1 Layer Implementations

In this question, you will implement the layers needed for basic classification neural networks. **For each part, you will be asked to 1) derive the gradients and 2) write the matching code.**

When doing the derivations, **please derive the gradients element-wise.** This means that rather than directly computing $\partial L/\partial X$ by manipulating entire tensors, we encourage you to compute $[\partial L/\partial X]_{ij}$ then stack those components back into a vector/matrix as appropriate. Also keep in mind that for all layers **your code must operate on mini-batches of data** and should not use loops to iterate over the training points individually. The code is marked with `YOUR CODE HERE` statements indicating what to implement and where. Please read the docstrings and the function signatures too.

1.1 Activation Functions

First, you will implement the ReLU activation function in `activations.py`. ReLU is a very common activation function that is typically used in the hidden layers of a neural network and is defined as

$$\sigma_{\text{ReLU}}(\gamma) = \begin{cases} 0 & \gamma < 0, \\ \gamma & \text{otherwise.} \end{cases}$$

Note that the activation function is applied element-wise to a vector input.

Instructions

1. **Derive** $\partial L/\partial Z$, the gradient of the downstream loss with respect to the batched input of the ReLU activation function, $Z \in \mathbb{R}^{m\times n}$. First derive the gradient element-wise, i.e. find an expression for $[\partial L/\partial Z]_{ij}$, and then stack these elements appropriately to obtain a simple vector/matrix expression for $\partial L/\partial Z$. **Write your final solution in terms of** $\partial L/\partial Y$ (the gradient of the loss w.r.t. the output $Y = \sigma_{\text{ReLU}}(Z)$ where $Y \in \mathbb{R}^{m\times n}$) and Z. Include your derivation in your writeup.

2. **Next, implement the forward and backward passes of the ReLU activation** in `activations.py`. **Do not iterate over training examples; use batched operations.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope.

Solution:

$$\left[\frac{\partial L}{\partial Z}\right]_{ij} = \left[\frac{\partial L}{\partial Y}\right]_{ij} \cdot \left[\frac{\partial Y}{\partial Z}\right]_{ij}$$

$$\left[\frac{\partial Y}{\partial Z}\right]_{ij} = \left(\frac{\partial \sigma_{\text{ReLU}}(Z)}{\partial Z}\right)_{ij} = \begin{cases} 0, & Z_{ij} \le 0 \\ 1, & Z_{ij} > 0 \end{cases}$$

$$\Rightarrow \frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \cdot 1(Z > 0)$$

where $1(\cdot)$ is the indicator function.

Implementation of `activations.ReLU`:

```
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 ###
        return dY * (Z > 0)
```

1.2 Fully-Connected Layer

Now you will implement the forward and backward passes for the fully-connected layer in the `layers.py` script. Write the fully-connected layer for a general input h that contains a mini-batch of m examples with d features. When implementing a new layer, it is important to manually verify correctness of the forward and backward passes. For debugging tips, look back at Section 2.2.

Instructions

1. **Derive $\partial L/\partial W$ and $\partial L/\partial b$**, the gradients of the loss with respect to the weight matrix $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$ and bias row vector $b \in \mathbb{R}^{1 \times n^{[l+1]}}$ in the fully-connected layer. First derive the gradient element-wise, i.e. find expressions for $[\partial L/\partial W]_{ij}$ and $[\partial L/\partial b]_{1i}$, and then stack these elements appropriately to obtain simple vector/matrix expressions for $\partial L/\partial W$ and $\partial L/\partial b$. Repeat this process to also derive the gradient of the loss with respect to the input of the layer $\partial L/\partial X$, which will be passed to lower layers, where $X \in \mathbb{R}^{m \times n^{[l]}}$ is the batched input. **Write your final solution for each of the gradients in terms of $\partial L/\partial Z$**, which you have already obtained in the previous subpart, where $Z = XW + 1b$ and $1 \in \mathbb{R}^{m \times 1}$ is a column of ones. Include your derivations in your writeup.

