CS 189 Introduction to Machine Learning
Spring 2023 Jonathan Shewchuk

HW6

Due: Wednesday, April 19 at 11:59 pm

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at
   - https://www.kaggle.com/competitions/cs189-hw6-cifar10-sp2023
2. The written portion:
   - Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment titled "Homework 6 Write-Up". Please see section 3.3 for an easy way to gather all your code for the submission (you are **not** required to use it, but we would strongly recommend using it).
   - In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. Whenever we say "include code", that means you can either include a screenshot of your code, or typeset your code in your submission (using markdown or LaTeX).
   - You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.**
   - If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
   - In your write-up, please state with whom you worked on the homework.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats.

     "I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."
3. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 6 Code". Yes, you must submit your code twice: in your PDF write-up following the directions as described above so the readers can easily read it, and once in compilable/interpretable form so the readers can easily run it. Do **NOT** include any data files we provided. Please include a short file named README listing your name, student ID, and instructions on how to reproduce your results. Please take care that your code doesn't take up inordinate amounts of time or memory. If your code cannot be executed, your solution cannot be verified.

1 Honor Code

Declare and sign the following statement:

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

Signature : _____

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

2 Background

This section will provide a background on neural networks that is designed to help you complete the assignment. There are no questions in this part. The questions for the homework begin in Section 4.

2.1 Neural Networks

Many of the most exciting recent breakthroughs in machine learning have come from "deep" (many-layered) neural networks, such as the deep reinforcement learning algorithm that learned to play Atari from pixels, or the ChatGPT model, which generates text that is nearly indistinguishable from human-generated text.

Neural network libraries such as Tensorflow and PyTorch have made training complicated neural network architectures very easy. However, we want to emphasize that neural networks begin with fundamentally simple models that are just a few steps removed from basic logistic regression. In this assignment, you will build two fundamental types of neural network models, all in plain `numpy`: a **feed-forward fully-connected network**, and a **convolutional neural network**. We will start with the essential elements and then build up in complexity.

A neural network model is defined by the following.

- An **architecture** defining the flow of information between computational layers. This defines the composition of functions that the network performs from input to output.
- A **cost function** (e.g. cross-entropy or mean squared error).
- An **optimization algorithm** (e.g. stochastic gradient descent with backpropagation).
- A set of **hyperparameters**. (Here we use this as a catch-all term that includes algorithm parameters that technically are not "hyperparameters" because they don't help you change the bias-variance tradeoff, such as the learning rate and the mini-batch size for stochastic gradient descent with mini-batches.)

Each *layer* is defined by the following components.

- A **parameterized function** that defines the layer's map from input to output (e.g. $f(x) = \sigma(Wx + b)$).
- An **activation function** σ (e.g. ReLU, sigmoid, etc.).
- A set of **parameters** (e.g. weights and bias terms).

Neural networks are commonly used for supervised learning problems, where we have a set of inputs and a set of labels, and we want to learn the function that maps inputs to labels. To learn this function, we need to update the parameters of the network (i.e. the weights, including the bias terms). We do this using **mini-batch gradient descent**. To compute the gradients for gradient descent, we use a dynamic programming algorithm called **backpropagation**.

In the backpropagation algorithm, we first compute what is called a "forward pass" of the network. In the forward pass, we send a mini-batch of input data (e.g. 50 training points) through the network. The output is a set of predicted labels, which we use as input to our loss function (along with the true labels from the training data). We then take the derivatives of the loss with respect to the parameters of each layer, starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the "backward pass." During the backward pass we compute the derivatives of the loss function with respect to each of the model parameters, starting from the last layer and "propagating" the information from the loss function backwards through the network. This lets us calculate derivatives with respect to all the parameters of our network while letting us avoid computing the same derivatives multiple times.

To summarize, training a neural network involves three steps.

1. Forward propagation of inputs.
2. Computing the cost.
3. Backpropagation and parameter updates.

2.2 Batching

When building neural networks, we have to carefully consider the data. In Homework 4, you coded both batch gradient descent and stochastic gradient descent for logistic regression. For the stochastic version, where only a single data point was used, the form of derivatives used in gradient descent were different than those of batch gradient descent. Neural networks usually operate on mini-batches, or subsets of the data matrix. This is because iterating on all the data at once (batch gradient descent) is inefficient for large data sets, whereas iterating on just one training point at a time introduces excessive stochasticity (randomness) and makes poor use of your computer's caches and potential for parallelism. Thus, every step of your neural network should be defined to operate on mini-batches of data. During a single operation of mini-batch gradient descent, you take a matrix of shape (B, d) where B is the mini-batch size and d is the number of features, and do a forward pass on B training points at once—ideally using vector operations to obtain some parallelism in your computations (as every training point is processed the same way). The input to a convolutional neural network for image recognition might be a four-dimensional array of shape (B, H, W, C) where B is the mini-batch size, H is the height of the image, W is the width of the image, and C is the number of channels in the image (3 for RGB—that is, red, green, and blue intensities).

As you are writing the gradient descent algorithm to work on mini-batches, all of your derivations must work for mini-batches. For that reason, many of the derivations you do for this homework will differ from those you have seen in class. Thinking in terms of mini-batches often changes the shapes and operations you do. **Your derivations must be for mini-batches and cannot use loops to iterate over individual data points.** Be prepared to spend some time working out the tricky details of how to do this.

2.3 Feed-Forward, Fully-Connected Neural Networks

A feed-forward, fully-connected neural network consists of layers of units alternating with layers of edges. Each layer of edges performs an affine transformation of an input, followed by a nonlinear activation func-

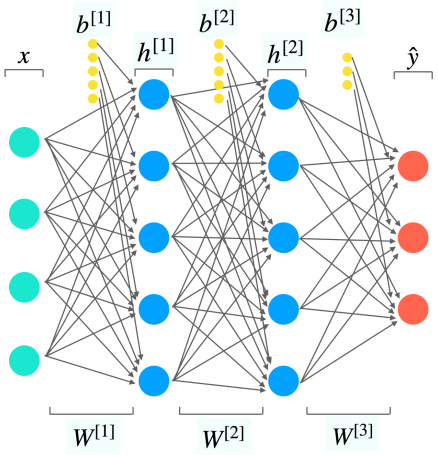


Figure 1: A 3-layer fully-connected neural network.

