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:

The ReLU activation function is defined element-wise as:

$$Y_{ij} = \sigma_{\text{ReLU}}(Z_{ij}) = \begin{cases} Z_{ij}, & \text{if } Z_{ij} \geq 0, \\ 0, & \text{if } Z_{ij} < 0. \end{cases}$$

The derivative of Y_{ij} with respect to Z_{ij} is:

$$\frac{\partial Y_{ij}}{\partial Z_{ij}} = \begin{cases} 1, & \text{if } Z_{ij} \geq 0, \\ 0, & \text{if } Z_{ij} < 0. \end{cases}$$

Using the chain rule, the gradient of the loss function L with respect to Z_{ij} is:

$$\frac{\partial L}{\partial Z_{ij}} = \frac{\partial L}{\partial Y_{ij}} \cdot \frac{\partial Y_{ij}}{\partial Z_{ij}}.$$

Thus, we can write the element-wise gradient more compactly as:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \odot \mathbb{I}(Z \geq 0),$$

where $\odot$ represents the element-wise (Hadamard) product, and $\mathbb{I}(Z \geq 0)$ is the indicator function applied element-wise on Z, which equals 1 when $Z \geq 0$ and 0 otherwise.

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:

Given the forward pass for a fully-connected layer:

$$Z = XW + \mathbf{1}b$$

where $X \in \mathbb{R}^{m \times n^{[l]}}$, $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$, $\mathbf{1} \in \mathbb{R}^{m \times 1}$ is a column vector of ones, and $b \in \mathbb{R}^{1 \times n^{[l+1]}}$, we will derive the gradients $\frac{\partial L}{\partial W}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial X}$.

Form Fradient of the Loss with Respect to the Weights W:

We first compute the element-wise gradient of the loss with respect to W_{ij}. The loss L depends on W through Z, so we use the chain rule:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z_{ki}} \cdot \frac{\partial Z_{ki}}{\partial W_{ij}}.$$

From the forward pass equation, we have:

$$Z_{ki} = \sum_{j} X_{kj} W_{ji} + b_i,$$

so

$$\frac{\partial Z_{ki}}{\partial W_{ij}} = X_{kj}.$$

Therefore, the gradient of the loss with respect to W is:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z_{ki}} \cdot X_{kj}.$$

In matrix form, we can express this more compactly as:

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

For Gradient of the Loss with Respect to the Bias b:

The bias term b is shared across all rows, so the gradient with respect to b_i is:

$$\frac{\partial L}{\partial b_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z_{ki}} \cdot \frac{\partial Z_{ki}}{\partial b_i}.$$

Since $\frac{\partial Z_{ki}}{\partial b_i} = 1$, this simplifies to:

$$\frac{\partial L}{\partial b_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z_{ki}}.$$

In matrix form:

$$\frac{\partial L}{\partial b} = \mathbf{1}^T \frac{\partial L}{\partial Z},$$

where $\mathbf{1}^T$ is a row vector of ones of length m.

For Gradient of the Loss with Respect to the Input X:

Again, we use the chain rule:

$$\frac{\partial L}{\partial X_{ij}} = \sum_{k=1}^{n^{[l+1]}} \frac{\partial L}{\partial Z_{ik}} \cdot \frac{\partial Z_{ik}}{\partial X_{ij}}.$$

Since

$$\frac{\partial Z_{ik}}{\partial X_{ij}} = W_{jk},$$

we have:

$$\frac{\partial L}{\partial X_{ij}} = \sum_{k=1}^{n^{[l+1]}} \frac{\partial L}{\partial Z_{ik}} \cdot W_{jk}.$$

In matrix form, this becomes:

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

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:

Given the softmax function:

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

We are asked to find the derivative of σ_i with respect to s_j.

