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:

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

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

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

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

```
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 max(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: TODO

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: TODO

1.4 Cross-Entropy Loss

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

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

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

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

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: TODO

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: TODO

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

   (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: TODO

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: TODO

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: TODO