   Note: the term $1b$ is a matrix (it's an outer product) here, whose each row is the row vector b so we are adding the same bias vector to each sample in a mini-batch during the forward pass: this is the mathematical equivalent of numpy broadcasting.

2. **Implement the forward and backward passes of the fully-connected layer** in `layers.py`. First, initialize the weights of the model using `_init_parameters`, which takes the shape of the data matrix X as input and initializes the parameters, cache, and gradients of the layer (you should initialize the bias vector to all zeros). The `backward` method takes in an argument `dLdY`, the derivative of the loss with respect to the output of the layer, which is computed by higher layers and backpropagated. This should be incorporated into your gradient calculation. **Do not iterate over training examples; use batched operations.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope.

Solution: Since Z = XW + 1b,

$$Z_{kj} = \sum_{i=1}^{n^{(l)}} X_{ki} W_{ij} + b_j$$

1. Gradient of loss in regards to W. Applying the chain rule:

$$\frac{\delta L}{\delta W} = \frac{\delta L}{\delta Z} \cdot \frac{\delta Z}{\delta W}$$

$$\left[\frac{\delta L}{\delta W}\right]_{ij} = \sum_{k=1}^{m} \frac{\delta L}{\delta Z_{kj}} \cdot \frac{\delta Z_{kj}}{\delta W_{ij}} = \sum_{k=1}^{m} \frac{\delta L}{\delta Z_{kj}} X_{ki} = \sum_{k=1}^{m} X_{ik}^T \frac{\delta L}{\delta Z_{kj}}$$

$$\implies \frac{\delta L}{\delta W} = X^T \cdot \frac{\delta L}{\delta Z}$$

2. Gradient of loss in regards to b. Applying the chain rule:

$$\left[\frac{\delta L}{\delta b}\right]_{1j} = \sum_{k=1}^{m} \frac{\delta L}{\delta Z_{kj}} \cdot \frac{\delta Z_{kj}}{\delta b_j} = \sum_{k=1}^{m} \frac{\delta L}{\delta Z_{kj}} \cdot 1_k, \quad 1 \in \mathbb{R}^m$$

$$\implies \frac{\delta L}{\delta b} = 1^T \frac{\delta L}{\delta Z}, \quad 1^T \in \mathbb{R}^{1 \times m}$$

3. Gradient of loss in regards to X. We know that

$$Z_{ik} = \sum_{j=1}^{n^{(l)}} X_{ij} W_{jk} + b_k$$

Applying the chain rule:

$$\left[\frac{\delta L}{\delta X}\right]_{ij} = \frac{\delta L}{\delta Z_{ik}} \cdot \frac{\delta Z_{ik}}{\delta X_{ij}} = \sum_{k=1}^{n^{(l+1)}} \frac{\delta L}{\delta Z_{ik}} \cdot W_{jk} = \sum_{k=1}^{n^{(l+1)}} \frac{\delta L}{\delta Z_{ik}} \cdot W_{kj}^T$$

$$\implies \frac{\delta L}{\delta X} = \frac{\delta L}{\delta Z} \cdot W^T$$

Implementation of `layers.FullyConnected`:

```
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({"X": None, "Z": None})  # 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 ###

    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 ###

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


        # perform an affine transformation and activation

        Z = np.dot(X, W) + b

        out = self.activation.forward(Z)

        # store information necessary for backprop in `self.cache`
        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 ###

        # unpack the cache
        X = self.cache["X"]
        Z = self.cache["Z"]
        W = self.parameters["W"]

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

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

        dW = np.dot(X.T, dLdZ)
        db = np.sum(dLdZ, axis=0, keepdims=True)
        dX = np.dot(dLdZ, 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"] = dW
        self.gradients["b"] = db


        ### END YOUR CODE ###

        return dX
```