Case 1: $i = j$ Taking the partial derivative of σ_i with respect to s_i:

$$\frac{\partial \sigma_i}{\partial s_i} = \frac{\partial}{\partial s_i} \left(\frac{e^{s_i}}{\sum_{l=1}^{k} e^{s_l}} \right)$$

Using the quotient rule:

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

Simplifying:

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

$$\frac{\partial \sigma_i}{\partial s_i} = \sigma_i\left(1 - \sigma_i\right)$$

Case 2: $i \neq j$ We differentiate σ_i with respect to s_j:

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

Since s_j only affects the denominator and not the numerator:

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

Simplifying:

$$\frac{\partial \sigma_i}{\partial s_j} = -\sigma_i\sigma_j$$

Thus, the Jacobian of the softmax function for an arbitrary pair (i, j) is:

$$\frac{\partial \sigma_i}{\partial s_j} = \begin{cases} \sigma_i(1 - \sigma_i) & \text{if } i = j \\ -\sigma_i\sigma_j & \text{if } i \neq j \end{cases}$$

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:

Given the cross-entropy loss function for a mini-batch of size m, we have:

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

Where: J is the loss; y_{ij} is the true label (one-hot encoded) for class j and sample i; $\hat{y}_{ij}$ is the predicted probability for class j and sample i; m is the number of samples in the batch; k is the number of classes.

To derive the gradient of the loss with respect to $\hat{y}_{ij}$, we compute the partial derivative of J with respect to $\hat{y}_{ij}$:

$$\frac{\partial J}{\partial \hat{y}_{ij}} = -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial \hat{y}_{ij}}\left(y_{ij}\log(\hat{y}_{ij})\right)$$

Applying the chain rule to the logarithm function:

$$\frac{\partial}{\partial \hat{y}_{ij}}\left(y_{ij}\log(\hat{y}_{ij})\right) = \frac{y_{ij}}{\hat{y}_{ij}}$$

Thus, the gradient becomes:

$$\frac{\partial J}{\partial \hat{y}_{ij}} = -\frac{1}{m}\left(\frac{y_{ij}}{\hat{y}_{ij}}\right)$$

Now, simplifying the equation for the entire batch:

$$\frac{\partial J}{\partial \hat{Y}} = -\frac{1}{m}\frac{Y}{\hat{Y}}$$

However, the simplification assumes that $\hat{y}_{ij}$ represents softmax output probabilities, leading to the well-known simplification:

$$\frac{\partial J}{\partial \hat{y}_{ij}} = \frac{1}{m}(\hat{y}_{ij} - y_{ij})$$

Finally, for the entire batch, we obtain the vectorized form:

$$\frac{\partial J}{\partial \hat{Y}} = \frac{1}{m}(\hat{Y} - Y)$$

where Y represents the true one-hot encoded labels, and $\hat{Y}$ represents the predicted probabilities (output of the softmax).

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:

I conducted experiments using four different configurations: (See below for their plots)

Configuration 1: 25 hidden layers with a learning rate of 0.01 and a constant learning rate decay. Configuration 2: 25 hidden layers with a learning rate of 0.1 and a constant learning rate decay. Configuration 3: 25 hidden layers with a learning rate of 0.1 and an exponential learning rate decay. Configuration 4: 100 hidden layers with a learning rate of 0.01 and a constant learning rate decay.

Comparison Between Configuration 1 and Configuration 2: Configuration 1, which uses a smaller learning rate of 0.01, provided more stable convergence. Both training and validation errors consistently decreased, leading to better generalization. In contrast, Configuration 2, with a higher learning rate of 0.1, caused oscillations in the error rates.

This instability suggests that the model struggled to find optimal weights and may have been prone to overfitting the training data due to the lack of smooth convergence. Although both learning rates eventually achieved low errors, the model with the 0.01 learning rate converged more smoothly and maintained a better balance between training and validation errors. The 0.1 learning rate, while initially faster, exhibited instability that could degrade performance on more complex datasets or during extended training sessions.

Comparison Between Configuration 1 and Configuration 4: Both Configuration 1 and Configuration 4 successfully fit the training data without overfitting, achieving low errors. However, the model with 25 hidden layers (Configuration 1) stabilized more quickly and exhibited fewer oscillations. In contrast, the model with 100 hidden layers (Configuration 4) took longer to stabilize and showed some instability in the early epochs before eventually converging, likely due to the increased number of parameters that needed to be optimized. Regarding validation behavior, both models achieved low validation errors, but the 25-layer model stabilized more rapidly with fewer fluctuations, whereas the 100-layer model displayed early instability before ultimately converging.

