm", A_batch, B_batch)

print("Batch Multiplication comparison, norm is: ", np.linalg.norm(mult_batch_np - mult_batch_einsum))
```

1.

```
Trace comparison, norm is:  0.0
```

2.

```
Multiplication comparison, norm is:  0.0
```

3.

```
Batch Multiplication comparison, norm is:  6.106226635438361e-16
```

6.2.1

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n]$$

$$= \sum_i \sum_j \sum_c W[i, j, c, n] \, X[d_1 + i, d_2 + j, c] + b[n]$$

$$\frac{\partial L}{\partial X[x, y, c]} = \sum_n \sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial Z[d_1, d_2, n]} \cdot \frac{\partial Z(d_1, d_2, n)}{\partial X[x, y, c]}$$

$$= \sum_n \sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial Z[d_1, d_2, n]} \cdot W[x - d_1, y - d_2, c, n]$$

$$\frac{\partial L}{\partial W[i, k, c, f]} = \sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial Z(d_1, d_2, f]} \cdot \frac{\partial Z(d_1, d_2, f)}{\partial W[i, k, c, f]}$$

$$= \sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial Z[d_1, d_2, f]} \cdot X[d_1 + i, d_2 + j, c]$$

$$\frac{\partial L}{\partial b[f]} = \sum_{d_1} \sum_{d_2} \frac{\partial L}{\partial Z[d_1, d_2, f]}$$

6.2.2.

```python
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass for convolutional layer. This layer convolves the input
    `X` with a filter of weights, adds a bias term, and applies an activation
    function to compute the output. This layer also supports padding and
    integer strides. Intermediates necessary for the backward pass are stored
    in the cache.

    Parameters
    ----------
    X  input with shape (batch_size, in_rows, in_cols, in_channels)

    Returns
    -------
    output feature maps with shape (batch_size, out_rows, out_cols, out_channels)
    """
    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)
    out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
    out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
    padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
    X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
    Z = np.zeros((n_examples, out_height, out_width ,out_channels))
    for i in range(out_height):
        for j in range(out_width):
            for f in range(out_channels):
                start_i = i * self.stride
                start_j = j * self.stride
                end_i = start_i + kernel_height
                end_j = start_j + kernel_width
                m = W[:,:,:,f] * X_padded[:, start_i:end_i, start_j:end_j, :]
                m = [np.sum(i) for i in m]
                Z[:, i, j, f] = m + b[0][f]
    out = self.activation.forward(Z)
    self.cache["X"] = X
    self.cache["Z"] = Z
    ### END YOUR CODE ###
    return out
```

```python
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    ### BEGIN YOUR CODE ###
    X = self.cache["X"]
    Z = self.cache["Z"]
    W = self.parameters["W"]
    b = self.parameters["b"]
    dLdZ = self.activation.backward(Z, dLdY)
    db = np.zeros(b.shape)
    dW = np.zeros(W.shape)
    dX = np.zeros(X.shape)
    db = np.sum(dLdZ, axis=(0, 1))
    self.gradients["b"] = db
    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)
    out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
    out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
    padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
    X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
    padded_X = np.pad(X, ((0,0), (0,0), (self.pad[0],self.pad[0]), (self.pad[1],self.pad[1])), 'constant')
    for i in range(out_height):
        for j in range(out_width):
            for f in range(out_channels):
                start_i = i * self.stride
                start_j = j * self.stride
                end_i = start_i + kernel_height
                end_j = start_j + kernel_width
                m = W[:,:,:,f] * X_padded[:, start_i:end_i, start_j:end_j, :]
                m = [np.sum(i) for i in m]
                Z[:, i, j, f] = m + b[0][f]

    ### END YOUR CODE ###
    return dX
```

6.3.1.

for max pooling I set the value which corresponds to the largest position to 1 and the rest of the values within the polling window to zero.

for averaging I set every entry to $\frac{1}{\text{size of the window}}$

Then I multiply with corresponding element of dLdy

6.3.2.

```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)
    """
    n_examples, in_rows, in_cols, out_channels = X.shape
    kernel_height = self.kernel_shape[0]
    kernel_width = self.kernel_shape[1]
    out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
    out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
    padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
    X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
    Z = np.zeros((n_examples, out_height, out_width ,out_channels))
    for i in range(out_height):
        for j in range(out_width):
            for f in range(out_channels):
                start_i = i * self.stride
                start_j = j * self.stride
                end_i = start_i + kernel_height
                end_j = start_j + kernel_width
                sh = X_padded[:, start_i:end_i, start_j:end_j, f]
                m = [self.pool_fn(s) for s in sh]
                Z[:, i, j, f] = np.array(m)
    self.cache["X"] = X
    self.cache["Z"] = Z
    ### END YOUR CODE ###
    X_pool = Z
    return X_pool
```

```python
def backward(self, dLdY: np.ndarray) -> np.ndarray:

    X = self.cache["X"]
    Z = self.cache["Z"]
    dX = np.zeros(X.shape)
    n_examples, in_rows, in_cols, out_channels = X.shape
    kernel_height = self.kernel_shape[0]
    kernel_width = self.kernel_shape[1]
    out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
    out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
    padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
    X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)

    if (self.mode == "average"):
        pass

    elif (self.mode == "max"):
        for i in range(out_height):
            for j in range(out_width):
                for f in range(out_channels):
                    start_i = i * self.stride
                    start_j = j * self.stride
                    end_i = start_i + kernel_height
                    end_j = start_j + kernel_width

                    windows = X[:, start_i:end_i, start_j:end_j, f]
                    v = Z[:, i, j, f]
                    v_reshaped = v[:, np.newaxis, np.newaxis]
                    target_values = np.full((X.shape[0], end_i - start_i, end_j - start_j), v_reshaped)
                    mask = (windows == target_values)
                    vy = dLdY[:,i,j,f]
                    vy_reshaped = vy[:, np.newaxis, np.newaxis]
                    target_values_y = np.full((X.shape[0], end_i - start_i, end_j - start_j), vy_reshaped)

                    dX[:, start_i:end_i, start_j:end_j, f] = mask * target_values_y
    return dX
```

7.1.1

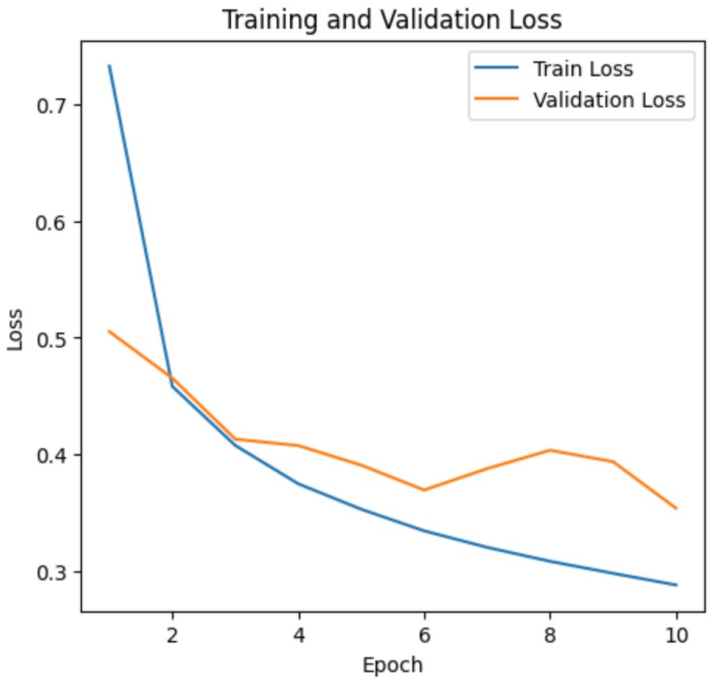
```
### YOUR CODE HERE ###
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(784, 128)
        self.layer2 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = self.output(x)
        return x

model = SimpleNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
model.train() # Put model in training mode
train_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffle=True)


epoch_count = []
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
validation_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=True)
```

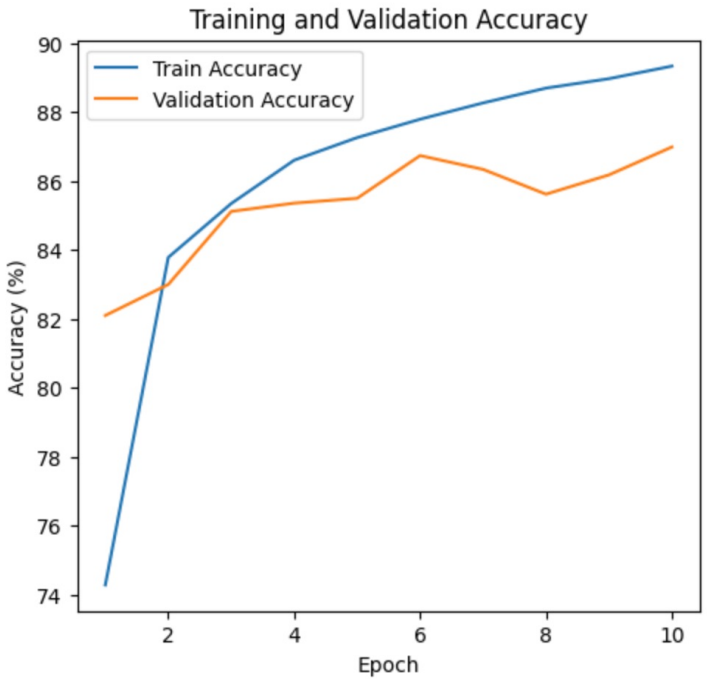
```
# Training loop
epochs = 15
for epoch in range(epochs):
    running_loss = 0
    correct = 0
    total = 0
    for x, y in tqdm.notebook.tqdm(train_loader, unit="batch"):
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        output = model(x)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    print("Finished Epoch", epoch + 1, ", training loss:", train_loss)
```

```
#val
model.eval()
val_loss = 0
correct = 0
total = 0
with torch.no_grad():
  for images, labels in validation_loader:
    images, labels = images.to(device), labels.to(device)
    outputs = model(images)
    loss = criterion(outputs, labels)
    val_loss += loss.item()
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

val_loss /= len(validation_loader)
val_accuracy = 100 * correct / total
val_losses.append(val_loss)
val_accuracies.append(val_accuracy)

model.train()
print(f'Epoch {epoch+1}, Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.2f}, Val Loss: {val_loss:.4f}, Val A
epoch_count.append(epoch+1)
```

I.1.2.

I.1.3


Training and Validation Loss
Train Loss
Validation Loss
Loss
Epoch
0.7
0.6
0.5
0.4
0.3
2
4
6
8
10


Training and Validation Accuracy
Train Accuracy
Validation Accuracy
Accuracy (%)
Epoch
90
88
86
84
82
80
78
76
74
2
4
6
8
10

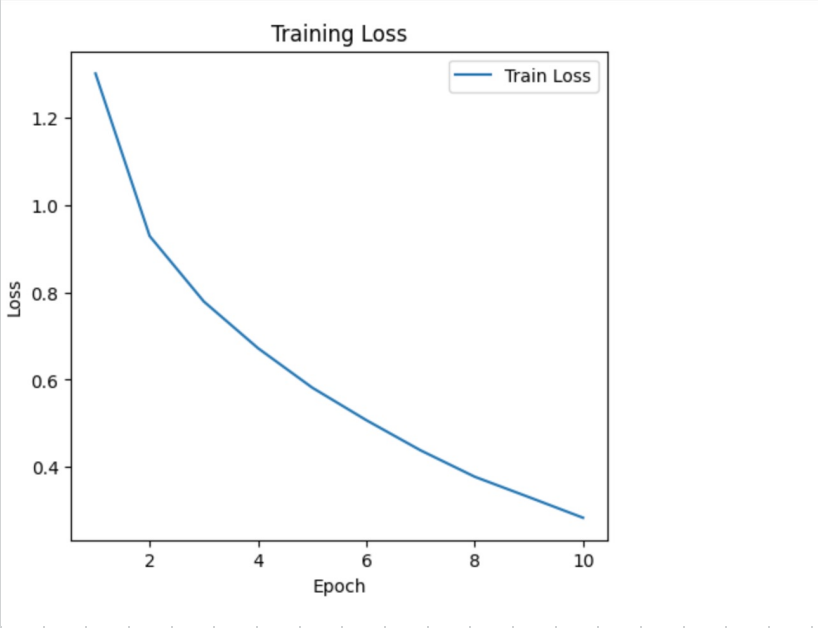
7.1

7.2.

Kaggle score user: Bayelle

0.701

7.2.3



7.2.4.

My first attempt only had 50% accuracy. I normalized images, also I switched SGD to Adam. I think the biggest improvement came from normalization, I tried AlexNet but it was very bad.

CS 189 HW 6: Neural Networks

Note: before starting this notebook, please make a copy of it, otherwise your changes will not persist.

This part of the assignment is designed to get you familiar with how engineerings in the real world train neural network systems. It isn't designed to be difficult. In fact, everything you need to complete the assignment is available directly on the pytorch website here. This note book will have the following components:

1. Understanding the basics of Pytorch (no deliverables)
2. Training a simple neural network on MNIST (Deliverable = training graphs)
3. Train a model on CIFAR-10 for Kaggle (Deliverable = kaggle submission and explanation of methods)

The last part of this notebook is left open for you to explore as many techniques as you want to do as well as possible on the dataset.

You will also get practice being an ML engineer by reading documentation and using it to implement models. The first section of this notebook will cover an outline of what you need to know -- we are confident that you can find the rest on your own.

Note that like all other assignments, you are free to use this notebook or not. You just need to complete the deliverables and turn in your code. If you want to run everything outside of the notebook, make sure to appropriately install pytorch to download the datasets and copy out the code for kaggle submission. If you don't want to use pytorch and instead want to use Tensorflow, feel free, but you may still need to install pytorch to download the datasets. That said, we will recommend pytorch over tensorflow since the latter has a somewhat steep learning curve and the former is more accessible to beginners.

Activation Function Implementations:

Implementation of `activations.Linear`:

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

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for f(z) = z.

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

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        return Z

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for f(z) = z.

        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
        """
        return dY
```

```
```

Implementation of `activations.Sigmoid`:

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

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for sigmoid function:
        f(z) = 1 / (1 + exp(-z))

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

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

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for sigmoid.

        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 ###
        return ...
```

Implementation of `activations.ReLU`:

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

    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)
```

```
        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 ###

            Z_batch = self.forward(Z)
            mask = Z_batch > 0
            return dY * mask
```

```
Implementation of `activations.SoftMax`:

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

    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 ###

        E = []
        for z in Z:
            m = np.max(z)
            elems = [np.exp(s - m) for s in z]
            elems = elems / np.sum(elems)
            E.append(elems)


        return np.array(E)

    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 ###


        dLdZ = np.zeros(Z.shape)
        Y = self.forward(Z)
        i = 0
        for y in Y:
            J = -np.outer(y, y)  # Outer product for all off-diagonal elements
            np.fill_diagonal(J, y * (1 - y))  # Diagonal elements
            dLdZ[i] = dY[i] @ J
            i +=1

        return dLdZ
```

```


### Layer Implementations:

Implementation of `layers.FullyConnected`:

```python
class FullyConnected(Layer):
    """A fully-connected layer multiplies its input by a weight matrix, adds
    a bias, and then applies an activation function.
    """

    def __init__(
        self, n_out: int, activation: str, weight_init="xavier_uniform"
    ) -> None:

        super().__init__()
        self.n_in = None
        self.n_out = n_out
        self.activation = initialize_activation(activation)