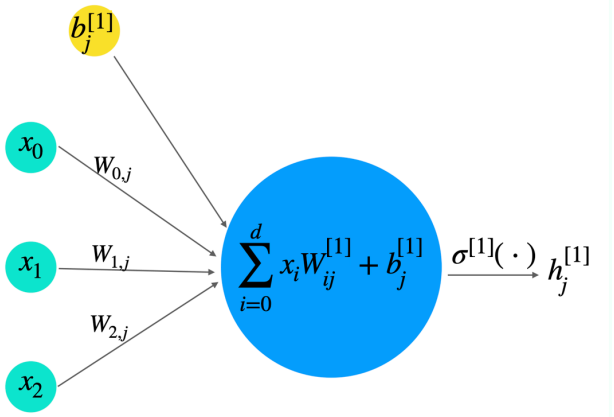


Figure 2: A single fully-connected neuron.

tion. "Fully-connected" means that a layer of edges connects every unit in one layer of units to every unit in the next layer of units. We use the following notation when defining fully-connected layers, with superscripts in brackets indexing layers (both layers of units and layers of edges) and subscripts indexing the vector/matrix elements. In this notation, we will use **row vectors** (not column vectors) to represent unit layers so that we can apply successive matrices (edge layers) to them from left to right.

- x: A single data vector, of shape $1 \times d$, where d is the number of features. You can think of it as "unit layer zero." We present a training point or a test point here.
- $\hat{y}$: A single output vector, of shape $1 \times k$, where k is the number of output units. These could be regression values or they could symbolize classifications (and you can mix output units of both types). Each training point x is accompanied by a label vector y, and the goal of training is to make x's output $\hat{y}$ be close to y.
- $n^{[l]}$: The number of units (neurons) in unit layer l.
- $W^{[l]}$: A matrix of weights connecting unit layer $l-1$ with unit layer l, of shape $n^{[l-1]} \times n^{[l]}$. This matrix represents the weights of the connections in edge layer l. At edge layer 1, the shape is $d \times n^{[1]}$.
- $b^{[l]}$: The bias vector for layer l, of shape $1 \times n^{[l]}$.
- $h^{[l]}$: The output of edge layer l. This is a vector of shape $1 \times n^{[l]}$.
- $\sigma^{[l]}(\cdot)$: The nonlinear *activation function* applied at layer l.

A fully-connected layer l is a function

$$\phi^{[l]}(h^{[l-1]}) = \sigma^{[l]}(h^{[l-1]}W^{[l]} + b^{[l]}) = h^{[l]}.$$

The output $h^{[l]}$ of edge layer l is computed then used as the input to edge layer $l+1$. (At edge layer 1, $h^{[0]}$ is simply the input vector x.) A neural network is thus a composition of functions. We want to find weights such that the network maps each training point x to its label y.

In a multiclass classification problem with more than two classes, it is common to set k equal to the number of classes and have each output unit represent a true/false value for one class. This is called *one-hot encoding*. A one-hot encoded output unit used for classification might employ the labels

$$y_i = \begin{cases} 1 & x \in \text{class } i, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for a classification problem with 3 classes, the label for a training point in class 3 might be $(0, 0, 1)$. However, the precise values you choose ought to depend on the activation function σ. Moreover, for reasons explained in lecture, you might get better results by using less extreme labels, such as 0.15 and 0.85 if σ is the logistic (sigmoid) function. If you have only two classes, there is usually no advantage to one-hot encoding; one output unit for the class label should suffice.

2.4 Convolutional Neural Networks

We will use the following notation when defining a convolutional neural network layer.

- X: A single image tensor (multi-dimensional array), of shape $d_1 \times d_2 \times c$, where d_1 and d_2 are the spatial dimensions, and c is the number of channels (of which there are typically three, encoding red, green, and blue pixel intensities).
- $\hat{y}$: A single output vector, of shape $1 \times k$.
- $n^{[l]}$: The number of image channels in layer l. Usually $n^{[l]} = c$.
- $(k_1, k_2)^{[l]}$: The size of the spatial dimensions of each filter/mask/kernel in layer l. Sometimes called the *kernel size*.[1]
- $W^{[l]}$: The tensor of filters convolved at edge layer l. This tensor has shape $k_1 \times k_2 \times n^{[l-1]} \times n^{[l]}$.
- $b^{[l]}$: The bias vector for layer l, of shape $1 \times n^{[l]}$.
- $H^{[l]}$: The output of layer l. This is a tensor of shape $r_1 \times r_2 \times n^{[l]}$, where (r_1, r_2) is the shape of output of the convolution operation. Below we will discuss how to calculate this.
- $\sigma^{[l]}(\cdot)$: The nonlinear *activation function* applied at layer l.

In a convolutional layer, each filter is convolved with the input image, across every image channel. This operation is, essentially, a sliding sum of element-wise products. Figure 3 gives a visual example. Let Z denote the intermediate result *just before we apply the activation function* σ to obtain H. To compute a single element in the intermediate output Z, for a single unit n, we compute

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n] = \sum_i \sum_j \sum_c W[i, j, c, n] X[d_1 + i, d_2 + j, c] + b[n].$$

Note: this formula is the **cross-correlation** formula from signal processing and NOT the convolution formula. Nevertheless this is what ML people call convolution, and so will we. It actually makes sense to use

[1] "Kernel" is an overloaded word. In the context of convolutional networks, the convolutional filter is also called the "convolutional kernel." This has no relationship with the "kernel" in the kernel trick and kernel methods. The convolutional kernel is also referred to as a "mask" in lecture.

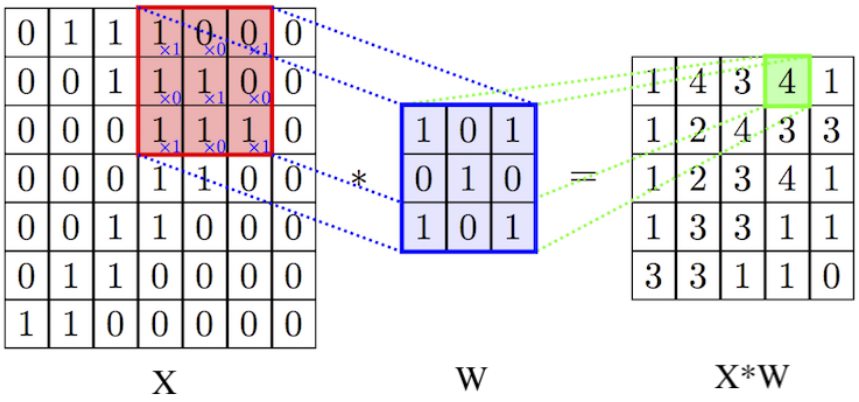


Figure 3: Figure showing an example of one convolution.