Comparison Between Configuration 2 and Configuration 3: Configuration 2 employed a constant decay, which generally resulted in a smoother training curve but sometimes struggled to achieve fine-tuned convergence for validation accuracy. On the other hand, Configuration 3 used an exponential decay, providing more initial flexibility but risking premature convergence if the learning rate decayed too quickly. This was particularly evident with the small Iris dataset, where exponential decay led to greater fluctuations in validation error. For this specific dataset and model, constant decay performed slightly better by maintaining more consistent validation accuracy towards the end of training, whereas exponential decay introduced more volatility in validation performance.

In summary, a smaller learning rate with constant decay (Configuration 1) offered the most stable and balanced performance across training and validation metrics. Increasing the learning rate or the number of hidden layers introduced instability, and while different decay strategies have their advantages, constant decay proved to be more effective for maintaining consistent validation accuracy in this context.

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 $f \in [1, \ldots, n]$?

   Solution:

   The output of a convolutional layer is generally expressed as:

   $$Z[f,i,j] = \sum_{c=1}^{C} \sum_{m=1}^{M} \sum_{n=1}^{N} W[f,c,m,n] \cdot X[c,i+m,j+n] + b[f]$$

   where $Z[f,i,j]$ is the pre-activation at spatial location (i,j) in feature map f; $W[f,c,m,n]$ is the kernel/filter weight for channel c and spatial dimensions $m \times n$; $X[c,i+m,j+n]$ is the input at channel c; $b[f]$ is the bias for feature map f.

   Since the bias $b[f]$ is added to every spatial location of the feature map $Z[f]$, the gradient of the loss with respect to the bias for feature map f is the sum of the gradients of the loss with respect to the pre-activation at all locations in the feature map:

   $$\frac{\partial L}{\partial b[f]} = \sum_{i,j} \frac{\partial L}{\partial Z[f,i,j]}$$

   where $\frac{\partial L}{\partial Z[f,i,j]}$ is the gradient of the pre-activation at location (i,j) with respect to the loss L.

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

   Solution:

   To compute the gradient of the loss with respect to the weight $W[i,k,c,f]$, we use the chain rule.

   The gradient of the loss with respect to $W[i,k,c,f]$ can be computed as:

   $$\frac{\partial L}{\partial W[i,k,c,f]} = \sum_{x,y} \frac{\partial L}{\partial Z[f,x,y]} \cdot \frac{\partial Z[f,x,y]}{\partial W[i,k,c,f]}$$

   Since $Z[f,x,y]$ is a linear function of $W[i,k,c,f]$, the partial derivative $\frac{\partial Z[f,x,y]}{\partial W[i,k,c,f]}$ is simply the corresponding input value $X[c,x+i,y+k]$. Thus, we have:

$$\frac{\partial L}{\partial W[i,k,c,f]} = \sum_{x,y} \frac{\partial L}{\partial Z[f,x,y]} \cdot X[c, x+i, y+k]$$

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

Solution:

We apply the chain rule:

$$\frac{\partial L}{\partial X[x,y,c]} = \sum_{f} \sum_{i,j} \frac{\partial L}{\partial Z[f,i,j]} \cdot \frac{\partial Z[f,i,j]}{\partial X[x,y,c]}$$

Since $Z[f,i,j]$ is a linear function of X, the partial derivative $\frac{\partial Z[f,i,j]}{\partial X[x,y,c]}$ is non-zero only when the input at $X[x,y,c]$ contributes to $Z[f,i,j]$, which occurs when $i+m=x$ and $j+n=y$ for some m and n. Thus:

$$\frac{\partial Z[f,i,j]}{\partial X[x,y,c]} = W[f,c,x-i,y-j]$$

Therefore, the gradient with respect to the input is:

$$\frac{\partial L}{\partial X[x,y,c]} = \sum_{f} \sum_{i,j} \frac{\partial L}{\partial Z[f,i,j]} \cdot W[f,c,x-i,y-j]$$

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:

In max pooling, the gradient is passed only to the maximum value in the pooling region. During the forward pass, we find the index of the maximum value for each pooling region. During backpropagation, the gradient $\frac{\partial L}{\partial Z}$ from the next layer is passed back to this index, while the gradients for all other elements in the pooling region are set to zero.

