CS 189 / 289 Introduction to Machine Learning
Fall 2024 Jennifer Listgarten, Saeed Saremi **HW3**

Due 10/09/24 11:59 pm PT

- Homework 3 is primarily coding with some math questions mixed in.
- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted for the written questions.
- In all of the questions, **show your work**, not just the final answer.

Deliverables:

1. Submit a PDF of your homework to the Gradescope assignment entitled "HW 3 Write-Up". **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. This should be on its own page and should be the first page that you submit.
   - In your write-up, please copy the following statement and sign your signature underneath. If you are using LaTeX, you can type your full name underneath instead. 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."*

   - **Replicate all of your code in an appendix**. Begin code for each coding question on a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from the appendix to correct questions.
2. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 3 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 as highlighted in the Submission section at the end, so it can be correctly parsed by the autograder.

1 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.

1.1 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. You don't even really need to understand how they work! With just a few lines of code, you can take a pre-defined neural network architecture and train it on your dataset. These libraries are wonderful for experienced practitioners who understand neural networks inside and out and want to work with a lot of complex machinery at a high level. They're also wonderful for those who don't care to dive deep into the inner workings of neural networks and want to just use pre-defined functions. But for those who want to dive deep and are just learning the material, they tend to obscure the fundamental simplicity and elegance of the inner workings of neural networks. It is easy to get lost in the complexity of the very many classes and parameters defined in these libraries.

In this assignment, we want to emphasize that neural networks begin with a fundamentally simple model that is 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 to also include algorithm parameters that technically are not "hyperparameters" in the traditional sense 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 biases).

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 gradients 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 gradients 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 gradients with respect to all the parameters of our network while letting us avoid computing the same gradients 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.

1.2 Batching

When building neural networks, we have to carefully consider the data. 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 perform 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. Thinking in terms of mini-batches often changes the shapes and operations you perform. **Your derivations must be batched and cannot use loops to iterate over individual data points.** Be prepared to spend some time working out the tricky details behind this.

1.3 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 function. "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.
- y: A single label 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 0, 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 $1 \times n^{[i]}$.
- $\sigma^{[l]}(\cdot)$: The nonlinear "activation function" applied at layer l.

A fully-connected layer l is a function

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

We will use the term $z^{[l]} = h^{[l-1]}W^{[l]} + b^{[l]}$ as shorthand for the intermediate result within layer l before applying the activation function σ. The output $h^{[l]}$ of edge layer l is computed, and then subsequently used as the input to edge layer $l + 1$ ($h^{[-1]}$ is simply the data vector x). A neural network is thus a *composition of functions*. We want to find the parameters such that the network maps each training point x to its label y.

In a multi-class 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

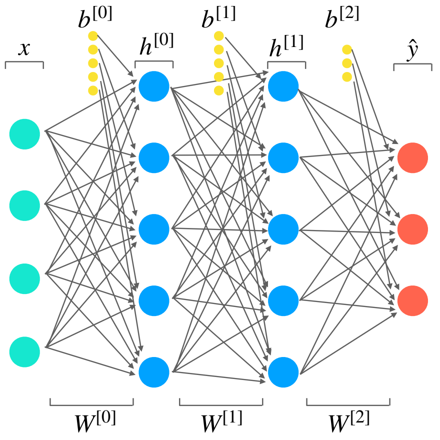


Figure 1: A 3-layer fully-connected neural network.

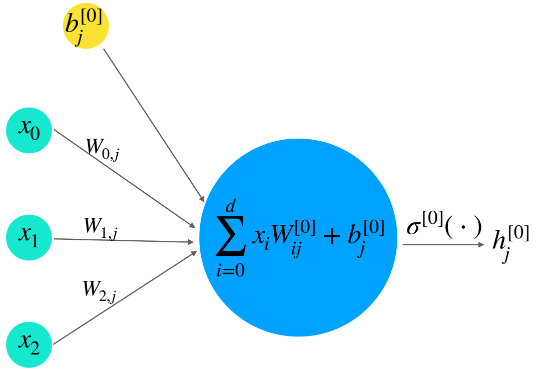


Figure 2: A single fully-connected neuron.

called *one-hot encoding*. A one-hot encoded label vector is a binary vector whose elements are computed according to the following function:

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

For example, for a classification problem with 3 classes, the label for a training point in class 3 might be $(0,0,1)$ and the label for a training point in class 2 might be $(0,1,0)$. 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, in lieu of 0 and 1, 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.