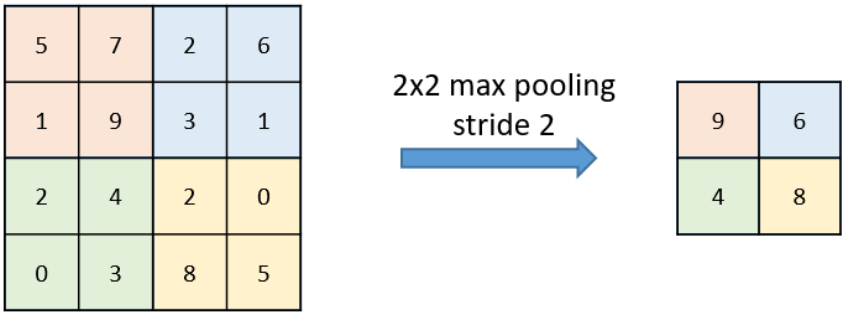


Figure 4: Figure showing an example of a max pooling layer with a kernel size of 2 and stride of 2.

cross-correlation instead of using convolution because the former can be interpreted as producing an output which is higher at locations where the image has the pattern in the filter and low elsewhere. Convolution is the same as cross-correlation with a flipped filter, and our filters are learned, so it makes no difference operationally whether you implement convolution or cross-correlation. However, to pass our tests, you must implement **cross-correlation** and call that convolution because that's how we do it in ML-land.

In this equation, we drop the layer superscripts for clarity, and index elements of the matrices in brackets. The pre-output Z is what we call a "feature map," which essentially captures the strength of each filter at every region in the image. In the equation above, we slide the filter over the image in increments of one pixel. We can choose to take a larger steps instead. The size of the step taken in the convolution operation is referred to as the *stride*.

The output of the convolutional layer is

$$H^{[l]} = \sigma^{[l]}(Z^{[l]}).$$

A pooling layer is used to downsample the input feature maps. It takes an input array of shape $d_1 \times d_2 \times n$ and outputs an array of shape $r_1 \times r_2 \times n$. Note that it does **not** change the number of channels, but typically reduces the number of spatial dimensions, i.e., $r_1 < d_1$ and $r_2 < d_2$. In order to do this, we have a **kernel** of shape $k_1 \times k_2$ and a stride s. For each channel, we take either the max or the average of all the points in the window of size $k_1 \times k_2$. Then we slide the window by s pixels and repeat until we have performed this operation over the entire input image. This is illustrated in Figure 4. When the operation performed over each sliding window is max, it is called **max pooling**, whereas when the operation is averaging, then it is

called **average pooling**. Using similar notation as above, the function computed by a max pooling layer is

$$Z[r_1, r_2, c] = \text{MaxPool}(X)[r_1, r_2, c] = \max\{X[r_1 s : r_1 s + k_1 - 1, r_2 s : r_2 s + k_2 - 1, c]\}.$$

Replacing the max function with the average function, we get an average pooling layer. This function is abstracted away as `pool_fn` in the code. Note that `pool_fn` takes an array as input and outputs a single float. The notation on the right hand side should be read as array slicing as in `numpy`.

Traditional CNNs operate on images by combining convolutional layers with pooling layers to progressively "shrink" the spatial size of the input until it is small enough to be fed to fully-connected network layers for classification.

3 The Neural Nets Package

3.1 Structure

We have provided a modularized codebase for constructing neural networks. The codebase has the following structure.

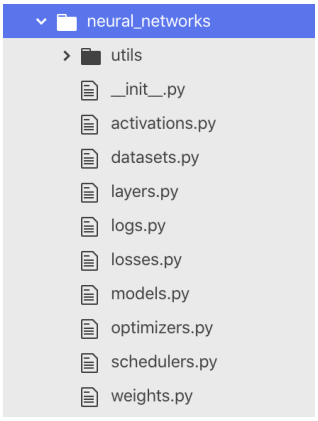


Figure 5: The structure of the starter codebase.

As you can see, the modules in the codebase reflect the structure outlined above. Different losses, activations, layers, optimizers, hyperparameters, and neural network architectures can be combined to yield different architectures.

In the codebase we have provided, each layer is an object with a few relevant attributes.

- `parameters`: An `OrderedDict` containing the weights and biases of the layer.
- `gradients`: An `OrderedDict` containing the derivatives of the loss with respect to the weights and biases of the layer, with the same keys as `parameters`.
- `cache`: An `OrderedDict` containing intermediate quantities calculated in the forward pass that are useful for the backward pass.
- `activation`: An `Activation` instance that is the activation function applied by this layer.
- `n_in`: The number of input units (input channels in CNN).

- `n_out`: The number of output units (output channels in CNN).

You will pass the layer a parameter that selects an activation function from those defined in `activations.py`. This will be stored as an attribute of the layer, which can be called as `layer.activation()`. The forward and backward passes of the layer are defined by the following methods.

- `forward` This method takes as input the output `X` from the previous layer (or input data). This method computes the function $\phi(\cdot)$ from above, combining the input with the weights `W` and bias `b` that are stored as attributes. It returns an output `out` and saves the intermediate value `Z` to the `cache` attribute, as it is needed to compute gradients in the backward pass.

```
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.
    """
    ...
```

- `backward` This method takes the gradient of the downstream loss (i.e., the derivative of the loss with respect to the output of the current layer) as input. Then it uses the `cache` from the forward pass to compute the gradient of its output with respect to its inputs and weights. Via chain rule, it then returns the gradient of the loss with respect to the input of the layer.

```
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)
    """
    ...
```

Each activation function has a similar (but simpler) structure:

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

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

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

3.2 Testing

We are using the `unittest` module built into Python for testing. Note that **the tests are not comprehensive**, so it is possible that your code passes all the tests but is still broken and gives you poor performance. We test the `forward` and `backward` methods of all the layers that you implement, along with a few that you don't have to implement (which should already be passing their tests). We also test the weight initialization of the `FullyConnected` layer so that you have some guidance when you implement a layer for the first time. When you are done with the homework, **you must make sure that your code passes all the tests** except the tests for the `L2` loss function and the `Sigmoid` activation function. You should have 21/25 tests passing

at least. (You can implement the `L2` loss and `Sigmoid` activation function if you want to make those tests pass.)

To run all the tests, make sure that you are in the root directory of the project (and **not** in either of the `neural_networks/` and `tests/` directories). Then run the following:

```
python -m unittest -v
```

25 tests will be run. Before you implement anything, 17 tests should fail and 8 should pass.

The tests are located in the `tests/` directory. **Please do not modify the pre-existing tests** because they rely on data generated by a correct implementation, and any modifications can cause the tests to fail. The data is located in the `tests/data` directory. However, if you wish to write your own tests in addition to the ones that are already provided, then please refer to this tutorial and the official documentation. Note that we don't require you to write your own tests as part of this homework.