Mathematically, for a pooling region P where $Z_{\max} = \max(P)$, the gradient $\frac{\partial L}{\partial X}$ with respect to the input X is:

$$\frac{\partial L}{\partial X[i,j]} = \begin{cases} \frac{\partial L}{\partial Z[i',j']} & \text{if } X[i,j] = Z_{\max}, \\ 0 & \text{otherwise.} \end{cases}$$

In average pooling, the gradient is distributed evenly across all values in the pooling region. During the forward pass, we calculate the average value over each pooling region. During backpropagation, the gradient from the next layer is divided equally among all the elements of the pooling region.

Mathematically, for a pooling region of size $k \times k$, the gradient $\frac{\partial L}{\partial X}$ with respect to the input is:

$$\frac{\partial L}{\partial X[i,j]} = \frac{1}{k^2} \frac{\partial L}{\partial Z[i',j']}.$$

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:

```
# Define the MLP model (multi-layer perceptron)
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # Flatten the 28x28 image into a single vector of size 784 (28*28)
        self.flatten = nn.Flatten()

        # First fully connected layer: input size is 784 (flattened image) and output size is 256
        self.fc1 = nn.Linear(28 * 28, 256)

        # Second fully connected layer: takes 256 inputs, gives 128 outputs
        self.fc2 = nn.Linear(256, 128)

        # Output layer: 10 classes for the 10 categories in FashionMNIST
        self.fc3 = nn.Linear(128, 10)

        # ReLU as activation function for non-linearity
        self.relu = nn.ReLU()

    # Forward pass: Defines how data flows through the network
    def forward(self, x):
        x = self.flatten(x)  # Flatten the input image
        x = self.relu(self.fc1(x))  # Apply first layer and ReLU activation
        x = self.relu(self.fc2(x))  # Apply second layer and ReLU activation
        x = self.fc3(x)  # Apply output layer (no activation because we use CrossEntropyLoss)
        return x

# Set hyperparameters for training
batch_size = 64       # How many images to process at once
learning_rate = 0.001 # Step size for the optimizer
epochs = 10           # Number of complete passes through the training dataset

# Dataloaders to handle mini-batches of data for training and validation
train_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=False)

# Initialize the model, loss function, and optimizer
model = MLP()  # Instantiate the MLP model
loss_fn = nn.CrossEntropyLoss()  # Cross entropy loss, used for multi-class classification
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)  # Adam optimizer with set learning rate

# Lists to track the performance over epochs
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Check if GPU (CUDA) is available and move the model to the GPU if possible
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = model.to(device)

# Training and validation loop
for epoch in range(epochs):
    model.train()  # Set the model to training mode
    train_loss = 0  # Reset training loss for this epoch
    correct_train = 0  # Reset training accuracy counter
    total_train = 0  # Total training samples seen in this epoch

    # Loop through mini-batches of training data
    for images, labels in tqdm.tqdm(train_loader):
        images, labels = images.to(device), labels.to(device)  # Move data to GPU if available
        outputs = model(images)  # Forward pass
        loss = loss_fn(outputs, labels)  # Compute loss

        optimizer.zero_grad()  # Reset gradients from the previous step
        loss.backward()  # Backpropagation to compute gradients
        optimizer.step()  # Update model parameters

        # Accumulate the training loss and accuracy
        train_loss += loss.item()
        _, predicted = outputs.max(1)  # Get the predicted class
        total_train += labels.size(0)  # Total number of training samples in this batch
        correct_train += (predicted == labels).sum().item()  # Correct predictions in this batch
    # Append average training loss and accuracy
    train_losses.append(train_loss / len(train_loader))
    train_accuracies.append(100 * correct_train / total_train)

    # Switch to evaluation mode for validation
    model.eval()
    val_loss = 0
    correct_val = 0
    total_val = 0

    # Loop through validation data (same as training, but no weight updates)
    with torch.no_grad():  # Disable gradient calculations for validation
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = loss_fn(outputs, labels)
            val_loss += loss.item()

            _, predicted = outputs.max(1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    # Append average validation loss and accuracy
    val_losses.append(val_loss / len(val_loader))
    val_accuracies.append(100 * correct_val / total_val)

    # Print epoch summary for both training and validation
    print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4f}, T

# Plotting training and validation loss and accuracy over epochs
plt.figure(figsize=(12, 5))

# Loss plot
plt.subplot(1, 2, 1)
plt.plot(range(1, epochs+1), train_losses, label='Training Loss')
plt.plot(range(1, epochs+1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Accuracy plot
plt.subplot(1, 2, 2)
plt.plot(range(1, epochs+1), train_accuracies, label='Training Accuracy')
plt.plot(range(1, epochs+1), val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.tight_layout()
plt.show()
```