        # instantiate the weight initializer
        self.init_weights = initialize_weights(weight_init, activation=activation)

    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 = self.init_weights([self.n_in, self.n_out])
        b = np.zeros((1, self.n_out))

        self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
        self.cache: OrderedDict = OrderedDict()  # cache for backprop
        self.gradients: OrderedDict = OrderedDict({"W": np.zeros((self.n_in, self.n_out)), "b": np.zeros((1, s
                                        # MUST HAVE THE SAME KEYS AS `self.parameters`

        ### END YOUR CODE ###

    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 = X.dot(self.parameters["W"]) + self.parameters["b"]
            out = self.activation.forward(Z)

            self.cache["X"] = X
            self.cache["Z"] = Z
            ### END YOUR CODE ###

            return out

        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 = self.cache["X"]
            Z = self.cache["Z"]
            dLdZ = self.activation.backward(Z, dLdY)
            # unpack the cache

            # compute the gradients of the loss w.r.t. all parameters as well as the
            # input of the layer

            dX = dLdZ @ self.parameters["W"].T

            # store the gradients in `self.gradients`
            # the gradient for self.parameters["W"] should be stored in
            # self.gradients["W"], etc.

            self.gradients["W"] = X.T @ dLdZ
            self.gradients["b"] = np.sum(dLdZ, axis=0, keepdims=True)

            ### END YOUR CODE ###

            return dX
```

Implementation of `layers.Pool2D`:

```python
class Pool2D(Layer):
    """Pooling layer, implements max and average pooling."""

    def __init__(
        self,
        kernel_shape: Tuple[int, int],
```

```
            mode: str = "max",
            stride: int = 1,
            pad: Union[int, Literal["same"], Literal["valid"]] = 0,
        ) -> None:

            if type(kernel_shape) == int:
                kernel_shape = (kernel_shape, kernel_shape)

            self.kernel_shape = kernel_shape
            self.stride = stride

            if pad == "same":
                self.pad = ((kernel_shape[0] - 1) // 2, (kernel_shape[1] - 1) // 2)
            elif pad == "valid":
                self.pad = (0, 0)
            elif isinstance(pad, int):
                self.pad = (pad, pad)
            else:
                raise ValueError("Invalid Pad mode found in self.pad.")

            self.mode = mode

            if mode == "max":
                self.pool_fn = np.max
                self.arg_pool_fn = np.argmax
            elif mode == "average":
                self.pool_fn = np.mean

            self.cache = {
                "out_rows": [],
                "out_cols": [],
                "X_pad": [],
                "p": [],
                "pool_shape": [],
            }
            self.parameters = {}
            self.gradients = {}

        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)
            """
            n_examples, in_rows, in_cols, out_channels = X.shape
            kernel_height = self.kernel_shape[0]
            kernel_width = self.kernel_shape[1]
            out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
            out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
            padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
            X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
            Z = np.zeros((n_examples, out_height, out_width ,out_channels))
            for i in range(out_height):
                for j in range(out_width):
                    for f in range(out_channels):
                        start_i = i * self.stride
                        start_j = j * self.stride
```

```
                    end_i = start_i + kernel_height
                    end_j = start_j + kernel_width
                    sh = X_padded[:, start_i:end_i, start_j:end_j, f]
                    m = [self.pool_fn(s) for s in sh]
                    Z[:, i, j, f] = np.array(m)
        self.cache["X"] = X
        self.cache["Z"] = Z
        ### END YOUR CODE ###
        X_pool = Z
        return X_pool

    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """Backward pass for pooling layer.

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

        Returns
        -------
        gradient of loss with respect to the input of this layer
        shape (batch_size, in_rows, in_cols, channels)
        """
        X = self.cache["X"]
        Z = self.cache["Z"]
        dX = np.zeros(X.shape)
        n_examples, in_rows, in_cols, out_channels = X.shape
        kernel_height = self.kernel_shape[0]
        kernel_width = self.kernel_shape[1]
        out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
        out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
        padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
        X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)

        if (self.mode == "average"):
            pass

        elif (self.mode == "max"):
            for i in range(out_height):
                for j in range(out_width):
                    for f in range(out_channels):
                        start_i = i * self.stride
                        start_j = j * self.stride
                        end_i = start_i + kernel_height
                        end_j = start_j + kernel_width

                        windows = X[:, start_i:end_i, start_j:end_j, f]
                        v = Z[:, i, j, f]
                        v_reshaped = v[:, np.newaxis, np.newaxis]
                        target_values = np.full((X.shape[0], end_i - start_i, end_j - start_j), v_reshaped)
                        mask = (windows == target_values)
                        vy = dLdY[:,i,j,f]
                        vy_reshaped = vy[:, np.newaxis, np.newaxis]
                        target_values_y = np.full((X.shape[0], end_i - start_i, end_j - start_j), vy_reshaped)

                        dX[:, start_i:end_i, start_j:end_j, f] = mask * target_values_y


        return dX
```

```

Implementation of `layers.Conv2D.__init__`:

```python
    def __init__(
```
```

```
        def __init__(
            self,
            n_out: int,
            kernel_shape: Tuple[int, int],
            activation: str,
            stride: int = 1,
            pad: str = "same",
            weight_init: str = "xavier_uniform",
        ) -> None:

            super().__init__()
            self.n_in = None
            self.n_out = n_out
            self.kernel_shape = kernel_shape
            self.stride = stride
            self.pad = pad

            self.activation = initialize_activation(activation)
            self.init_weights = initialize_weights(weight_init, activation=activation)
```

Implementation of `layers.Conv2D._init_parameters`:

```
    def _init_parameters(self, X_shape: Tuple[int, int, int, int]) -> None:
        """Initialize all layer parameters and determine padding."""
        self.n_in = X_shape[3]

        W_shape = self.kernel_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 ini
                                                                                     # MUST HAVE THE SAME KEYS

        if self.pad == "same":
            self.pad = ((W_shape[0] - 1) // 2, (W_shape[1] - 1) // 2)
        elif self.pad == "valid":
            self.pad = (0, 0)
        elif isinstance(self.pad, int):
            self.pad = (self.pad, self.pad)
        else:
            raise ValueError("Invalid Pad mode found in self.pad.")
```

Implementation of `layers.Conv2D.forward`:

```
    def forward(self, X: np.ndarray) -> np.ndarray:
        """Forward pass for convolutional layer. This layer convolves the input
        `X` with a filter of weights, adds a bias term, and applies an activation
        function to compute the output. This layer also supports padding and
        integer strides. Intermediates necessary for the backward pass are stored
        in the cache.

        Parameters
        ----------
        X  input with shape (batch_size, in_rows, in_cols, in_channels)

        Returns
        -------
        output feature maps with shape (batch_size, out_rows, out_cols, out_channels)
        """
```

```
            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)
            out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
            out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
            padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
            X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
            Z = np.zeros((n_examples, out_height, out_width ,out_channels))
            for i in range(out_height):
                for j in range(out_width):
                    for f in range(out_channels):
                        start_i = i * self.stride
                        start_j = j * self.stride
                        end_i = start_i + kernel_height
                        end_j = start_j + kernel_width
                        m = W[:,:,:,f] * X_padded[:, start_i:end_i, start_j:end_j, :]
                        m = [np.sum(i) for i in m]
                        Z[:, i, j, f] = m + b[0][f]
            out = self.activation.forward(Z)
            self.cache["X"] = X
            self.cache["Z"] = Z
            ### END YOUR CODE ###
            return out
```