1.3 Softmax Activation

Next, we need to define an activation function for the output layer. The ReLU activation function returns continuous values that are (potentially) unbounded to the right. Since we are building a classifier, we want to return *probabilities* over classes. The softmax function has the desirable property that it outputs a valid probability distribution: it takes in a vector s of k un-normalized values $s_1, \ldots, s_k$, maps each component to $s_i \mapsto e^{s_i} > 0$, and normalizes the result so that all components add up to 1. That is, the softmax activation squashes continuous values in the range $(-\infty, \infty)$ to the range $(0, 1)$ so the output can be interpreted as a probability distribution over k possible classes. For this reason, many classification neural networks use the softmax activation as the output activation after the final layer. Mathematically, the forward pass of the softmax activation on input s_i is

$$\sigma_i = \frac{e^{s_i}}{\sum_{l=1}^{k} e^{s_l}},$$

Due to issues of numerical stability, the following modified version of this function is commonly used in practice instead:

$$\sigma_i = \frac{e^{s_i - m}}{\sum_{l=1}^{k} e^{s_l - m}},$$

where $m = \max_{j=1}^{k} s_j$. We recommend implementing this method. You can verify yourself why these two formulations are equivalent mathematically.

Instructions

1. **Derive the Jacobian** of the softmax activation function. You do not need to write out the entire matrix, but please write out what $\partial\sigma_i/\partial s_j$ is for an arbitrary (i, j) pair. This question does not require bathched inputs; an answer for a single training point is acceptable. Include your derivation in your writeup.

2. **Implement the forward and backward passes of the softmax activation** in the file `activations.py`. We recommend vectorizing the backward pass for efficiency. However, if you wish, **for this question only, you may use a "for" loop over the training points in the mini-batch.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope.

Solution:

$$(J)_{ij} = \frac{\partial \sigma_i}{\partial s_j}, \quad \sigma_i = \frac{e^{s_i - m}}{\sum_{l=1}^{k} e^{s_l - m}} \tag{1}$$

Using the quotient rule, we get:

$$\frac{d}{ds_j}\left(\frac{u}{v}\right) = \frac{v\frac{du}{ds_j} - u\frac{dv}{ds_j}}{v^2}$$

Let $u = e^{s_i - m}$ and $v = \sum_{l=1}^{k} e^{s_l - m}$

$$\frac{\partial u}{\partial s_j} = \begin{cases} 0 & \text{if } i \neq j \\ e^{s_i - m} & \text{if } i = j \end{cases} = e^{s_j - m} \cdot \delta_{ij} \quad \text{with } \delta_{ij} \text{ being Kronecker delta}$$

$$\frac{\partial v}{\partial s_j} = e^{s_j - m}$$

Plugging the values of the partial derivates into the main equation yields:

$$\frac{\partial \sigma_i}{\partial s_j} = \frac{\sum_{l=1}^{k} e^{s_l - m}\ e^{s_j - m}\delta_{ij} - e^{s_i - m}e^{s_j - m}}{\left(\sum_{l=1}^{k} e^{s_l - m}\right)^2}$$

$$= \frac{e^{s_j - m}\delta_{ij}}{\sum_{l=1}^{k} e^{s_l - m}} - \frac{e^{s_i - m} e^{s_j - m}}{\left(\sum_{l=1}^{k} e^{s_l - m}\right)^2}$$

$$\text{with} \quad \frac{e^{s_j - m}}{\sum_{l=1}^{k} e^{s_l - m}} = \sigma_j,$$

$$= \sigma_j \delta_{ij} - \sigma_i \sigma_j = \sigma_j(\delta_{ij} - \sigma_i)$$