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:

```
# Imports
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import tqdm.notebook as tqdm

# Setting up the device to utilize GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define data transforms (add normalization and augmentations)
transform = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomCrop(32, padding=4),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 training and test data
training_data = torchvision.datasets.CIFAR10(
    root="data", train=True, download=True, transform=transform
)

test_data = torchvision.datasets.CIFAR10(
    root="data", train=False, download=True, transform=transform
)

# Create data loaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False)

# CNN Architecture with Batch Normalization
```

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

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)  # Batch Norm after conv1
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)  # Batch Norm after conv2
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)  # Batch Norm after conv3
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(128 * 4 * 4, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.pool(self.relu(self.bn2(self.conv2(x))))
        x = self.pool(self.relu(self.bn3(self.conv3(x))))

        x = x.view(-1, 128 * 4 * 4)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Model, loss function, and optimizer
model = CNN().to(device)
loss_fn = nn.CrossEntropyLoss()

# Switching optimizer to SGD with momentum and adding learning rate scheduler
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)

def train_model(model, train_loader, test_loader, epochs):
    train_losses, test_losses = [], []  # Correct initialization
    train_accuracies, test_accuracies = [], []  # Initialize two separate empty lists

    for epoch in range(epochs):
        model.train()
        total_train_loss, correct_train, total_train = 0, 0, 0

        # Training loop
        for images, labels in tqdm.tqdm(train_loader):
            images, labels = images.to(device), labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = loss_fn(outputs, labels)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_train_loss += loss.item()
            _, predicted = outputs.max(1)
            correct_train += (predicted == labels).sum().item()
            total_train += labels.size(0)

        # Adjust learning rate
        scheduler.step()

        # Average training loss and accuracy
        train_loss = total_train_loss / len(train_loader)
        train_acc = correct_train / total_train * 100
        train_losses.append(train_loss)
        train_accuracies.append(train_acc)

        # Validation loop
        model.eval()
        total_test_loss, correct_test, total_test = 0, 0, 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = loss_fn(outputs, labels)
                total_test_loss += loss.item()
                _, predicted = outputs.max(1)
                correct_test += (predicted == labels).sum().item()
                total_test += labels.size(0)

        # Average test loss and accuracy
        test_loss = total_test_loss / len(test_loader)
        test_acc = correct_test / total_test * 100
        test_losses.append(test_loss)
        test_accuracies.append(test_acc)

        print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, Test

    return train_losses, test_losses, train_accuracies, test_accuracies

# Train the model for 30 epochs
```

```
train_losses, test_losses, train_accuracies, test_accuracies = train_model(model, train_loader, test_loader

# Plotting the training and validation losses
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss per Epoch')
plt.legend()

# Plotting the training and validation accuracies
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label="Train Accuracy")
plt.plot(test_accuracies, label="Test Accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy per Epoch')
plt.legend()

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

Design Choices:
Convolutional Layers: The three convolutional layers use 32, 64, and 128 filters, respectively. These progressively deeper layers help capture both low-level and high-level features. The kernel size is 3x3, which is a standard choice that effectively captures spatial hierarchies in small datasets like CIFAR-10.

Max Pooling: A 2x2 max-pooling layer is applied after each convolutional layer to reduce the spatial dimensions and computation, while retaining important features. This downsampling also makes the model less sensitive to small translations in the input data.

Activation Function (ReLU): ReLU is applied after each convolutional and fully connected layer to introduce non-linearity. This helps the network learn more complex functions and improves convergence.

Fully Connected Layers: Two fully connected layers (256 and 128 units) are used after the convolutional layers. These layers integrate the features learned by the convolutional layers and prepare them for classification. A final fully connected layer with 10 outputs corresponds to the 10 classes of CIFAR-10.