1.4 Convolutional Neural Networks

With fully-connected networks, we represent every datapoint as a 1-dimensional vector. We also generally assume that element d in the vector is independent from element $d+1$; there's no inherent relationship between different elements of the vector. But what if we want to classify *images*? Some of these assumptions break down. Images are inherently 2-dimensional. And there are dependencies between neighboring pixels; if you see part of an image containing a line oriented at 45 degrees, you can probably fill in the rest of the image, extending that line through the 2D plane. To capture these properties, we will need to switch from representing datapoints as 1D vectors to a format that includes 2 spatial dimensions. More generally, we will represent images as *3-tensors* with a third dimension that captures the number of "channels" in the image. For color images, we typically use 3 channels—red, green, and blue (RGB).

We'll also want the "features" (weights/filters) learned by our network to have 2 spatial dimensions, so that we can detect things like circles and eyes and faces. If our input is an $d_1 \times d_2$ image X, we could imagine building a network where our weights in a given layer are stored in a tensor of shape $d_1 \times d_2 \times c \times n$, where n is the number of neurons in that layer and c is the number of channels. But we will quickly face a combinatorial explosion in the number of weights needed to learn useful features from natural images. Imagine that one of the weights learns to represent a cat in the top left corner of the image. What if our dataset includes images with cats in the bottom right corner as well? The top-left-cat feature will be useless for those images and the network would have to dedicate a different weight to representing bottom-right-cat. But it's worse than that. If cats could be expected to appear in any arbitrary location in the image, the network would need to learn a separate weight for every possible position. It would also have to do this for every possible feature that might be needed for the task at hand, such as human faces or hands or buildings. Rapidly, we face a combinatorial explosion.

We can avoid this problem by allowing the weights of the network to be *translation invariant*, conforming the structure of the neural network architecture to the translation invariant structure of natural images. **Convolutional neural networks** do exactly this. Convolutional neural networks were inspired by models of the visual cortex. In the classical model of the visual cortex, each neuron responds to a particular feature in a particular region of the visual field, called the neuron's "receptive field". Information processing in the visual cortex is hierarchical. Neurons in regions of the brain involved in early visual processing extract simple features such as dots and oriented straight lines. As we move up towards later stages of processing, we find neurons representing more complex features, such as curves and crosses, and in the highest areas of visual processing, we find neurons that are selective for faces and other objects. These more complex features are computed as compositions of the simpler features represented in earlier stages. As we move through the visual hierarchy, the size of each neuron's receptive field increases as well, with highly localized features represented in early stages and larger features that dominate most of the visual scene represented in later stages.

Convolutional neural networks achieve these properties by incorporating the following.

- **Convolutional filters**: The weights of a convolutional network are typically referred to as *filters* or *kernels*. In a convolutional network, filters with 2 spatial dimensions (generally smaller than the original image) are *convolved* with the image. This is often referred to as "weight sharing," as the same weights are applied across many different locations in the image.

- **Pooling layers**: Convolutional neural networks typically incorporate layers that downsample the image so that later layers represent the image at a coarser level of resolution, mimicking the increase in receptive field size observed in biological brains.

- **Deep, hierarchical processing**: The convolutional networks used in state-of-the-art image processing are typically very deep (on the order of 15–25 layers). This allows the network to buld up complex features as compositions of simpler features.

Remarkably, visualizations of features learned by a convolutional neural network bear resemblance to many of the features observed in biological neurons in visual cortex, such as oriented lines and curves. If you're interested in understanding the representations learned by convolutional neural networks trained on images, we highly recommend checking out this work: `https://distill.pub/2017/feature-visualization/`