Vectorized solution:

$$\left[\frac{\delta L}{\delta z}\right]_i = \sum_{j=1}^{k} \frac{\delta L}{\delta y_j} \cdot \frac{\delta y_j}{\delta z_i}$$

$$\frac{\delta L}{\delta y_j} = dy_j$$

$$\frac{\delta y_j}{\delta z_i} = \frac{\delta \sigma_j}{\delta z_i} = \sigma_i(1 - \sigma_i) \quad \text{if } i = j$$

$$\frac{\delta y_j}{\delta z_i} = \frac{\delta \sigma_j}{\delta z_i} = -\sigma_i \sigma_j \quad \text{if } i \neq j$$

$$\text{In each row, the sum of } \frac{\delta y_j}{\delta z_i} \text{ has to equal } \sigma_i + \sum_{j=1}^{k} -\sigma_i\sigma_j, \ +1 \text{ for the one position where } i = j$$

$$\left[\frac{\delta L}{\delta z}\right]_i = \sum_{j=1, j\neq i}^{k} -dy_j \cdot \sigma_j \sigma_i + (dy_i \cdot \sigma_i - dy_i \cdot \sigma_i \sigma_i)$$

$$= -\sum_{j=1, j\neq i}^{k} dy_j \cdot \sigma_j \sigma_i + dy_i \cdot \sigma_i$$

$$= \sigma_i \left(dy_i - \sum_{j=1}^{k} dy_j \sigma_j \right)$$

$$= y_i \left(dy_i - \sum_{j=1}^{k} dy_j y_j \right)$$

Implementation of `activations.SoftMax`:

```
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 ###
        m = np.max(Z, axis=1, keepdims=True)

        sum_exp = np.sum(np.exp(Z - m), axis=1, keepdims=True)

        return np.exp(Z - m) / sum_exp

    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 ###

        Y = self.forward(Z)
        return Y * (dY - np.sum(Y * dY, axis=1, keepdims=True))
```

1.4 Cross-Entropy Loss

For this classification network, we will be using the multi-class cross-entropy loss function

$$L = -y \cdot \ln(\hat{y}),$$

where y is the binary one-hot vector encoding the ground truth labels and $\hat{y}$ is the network's output, a vector of probabilities over classes. Note that $\ln \hat{y}$ is $\hat{y}$ with the natural log applied elementwise to it and $\cdot$ represents the dot product between y and $\ln \hat{y}$. The cross-entropy cost calculated for a mini-batch of m samples is

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

Let $Y \in \mathbb{R}^{m\times k}$ and $\hat{Y} \in \mathbb{R}^{m\times k}$ be the one-hot labels and network outputs for the m samples, stacked in a matrix. Then, y_i and $\hat{y}_i$ in the expression above are just the ith rows of Y and $\hat{Y}$.

Instructions

1. **Derive** $\partial L/\partial \hat{Y}$ the gradient of the cross-entropy cost with respect to the network's predictions, $\hat{Y}$. First derive the gradient element-wise, i.e. find an expression for $[\partial L/\partial \hat{Y}]_{ij}$, and then stack these elements appropriately to obtain a simple vector/matrix expression for $\partial L/\partial \hat{Y}$. **You must use batched inputs.** Include your derivation in your writeup.
2. **Implement the forward and backward passes of the cross-entropy cost in `losses.py`**. Note that in the codebase we have provided, we use the words "loss" and "cost" interchangeably. This is consistent with most large neural network libraries, though technically "loss" denotes the function computed for a single datapoint whereas "cost" is computed for a batch. You will be computing the cost over mini-batches. **Do not iterate over training examples; use batched operations.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope.

Solution:

The cost function J is given by:

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

Using the definition of the dot product, this can be expanded to:

$$J = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{j=1}^{k} y_{ij} \cdot \ln(\hat{y}_{ij})\right)$$

The element-wise gradient of J with respect to $\hat{y}_{ij}$ is:

$$\left[\frac{\partial J}{\partial \hat{Y}}\right]_{ij} = -\frac{1}{m}\frac{\partial}{\partial \hat{y}_{ij}}\left(\sum_{i=1}^{m}\sum_{j=1}^{k} y_{ij} \cdot \ln(\hat{y}_{ij})\right) = -\frac{1}{m}\cdot\frac{y_{ij}}{\hat{y}_{ij}}$$