Dropout (0.5): Dropout is applied to the first fully connected layer to prevent overfitting by randomly dropping half of the neurons during training. This forces the network to learn more robust features and improves generalization.

Adam Optimizer (learning rate = 0.001): Adam is used as the optimizer because it adapts the learning rate during training, making it efficient for faster convergence without needing extensive hyperparameter tuning.

Data Augmentation (random horizontal flip, random crop): Data augmentation is introduced to improve generalization by generating more diverse training samples. Random horizontal flips and random cropping ensure the model is exposed to slightly varied versions of the images, helping it generalize better to unseen data.

5 Honor Code

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

   Solution: N/A

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*!

   Solution: Zhe Wee Ng

6 Appendix

1. **Q1.1.2: activations.py**

```
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 ###
        dZ = np.where(Z > 0, dY, 0)
        return dZ
```

2. **Q1.2.2: layers.py**

```
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 ###

        # Initialize weights and bias
        W = self.init_weights((self.n_in, self.n_out))  # W has shape (n_in, n_out)
        b = np.zeros((1, self.n_out))  # Bias is initialized to zeros with shape (1, n_out)

        # Store parameters and initialize cache and gradients
        self.parameters = OrderedDict({"W": W, "b": b})  # store W and b
        self.cache = OrderedDict()  # cache for storing forward pass values
        self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)})  # gradients init

        ### 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 ###

        # perform an affine transformation and activation

        # Affine transformation (W * X + b)
        Z = X @ self.parameters["W"] + self.parameters["b"]

        # Apply activation function
        A = self.activation.forward(Z)
        out = A

        # store information necessary for backprop in ‘self.cache‘
        self.cache["X"] = X  # input
        self.cache["Z"] = Z  # pre-activation output

        ### 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"]

        # Derivative of the activation function
        dLdZ = self.activation.backward(Z, dLdY)
        # compute the gradients of the loss w.r.t. all parameters as well as the
        # input of the layer
        # 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  # Gradient w.r.t. W (input transposed times dLdZ)
        self.gradients["b"] = np.sum(dLdZ, axis=0, keepdims=True)  # Gradient w.r.t. b (sum of dLdZ al
        dLdX = dLdZ @ self.parameters["W"].T  # Gradient w.r.t. input X (dLdZ times W transpose)
        dX = dLdX

        ### END YOUR CODE ###

        return dX
```

3. **Q1.3.2: activations.py**

```
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 ###
        # Subtract the max for numerical stability
        shift_z = Z - np.max(Z, axis=1, keepdims=True)
        exp_z = np.exp(shift_z)
        softmaxOutput = exp_z / np.sum(exp_z, axis=1, keepdims=True)

        # Store the output for use in the backward pass
        self.cache = softmaxOutput
        return softmaxOutput

    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 ###
        softmaxOutput = self.cache

        # Initialize an empty matrix to hold gradients
        dZ = np.zeros_like(dY)

        # Use a loop over the mini-batch
        for i in range(dY.shape[0]):
            y_i = softmaxOutput[i].reshape(-1, 1)
            jacobian = np.diagflat(y_i) - np.dot(y_i, y_i.T)  # Jacobian of the softmax
            dZ[i] = np.dot(jacobian, dY[i])

        return dZ
```

4. **Q1.4.2: losses.py**

```
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 ###
        # Number of samples in the batch
        m = Y.shape[0]

        # Compute the cross-entropy loss (avoid log(0) by adding a small epsilon)
        epsilon = 1e-15
        Y_hat_clipped = np.clip(Y_hat, epsilon, 1 - epsilon)  # Ensure no 0s in log
        loss = -np.sum(Y * np.log(Y_hat_clipped)) / m

        return loss

    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 ###
        # Number of samples in the batch
        m = Y.shape[0]

        # Gradient of the loss with respect to Y_hat
        dY_hat = (Y_hat - Y) / m

        return dY_hat
```

