CS 189 / 289 Introduction to Machine Learning
Fall 2023 Jennifer Listgarten, Jitendra Malik HW3

Due 10/06/23 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" (read: many-layer) neural networks, such as the deep reinforcement learning algorithm that learned to play Atari from pixels, or the GPT-2 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 a **feed-forward fully-connected network**, all in plain `numpy`. 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 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** (e.g. learning rate, batch size, etc.).

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 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.

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).

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

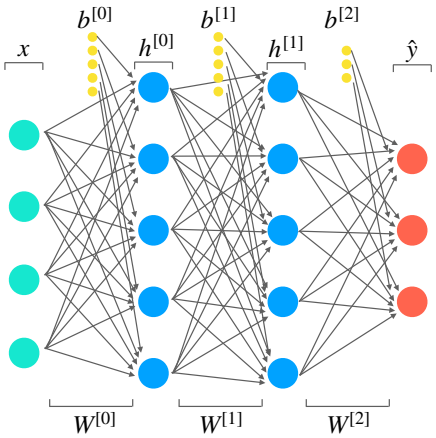

Figure 1: A 3-layer fully-connected neural network.

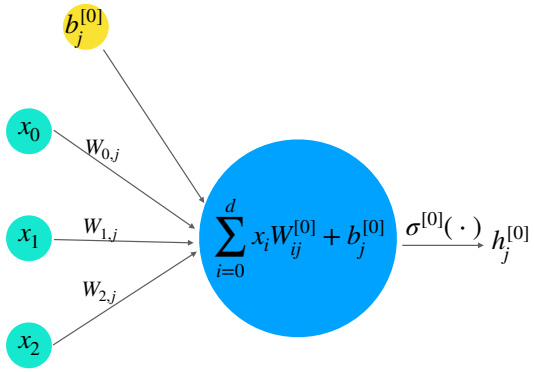

Figure 2: A single fully-connected neuron.

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 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 $1 \times n^{[i]}$.
- $\sigma^{[l]}(\cdot)$: 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 $l + 1$ (at layer 0, $h^{[l-1]}$ 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 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 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 an example from class 3 might be $[0, 0, 1]$.

2 The Neural Nets Package

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

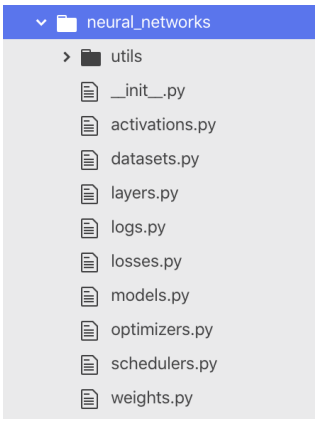

Figure 3: The structure of the starter codebase.

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.
- `n_out`: The number of output units.

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 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).

2.2 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

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.

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

3.1 Activation Functions

First, you will implement an activation function in `activations.py`. You will implement the forward and backward passes for the ReLU activation function, commonly used in the hidden layers of neural networks,

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

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

Instructions

1. 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`. **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. Please read the docstrings and the function signatures before you begin writing any code. Then, 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 2.1.

Instructions

1. First, derive the partial derivatives of the downstream loss L with respect to W and b in the fully-connected layer, i.e., $\frac{\partial L}{\partial W}$ 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, i.e. $\frac{\partial L}{\partial X}$, which will be passed to the lower layers. **Again,**

you must arrive at a solution for batched X and Z. Please express your solution in terms of $\frac{\partial L}{\partial Z}$, which you have already obtained in the previous question, 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 data matrix X 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. **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 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 probabiity 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 $m = \max_{j=0}^{k} s_j$. We recommend implementing this method. You can verify yourself why these two formulations are equivalent.

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. **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

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

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.** Include your code in the appendix and select the appropriate pages when submitting to Gradescope.

3.5 Two-Layer Networks

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

Instructions

1. Fill in the `forward` and `backward` 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 the Iris dataset has not been normalized or standardized in any way.
    - The stochastic gradient descent optimizer (in `optimizers.py`), which performs the gradient updates and optionally incorporates a momentum term.
    - 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

   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 plots showing the training and validation loss across different epochs and report your final test error.

4 PyTorch

Please go through the Google Colab Notebook here that will introduce you to PyTorch, and will contain the questions to be completed for this section.

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

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

4.1 MLP for Fashion MNIST

Deliverables:

- Provide the code for training an MLP on the fashion MNIST dataset (include it in your appendix).
- 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%.

4.2 CNNs for CIFAR-10

Deliverables:

- Provide the code for training a CNN model on the CIFAR-10 dataset (include it in the appendix).
- 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.

5 Code Submission

In order to turn in your code, create a folder called `hw3` that contains the following:

- The entire `neural_networks` directory. This includes any files that you edited when writing your forward and backward passes as well as any files that went untouched.
- Your `train_ffnn.py` script and a `README` containing instructions for reproducing your results (ex. the plots in section 3.5).
- The `predictions.csv` file containing your predictions on the provided CIFAR-10 test set.
- The completed python notebook `CS189_HW_NN.ipynb` itself.

The unzipped version of your submission should have the following file structure:

```
hw3/
├── neural_networks
│   ├── utils/
│   │   └── ...
│   ├── __init__.py
│   ├── activations.py
│   ├── ...
│   └── weights.py
├── train_ffnn.py
├── README.md
├── predictions.csv
└── CS189_HW_NN.ipynb
```

Now, compress this folder into a single zip file. If you are a Mac user, **do not use the default "Compress" option to create the zip**. It creates artifacts that the autograder does not like. You may instead use `zip -vr hw3.zip hw3 -x "*.DS_Store"` from your terminal. Submit the `hw3.zip` file to the Gradescope assignment called "Homework 3 Code."