Thus, the full gradient can be expressed in matrix form as:

$$\frac{\partial J}{\partial \hat{Y}} = -\frac{1}{m}\cdot\frac{Y}{\hat{Y}}, \quad \text{where } \frac{Y}{\hat{Y}} \text{ is element-wise division}$$

Implementation of `losses.CrossEntropy`:

```
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
        """
        ### YOUR CODE HERE ###
        m = Y.shape[0]
        return -1/m * np.sum(Y * np.log(Y_hat))

    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 ###
        m = Y.shape[0]
        return -1/m * (Y / Y_hat)
```

2 Two-Layer Networks

Now, you will use the methods you've written to train a two-layer network (also referred to as a one-*hidden*-layer network). You will use the Iris Dataset, which contains 4 features for 3 different classes of irises.

Instructions

1. **Fill in the `forward`, `backward`, `predict` methods for the `NeuralNetwork` class in `models.py`.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope. Define the parameters of your network in `train_ffnn.py`. We have provided you with several other classes that are critical for the training process.

   - The data loader (in `datasets.py`), which is responsible for loading batches of data that will be fed to your model during training. You may wish to alter the data loader to handle data pre-processing. Note that all datasets you are given have not been normalized or standardized.
   - The stochastic gradient descent optimizer (in `optimizers.py`), which performs the gradient updates and optionally incorporates a momentum term.
   - The learning rate scheduler (in `schedulers.py`), which handles the optional learning rate decay. You may choose to use either a constant or exponentially decaying learning rate.
   - Weight initializers (in `weights.py`). We provide you with many options to explore, but we recommend using `xavier_uniform` as a default.
   - A logger (in `logs.py`), which saves hyperparameters and learned parameters and plots the loss as your model trains.

   Outputs will be saved to the folder `experiments/`. You can change the name of the folder a given run saves to by changing the parameter called `model_name`. Be careful about overwriting folders; if you forget to change the name and perform a run with identical hyperparameters, your previous run will be overwritten!

2. **Train a 2-layer neural network on the Iris Dataset by running `train_ffnn.py`.** Vary the following hyperparameters.

   - Learning rate
   - Hidden layer size

   You must try at least 4 different *combinations* of these hyperparameters. Report the results of your exploration, including the values of the parameters you tried and which set of parameters yielded the best test error. Comment on how changing these hyperparameters affected the test error. Provide a plot showing the training and validation loss across different epochs for your best model and report your final test error.

Solution:

PART 1

Implementation of `models.NeuralNetwork.forward`:

```
    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 layer in self.layers:
            X = layer.forward(X)

        return X
```

Implementation of `models.NeuralNetwork.backward`:

```
    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.

        L = self.loss.forward(target, out)
        dLdY = self.loss.backward(target, out)
        for layer in reversed(self.layers):
            dLdY = layer.backward(dLdY)

        return L
```

Implementation of `models.NeuralNetwork.predict`:

```
    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)
        L = self.backward(Y, Y_hat)
        return Y_hat, L
```

PART 2

I first tried various hidden layer sizes (25, 50, 10, 5, 3, 2) while keeping learning rate constant (0.01)

