EE 519: Deep Learning Theory & Fundamentals **Spring 2023**

Homework 1
Due: April 21, 2023, 11:59PM PT

Student Name: *Instructor Name: John Lipor*

For this assignment, you will begin coding your own object-oriented deep learning library called *MyTorch*. This library will have the below file structure, allowing you to call functions in a manner similar to PyTorch (see included code). Your job is to complete the respective functions and test your deep learning library.

Note: This is a pure `numpy` implementation, so you may not import any additional libraries.

```
hw01files
└── mytorch
│   ├── nn
│   │   ├── activation.py
│   │   ├── linear.py
│   │   └── loss.py
│   └── optim
│       └── sgd.py
└── models
    └── mlp.py
```

Code: Please attach your code to the end of your final document. You do not need to link your code for each problem within gradescope, only the outputs of the respective jupyter notebooks.

Problem 1 `linear.py` (1 pt, 2 pts, 4 pts)

Your first task is to complete the function `linear.py`, which initializes a single linear layer with the given input and output sizes, and performs the forward and backward operations. Recall that the forward pass of a linear layer performs the operation

$$O = AW + 1_N b^T, \tag{1}$$

where $A \in \mathbb{R}^{N\times d}$ is the input data matrix of N examples, $W \in \mathbb{R}^{d\times h}$ is the weight matrix, $b \in \mathbb{R}^h$ is the bias term, and $1_N \in \mathbb{R}^N$ denotes the vector of all ones.

To perform backpropagation, we require the gradient of the loss with respect to the output O, which will be implemented in a later problem. We then wish to output the gradients with respect to the parameters W and b as well as the inputs A. Assuming we have access to $\partial L/\partial O$, these gradients are defined by the equations

$$\begin{aligned} \frac{\partial L}{\partial A} &= \frac{\partial L}{\partial O}\left(\frac{\partial O}{\partial A}\right)^T \\ \left(\frac{\partial L}{\partial W}\right)^T &= \left(\frac{\partial L}{\partial O}\right)^T \frac{\partial O}{\partial W} \\ \frac{\partial L}{\partial b} &= \left(\frac{\partial L}{\partial O}\right)^T \frac{\partial O}{\partial b}. \end{aligned} \tag{2}$$

(a) Implement the `__init__` function, initializing W using a normal distribution with variance 0.01 and b to be all zeros.

(b) Implement the `forward` method as defined by (1). This method receives the input data matrix A and returns the output data matrix O.

(c) Implement the `backward` method as defined by (2). Note that you need to determine the gradients of the output with respect to the various inputs yourself. This method receives $\partial L/\partial O$ as input, returns $\partial L/\partial A$, and stores $\partial L/\partial W$ and $\partial L/\partial b$ to the `linear` object.

Turn in the output of the notebook `LinearTester.ipynb`. Note that you can easily save the output in pdf form and attach to a LaTeXdocument using the `include` command.

Problem 2 `loss.py` (2 pts each)

Your next task is to implement both the mean squared error (MSE) and cross entropy losses as their own classes in `loss.py`. Defining these classes allows us to treat losses as objects, which allows for storing and passing gradients.

(a) Implement the `forward` method for the `MSELoss` class. This method takes as inputs the output matrix $O \in \mathbb{R}^{N \times q}$ and returns the MSE with respect to the true targets $Y \in \mathbb{R}^{N \times q}$. Note that the MSE is normalized by dividing by Nq rather than just N.

(b) Implement the `backward` method for the `MSELoss` class. This method takes no input arguments and uses the stored variables O and Y to compute and return $\partial L / \partial O$.

(c) Implement the `forward` method for the `CrossEntropyLoss` class. This method takes as inputs the output matrix $O \in \mathbb{R}^{N \times q}$ and returns the MSE with respect to the true targets $Y \in \mathbb{R}^{N \times q}$. Note that the cross entropy loss is normalized by dividing by N only.

(d) Implement the `backward` method for the `CrossEntropyLoss` class. This method takes no input arguments and uses the stored variables O and Y to compute and return $\partial L / \partial O$.

Turn in the output of the notebook `LossTester.ipynb`.