The tests are organized into separate classes that mirror the structure of the classes that they test. For example, the `forward` method of the `Conv2D` class located in `neural_networks/layers.py` is tested by the `test_forward` method of the `TestConv2D` class located in `tests/test_layers.py`. If you want to run that specific test, you can use the following command **root directory of the project**:

```
python -m unittest -v tests.test_layers.TestConv2D.test_forward
```

If you instead want to run all the tests defined in the `TestConv2D` class, you should run:

```
python -m unittest -v tests.test_layers.TestConv2D
```

And if you want to run all the tests in `test_layers.py`, then you should run:

```
python -m unittest -v tests.test_layers
```

We have also included a Jupyter Notebook named `check_gradients.ipynb`, which only contains gradient checks for your layers, which you can use for debugging your layers' gradient computations. This can provide more debuggability than simply running the provided unit tests. Note that we do **not** require you to provide us the output of those tests, since gradients are implicitly tested in the `test_backward` methods of the testing classes.

3.3 Submission

Please run

```
python3 generate_submission.py --help
```

for instructions on how to use the script to extract your implementations for the submission.

The script `generate_submission.py` extracts all the code from your implementations and produces either a LaTeX or markdown file containing your code implementations, when given the flag `--format latex` and `--format markdown` respectively. The generated document contains your functions in separate sections for activation functions, layers, losses, and the model. These can be sections, subsections, or subsubsections depending on whether you supply the flag `--heading_level 1`, `--heading_level 2`, or `--heading_level 3`.

For example, if you want LaTeX output with each part (activations, layers, losses, and the model) in a different subsection, with the output saved to `submission.tex`, you would run the following:

```
python3 generate_submission.py --format latex --heading_level 2 --output submission.tex
```

whereas if you want markdown output with each part (activations, layers, losses, and the model) in a different subsubsection, with the output saved to `submission.md`, you would run the following:

```
python3 generate_submission.py --format markdown --heading_level 3 --output submission.md
```

We would suggest running these commands to see exactly what they do.

The markdown document will compile by itself, but you would most likely want to create a markdown cell in a Jupyter Notebook and copy-paste the generated markdown into that cell. That should work seamlessly provided you can already compile Jupyter Notebooks into PDFs.

Note that the LaTeX document will **not** compile by itself. It is meant to generate code that you can then `\input{}` into your LaTeX document.

Feel free to play around with the script if you want to. Notably, if you change some function which is not in the `student_implementations` list, then you could add that function to the `student_implementations` list to have the script automatically gather your code from that function (but we think that most students will **not** have to do this).

Questions

The background section of the homework has ended. The following sections contain the questions you must complete.

4 Basic Network Layers

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, 2) write the matching code, 3) pass the tests.

Keep in mind that your solutions to all layers **must operate on mini-batches of data** and should not use loops to iterate over the training points in a mini-batch.

4.1 ReLU

First, you will implement the ReLU activation function in `activations.py`.

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

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

Instructions

1. First, derive the gradient of the downstream loss with respect to the input of the ReLU activation function, Z. **You must arrive at a solution in terms of $\frac{dL}{dy}$, the gradient of the loss w.r.t. the output of ReLU, and a batched input Z, i.e., where $Z \in \mathbb{R}^{N \times M}$.**
2. Next, implement the forward and backward passes of the ReLU activation in the script `activations.py`. Include your code in your writeup (either a screenshot or typesetting is fine).**Do not iterate over training examples, use batched operations.**
3. Include the output of running `python -m unittest -v tests.test_activations.TestReLU`. Make sure that the tests are passing.

Solution: 1. Assume that $y = \max\{x, 0\}$ and $x \in N \times M$. Also assume $\frac{\partial \text{loss}}{\partial y} = \text{dout} \in N \times M$. Then

$$\frac{\partial \text{loss}}{\partial x} = \text{dout} \odot \mathbb{1}_{x>0}$$

Here $\odot$ denotes elementwise multiplication and $\mathbb{1}_{x>0}$ denotes a 0/1 matrix with the same size as x where 1 is filled when the corresponding entry in x is bigger than 0 and 0 is filled otherwise.

2.

```
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
```

```
        """
        return np.maximum(0, Z)

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for relu activation.
        f'(z) = 1 if z >= 0
                0 otherwise
        """
        backward = np.copy(Z)
        backward[Z < 0] = 0
        backward[Z >= 0] = 1
        return dY * backward
```

3.

```
test_backward (tests.test_activations.TestReLU) ... ok
test_forward (tests.test_activations.TestReLU) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.082s

OK
```

4.2 Fully-Connected Layer

Now you will implement the forward and backward passes for the fully-connected layer in the `layers.py` script. The code is marked with `YOUR CODE HERE` statements indicating what to implement and where. Please read the docstrings and the function signatures too. 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. We have provided a Jupyter notebook `check_gradients.ipynb` for you to use to numerically check the gradients of your layer implementations. Simply run the cell corresponding to each layer. The printed errors should be very small, usually on the order of 10^{-8} or smaller.

Instructions

1. First, derive the gradients of the downstream loss L with respect to W and b in the fully-connected layer, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. You will also need to take the derivative of the loss with respect to the input of the layer $\frac{\partial L}{\partial X}$, which will be passed to lower layers. **Again, you must assume must arrive at a solution for batched X and Z. Please express your solution in terms of $\frac{dL}{dZ}$, which you have already obtained in 4.1, and $Z = XW + b$**

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 design matrix as input and initializes the parameters, cache, and gradients of the layer. 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. In your writeup, include the code you have implemented. **Do not loop over training points; use batched operations.**

3. Include the output of `python -m unittest -v tests.test_layers.TestFullyConnected`. Make sure that the tests are passing.

Solution: 1. Assume $z = xW + b$ and $y = \sigma(z)$, where $x \in 1 \times n^{[l]}$, $W \in n^{[l]} \times n^{[l+1]}$, $b \in 1 \times n^{[l+1]}$ and $y \in 1 \times k$. Also assume the gradient from the upper stream is $\frac{\partial L}{\partial y} \in 1 \times k$ and the gradient of the activation function $\sigma(\cdot)$ is $\frac{\partial y}{\partial z}$.

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

Note that $\frac{\partial L}{\partial z} \in 1 \times n^{[l+1]}$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial L}{\partial z}W^\top$$

Note that the order here switches.

$$\frac{\partial L}{\partial W} = \frac{\partial z}{\partial W}\frac{\partial L}{\partial z} = x^\top\frac{\partial L}{\partial z}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$$

We can the generalize this to batches: $Z = XW + b$.