1. Iteration (n_out = 25, LR = 0.01): Epoch 99 Training Loss: 0.0846, Training Accuracy: 0.952, Validation Loss: 0.0201, Validation Accuracy: 1.0, Test Loss: 0.3021, Test Accuracy: 0.88
2. Iteration (n_out = 50, LR = 0.01): Epoch 99 Training Loss: 0.0541, Training Accuracy: 0.984, Validation Loss: 0.0447, Validation Accuracy: 1.0, Test Loss: 0.2547, Test Accuracy: 0.88
3. Iteration (n_out = 10, LR = 0.01): Epoch 99 Training Loss: 0.0653 Training Accuracy: 0.976 Val Loss: 0.0111 Val Accuracy: 1.0, Test Loss: 0.1349 Test Accuracy: 0.96
4. Iteration (n_out = 5, LR = 0.01): Epoch 99 Training Loss: 0.1127 Training Accuracy: 0.92 Val Loss: 0.056 Val Accuracy: 1.0 Test Loss: 0.067 Test Accuracy: 0.96
5. Iteration (n_out = 3, LR = 0.01): Epoch 99 Training Loss: 0.1911 Training Accuracy: 0.968 Val Loss: 0.1197 Val Accuracy: 1.0 Test Loss: 0.2112 Test Accuracy: 0.98
6. Iteration (n_out = 2, LR = 0.01): Epoch 99 Training Loss: 0.1925 Training Accuracy: 0.976 Val Loss: 0.1065 Val Accuracy: 1.0 Test Loss: 0.1941 Test Accuracy: 0.98
7. Iteration (n_out = 1, LR = 0.01): Epoch 99 Training Loss: 1.1014 Training Accuracy: 0.352 Val Loss: 1.1311 Val Accuracy: 0.2 Test Loss: 1.0927 Test Accuracy: 0.46

   It was very surprising that fewer hidden layers lead to higher test accuracy - intuitively, one would think different. Setting hidden layer size to 1, however, makes the model a lot worse.

   Next, I tried varying the learning rate and applying it to different hidden layer numbers:

8. Iteration (n_out = 3, LR = 0.1): Epoch 99 Training Loss: 1.1168 Training Accuracy: 0.384 Val Loss: 1.2223 Val Accuracy: 0.2 Test Loss: 1.1255 Test Accuracy: 0.46
9. Iteration (n_out = 50, LR = 0.1): Epoch 99 Training Loss: 0.0671 Training Accuracy: 0.968 Val Loss: 0.0827 Val Accuracy: 0.9333 Test Loss: 0.1802 Test Accuracy: 0.9
10. Iteration (n_out = 2, LR = 0.005): Epoch 99 Training Loss: 0.375 Training Accuracy: 0.976 Val Loss: 0.2427 Val Accuracy: 1.0 Test Loss: 0.3714 Test Accuracy: 0.98

    The higher learning rate made the model with 3 hidden layers drastically worse, while the model with 50 hidden layers also got worse (although not that big of a difference compared to the other model)

    After a few more iterations (which I am not going to explicitely show here), my final model that performed the best was with 2 hidden layers and a learning rate = 0.014. It achieved a test accuracy of 0.98 while having a test loss of 0.1705.

11. Iteration (n_out = 2, LR = 0.014): Epoch 99 Training Loss: 0.1609 Training Accuracy: 0.928 Val Loss: 0.0758 Val Accuracy: 1.0 Test Loss: 0.1705 Test Accuracy: 0.98

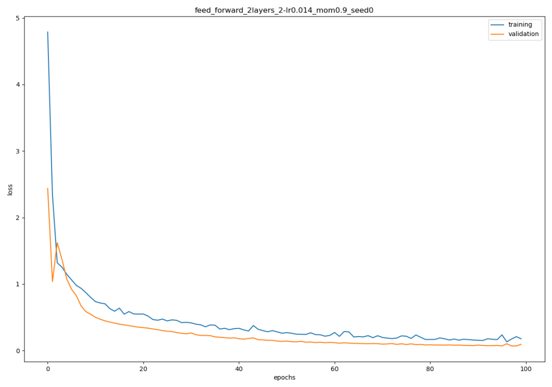

Figure 1: Training and validation loss for hidden layer size = 2 and learning rate = 0.014

3 CNN Layers

In this problem, you will only derive the gradients for the convolutional and pooling layers used within CNNs. **There is no coding portion for this question.**

3.1 Convolutional and Pooling Layers

Instructions

1. **Derive the gradient of the loss with respect to the input and parameters (kernels and biases) of a convolutional layer.** For this question your answer may be in the form of individual component partial derivatives. Assume you have access to the full 3d array $\frac{\partial L}{\partial Z[d_1, d_2, n]}$, which is the gradient of the pre-activation w.r.t. the loss. **You do not need to use batched inputs for this question; an answer for a single training point is acceptable.** For the sake of simplicity, you may also ignore stride and assume both the filter and image are infinitely zero-padded outside of their bounds.

