Lab 6

Due: Friday 11/08/2024 @ 11:59pm EST

The purpose of labs is to give you some hands on experience programming the things we've talked about in lecture. This lab will focus on learning (i.e. "fitting") using gradient descent on a Logistic Regression model.

Task 0: Setup

Included with this file is a new file called `requirements.txt`. This file is rather special in Python: it is the convention used to communicate dependencies that your code depends on. For instance, if you open this file, you will see entries for `numpy` and `scikit-learn`: two pythn packages we will need in order to run the code in this lab. You can download and install these python packages through `pip`, which is python's package manager. While there are many ways of invoking `pip`, I like to do the following:

```
python3 -m pip install -r requirements.txt
```

(I let `python3` figure out which `pip` is attached to it rather than the more common usage: `pip install -r requirements.txt`)

Task 1: Code Organization

In this lab you will be working on a file called `lr.py`. Inside of this file is quite a lot of code. Whenever we want to do gradient descent, we often organize our code into a class heirarchy to take advantage of *chain rule*. Chain rule is the mathematical equivalent of divide and conquer for computing gradients. Chain rule lets us split up complicated gradient equations into a collection of smaller, easier to calculate derivatives, and then assemble them together into the final gradient. When doing logistic regression, we will be calculating predictions like so:

$$z^{(i)} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j^{(i)}$$

$$\hat{y}^{(i)} = \sigma(z^{(i)}) \qquad \text{where } \sigma(z) = \frac{1}{1+e^{-z}}$$

We then evaluate the loss (i.e. error) of these predictions by comparing them against the ground truth using the following loss function:

$$L(\vec{\hat{y}}, \vec{y}_{gt}) = \sum_{i=1}^{N} y_{gt}^{(i)} \log(\hat{y}^{(i)}) + (1 - y_{gt}^{(i)}) \log(1 - \hat{y}^{(i)})$$

When training logistic regression using gradient descent, we want to use chain rule to split up the complicated derivative $\frac{\partial L}{\partial \vec{\beta}}$ into its smaller pieces $\frac{\partial L}{\partial \vec{\hat{y}}}$, $\frac{\partial \vec{\hat{y}}}{\partial \vec{z}}$, and $\frac{\partial \vec{z}}{\partial \vec{\beta}}$.

Our code implements this division of labor using chain rule. Every operation we can perform in logistic regression shares the same root of an object heirarchy called a `Module`. The `Module` class defines an api that all child classes must follow. There are three methods:

- `forward(X)`. This method, given input to the `Module`, will execute the equation that the `Module` is supposed to do. For example, if we had a `Sigmoid` class than extends `Module`, then the `forward(X)` method of the `Sigmoid` class would compute $\frac{1}{1+e^{-X}}$.

- `backward(X, dLoss_dModule)`. This method is where chain rule happens. Inside of the `backwards` method, the tiny piece of chain rule for this `Module` needs to be computed, and then combined with the gradient of the output of this `Module`. For instance, a `Sigmoid` class would need to compute $\frac{\partial \vec{\hat{y}}}{\partial \vec{z}}$ and then combine this tiny piece of chain rule with the incoming gradient $\frac{\partial L}{\partial \vec{\hat{y}}}$ to produce $\frac{\partial L}{\partial \vec{z}}$.
- `parameters()`. This method returns whatever parameters (stored as `Parameter` objects) that the `Module` contains within it. For instance, `Sigmoid` has no parameters inside it, but if we had class called `LinearRegression` that extends `Module`, then its `paramters` method would return a list with the `Parameter` object for $\vec{\beta}$ inside.

Any loss function cannot itself be a `Module` because of the need for a slightly different api. So, loss functions have their own base class called `LossFunction`. The api for a loss function is very similar (`forwards` and `backwards` methods), but the required arguments are slightly different.

Task 1: `class BCE` (25 points)

In the file `lr.py`, you will find a class called `BCE`. This class is for the binary cross entropy loss function that we derived in class. The version we derived in class looks like this:

$$L(\vec{\hat{y}}, \vec{y}_{gt}) = \sum_{i=1}^{N} y_{gt}^{(i)} \log(\hat{y}^{(i)}) + (1 - y_{gt}^{(i)}) \log(1 - \hat{y}^{(i)})$$

in code however, we often like to keep gradients small (for numerical stability). We can achieve this by scaling down the loss function like so:

$$L(\vec{\hat{y}}, \vec{y}_{gt}) = \frac{1}{N} \sum_{i=1}^{N} y_{gt}^{(i)} \log(\hat{y}^{(i)}) + (1 - y_{gt}^{(i)}) \log(1 - \hat{y}^{(i)})$$

Your task here is to complete the `backwards` method of this class. You will need to populate the entries in the variable `dLoss_dY_Hat` with the correct values by first differentiating (on paper) the above loss function equation and then implementing this equation to set the values.

Task 2: Sigmoid (25 points)

In the file `lr.py`, you will find a class called `Sigmoid`. This class is for the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Note that this function is element-wise independent. If you were to calculate sigmoid of a vector or a matrix, you would apply sigmoid to each element separately.

Your task here is to complete the `backwards` method of this class. You will need to populate entries in the variable `dModule_dX` with the correct values by first differentiating (on paper) the sigmoid activation function. Let $y = \sigma(x)$. You will first need to differentiate $\frac{\partial y}{\partial x}$ on paper, and then implement this equation in your code to set the correct values of `dModule_dX`.

You will then need to combine `dModule_dX` with `dLoss_dModule` to set the variable `dLoss_dX`.

Task 3: LinearRegression (50 points)

In the file `lr.py`, you will find a class called `LinearRegression`. This class is for the linear regression model that logistic regression uses internally:

$$z^{(i)} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j^{(i)}$$

Your task here is to complete the `backwards` method of this class. There are a few things you need to do here. Since `LinearRegression` contains the parameters of the model (i.e. the beta values), you first need to set the value of the variable `dModule_dBeta`. This tiny piece of chain rule contains the derivative $\frac{\partial z^{(i)}}{\partial \vec{\beta}}$. You then need to combine it with the argument `dLoss_dModule` which contains $\frac{\partial L}{\partial \vec{z}}$ to compute the overall gradient $\frac{\partial L}{\partial \vec{\beta}}$. Note that this value is used to increment the gradient of the beta `Parameter` object.

You then need to, just like every other `Module`, need to compute `dModule_dX` and then combine it with `dLoss_dModule` to get `dLoss_dX`.

Task 4: Submit Your Lab

To complete your lab, please **only turn in the `lr.py` file** on Gradescope. You shouldn't have to worry about zipping it up or anything, just drag and drop it in.