```

Implementation of `layers.Conv2D.backward`:

```python
    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """Backward pass for conv layer. Computes the gradients of the output
        with respect to the input feature maps as well as the filter weights and
        biases.

        Parameters
        ----------
        dLdY  gradient of loss with respect to output of this layer
              shape (batch_size, out_rows, out_cols, out_channels)

        Returns
        -------
        gradient of the loss with respect to the input of this layer
        shape (batch_size, in_rows, in_cols, in_channels)
        """
        ### BEGIN YOUR CODE ###
        X = self.cache["X"]
        Z = self.cache["Z"]
        W = self.parameters["W"]
        b = self.parameters["b"]



        dLdZ = self.activation.backward(Z, dLdY)


        db = np.zeros(b.shape)
        dW = np.zeros(W.shape)
        dX = np.zeros(X.shape)

        print(dLdZ.shape)

        db = np.sum(dLdZ, axis=(0, 1))
        self.gradients["b"] = db
```
```

```
            self.gradients[ b ] = db
            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)
            out_height = (in_rows - kernel_height + 2 * self.pad[0]) // self.stride + 1
            out_width = (in_cols - kernel_width + 2 * self.pad[1]) // self.stride + 1
            padding_config = ((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0))
            X_padded = np.pad(X, padding_config, mode='constant', constant_values=0)
            padded_X = np.pad(X, ((0,0), (0,0), (self.pad[0],self.pad[0]), (self.pad[1],self.pad[1])), 'constant')
            for i in range(out_height):
                for j in range(out_width):
                    for f in range(out_channels):
                        start_i = i * self.stride
                        start_j = j * self.stride
                        end_i = start_i + kernel_height
                        end_j = start_j + kernel_width
                        m = W[:,:,:,f] * X_padded[:, start_i:end_i, start_j:end_j, :]
                        m = [np.sum(i) for i in m]
                        Z[:, i, j, f] = m + b[0][f]



            ### END YOUR CODE ###

            return dX
```

```

### Loss Function Implementations:

Implementation of `losses.CrossEntropy`:

```python
class CrossEntropy(Loss):
    """Cross entropy loss function."""

    def __init__(self, name: str) -> None:
        self.name = name

    def __call__(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
        return self.forward(Y, Y_hat)

    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
        """
        sum = np.sum(Y * np.log(Y_hat), axis=1)
        return -1/len(Y) * np.sum(sum)

    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`
        """

        return -1/len(Y) * Y / Y_hat
```

Model Implementations:

Implementation of `models.NeuralNetwork.forward`:

```python
    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.
        for i in range(self.n_layers):
            X = self.layers[i].forward(X)
        return X
```

Implementation of `models.NeuralNetwork.backward`:

```python
    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.
        # Backpropagate through the network's layers.
        loss = self.loss.forward(target, out)
        dy = self.loss.backward(target, out)
        for i in range(self.n_layers - 1, -1, -1):
            dy = self.layers[i].backward(dy)
        return loss
```

```

Implementation of `models.NeuralNetwork.predict`:

```python
    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.
        Y_hat = self.forward(X)
        return (Y_hat, self.backward(Y, Y_hat))

```

```
# Imports for pytorch
import numpy as np
import torch
import torchvision
from torch import nn
import matplotlib
from matplotlib import pyplot as plt
import tqdm
```

1. Understanding Pytorch

Pytorch is based on the "autograd" paradigm. Essentially, you perform operations on multi-dimensional arrays like in numpy, except pytorch will automatically handle gradient tracking. In this section you will understand how to use pytorch.

This section should help you understand the full pipeline of creating and training a model in pytorch. Feel free to re-use code from this section in the assigned tasks.

Content in this section closely follows this pytorch tutorial: https://pytorch.org/tutorials/beginner/basics/intro.html

Tensors

Tensors can be created from numpy data or by using pytorch directly.
```

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

np_array = np.array(data)
x_np = torch.from_numpy(np_array)

shape = (2,3,)
rand_tensor = torch.rand(shape)
np_rand_array = rand_tensor.numpy()

print(f"Tensor from np: \n {x_np} \n")
print(f"Rand Tensor: \n {rand_tensor} \n")
print(f"Rand Numpy Array: \n {np_rand_array} \n")
```

```
Tensor from np:
 tensor([[1, 2],
        [3, 4]])

Rand Tensor:
 tensor([[0.5953, 0.0580, 0.7210],
        [0.3598, 0.7800, 0.5093]])

Rand Numpy Array:
 [[0.5952586  0.05800319 0.72102785]
 [0.3598193  0.7799766  0.50928503]]
```

They also support slicing and math operations very similar to numpy. See the examples below:

```
# Slicing
tensor = torch.ones(4, 4)
print('First row: ',tensor[0])
print('First column: ', tensor[:, 0])

# Matrix Operations
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

# Getting a single item
scalar = torch.sum(y1) # sums all elements
item = scalar.item()
print("Sum as a tensor:", scalar, ", Sum as an item:", item)
```

```
First row:  tensor([1., 1., 1., 1.])
First column:  tensor([1., 1., 1., 1.])
Sum as a tensor: tensor(64.) , Sum as an item: 64.0
```

Autograd

This small section shows you how pytorch computes gradients. When we create tenors, we can set `requires_grad` to be true to indicate that we are using gradients. For most of the work that you actually do, you will use the `nn` package, which automatically sets all parameter tensors to have `requires_grad=True`.

```
# Below is an example of computing the gradient for a single data point in logistic regression using pytorch'

x = torch.ones(5)  # input tensor
y = torch.zeros(1) # label
w = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
pred = torch.sigmoid(torch.matmul(x, w) + b)
loss = torch.nn.functional.binary_cross_entropy(pred, y)
loss.backward() # Computers gradients
print("W gradient:", w.grad)
print("b gradient:", b.grad)

# when we want to actually take an update step, we can use optimizers:
optimizer = torch.optim.SGD([w, b], lr=0.1)
print("Weight before", w)
optimizer.step() # use the computed gradients to update
# Print updated weights
print("Updated weight", w)

# Performing operations with gradients enabled is slow...
# You can disable gradient computation using the following enclosure:
with torch.no_grad():
    # Perform operations without gradients
    ...

    W gradient: tensor([[0.0202],
            [0.0202],
            [0.0202],
            [0.0202],
            [0.0202]])
    b gradient: tensor([0.0202])
    Weight before tensor([[-0.0768],
            [-0.5657],
            [-0.1409],
            [-0.4327],
            [-0.3847]], requires_grad=True)
    Updated weight tensor([[-0.0788],
            [-0.5678],
            [-0.1429],
            [-0.4347],
            [-0.3868]], requires_grad=True)
```

Devices

Pytorch supports accelerating computation using GPUs which are available on google colab. To use a GPU on google colab, go to runtime -> change runtime type -> select GPU.

Note that there is some level of strategy for knowing when to use which runtime type. Colab will kick users off of GPU for a certain period of time if you use it too much. Thus, its best to run simple models and prototype to get everything working on CPU, then switch the instance type over to GPU for training runs and parameter tuning.

Its best practice to make sure your code works on any device (GPU or CPU) for pytorch, but note that numpy operations can only run on the CPU. Here is a standard flow for using GPU acceleration:

```
# Determine the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)
# Next create your tensors
tensor = torch.zeros(4, 4, requires_grad=True)
# Move the tensor to the device you want to use
tensor = tensor.to(device)

# Perform whatever operations you want.... (often this will involve gradients)
# These operations will be accelerated by GPU.
tensor = 10*(tensor + 1)

# bring the tensor back to CPU, first detaching it from any gradient computations
tensor = tensor.detach().cpu()

tensor_np = tensor.numpy() # Convert to numpy if you want to perform numpy operations.
```

```
Using device cpu
```

The NN Package

Pytorch implements composable blocks in `Module` classes. All layers and modules in pytorch inherit from `nn.Module`. When you make a module you need to implement two functions: `__init__(self, *args, **kwargs)` and `foward(self, *args, **kwargs)`. Modules also have some nice helper functions, namely `parameters` which will recursively return all of the parameters. Here is an example of a logistic regression model:

```
class Perceptron(nn.Module):
  def __init__(self, in_dim):
    super().__init__()
    self.layer = nn.Linear(in_dim, 1) # This is a linear layer, it computes Xw + b

  def forward(self, x):
    return torch.sigmoid(self.layer(x)).squeeze(-1)

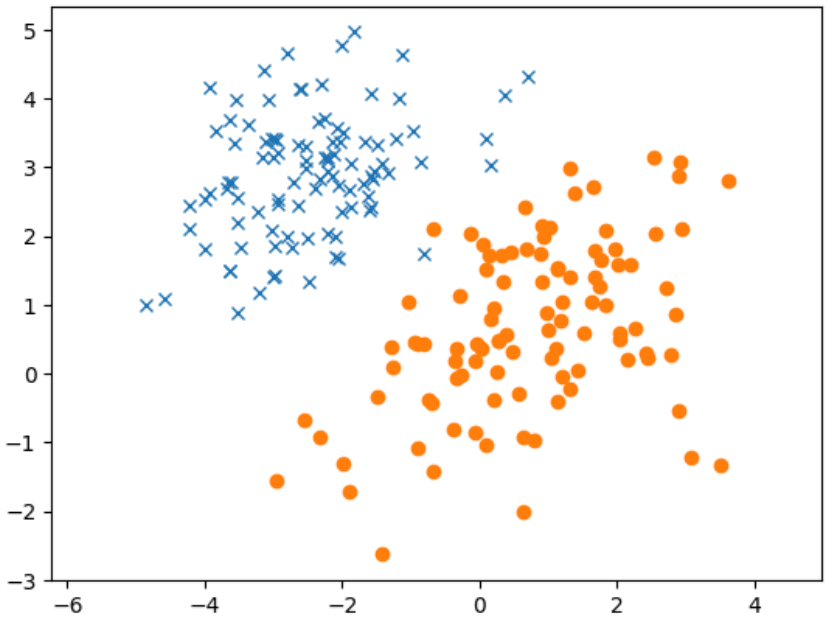
perceptron = Perceptron(10)
perceptron = perceptron.to(device) # Move all the perceptron's tensors to the device
print("Parameters", list(perceptron.parameters()))
```

```
Parameters [Parameter containing:
tensor([[-0.2326, -0.0355,  0.1614, -0.3140, -0.2766,  0.3156,  0.2773, -0.1832,
         -0.0191, -0.1102]], requires_grad=True), Parameter containing:
tensor([0.3120], requires_grad=True)]
```

Datasets

Pytorch has nice interfaces for using datasets. Suppose we create a logistic regression dataset as follows:

```
c1_x1, c1_x2 = np.random.multivariate_normal([-2.5,3], [[1, 0.3],[0.3, 1]], 100).T
c2_x1, c2_x2 = np.random.multivariate_normal([1,1], [[2, 1],[1, 2]], 100).T
c1_X = np.vstack((c1_x1, c1_x2)).T
c2_X = np.vstack((c2_x1, c2_x2)).T
train_X = np.concatenate((c1_X, c2_X))
train_y = np.concatenate((np.zeros(100), np.ones(100)))
# Shuffle the data
permutation = np.random.permutation(train_X.shape[0])
train_X = train_X[permutation, :]
train_y = train_y[permutation]
# Plot the data
plt.plot(c1_x1, c1_x2, 'x')
plt.plot(c2_x1, c2_x2, 'o')
plt.axis('equal')
plt.show()
```



We can then create a pytorch dataset object as follows. Often times, the default pytorch datasets will create these objects for you. Then, we can apply dataloaders to iterate over the dataset in batches.

```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(train_X), torch.from_numpy(train_y))
# We can create a dataloader that iterates over the dataset in batches.
dataloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True)
for x, y in dataloader:
    print("Batch x:", x)
    print("Batch y:", y)
    break

# Clean up the dataloader as we make a new one later
del dataloader
```

```
Batch x: tensor([[ 3.0811, -1.2286],
        [ 2.8433,  0.8548],
        [ 1.3224,  2.9873],
        [-1.9880, -1.2990],
        [-2.2946,  4.2071],
        [-2.0297,  3.3661],
        [ 0.2093, -0.3754],
        [ 1.6876,  1.3974],
        [ 1.1358,  1.5174],
        [-2.9899,  1.3939]], dtype=torch.float64)
Batch y: tensor([1., 1., 1., 1., 0., 0., 1., 1., 1., 0.], dtype=torch.float64)
```

Training Loop Example

Here is an example of training a full logistic regression model in pytorch. Note the extensive use of modules -- modules can be used for storing networks, computation steps etc.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

epochs = 10
batch_size = 10
learning_rate = 0.01

num_features = dataset[0][0].shape[0]
model = Perceptron(num_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
criterion = torch.nn.BCELoss()
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)

model.train() # Put model in training mode
for epoch in range(epochs):
    training_losses = []
    for x, y in tqdm.notebook.tqdm(dataloader, unit="batch"):
        x, y = x.float().to(device), y.float().to(device)
        optimizer.zero_grad() # Remove the gradients from the previous step
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        training_losses.append(loss.item())
    print("Finished Epoch", epoch + 1, ", training loss:", np.mean(training_losses))

# We can run predictions on the data to determine the final accuracy.
with torch.no_grad():
    model.eval() # Put model in eval mode
    num_correct = 0
    for x, y in dataloader:
        x, y = x.float().to(device), y.float().to(device)
        pred = model(x)
        num_correct += torch.sum(torch.round(pred) == y).item()
    print("Final Accuracy:", num_correct / len(dataset))
    model.train() # Put model back in train mode
```

```
Using device cpu
100%                                          20/20 [00:00<00:00, 159.06batch/s]
Finished Epoch 1 , training loss: 0.6159424811601639
100%                                          20/20 [00:00<00:00, 57.20batch/s]
Finished Epoch 2 , training loss: 0.5751470535993576
100%                                          20/20 [00:00<00:00, 96.14batch/s]
Finished Epoch 3 , training loss: 0.5435953810811043
100%                                          20/20 [00:00<00:00, 121.38batch/s]
Finished Epoch 4 , training loss: 0.5181494861841202
100%                                          20/20 [00:00<00:00, 188.27batch/s]
Finished Epoch 5 , training loss: 0.4970920242369175
100%                                          20/20 [00:00<00:00, 109.27batch/s]
Finished Epoch 6 , training loss: 0.4789320185780525
100%                                          20/20 [00:00<00:00,  8.46batch/s]
Finished Epoch 7 , training loss: 0.4630540765821934
100%                                          20/20 [00:00<00:00, 109.73batch/s]
Finished Epoch 8 , training loss: 0.44911158084869385
100%                                          20/20 [00:00<00:00, 57.05batch/s]
Finished Epoch 9 , training loss: 0.4363022968173027
100%                                          20/20 [00:00<00:00, 95.37batch/s]
Finished Epoch 10 , training loss: 0.42487354949116707
Final Accuracy: 0.77
```

Task 1: MLP For FashionMNIST

Earlier in this course you trained SVMs and GDA models on MNIST. Now you will train a multi-layer perceptron model on an MNIST-like dataset. Your deliverables are as follows:

1. Code for training an MLP on MNIST (can be in code appendix, tagged in your submission).
2. A plot of the training loss and validation loss for each epoch of training after trainnig for at least 8 epochs.
3. A plot of the training and validation accuracy, showing that it is at least 82% for validation by the end of training.

Below we will create the training and validation datasets for you, and provide a very basic skeleton of the code. Please leverage the example training loop from above.

Some pytorch components you should definitely use:

1. `nn.Linear`
2. Some activation function like `nn.ReLU`
3. `nn.CrossEntropyLoss` : if you choose to use `nn.CrossEntropyLoss` or `F.cross_entropy` , DO NOT add an explicit softmax layer in your neural network. PyTorch devs found it more numerically stable to combine softmax and cross entropy loss into a single module and if you explicitly attach a softmax layer at the end of your model, you would unintentionally be applying it twice, which can degrade performance.