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

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

$$\frac{\partial L}{\partial b} = 1^\top \frac{\partial L}{\partial Z}$$

2.

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

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

    self.parameters = OrderedDict({"W": W, "b": b})
    self.cache = OrderedDict({"Z": [], "X": []})
    self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)})
```

```
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.
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

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

    self.cache["Z"] = Z
    self.cache["X"] = X

    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)
    """
    W = self.parameters["W"]
    b = self.parameters["b"]
    Z = self.cache["Z"]
    X = self.cache["X"]

    dZ = self.activation.backward(Z, dLdY)
    dX = dZ @ W.T
    dW = X.T @ dZ
    dB = dZ.sum(axis=0, keepdims=True)

    self.gradients["W"] = dW
    self.gradients["b"] = dB

    return dX
```

3.

```
test_backward (tests.test_layers.TestFullyConnected) ... ok
```

```
test_forward (tests.test_layers.TestFullyConnected) ... ok
test_init_params (tests.test_layers.TestFullyConnected) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.463s

OK
```

4.3 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 probability distribution. That is, the softmax function squashes continuous values into the range $[0, 1]$ and normalizes the outputs so that they add up to 1. For this reason, many classification neural networks use the softmax activation. The softmax activation takes in a vector s of k un-normalized values $s_1, \ldots, s_k$ and outputs a probability distribution over the k possible classes. The forward pass of the softmax activation on input s_i is

$$\sigma_i = \frac{e^{s_i}}{\sum_{j=1}^{k} e^{s_j}},$$

where k ranges over all elements in s. Due to issues of numerical stability, the following modified version of this function is commonly used.

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

where $m = \max_{j=1}^{k} s_j$. We strongly recommend implementing the latter formula.

Instructions

1. Derive the Jacobian of the softmax activation function. **You do not need to use batched inputs for this question; an answer for a single training point is acceptable. You do not need to write out the entire matrix, but please write out what $\frac{\partial \sigma_i}{\partial s_j}$ is for an arbitrary (i, j) pair.**
2. Implement the forward and backward passes of the softmax activation in `activations.py`. We recommend vectorizing the backward pass for efficiency. **For this quesiton only may you use a "for" loop over the training points in the mini-batch.**
3. Include the output of `python -m unittest -v tests.test_activations.TestSoftMax`. Make sure that the tests are passing.

Solution:

1. The Jacobian matrix for the softmax function is

$$\frac{\partial \sigma}{\partial x} = \begin{bmatrix} \frac{\partial \sigma_0}{\partial x_0} & \frac{\partial \sigma_0}{\partial x_1} & \cdots & \frac{\partial \sigma_0}{\partial x_k} \\ \frac{\partial \sigma_1}{\partial x_0} & \frac{\partial \sigma_1}{\partial x_1} & \cdots & \frac{\partial \sigma_1}{\partial x_k} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial \sigma_k}{\partial x_0} & \frac{\partial \sigma_k}{\partial x_1} & \cdots & \frac{\partial \sigma_k}{\partial x_k} \end{bmatrix}.$$

For an arbitrary i and j,

$$\frac{\partial \sigma_i}{\partial x_j} = \frac{\partial}{\partial x_j} \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}.$$

By the quotient rule for derivatives, for $f(x) = \frac{g(x)}{h(x)}$, the derivative of $f(x)$ is

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}.$$

In our case, $g_i = e^{x_i}$ and $h_i = \sum_{k=0} K e^{x_k}$. For any x_j, the derivative of h_i with respect to x_j is always e^{x_j}:

$$\frac{\partial}{\partial x_j} h_i = \frac{\partial}{\partial x_j} \sum_{k=0}^{K} e^{x_k} = \sum_{k=0}^{K} \frac{\partial}{\partial x_j} e^{x_k} = e^{x_j},$$

because $\frac{\partial}{\partial x_j} e^{x_k} = 0$ for $k \neq j$. However, this is not the case for g_i. The term x_j contributes to g_i only when $i = j$. So the derivative of g_i with respect to x_j is e^{x_j} only when $i = j$. Otherwise, it's a constant and its derivative is 0.

Therefore, we know that the gradient along the diagonal of the Jacobian matrix (where $i = j$) is

$$\frac{\partial \sigma_i}{\partial x_j} = \frac{e^{x_i} \sum_{k=0}^{K} e^{x_k} - e^{x_j} e^{x_i}}{\left[\sum_{k=0}^{K} e^{x_k}\right]^2} \tag{1}$$

$$= \frac{e^{x_i} \sum_{k=0}^{K} e^{x_k} - e^{x_j}}{\left[\sum_{k=0}^{K} e^{x_k}\right]^2} \tag{2}$$

$$= \sigma_i(1 - \sigma_j). \tag{3}$$

The off-diagonal entries of the Jacobian are

$$\frac{\partial \sigma_i}{\partial x_j} = \frac{0 \sum_{k=0}^{K} e^{x_k} - e^{x_j} e^{x_i}}{\left[\sum_{k=0}^{K} e^{x_k}\right]^2} \tag{4}$$

$$= -\frac{e^{x_j}}{\sum_{k=0}^{K} e^{x_k}} \frac{e^{x_i}}{\sum_{k=0}^{K} e^{x_k}} \tag{5}$$

$$= -\sigma_j \sigma_i. \tag{6}$$

To summarize,

$$\frac{\partial \sigma_i}{\partial x_j} = \begin{cases} \sigma_i(1 - \sigma_j) & i = j, \\ -\sigma_j \sigma_i & i \neq j. \end{cases}$$

2.

```
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`
        """
        shifted = Z - np.max(Z, axis=-1, keepdims=True)
        exp = np.exp(shifted)
        out = np.divide(exp, np.sum(exp, axis=-1, keepdims=True))
        return out
```

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

        Parameters
        ----------
        Z   input to `forward` method
        dY  derivative of loss w.r.t. the output of this layer
            same shape as `Z`
        Returns
        -------
        derivative of loss w.r.t. input of this layer
        """
        p = self.forward(Z)
        backward = []
        for i, example in enumerate(p):
            diag = np.diagflat(example)
            example = example.reshape((-1, 1))
            J = diag - np.dot(example, example.T)
            backward.append(dY[i] @ J)
        return np.array(backward)
```

3.

```
test_backward (tests.test_activations.TestSoftMax) ... ok
test_forward (tests.test_activations.TestSoftMax) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.246s

OK
```

4.4 Cross-Entropy Loss

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

$$L = -y \ln \hat{y},$$

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. The cross-entropy cost calculated for a mini-batch of m samples is

$$J = -\frac{1}{m}\Big(\sum_{i=1}^{m} Y_i \ln(\hat{Y}_i)\Big).$$

Instructions

1. Derive the gradient of the cross-entropy cost with respect to the network's predictions, $\hat{Y}$. **You must use batched inputs.**
2. Implement the forward and backward passes of the cross-entropy cost. 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 over batches. **Do not iterate over training examples; use batched operations.**
3. Include the output of `python -m unittest -v tests.test_losses.TestCrossEntropy`. Make sure that the tests are passing.

