Programming Assignment 4: Backpropagation... From the Bottom Up

Due March 8, 2023

In the repository you have starter code for the fourth programming assignment, 'pa4.py.' In this assignment, you will work through implementing gradient descent using backpropagation for a fully connected neural network, with the objective of constructing a binary classification model, and each activation function the standard sigmoidal $\sigma(t) = (1 + \exp(-t))^{-1}$. Your primary task in this assignment is to define various gradients, so that you can successfully fit a classification model. Your starter code comes ready to go with a constructor for arbitrary input and hidden layer dimensions, as well as a forward method.

The output layer is $\mathbb{R}^2$ and you should use the cross entropy loss (c.f. chapter 3 of Nielsen's text). Unlike a vanilla classification model wherein $\tilde{y} : \mathcal{X} \to [0,1]$ denotes a score loosely interpretable as the likelihood that point $x \in \mathcal{X}$ belongs or is associated to $y = 1$,[*] our network $\nu : \mathcal{X} \to \mathbb{R}^2$ maps $x \mapsto \nu(x) = (\nu_0(x), \nu_1(x))$ where now you may think of $\nu_j(x) \approx \mathbb{P}_{\mathcal{Y}|\mathcal{X}}(y = j|x)$. Of course, there is no apriori reason why $\nu_0 + \nu_1 \equiv 1$, but there are easy ways to normalize output to maintain this probability interpretation, e.g. see the softmax function. We do not need to worry about enforcing the constant sum with normalization. Labeled data now lives in $\mathcal{Y} = \{0,1\}^2$ though all labels are either $(1,0)$ (corresponding to 0) or $(0,1)$ (corresponding to 1); in other words, you should not see either $(0,0)$ or $(1,1)$ in your *labeled* data. Depending on how you associate $\nu(x)$ to an element of $\mathcal{Y}$, however, the model *may* suggest more than one class $(1,1)$ or neither $(0,0)$ (e.g. if you choose to set predicted label by threshold instead of argmax). For binary classification, cross-entropy may seem to obfuscate our problem, but observe that extending to multi-class classification becomes straightforward: for k classes, $\mathcal{Y} = \{0,1\}^k$ and the label for jth class is the standard basis vector e_j with a 1 in the jth component and zeros elsewhere (using pythonic indexing for basis $\{e_0, e_1, \ldots, e_{k-1}\}$ of $\mathbb{R}^k$).

Recall that a neural network $\nu : \mathcal{X} \to \mathcal{Y}$ is "simply" a composition of functions

$$\begin{tikzcd}
\mathbb{R}^{n_0} \arrow[r, "\cdots"] \arrow[rrrrrr, bend left, "\nu"] & \mathbb{R}^{n_{j-1}} \arrow[rr, "a^j \circ z^{j-1}"] \arrow[dr, "z^j"'] & & \mathbb{R}^{n_j} \arrow[rr, "a^{j+1} \circ z^j"] \arrow[dr, "z^{j+1}"'] & & \mathbb{R}^{n_{j+1}} \arrow[r, "\cdots"] & \mathbb{R}^{n_{L+1}} \arrow[dr, "c(\cdot,y)"] & \\
& & \mathbb{R}^{n_j} \arrow[ur, "a^j"'] & & \mathbb{R}^{n_{j+1}} \arrow[ur, "a^{j+1}"'] & & & \mathbb{R}.
\end{tikzcd}$$

The last diagonal is not part of the network; the cost function $c(\cdot, y) : \mathbb{R}^{n_{L+1}} \to \mathbb{R}$ maps $\nu(x) \mapsto c_\nu(x,y)$. You will iteratively compute $\frac{\partial c(\cdot,y)}{\partial w^j}$ and $\frac{\partial c(\cdot,y)}{\partial b^j}$, liberally relying on various curried forms of $c(\cdot, y)$. For fixed *input* $(x,y) \in \mathcal{X} \times \mathcal{Y}$, and model-defining parameters $(w_{\setminus j}, b_{\setminus j}) := (w^0, b^0, \ldots, w^{j-1}, b^{j-1}, w^{j+1}, b^{j+1}, \ldots, w^L, b^L)$, we define

$$c^j_{(x,y,w_{\setminus j},b_{\setminus j})}(\cdot) : \mathbb{R}^{(n_j+1)\cdot n_{j+1}} \to \mathbb{R}$$

by the composition

$$(w^j, b^j) \mapsto (w, b) := (w^0, \ldots, w^{j-1}, b^{j-1}, w^j, b^j, w^{j+1}, b^{j+1}, \ldots, b^L) \mapsto c_{\tilde{y}_{(w,b)}}(x,y).$$

Therefore, the gradient $\nabla_{(w^j,b^j)} c^j(x, y, w_{\setminus j}, b_{\setminus j}) \in \mathbb{R}^{(n_j+1)\cdot n_{j+1}}$. Practically speaking, the gradient w.r.t. w^j is an $n_j \cdot n_{j+1}$ *matrix* while the gradient wrt to b^j is a vector squarely in $\mathbb{R}^{n_{j+1}}$. If the formalism is annoying, then ignore it!, just be careful to note that $w^j \mapsto w^j a^j + b^j$ (w.r.t. which you will be computing derivatives) is *not* the same map as $a^j \mapsto w^j a^j + b^j$: the gradient in the latter case is generally smaller.

Follow steps as outlined in comments under the main function. You will start by generating data and defining the cost function in class FCNetFS. This cost should return both the cross entropy, an array of size $1 \times m$ for data $((x_1, y_1), \ldots, (x_m, y_m)) \in (\mathcal{X} \times \mathcal{Y})^m$, and an array of the gradient $(\nabla_\nu c_\nu(x_1, y_1), \ldots, \nabla_\nu c(x_m, y_m))$ of size $2 \times m$. You should sanity check that this gradient is pointed in the right direction. In particular, you should first of all understand which direction it should be pointing![†] Then you will fill in the backward() method. Notice the input 'y_layers' which is a *list* $[a^0, a^1, \ldots, a^{L+1}]$ of each layer's activation. You will iterate through each layer, starting at the end, noticing that the chain rule will successively map gradients, starting with $\nabla_\nu c_\nu(x,y)$, *backward*:

$$\begin{tikzcd}
\mathbb{R}^{n_0} & \mathbb{R}^{n_{j-1}} \arrow[l, "\cdots"'] & \mathbb{R}^{n_j} \arrow[l] & \mathbb{R}^{n_{j+1}} \arrow[l] & \mathbb{R}^{n_{L+1}} \arrow[l, "\cdots"']
\end{tikzcd}$$

[*] Associated how?, w.r.t. measure $\mathbb{P}_{\mathcal{X}\times\mathcal{Y}}$ from which $(x,y) \sim \mathbb{P}_{\mathcal{X}\times\mathcal{Y}}$ is sampled!

[†] Keep in mind that a gradient update for cost-minimization would *subtract* the gradient.

You may refer to class notes and Nielsen's text (chapter 2) if you get stuck, but I highly encourage you to work out the math manually: your biggest headache will be making sizes align.