Here are challenges you will need to overcome:

1. The data is default configured in image form i.e. (1 x 28 x 28), versus one feature vector. You will need to reshape it somewhere to feed it in as vector to the MLP. There are many ways of doing this.
2. You need to write code for plotting.
3. You need to find appropriate hyper-parameters to achieve good accuracy.

Your underlying model must be fully connected or dense, and may not have convolutions etc., but you can use anything in torch.optim or any layers in torch.nn besides nn.Linear that do not have weights.

```
# Creating the datasets
transform = torchvision.transforms.ToTensor() # feel free to modify this as you see fit.

training_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=transform,
)

validation_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=transform,
)
```

```
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 data
100%|██████████| 26421880/26421880 [00:01<00:00, 17258684.67it/s]
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/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 data
100%|██████████| 29515/29515 [00:00<00:00, 336958.60it/s]
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/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 data/
```

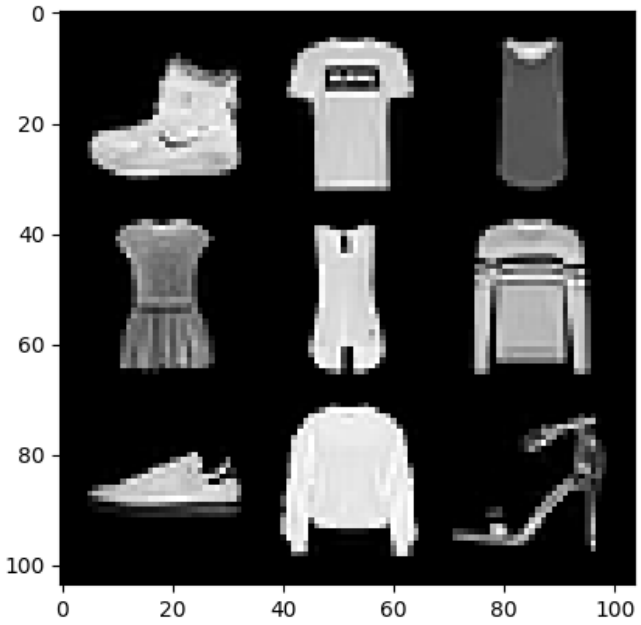
```
100%|██████████| 4422102/4422102 [00:00<00:00, 6257543.36it/s]
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

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 data/
100%|██████████| 5148/5148 [00:00<00:00, 4671630.68it/s]Extracting data/FashionMNIST/raw/t10k-labels-idx1
```

Before training a neural network, let's visualize our data first! Running the cell below will display the first 9 images in a 3 by 3 grid.

```
images = [training_data[i][0] for i in range(9)]
plt.imshow(torchvision.utils.make_grid(torch.stack(images), nrow=3, padding=5).numpy().transpose((1, 2, 0)))
```

```
<matplotlib.image.AxesImage at 0x7a98a62e9f00>
```



```
### YOUR CODE HERE ###
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(784, 128)
        self.layer2 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = self.output(x)
        return x

model = SimpleNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
model.train() # Put model in training mode
train_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffle=True)


epoch_count = []
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
validation_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=True)



# Training loop
epochs = 15
for epoch in range(epochs):
    running_loss = 0
    correct = 0
    total = 0
    for x, y in tqdm.notebook.tqdm(train_loader, unit="batch"):
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        output = model(x)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    print("Finished Epoch", epoch + 1, ", training loss:", train_loss)


    #val
    model.eval()
    val_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
      for images, labels in validation_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
```

```
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        val_loss /= len(validation_loader)
        val_accuracy = 100 * correct / total
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)

        model.train()
        print(f'Epoch {epoch+1}, Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.2f}, Val Loss: {val_lo
        epoch_count.append(epoch+1)
```

```
100%                                      6000/6000 [00:19<00:00, 350.84batch/s]
Finished Epoch 1 , training loss: 0.7376522328043357
Epoch 1, Train Loss: 0.7377, Train Acc: 74.09, Val Loss: 0.5012, Val Acc: 82.40
100%                                      6000/6000 [00:19<00:00, 334.79batch/s]
Finished Epoch 2 , training loss: 0.45454469227294125
Epoch 2, Train Loss: 0.4545, Train Acc: 84.01, Val Loss: 0.4604, Val Acc: 83.62
100%                                      6000/6000 [00:18<00:00, 328.44batch/s]
Finished Epoch 3 , training loss: 0.40566123129085946
Epoch 3, Train Loss: 0.4057, Train Acc: 85.48, Val Loss: 0.4185, Val Acc: 84.97
100%                                      6000/6000 [00:18<00:00, 366.20batch/s]
Finished Epoch 4 , training loss: 0.37459908310851703
Epoch 4, Train Loss: 0.3746, Train Acc: 86.52, Val Loss: 0.3975, Val Acc: 85.74
100%                                      6000/6000 [00:19<00:00, 341.57batch/s]
Finished Epoch 5 , training loss: 0.35424619556063164
Epoch 5, Train Loss: 0.3542, Train Acc: 87.26, Val Loss: 0.3892, Val Acc: 85.72
100%                                      6000/6000 [00:19<00:00, 278.95batch/s]
Finished Epoch 6 , training loss: 0.3356960860251759
Epoch 6, Train Loss: 0.3357, Train Acc: 87.78, Val Loss: 0.3755, Val Acc: 86.43
100%                                      6000/6000 [00:17<00:00, 347.17batch/s]
Finished Epoch 7 , training loss: 0.32217601146215263
Epoch 7, Train Loss: 0.3222, Train Acc: 88.31, Val Loss: 0.3676, Val Acc: 86.54
100%                                      6000/6000 [00:18<00:00, 366.10batch/s]
Finished Epoch 8 , training loss: 0.3123551459130443
Epoch 8, Train Loss: 0.3124, Train Acc: 88.63, Val Loss: 0.3720, Val Acc: 86.44
100%                                      6000/6000 [00:17<00:00, 354.74batch/s]
Finished Epoch 9 , training loss: 0.3000898987756421
Epoch 9, Train Loss: 0.3001, Train Acc: 89.04, Val Loss: 0.3496, Val Acc: 87.24
100%                                      6000/6000 [00:17<00:00, 343.67batch/s]
Finished Epoch 10 , training loss: 0.2910607894936111
Epoch 10, Train Loss: 0.2911, Train Acc: 89.34, Val Loss: 0.3463, Val Acc: 87.69
100%                                      6000/6000 [00:19<00:00, 331.63batch/s]
Finished Epoch 11 , training loss: 0.28279875379203195
Epoch 11, Train Loss: 0.2828, Train Acc: 89.56, Val Loss: 0.3501, Val Acc: 87.56
100%                                      6000/6000 [00:18<00:00, 366.22batch/s]
Finished Epoch 12 , training loss: 0.2758237444705155
Epoch 12, Train Loss: 0.2758, Train Acc: 89.88, Val Loss: 0.3500, Val Acc: 87.64
100%                                      6000/6000 [00:18<00:00, 347.02batch/s]
Finished Epoch 13 , training loss: 0.26832965893448757
Epoch 13, Train Loss: 0.2683, Train Acc: 90.19, Val Loss: 0.3385, Val Acc: 88.11
100%                                      6000/6000 [00:19<00:00, 322.93batch/s]
Finished Epoch 14 , training loss: 0.2619901296153451
Epoch 14, Train Loss: 0.2620, Train Acc: 90.42, Val Loss: 0.3379, Val Acc: 87.91
100%                                      6000/6000 [00:17<00:00, 345.70batch/s]
Finished Epoch 15 , training loss: 0.2552245488592501
Epoch 15, Train Loss: 0.2552, Train Acc: 90.55, Val Loss: 0.3321, Val Acc: 88.44
```

```
validation_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=True)

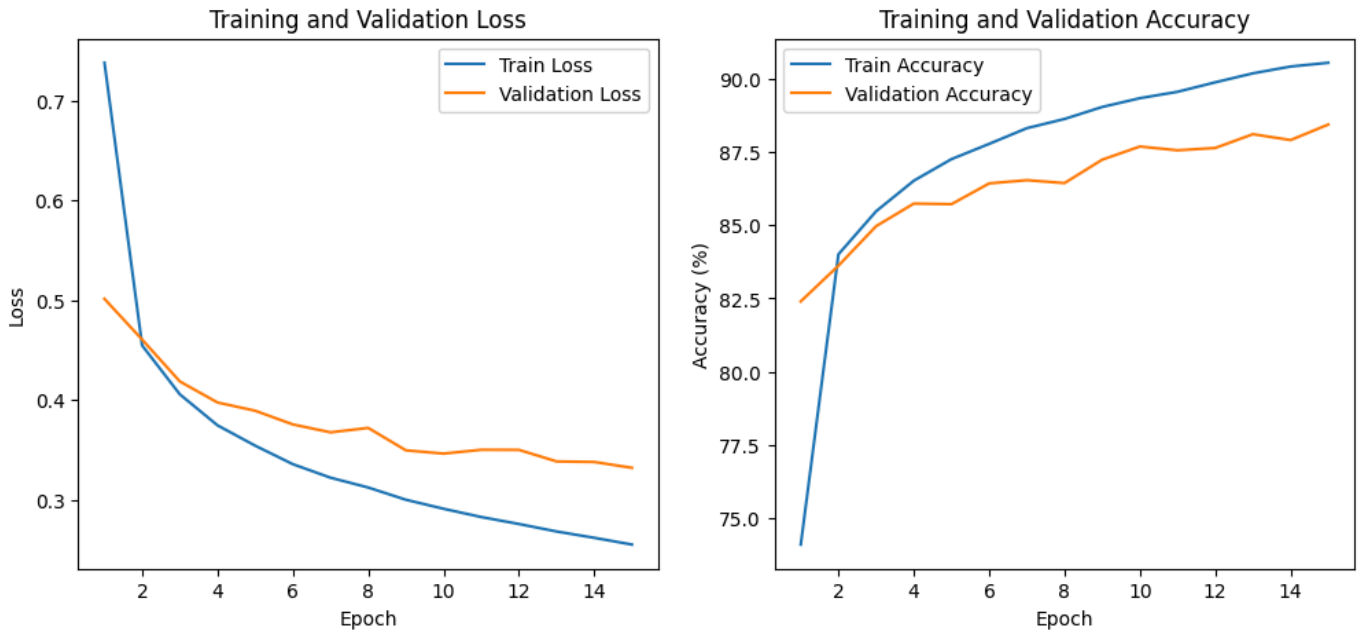

with torch.no_grad():
    model.eval() # Put model in eval mode
    num_correct = 0
    for x, y in validation_loader:
        x, y = x.float().to(device), y.float().to(device)
        pred = model(x)
        num_correct += torch.sum(torch.round(pred) == y).item()
    print("Final Accuracy:", num_correct / len(validation_data))
    model.train() # Put model back in train mode

     Final Accuracy: 0.4123


plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epoch_count, train_losses, label='Train Loss')
plt.plot(epoch_count, val_losses, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plotting training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(epoch_count, train_accuracies, label='Train Accuracy')
plt.plot(epoch_count, val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

plt.show()
```




```
Start coding or generate with AI.
```