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

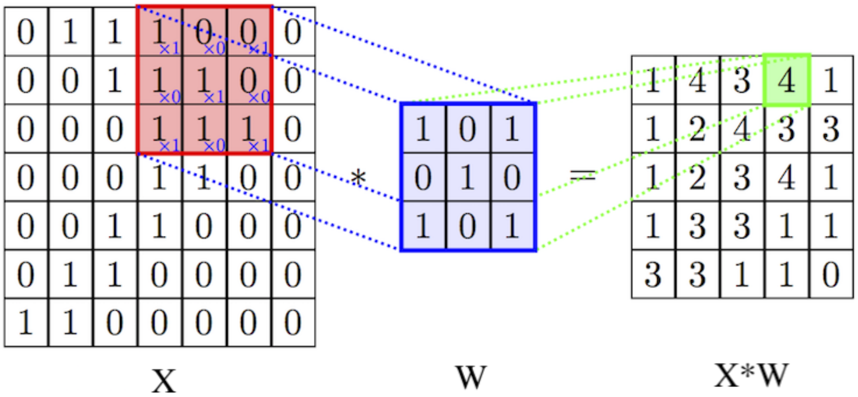


Figure 3: Figure showing an example of one convolution.

- 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.
- $(k_1, k_2)^{[l]}$: 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 elementwise 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

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n] = \sum_i \sum_j \sum_c W[i, j, c, n] X[d_1 + i, d_2 + j, c] + b[n]. \qquad (1)$$

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 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

[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.

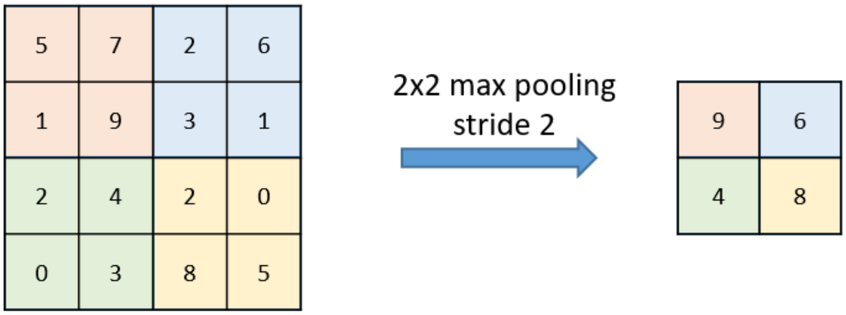


Figure 4: Figure showing an example of a max pooling layer with a kernel size of 2 and stride of 2.

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

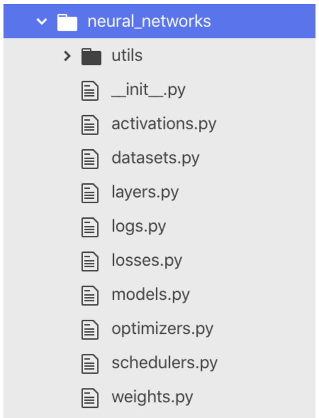
$$Z[r_1, r_2, c] = \text{MaxPool}(X)[r_1, r_2, c] = \max\{X[r_1 s : r_1 s + k_1, r_2 s : r_2 s + k_2, c]\}.$$

The notation on the right hand side should be read as array slicing as in `numpy` (with exclusive end coordinates).

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.

2 The Neural Nets Package

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



```
neural_networks
  utils
  __init__.py
  activations.py
  datasets.py
  layers.py
  logs.py
  losses.py
  models.py
  optimizers.py
  schedulers.py
  weights.py
```

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.

Each type of neural network architecture builds in certain assumptions about the structure of the data it receives. We will begin with a feed-forward, fully-connected network, which makes the fewest assumptions and will build up in complexity from there.

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 (or, input channels in the case of a CNN).
- `n_out`: The number of output units (or, output channels in the case of 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.
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

    # unpack model parameters
    W = self.parameters["W"]
    b = self.parameters["b"]

    # perform an affine transformation and activation
    Z = # some intermediate quantity
    out = # the output

    # store information necessary for backprop in ‘self.cache‘
    self.cache[...] = # something useful for backpropagation
    self.cache[...] = ...

    return out
```

- `backward`: This method takes the gradient of the downstream loss as input and uses the cached values to compute gradients with respect to its inputs and weights. It 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)
    """

    # unpack the cache
    ... = self.cache[...]

    # use values in the cache, along with dLdY to compute derivatives
    dX = # Derivative of loss with respect to X
    dW = # Derivative of loss with respect to W
    dB = # Derivative of loss with respect to b

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

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

    return dX
```

