DS 102 Discussion 7

Wednesday, October 20, 2021

1. **Activation Functions for Neural Networks**

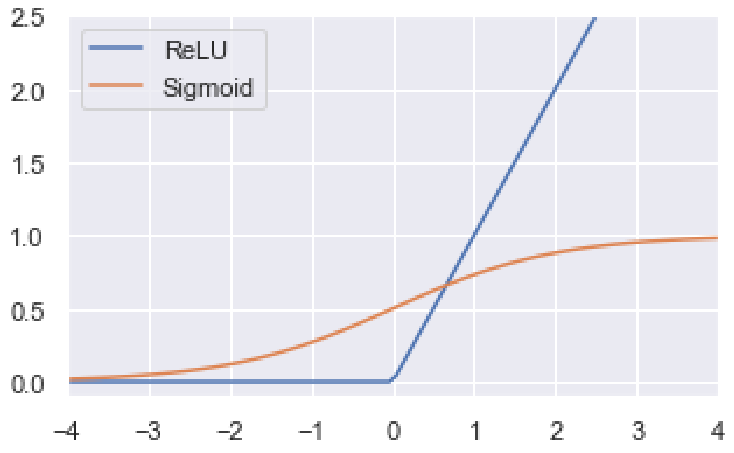
   We saw in lecture that neural networks can learn complex, non-linear patterns within data using activation functions. One simple activation function which you've seen in this class before is the *sigmoid* function used in Logistic Regression, defined as:

$$\sigma(x) = \frac{1}{1 + \exp(-x))}$$

   However, most modern-day implementations of neural networks avoid using sigmoid functions for the nonlinearity, and instead favor functions like the *REstricted Linear Unit*, commonly abbreviated as ReLU. This function is defined as follows:

$$\text{ReLU}(x) = \max(0, x)$$

   We can visualize what these activation functions look like in the plot below:



   (a) *Sketching the Derivatives*

   Referencing the plot above, sketch the derivatives of the sigmoid and ReLU activation functions, overlaid on the same plot.

(b) *Behavior of Activation Functions*

What happens to the derivative for each of the activation functions as the input gets large?

(c) *The Vanishing Gradient Problem*

When optimizing neural networks, why is it bad for the gradient values to be 0? What can you do to avoid this problem?

2. **Backpropagation for a Two-Layer Neural Network**

Consider a two-layer neural network that computes a real-valued function of the form $f_{Ab}(x) = b^T\sigma(Ax)$ where $x \in \mathbb{R}^m$, $A \in \mathbb{R}^{h\times m}$, $b \in \mathbb{R}^h$, and σ is the element-wise sigmoid function given by $\sigma(x) = 1/(1 + \exp(-x))$ (the subscript notation in f_{Ab} is used to emphasize that A and b are the parameters of the function). In other words, the neural network has input size m, h units in the hidden layer, and a single scalar output. Note that this model is a simplification of what we saw in Lecture since we do not have a bias term.

The neural network f_{Ab} can be trained to predict a real-valued output given an m-dimensional input (a regression problem). Given a dataset of n input-output pairs, $\{(x_i, y_i)\}_{i=1}^n$, a common way of training a neural network to perform this task is to find the parameter values (values of the matrix A and the vector b) that minimize the squared error loss over the dataset:

$$\underset{A,b}{\operatorname{argmin}} \sum_{i=1}^{n}(y_i - f_{Ab}(x_i))^2 = \underset{A,b}{\operatorname{argmin}} \sum_{i=1}^{n}(y_i - b^T\sigma(Ax_i))^2.$$

To perform this minimization, gradient descent is conducted on the loss with respect to the parameters A, b.

For simplicity, here we'll just focus on the partial derivatives of the squared error loss evaluated on a single data point, (x, y):

$$\mathcal{L}(A, b) = (y - f_{Ab}(x))^2 = (y - b^T\sigma(Ax))^2. \tag{1}$$

Backpropagation leverages the chain rule, along with dynamic programming, to compute the required partial derivatives $\frac{\partial\mathcal{L}(A,b)}{\partial A}$ and $\frac{\partial\mathcal{L}(A,b)}{\partial b}$ in an efficient way. This requires first computing intermediate quantities in the computation graph in what's called a "forward pass". That is, backpropagation first computes $\mathcal{L}(A, b)$ by computing the quantities $z_1 = Ax$, $z_2 = \sigma(z_1)$, $z_3 = b^Tz_2$, the error $z_4 = y - z_3$, then finally the loss $\mathcal{L}(A, b) = z_4^2$. Backpropagation then performs a "backward pass" to compute the partial derivatives, starting with $\frac{\partial\mathcal{L}(A,b)}{\partial b}$.

(a) *Drawing a Computation Graph*

For the loss function defined in Equation (1), draw the corresponding computation graph. Label intermediate quantities z_1, z_2, z_3, and z_4 in the graph.

(b) *Updating* b

Using the chain rule, write down an expression for $\frac{\partial \mathcal{L}(A,b)}{\partial b}$. Use intermediate quantities from the forward pass (the z variables) listed above wherever possible, since these have already been computed after the forward pass.

Hint: Note that b is an h-dimensional vector, so the partial derivative will be an h-dimensional vector. The expression $b^T\sigma(Ax) = b^T z_2$ is a dot product between the vector b and the vector z_2. Recall that for a dot product between two vectors $v^T w$, we have $\frac{\partial v^T w}{\partial v} = w$.

(c) *Updating* A

Using the chain rule, write down an expression for $\frac{\partial \mathcal{L}(A,b)}{\partial A}$. Once again, use the intermediate quantities from the forward pass wherever possible.

Hint: A is an $h \times m$-dimensional matrix, so the partial derivative will be an $h \times m$-dimensional matrix. You can approach this problem by noting that

$$\frac{\partial \mathcal{L}(A,b)}{\partial A} = 2(y - b^T\sigma(Ax)) \cdot -\frac{\partial b^T\sigma(Ax)}{\partial A}$$

and finding the partial derivative of $b^T\sigma(Ax)$ with respect to each element A_{ij} of A. Use this result to write the partial derivative of A in terms of matrices and/or vectors. Note that the derivative of the sigmoid function is $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$.

(d) *Why use Backpropagation?*

Go back to the computation graph you drew in Part (a). If you were to compute the derivatives for weights A and b one at a time, how many total derivatives would you need to compute to perform one round of weight updates? In comparison, how many derivatives many does the backpropagation algorithm actually compute? How would these numbers change as the neural network becomes more complex (e.g. adding more layers)?