   (a) What is $\frac{\partial L}{\partial b[f]}$ for an arbitrary $f \in [1, \dots, n]$?

   (b) What is $\frac{\partial L}{\partial W[i, k, c, f]}$ for arbitrary i, k, c, f indexes?

   (c) What is $\frac{\partial L}{\partial X[x, y, c]}$ for arbitrary x, y, c indexes?

   Include your derivations in your writeup.

2. **Explain how we can use the backprop algorithm to compute gradients through the max pooling and average pooling operations.** (A plain English answer, explained clearly, will suffice; equations are optional.)

Solution: 1.

$$\text{(a)} \quad \frac{\partial L}{\partial b[f]} = \sum_{d_1, d_2} \frac{\partial L}{\partial Z[d_1, d_2, f]} \cdot \frac{\partial Z[d_1, d_2, f]}{\partial b[f]} = \sum_{d_1, d_2} \frac{\partial L}{\partial Z[d_1, d_2, f]}$$

$$\text{where} \quad \frac{\partial Z[d_1, d_2, f]}{\partial b[f]} = 1 \quad \text{for arbitrary } f \in [1, \dots, n]$$

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

$$\frac{\partial Z[d_1,d_2,f]}{\partial W[i,k,c,f]} = X[d_1+i, d_2+k, c]$$

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

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

$$x = d_1 + i \quad y = d_2 + j \quad c = c$$

$$d_1 = x - i \quad d_2 = y - j \quad i = x - d_1 \quad j = y - d_2$$

$$\frac{\partial Z[d_1,d_2,n]}{\partial X[x,y,c]} = W[x-d_1, y-d_2, c, n]$$

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

2.

Average Pooling: Forward pass computes the average value of the input and it on. In the backward pass, the gradient fof loss regarding the input is calculated by taking gradient of loss regarding the output and equally distributing it, i.e. dividing by number of inputs, to all the input positions equally.

Max pooling: Forward pass records the position of the maximum value in each pooling window, and in the backward pass, the gradient of the loss regarding the input is equal to gradient of loss regarding output, only for the input with max value in the window, and will be 0 otherwise.

4 PyTorch

In this section, you will train neural networks in PyTorch. Please make a copy of the Google Colab Notebook here and find the necessary data files in the `datasets/` folder of the starter code. **The Colab Notebook will walk you through all the steps for completing for this section, where we have copied the deliverables for your writeup below.**

As with every homework, you are allowed to use any setup you wish. However, we highly recommend using Google Colab for it provides free access to GPUs, which will significantly improve the training speed for neural networks. Instructions on using the Colab-provided GPUs are within the notebook itself. If you have access to your own GPUs, feel free to run the notebook locally on your computer.

4.1 MLP for Fashion MNIST

Deliverables

- Code for training an MLP on FashionMNIST.
- A plot of the training and validation loss for at least 8 epochs.
- A plot of the training and validation accuracy for each epoch, achieving a final validation accuracy of at least 82%.

Solution: TODO

4.2 CNNs for CIFAR-10

Deliverables

- Code for training a CNN on CIFAR-10.
- Provide at least 1 training curve for your model, depicting loss per epoch after training for at least 8 epochs.
- Explain the components of your final model, and how you think your design choices contributed to its performance.
- A `predictions.csv` file that will be submitted to the Gradescope autograder, achieving a final test accuracy of at least 75%. You will receive half credit if you achieve an accuracy of at least 70%.

Solution: TODO

5 Honor Code

1. **List all collaborators. If you worked alone, then you must explicitly state so.**

   Solution: 1. Adil Köken

2. **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 the consequences of academic misconduct are *particularly severe*!