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
```

2.1 Autograder

To test your code for correctness, you may submit your code to Gradescope by following the instructions on the front page of this PDF. This will kick off an autograder which will take a few minutes to run. You are welcome to submit to the autograder as frequently as you wish.

2.2 Debugging

We have included a python notebook named `check_gradients.ipynb`, which you can use for debugging your layers' gradient computations. It will compare the gradients you implement for various layers against gradients computed numerically as

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\ldots, a_k + \epsilon, \ldots) - f(\ldots, a_k - \epsilon, \ldots)}{2\epsilon}$$

for some small ϵ. If your gradient implementations are correct, they should be close enough to the numerical approximations such that the error between them is very small (usually on the order of 10^{-8} or smaller).

Something to note, however, is that this notebook only checks if the gradients you implement for a layer's backward pass are consistent with the implementation of that layer's forward pass. It's possible to have an incorrect forward pass implementation and a consequently incorrect backward pass implementation that still yields low errors in the notebook. To check if your implementations are correct, you should submit to the Gradescope autograder!

2.3 (Optional) Generating LaTex or Markdown for your write-up

Please run

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

for instructions on how to use the script.

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).

3 Layer Implementations

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

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.

3.1 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 the script `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.

3.2 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.

3.3 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.

3.4 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.

4 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.

5 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.**

5.1 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 (see equation 1). **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.)

6 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.

6.1 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%.

6.2 CNNs for CIFAR-10

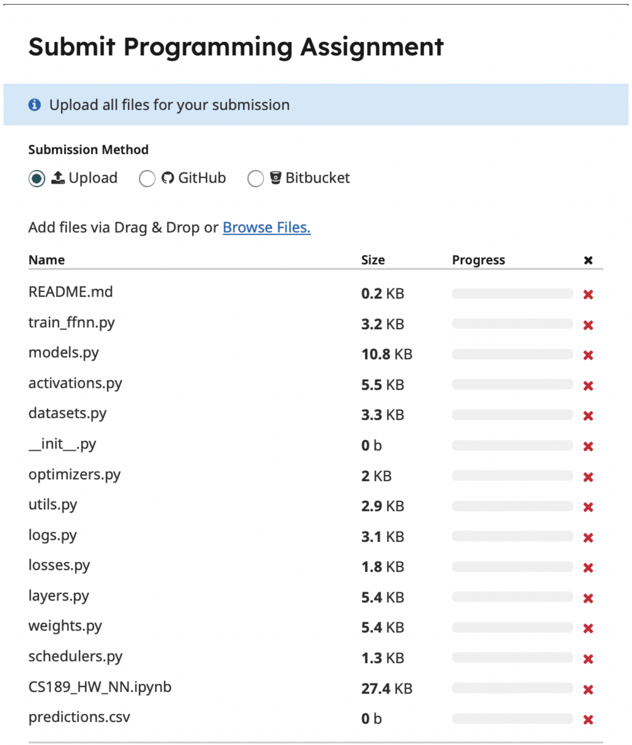
Deliverables

- 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%.

7 Code Submission

Code submission instructions:

- Your submission should include all the files in the neural networks package. You should be able to simply drag and drop these files directly into the Gradescope submission box; **don't package the files in a directory or zip them in any way.**
- Do **NOT** submit any data files that we provided.
- Please also include your final `train_ffnn.py` script and a `README` containing instructions for reproducing your results (ex. the plots in section 4).
- Download the Colab notebook containing all of the code you wrote for the deliverables in the PyTorch section, and include it with the rest of your submission.
- Finally, also attach the `predictions.csv` file containing your predictions on the provided CIFAR-10 test set.
- At the end, your Gradescope submission screen might look like the following:



Submit Programming Assignment

Upload all files for your submission

Submission Method

● Upload ○ GitHub ○ Bitbucket

Add files via Drag & Drop or Browse Files.

Name	Size	Progress	✖
README.md	**0.2** KB		✖
train_ffnn.py	**3.2** KB		✖
models.py	**10.8** KB		✖
activations.py	**5.5** KB		✖
datasets.py	**3.3** KB		✖
__init__.py	**0** b		✖
optimizers.py	**2** KB		✖
utils.py	**2.9** KB		✖
logs.py	**3.1** KB		✖
losses.py	**1.8** KB		✖
layers.py	**5.4** KB		✖
weights.py	**5.4** KB		✖
schedulers.py	**1.3** KB		✖
CS189_HW_NN.ipynb	**27.4** KB		✖
predictions.csv	**0** b		✖

8 Honor Code

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