Solution:

1. $\frac{\partial J}{\partial \hat{Y}} = -\frac{1}{m}\frac{Y}{\hat{Y}}$.
2.

```
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
        """
        return -np.sum(Y * np.log(Y_hat + np.finfo(float).eps)) / Y.shape[0]

    def backward(self, Y: np.ndarray, Y_hat: np.ndarray) -> np.ndarray:
        """Backward pass of cross-entropy loss.
        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 derivative of the cross-entropy loss with respect to the vector of
        predictions, `Y_hat`
        """
        return - Y / (Y_hat * Y.shape[0])
```

3.

```
test_backward (tests.test_losses.TestCrossEntropy) ... ok
test_forward (tests.test_losses.TestCrossEntropy) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.048s

OK
```

5 Two-Layer Fully Connected 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`, and `predict` methods for the `NeuralNetwork` class in `models.py`. 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 while varying the following hyperparameters.

   - Learning rate
   - Hidden layer size (number of units per hidden layer)

   You must try at least 3 different *combinations* of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations for your best model and report your final test error.

Solution:

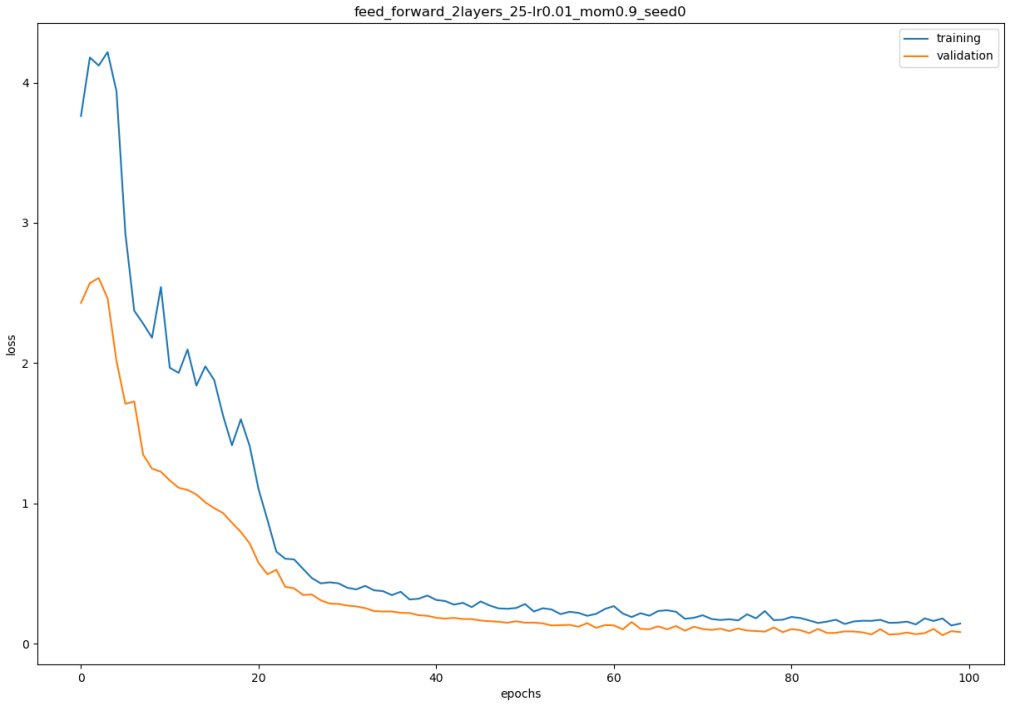
1.

```
    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
        """
        Y = X
```

```
        for layer in self.layers:
            Y = layer.forward(Y)
        return Y

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

2.



6 CNN Layers

In this problem, you will write the forward and backward passes of a convolutional neural network layer. Convolutional neural networks take considerably longer to train than the feedforward layers we have built, and the `numpy` implementation you will write here will be impractically slow. So, you will not be asked to train your network. Instead, we will simply run tests on the forward and backward passes of your network.

Again, remember that all your implementations and derived gradients must be for mini-batches, and not single examples. For CNN's, the input tensor will be of shape (m, H, W, C).

6.1 The Einsum Function

The np.einsum function is a powerful tool for performing various linear algebra operations on arrays in NumPy, a popular numerical computing library in Python. It allows for efficient computation of multi-dimensional array operations through a concise and flexible notation.

The np.einsum function takes in one or more input arrays and a string specifying the desired output format. The string consists of subscripts that define how the arrays should be multiplied, summed, and contracted. The output of the function is a new array that represents the result of the specified operation.

Here are some basic rules that help you understand how to use np.einsum, as this may help you writing the other parts in the CNN section.

- Einsum notation uses subscripts to represent the dimensions of the input arrays. Each subscript is a single uppercase or lowercase letter or a range of letters separated by colons. For example, "i,j" represents two dimensions, and "i:j,k" represents a range of dimensions from i to j (inclusive) and one additional dimension k.
- The einsum string consists of two or more subscript strings separated by a comma, followed by an arrow (->), and a final subscript string that specifies the dimensions of the output array. For example, "i,j->ij" specifies the multiplication of two arrays with dimensions i and j, resulting in an output array with dimensions ij.
- The same subscript letter can appear in multiple input subscript strings, indicating that the corresponding dimensions should be multiplied. For example, "ij,j->i" performs a sum over the second dimension of the second input array and the third dimension of the third input array, resulting in an output array with dimensions i.
- If a subscript appears only in the output subscript string, the corresponding dimension is preserved in the output array. For example, "ij->j" returns a one-dimensional array with dimensions j.
- Ellipses (...) can be used to represent a range of dimensions in an input or output subscript string. For example, "i...->i" performs a summation over all dimensions except the first of the input array, resulting in a one-dimensional output array with dimensions i.

For more details please refer to the NumPy API.

Please accomplish the following tasks in NumPy using np.einsum. Confirm you got the same answer by subtracting the two results and show that its norm is zero.

1. For any matrix $A \in \mathbb{R}^{5\times5}$, please derive the trace of it using np.einsum.

2. Please do matrix multiplication of $A \in \mathbb{R}^{5\times5}$ and $b \in \mathbb{R}^5$ in np.einsum for any A, b.

3. Please do vector outer product of $a \in \mathbb{R}^5$ and $b \in \mathbb{R}^5$ in np.einsum for any a, b.

Solution: 1.