5. **Q2.1: models.py**

```
from abc import ABC, abstractmethod
import numpy as np

from neural_networks.losses import initialize_loss
from neural_networks.optimizers import initialize_optimizer
from neural_networks.layers import initialize_layer
from collections import OrderedDict
import pickle
from tqdm import tqdm
import pandas as pd

# imports for typing only
from neural_networks.utils import AttrDict
from neural_networks.datasets import Dataset
from typing import Any, Dict, List, Sequence, Tuple


def initialize_model(name, loss, layer_args, optimizer_args, logger=None, seed=None):

    return NeuralNetwork(
        loss=loss,
        layer_args=layer_args,
        optimizer_args=optimizer_args,
        logger=logger,
        seed=seed,
    )


class NeuralNetwork(ABC):
    def __init__(
        self,
        loss: str,
        layer_args: Sequence[AttrDict],
        optimizer_args: AttrDict,
        logger=None,
        seed: int = None,
    ) -> None:

        self.n_layers = len(layer_args)
        self.layer_args = layer_args
        self.logger = logger
        self.epoch_log = {"loss": {}, "error": {}}

        self.loss = initialize_loss(loss)
        self.optimizer = initialize_optimizer(**optimizer_args)
        self._initialize_layers(layer_args)
    def _initialize_layers(self, layer_args: Sequence[AttrDict]) -> None:
        self.layers = []
        for l_arg in layer_args[:-1]:
            l = initialize_layer(**l_arg)
            self.layers.append(l)

    def _log(self, loss: float, error: float, validation: bool = False) -> None:

        if self.logger is not None:
            if validation:

                self.epoch_log["loss"]["validate"] = round(loss, 4)
                self.epoch_log["error"]["validate"] = round(error, 4)
                self.logger.push(self.epoch_log)
                self.epoch_log = {"loss": {}, "error": {}}
            else:
                self.epoch_log["loss"]["train"] = round(loss, 4)
                self.epoch_log["error"]["train"] = round(error, 4)

    def save_parameters(self, epoch: int) -> None:
        parameters = {}
        for i, l in enumerate(self.layers):
            parameters[i] = l.parameters
        if self.logger is None:
            raise ValueError("Must have a logger")
        else:
            with open(
                self.logger.save_dir + "parameters_epoch{}".format(epoch), "wb"
            ) as f:
                pickle.dump(parameters, f)

    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.
        self.cache = [X]  # To store intermediate activations for backpropagation
        out = X
        for layer in self.layers:
            out = layer.forward(out)
            self.cache.append(out)
        return out

    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
        lossValue = self.loss.forward(target, out)

        #  Compute the initial gradient of the loss with respect to the output
        dL = self.loss.backward(target, out)

        # Backpropagate through the network’s layers
        for i in reversed(range(len(self.layers))):
            dL = self.layers[i].backward(dL)

        return lossValue

    def update(self, epoch: int) -> None:
        """One step of gradient update using the derivatives calculated by
        ‘self.backward‘.

        Parameters
        ----------
        epoch  the epoch we are currently on
        """
        param_log = {}
        for i, layer in enumerate(self.layers):
            for param_name, param in layer.parameters.items():
                if param_name != "null":  # FIXME: possible change needed to ‘is not‘
                    param_grad = layer.gradients[param_name]
                    # Optimizer needs to keep track of layers
                    delta = self.optimizer.update(
                        param_name + str(i), param, param_grad, epoch
                    )
                    layer.parameters[param_name] -= delta
                    if self.logger is not None:
                        param_log["{}{}".format(param_name, i)] = {}
                        param_log["{}{}".format(param_name, i)]["max"] = np.max(param)
                        param_log["{}{}".format(param_name, i)]["min"] = np.min(param)
            layer.clear_gradients()
        self.epoch_log["params"] = param_log

    def error(self, target: np.ndarray, out: np.ndarray) -> float:
        """Only calculate the error of the model’s predictions given ‘target‘.

        For classification tasks,
            error = 1 - accuracy

        For regression tasks,
            error = mean squared error

        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 features corresponding to
                ‘target‘

        Returns
        -------
        the error of the model given the training inputs and targets
        """
        # Print the shape of ‘out‘ and ‘target‘
        print(f"Shape of out (Y_hat): {out.shape}")
        print(f"Shape of target (Y): {target.shape}")

        # classification error
        if self.loss.name == "cross_entropy":