Task 2: CNNs for CIFAR-10

In this section, you will create a CNN for the CIFAR dataset, and submit your predictions to Kaggle. It is recommended that you use GPU acceleration for this part.

Here are some of the components you should consider using:

1. `nn.Conv2d`
2. `nn.ReLU`
3. `nn.Linear`
4. `nn.CrossEntropyLoss`: if you choose to use `nn.CrossEntropyLoss` or `F.cross_entropy`, DO NOT add an explicit softmax layer in your neural network. PyTorch devs found it more numerically stable to combine softmax and cross entropy loss into a single module and if you explicitly attach a softmax layer at the end of your model, you would unintentionally be applying it twice, which can degrade performance.
5. `nn.MaxPooling2d` (though many implementations without it exist; for example, you can also do strided convolutions instead of a pooling layer!)

We encourage you to explore different ways of improving your model to get higher accuracy. Here are some suggestions for things to look into:

1. CNN architectures: AlexNet, VGG, ResNets, etc.
2. Different optimizers and their parameters (see torch.optim)
3. Image preprocessing / data augmentation (see torchvision.transforms)
4. Regularization or dropout (see torch.optim and torch.nn respectively)
5. Learning rate scheduling: https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate
6. Weight initialization: https://pytorch.org/docs/stable/nn.init.html

Though we encourage you to explore, there are some rules:

1. You are not allowed to install or use packages not included by default in the Colab Environment.
2. You are not allowed to use any pre-defined architectures or feature extractors in your network.
3. You are not allowed to use **any** pretrained weights, ie no transfer learning.
4. You cannot train on the test data.

Otherwise everything is fair game.

Your deliverables are as follows:

1. Submit to Kaggle and include your test accuracy in your report.
2. Provide at least (1) training curve for your model, depicting loss per epoch or step after training for at least 8 epochs.
3. Explain the components of your final model, and how you think your design choices contributed to it's performance.

After you write your code, we have included skeleton code that should be used to submit predictions to Kaggle. **You must follow the instructions below under the submission header**. Note that if you apply any processing or transformations to the data, you will need to do the same to the test data otherwise you will likely achieve very low accuracy.

It is expected that this task will take a while to train. Our simple solution achieves a training accuracy of 90.2% and a test accuracy of 74.8% after 10 epochs (be careful of overfitting!). This easily beats the best SVM based CIFAR10 model submitted to the HW 1 Kaggle! It is possible to achieve 95% or higher test accuracy on CIFAR 10 with good model design and tuning.

```
# Creating the datasets, feel free to change this as long as you do the same to the test data.
# You can also modify this to split the data into training and validation.
# See https://pytorch.org/docs/stable/data.html#torch.utils.data.random_split

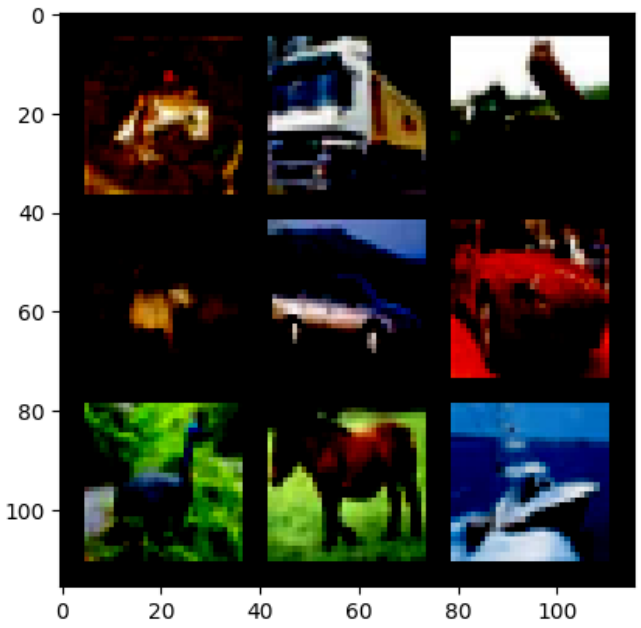
transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
training_data = torchvision.datasets.CIFAR10(
    root="data",
```