```
A = np.random.rand(5,5)
trace = np.einsum('ii', A)
#or alternatively
trace = np.einsum(A, [0,0])
print(np.linalg.norm(np.trace(A)-trace))
```

2.

```
A = np.random.rand(5,5)
b = np.random.rand(5)
product = np.einsum('ij,j', A, b)
#or alternatively
product = np.einsum(A, [0,1], b, [1])
print(np.linalg.norm(A@b-product))
```

3.

```
a = np.random.rand(5)
b = np.random.rand(5)
product = np.einsum('i,j', a, b)
#or alternatively
product = np.einsum(a, [0], b, [1])
print(np.linalg.norm(a@b.T-product))
```

6.2 Convolutional Layer

1. Derive the gradient of the loss with respect to the input and parameters (weights 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 derivative of the pre-activation w.r.t the loss** 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?

2. Fill in the forward and backward passes of the `Conv2D` layer in `layers.py`. Include your code in your submission. **Do not iterate over training examples; use batched operations.**

3. Run `python -m unittest -v tests.test_layers.TestConv2D`. Make sure the tests are passing and include the output in your submission.

Solution: 1. Assume $\frac{\partial L}{\partial y}$ is known where y is the output of the convolution. We want to compute $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$.

Using the same notation as the background section, we have that:

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n] = \sum_i \sum_j \sum_c W[i, j, c, n] X[d_1 + i, d_2 + j, c] + b[n].$$

Let's start with the easiest case: the bias.

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

Next, the weights:

By the chain rule, we have

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

By looking at the cross correlation equation, we can easily see the following:

$$\frac{\partial Z[d_1, d_2, f]}{\partial W[x, y, c, f]} = X[d_1 + j, d_2 + j, c]$$

Note that all terms that are not for the same output channel are zero! We can thus conclude the following:

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

Now, we can do the input.

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

Consider just $\frac{\partial Z[d_1,d_2,n]}{\partial X[x,y,c]}$. Notice that this is zero if the $Z[d_1, d_2, n]$ and $X[x, y, c]$ do not overlap via the convolutional filter. If they do overlap, then:

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

Thus, we reorder the original sum to be localized around the positions x, y of the output where the derivatives are non-zero. To see this, set $d_1 + i = x$, and you get that $i = x - d_1$.

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

2.

```python
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass for convolutional layer. This layer convolves the input
    `X` with a filter of weights, adds a bias term, and applies an activation
    function to compute the output. This layer also supports padding and
    integer strides. Intermediates necessary for the backward pass are stored
    in the cache.
    Parameters
    ----------
    X  input with shape (batch_size, in_rows, in_cols, in_channels)
    Returns
    -------
    output feature maps with shape (batch_size, out_rows, out_cols, out_channels)
    """
    if self.n_in is None:
        self._init_parameters(X.shape)

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

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

    out_rows = int((in_rows + 2*self.pad[0] - kernel_height) / self.stride + 1)
    out_cols = int((in_cols + 2*self.pad[1] - kernel_width) / self.stride + 1)

    X_pad = np.pad(X, pad_width=((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0)), mode=
    ↪ 'constant')

    Z = np.empty((n_examples, out_rows, out_cols, out_channels), dtype=X.dtype)
    for r in range(out_rows):
        for c in range(out_cols):
            # Note: you can also get rid of this for loop by using one more broadcasting operation.
            for oc in range(out_channels):
                Z[:, r, c, oc] = (
                    np.sum(
                        X_pad[
                            :,
                            r * self.stride : r * self.stride + kernel_height,
                            c * self.stride : c * self.stride + kernel_width,
                            :,
                        ]
                        * W[:, :, :, oc],
                        axis=(1, 2, 3),
                    )
                    + b[:, oc]
                )

    out = self.activation(Z)

    self.cache["Z"] = Z
```

```
    self.cache["X"] = X

    return out
```

```
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for conv layer. Computes the gradients of the output
    with respect to the input feature maps as well as the filter weights and
    biases.
    Parameters
    ----------
    dLdY  derivative of loss with respect to output of this layer
          shape (batch_size, out_rows, out_cols, out_channels)
    Returns
    -------
    derivative of the loss with respect to the input of this layer
    shape (batch_size, in_rows, in_cols, in_channels)
    """
    W = self.parameters["W"]
    b = self.parameters["b"]
    Z = self.cache["Z"]
    X = self.cache["X"]

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

    out_rows = int((in_rows + 2*self.pad[0] - kernel_height) / self.stride + 1)
    out_cols = int((in_cols + 2*self.pad[1] - kernel_width) / self.stride + 1)

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

    X_pad = np.pad(X, pad_width=((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0)), mode=
    ↪ 'constant')
    dX_pad = np.zeros_like(X_pad)

    dW = np.zeros_like(W)
    dB = dZ.sum(axis=(0, 1, 2)).reshape(1, -1)

    for r in range(out_rows):
        for c in range(out_cols):
            # Note: you can also get rid of this for loop by using one more broadcasting operation.
            for oc in range(out_channels):
                dX_pad[:,
                       r*self.stride:r*self.stride + kernel_height,
                       c*self.stride:c*self.stride + kernel_width,
                       :] += W[np.newaxis, :, :, :, oc] * dZ[:, r:r+1, c:c+1, np.newaxis, oc]

                dW[:, :, :, oc] += np.sum(
                    X_pad[:, r*self.stride:r*self.stride + kernel_height, c*self.stride:c*self.stride +
    ↪ kernel_width, :] *
                    dZ[:, r:r+1, c:c+1, np.newaxis, oc],
                    axis=0
                )

    self.gradients["W"] = dW
    self.gradients["b"] = dB

    dX = dX_pad[:, self.pad[0]:in_rows+self.pad[0], self.pad[1]:in_cols+self.pad[1], :]
    return dX
```

3.