            predictions = np.argmax(out, axis=1)
            target_idxs = np.argmax(target, axis=1)
            error = np.mean(predictions != target_idxs)

        # Error!
        else:
            raise NotImplementedError(
                "Error for {} loss is not implemented".format(self.loss)
            )

        return error
    def train(self, dataset: Dataset, epochs: int) -> None:
        """Train the neural network on using the provided dataset for ‘epochs‘
        epochs. One epoch comprises one full pass through the entire dataset, or
        in case of stochastic gradient descent, one epoch comprises seeing as
        many samples from the dataset as there are elements in the dataset.

        Parameters
        ----------
        dataset  training dataset
        epochs   number of epochs to train for
        """
        # Initialize output layer
        args = self.layer_args[-1]
        args["n_out"] = dataset.out_dim
        output_layer = initialize_layer(**args)
        self.layers.append(output_layer)

        for i in range(epochs):
            training_loss = []
            training_error = []
            for _ in tqdm(range(dataset.train.samples_per_epoch)):
                X, Y = dataset.train.sample()
                # Log the batch size
                print(f"Training batch size: {X.shape[0]}")
                Y_hat = self.forward(X)
                L = self.backward(np.array(Y), np.array(Y_hat))
                error = self.error(Y, Y_hat)
                self.update(i)
                training_loss.append(L)
                training_error.append(error)
            training_loss = np.mean(training_loss)
            training_error = np.mean(training_error)
            self._log(training_loss, training_error)

            validation_loss = []
            validation_error = []
            for _ in range(dataset.validate.samples_per_epoch):
                X, Y = dataset.validate.sample()
                Y_hat = self.forward(X)
                L = self.loss.forward(Y, Y_hat)
                error = self.error(Y, Y_hat)
                validation_loss.append(L)
                validation_error.append(error)
            validation_loss = np.mean(validation_loss)
            validation_error = np.mean(validation_error)
            self._log(validation_loss, validation_error, validation=True)

            print("Example target: {}".format(Y[0]))
```

```
                print("Example prediction: {}".format([round(x, 4) for x in Y_hat[0]]))
                print(
                    "Epoch {} Training Loss: {} Training Accuracy: {} Val Loss: {} Val Accuracy: {}".forma
                        i,
                        round(training_loss, 4),
                        round(1 - training_error, 4),
                        round(validation_loss, 4),
                        round(1 - validation_error, 4),
                    )
                )

        def test(
            self, dataset: Dataset, save_predictions: bool = False
        ) -> Dict[str, List[np.ndarray]]:
            """Makes predictions on the data in ‘datasets‘, returning the loss, and
            optionally returning the predictions and saving both.

            Parameters
            ----------
            dataset  test data
            save_predictions  whether to calculate and save the predictions

            Returns
            -------
            a dictionary containing the loss for each data point and optionally also
            the prediction for each data point
            """
            test_log = {"loss": [], "error": []}
            if save_predictions:
                test_log["prediction"] = []
            for _ in range(dataset.test.samples_per_epoch):
                X, Y = dataset.test.sample()
                Y_hat, L = self.predict(X, Y)
                error = self.error(Y, Y_hat)
                test_log["loss"].append(L)
                test_log["error"].append(error)
                if save_predictions:
                    test_log["prediction"] += [x for x in Y_hat]
            test_loss = np.mean(test_log["loss"])
            test_error = np.mean(test_log["error"])
            print(
                "Test Loss: {} Test Accuracy: {}".format(
                    round(test_loss, 4), round(1 - test_error, 4)
                )
            )
            if save_predictions:
                with open(self.logger.save_dir + "test_predictions.p", "wb") as f:
                    pickle.dump(test_log, f)
            return test_log
```

```
    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.
        print(f"Batch size: {X.shape[0]}")

        # Perform the forward pass
        output = self.forward(X)

        # Ensure that the output is a 2D array for classification
        if output.ndim == 1:
            output = np.expand_dims(output, axis=1)

        # Get the loss
        loss = self.loss.forward(Y, output)

        # Return the predictions (softmax applied) and the calculated loss
        return output, loss
```

6. **Q2.2: Plots for Combinations of Hyperparameters**