Again, let's first visualize our data.

```
    transform=transform,
```

```
images = [training_data[i][0] for i in range(9)]
plt.imshow(torchvision.utils.make_grid(torch.stack(images), nrow=3, padding=5).numpy().transpose((1, 2, 0)))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
<matplotlib.image.AxesImage at 0x7c705ef02c80>
```




```
### YOUR CODE HERE ###
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, padding=2)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool1(self.act1(self.conv1(x)))
        x = self.pool2(self.act2(self.conv2(x)))
        x = torch.flatten(x, 1)
```

```
### YOUR CODE HERE ###
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, padding=2)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool1(self.act1(self.conv1(x)))
        x = self.pool2(self.act2(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = self.act3(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

trainloader = torch.utils.data.DataLoader(training_data, batch_size=16, shuffle=True)

epoch_count = []
train_losses = []

num_epochs = 10

for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

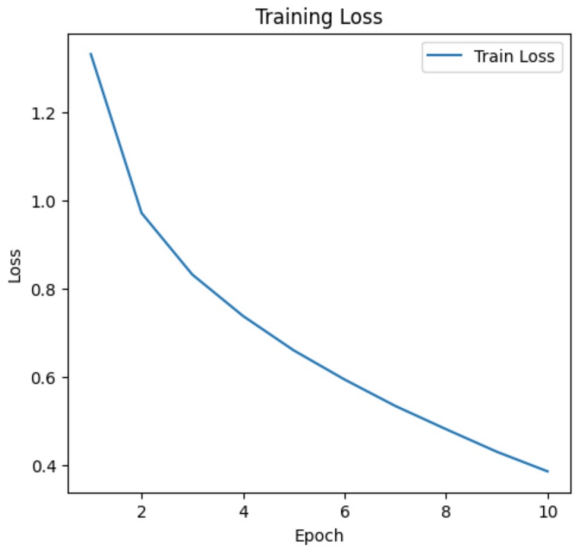
    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)
    epoch_count.append(epoch+1)

print('Finished Training')
```

```
Finished Training
```

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epoch_count, train_losses, label='Train Loss')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
plt.legend()
plt.show()
```




```
### YOUR CODE HERE ###

# Recommendation: create a `test_dataloader` from torch.utils.data.DataLoader with `shuffle=False` to iterate ov

# Store a numpy vector of the predictions for the test set in the variable `predictions`.

test_dataloader = torch.utils.data.DataLoader(testing_data, shuffle=False)

model.eval()
with torch.no_grad():
  predictions = []
  for data in test_dataloader:
      inputs = data.to(device)
      outputs = model(inputs)
      _, predicted = torch.max(outputs.data, 1)
      predictions.extend(predicted.cpu().numpy())

predictions = np.array(predictions)
print(predictions.shape)
print(len(testing_data))
```

```
(10000,)
10000
```

```
[ ] # This code below will generate kaggle_predictions.csv file. Please download it and submit to kaggle.
    import pandas as pd

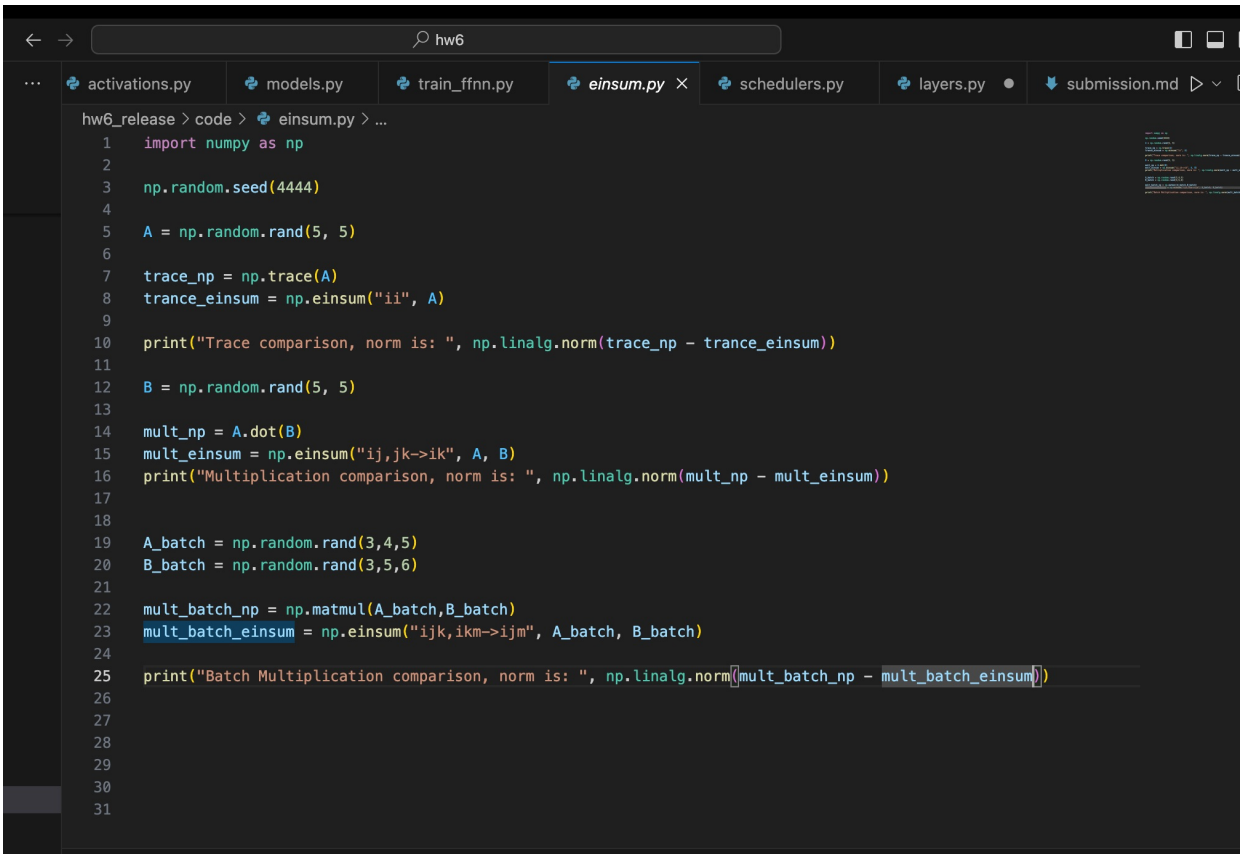
    if isinstance(predictions, np.ndarray):
        predictions = predictions.astype(int)
    else:
        predictions = np.array(predictions, dtype=int)
    assert predictions.shape == (len(testing_data),), "Predictions were not the correct shape"
    df = pd.DataFrame({'Category': predictions})
    df.index += 1  # Ensures that the index starts at 1.
    df.to_csv('submission.csv', index_label='Id')

    # Now download the submission.csv file to submit.
```

```
[1] from google.colab import output
    output.enable_custom_widget_manager()
```

My first attempt only had 50% accuracy. I normalized images, also I switched SGD to Adam. I think the biggest improvement came from normalization, I tried AlexNet but it was very bad.

Congrats! You made it to the end.


hw6
activations.py
models.py
train_ffnn.py
einsum.py
schedulers.py
layers.py
submission.md
hw6_release > code > einsum.py > ...
import numpy as np
np.random.seed(4444)
A = np.random.rand(5, 5)
trace_np = np.trace(A)
trance_einsum = np.einsum("ii", A)
print("Trace comparison, norm is: ", np.linalg.norm(trace_np - trance_einsum))
B = np.random.rand(5, 5)
mult_np = A.dot(B)
mult_einsum = np.einsum("ij,jk->ik", A, B)
print("Multiplication comparison, norm is: ", np.linalg.norm(mult_np - mult_einsum))
A_batch = np.random.rand(3,4,5)
B_batch = np.random.rand(3,5,6)
mult_batch_np = np.matmul(A_batch,B_batch)
mult_batch_einsum = np.einsum("ijk,ikm->ijm", A_batch, B_batch)
print("Batch Multiplication comparison, norm is: ", np.linalg.norm(mult_batch_np - mult_batch_einsum))