Problem 3 `sgd.py` (2 pts)

Now that we have the ability to pass gradients, we can use stochastic gradient descent (SGD) to optimize the parameters of our linear layer. Your task is to implement the `step` method, which updates both the weight matrix W and bias vector b. Note that the provided implementation allows for multiple layers, but the update equation is the same for either case.
Turn in the output of the notebook `SGDTester.ipynb`.

Problem 4 `activation.py` (2 pts each)

At this point, we have implemented a single-layer neural network that can be used for either classification or regression. You can (and probably should) test your network using the notebook `TrainingTester.ipynb` to see how it performs on these tasks. To allow our network to learn nonlinear functions, we need to utilize activation functions, which are implemented in `activation.py`.

Consider a multi-layer perceptrion (MLP) with a single hidden layer. Let $H \in \mathbb{R}^{N \times h}$ be the output of hidden variables after the first linear layer. For an activation function $\phi : \mathbb{R} \to \mathbb{R}$, let $A \in \mathbb{R}^{N \times h}$ be the output after applying ϕ element-wise to H.

(a) Implement the sigmoid activation function. Complete the `forward` method, which takes the hidden variables H as input and outputs the the activation function applied element-wise to H. Next, complete the `backward` method, which utilizes the stored variable H to compute the element-wise gradient of the activation function with respect to H. Note that the gradient $\partial A / \partial H$ should have the same dimensions as H.

(b) Implement the tanh activation function, completing both the `forward` and `backward` methods.

(c) Implement the ReLU activation function, completing both the `forward` and `backward` methods.

Turn in the output of the notebook `ActivationTester.ipynb`.

Problem 5 `mlp.py` (4 pts each)

Having put into place the tools for optimizing a linear layer, selecting the appropriate cost function, and adding nonlinearities, we are now ready to implement a MLP with an arbitrary number of layers. In this problem, we will complete the `mlp.py` file, implementing MLPs with 0, 1, and 4 hidden layers.

(a) Complete the class `MLP0` to implement an MLP with 0 hidden layers. The class structure is as follows.

- `__init__`: Defines the list of layers in `self.layers` and activation functions in `self.f`. For the MLP0, we have one linear layer and a single Identity activation function.
- `forward`: Performs a forward pass, first through the linear layer, then through the activation function. The parameters follow the equations

$$\begin{aligned} H^{(1)} &= XW^{(1)} + 1_N(b^{(1)})^T \\ A^{(1)} &= \phi\left(H^{(1)}\right) \text{ (element wise)} \end{aligned}$$

- `backward`: Performs a backward pass through the network. This method takes the gradient $\partial L/\partial A^{(1)}$ as input, then computes $\partial A^{(1)}/\partial H^{(1)}$ via the `backward` method for the corresponding activation function. It then computes

$$\frac{\partial L}{\partial H^{(1)}} = \frac{\partial L}{\partial A^{(1)}} \odot \frac{\partial A^{(1)}}{\partial H^{(1)}}.$$

  Noting that $H^{(1)}$ is the output of our single linear layer, it uses $\partial L/\partial H^{(1)}$ to compute the gradients of the loss with respect to $W^{(1)}$, $b^{(1)}$, and the inputs X. These gradients can be used with SGD or passed to additional layers, as in parts (b) and (c) of this problem.

(b) Complete the class `MLP1` to implement an MLP with 1 hidden layer. In this case, our forward equations become

$$\begin{aligned} H^{(1)} &= XW^{(1)} + 1_N(b^{(1)})^T \\ A^{(1)} &= \phi\left(H^{(1)}\right) \text{ (element wise)} \\ H^{(2)} &= A^{(1)}W^{(2)} + 1_N(b^{(2)})^T \\ A^{(2)} &= \phi\left(H^{(2)}\right) \text{ (element wise).} \end{aligned}$$

(c) Follow the above sequence to complete the class `MLP4`.

Turn in the output of the notebook `MLPTester.ipynb`.

Problem 6 `TrainingTester.ipynb` (0 pts)

Congratulations! If you've reached this point, you have coded a pure-`numpy`, object-oriented implementation of an MLP with arbitrary number of layers and activation functions, capable exciting tasks such as regressing synthetic data and classifying pictures of clothing! In `TrainingTester.ipynb`, I have implemented the framework to allow for training neural networks with MyTorch using tools from the D2L library. Have a look at the code in this notebook to observe what changes were required to utilize our library, as well as how to perform the same tasks using PyTorch.

Turn in nothing.