```
test_backward (tests.test_layers.TestConv2D) ... ok
test_forward (tests.test_layers.TestConv2D) ... ok

----------------------------------------------------------------------
Ran 2 tests in 8.301s
```

```
OK
```

6.3 Pooling Layers

1. Explain how we can use the backprop algorithm to compute derivatives through the max pooling and average pooling operations. (A plain English answer will suffice; equations are optional.) Carefully consider the values you must keep track of!

2. Fill in the forward and backward passes of the `Pool2D` layer in `layers.py`. Include your code in your submission. **Do not iterate over training examples, use batched operations.**

3. Run `python -m unittest -v tests.test_layers.TestPool2D`. Make sure the tests are passing and include the output in your submission.

Solution: 1. Similar to how we computed the derivatives through a convolution layer, we'll be given the derivative with respect to the output of the maxpool layer.

The gradient from the next layer is passed back only to the neuron which achieved the max. All other neurons get zero gradient.

Because maxpooling doesn't have any trainable parameters, we won't need to worry about calculating any derivatives for weights.

Once we have the derivative with respect to the input, the backprop algorithm can continue on to the layer before the maxpool by using this derivative.

Average pooling is analogous except with the average operation instead of the max.

2.

```
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: use the pooling function to aggregate local information
    in the input. This layer typically reduces the spatial dimensionality of
    the input while keeping the number of feature maps the same.
    As with all other layers, please make sure to cache the appropriate
    information for the backward pass.
    Parameters
    ----------
    X  input array of shape (batch_size, in_rows, in_cols, channels)
    Returns
    -------
    pooled array of shape (batch_size, out_rows, out_cols, channels)
    """
    n_examples, in_rows, in_cols, in_channels = X.shape
    kernel_height, kernel_width = self.kernel_shape

    out_rows = int((in_rows + 2*self.pad[0] - kernel_height) / self.stride + 1)
    out_cols = int((in_cols + 2*self.pad[1] - kernel_width) / self.stride + 1)

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

    X_pad = np.pad(X, pad_width=((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0)), mode=
    ↪ 'constant')
    X_pool = np.zeros((n_examples, out_rows, out_cols, in_channels))
    for i in range(out_rows):
        for j in range(out_cols):
            i0, i1 = i * self.stride, (i * self.stride) + kernel_height
            j0, j1 = j * self.stride, (j * self.stride) + kernel_width
            X_pool[:, i, j, :] = pool_fn(X_pad[:, i0:i1, j0:j1, :], axis=(1, 2))

    self.cache["X"] = X
```

```
    return X_pool
```

```
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for pooling layer.
    Parameters
    ----------
    dLdY  gradient of loss with respect to the output of this layer
          shape (batch_size, out_rows, out_cols, channels)
    Returns
    -------
    gradient of loss with respect to the input of this layer
    shape (batch_size, in_rows, in_cols, channels)
    """
        s = self.stride

        out_rows = self.cache["out_rows"]
        out_cols = self.cache["out_cols"]
        X = self.cache["X"]

        n_examples, in_rows, in_cols, in_channels = X.shape
        kernel_height, kernel_width = self.kernel_shape

        out_rows = int((in_rows + 2*self.pad[0] - kernel_height) / self.stride + 1)
        out_cols = int((in_cols + 2*self.pad[1] - kernel_width) / self.stride + 1)

        X_pad = np.pad(X, pad_width=((0, 0), (self.pad[0], self.pad[0]), (self.pad[1], self.pad[1]), (0, 0)),
    ↪ mode='constant')

        dX = np.zeros_like(X_pad)

        for i in range(out_rows):
            for j in range(out_cols):
                # calculate window boundaries, incorporating stride
                i0, i1 = i * s, (i * s) + kernel_height
                j0, j1 = j * s, (j * s) + kernel_width

                if self.mode == "max":
                    xi = X_pad[:, i0:i1, j0:j1, :]
                    xi_flat = xi.reshape(n_examples,
                                         kernel_height * kernel_width,
                                         in_channels)
                    idxs = np.argmax(xi_flat, axis=1)
                    mask = np.zeros_like(xi_flat)
                    n_idx, c_idx = np.indices((n_examples, in_channels))
                    mask[n_idx, idxs, c_idx] = 1
                    mask = mask.reshape(n_examples,
                                        kernel_height,
                                        kernel_width,
                                        in_channels)
                    dX[:, i0:i1, j0:j1, :] += mask * dLdY[:, i:i+1, j:j+1, :]

                elif self.mode == "average":
                    dX[:, i0:i1, j0:j1, :] += dLdY[:, i:i+1, j:j+1, :] / (kernel_height * kernel_width)
        return dX[:, self.pad[0]: in_rows + self.pad[0], self.pad[1]: in_cols + self.pad[1], :]
```

3.

```
test_backward (tests.test_layers.TestPool2D) ... ok
test_forward (tests.test_layers.TestPool2D) ... ok

----------------------------------------------------------------------
Ran 2 tests in 9.671s

OK
```

7 Kaggle

Please see the Google Colab Notebook here that contains the questions.

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

The following sections mirror the Colab Notebook and provide the deliverables for each question.

7.1 MLP for Fashion MNIST

Deliverables:

1. Provide code for training an MLP on the fashion MNIST dataset (can be tagged from colab notebook and in the code appendix)
2. A plot of the training and validation loss for each epoch of training for at least 8 epochs.
3. A plot of the training and validation accuracy for each epoch, achieving a final validation accuracy of at least 82%.

7.2 CNNs for CIFAR-10

Deliverables:

1. Provide the code for training your CNN model (can be in appendix).
2. Submit to Kaggle and include your test accuracy in your report. Our simple reference solution gets a test accuracy of 74.8% after 10 epochs.
3. Provide at least 1 training curve for your model, depicting loss per epoch or step after training for at least 8 epochs.
4. Explain the components of your final model, and how you think your design choices contributed to its performance.

Submission Checklist

Please ensure you have completed the following before your final submission.

At the beginning of your writeup...

1. Have you copied and hand-signed the honor code specified in Question 1?
2. Have you listed all students (Names and ID numbers) that you collaborated with?

In your writeup for Question 7...

1. Have you included your **Kaggle Score** and **Kaggle Username**?
2. Have you included your generated plots and visualizations?

At the end of the writeup...

1. Have you provided a code appendix including all code you wrote in solving the homework?

Executable Code Submission

1. Have you created an archive containing all ".py" files that you wrote or modified to generate your homework solutions?
2. Have you removed all data and extraneous files from the archive?
3. Have you included a `README.md` in your archive containing any special instructions to reproduce your results?

Submissions

1. Have you submitted your written solutions to the Gradescope assignment titled **HW6 Write-Up** and selected pages appropriately?
2. Have you submitted your executable code archive to the Gradescope assignment titled **HW6 Code**?
3. Have you submitted your test set predictions for **CIFAR-10** dataset to the appropriate Kaggle challenge?
4. Is your Kaggle submission in integer format? Submissions in decimal format will recieve a score of zero!

Congratulations! You have completed Homework 6.