DS 102 Discussion 6

Wednesday, March 9, 2022

1. **Trees, Forests, Bias, and Variance**

Recall that we can express the Frequentist Risk of a decision procedure $\delta(x)$ with respect to parameter θ as:

$$R(\theta) = \mathbb{E}\left[(\delta(x) - \theta)^2\right] = \underbrace{\mathbb{E}\left[(\delta(x) - \mathbb{E}[\delta(x)])^2\right]}_{\text{Variance of } \delta(x)} + \underbrace{(\mathbb{E}[\delta(x)] - \theta)^2}_{\text{Bias}^2 \text{ of } \delta(x)}$$

If our decision is a prediction for y that we call $\hat{y}$ and $\delta(x) = \hat{y}(x)$, then we can re-write the above expression as:

$$E[(\hat{y}(x) - y)^2] = \underbrace{E\left[(\hat{y}(x) - E[\hat{y}(x)])^2\right]}_{\text{Variance of prediction } \hat{y}(x)} + \underbrace{(E[\hat{y}(x)] - y)^2}_{\text{Bias}^2 \text{ of } \hat{y}(x)}$$

In this question, we will consider the bias-variance decomposition for two non-parametric models: Decision Trees and Random Forests.

(a) *Bias-Variance Decomposition for Decision Trees*

Consider a Decision Tree trained without a limit on depth. Describe this model's bias and variance.

(b) *Bias-Variance Decomposition for Random Forests*

Compare a Random Forest's bias and variance to those of a Decision Tree. Which model would you expect to generalize better to unseen data and why?

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

Consider a two-layer neural network that computes a real-valued function of the form $f_{W_1,w_2}(x) = w_2^T\sigma(W_1^T x)$ where $x \in \mathbb{R}^m$, $W_1 \in \mathbb{R}^{m\times h}$, $w_2 \in \mathbb{R}^h$, and σ is the element-wise sigmoid function given by $\sigma(x) = 1/(1+\exp(-x))$ (the subscript notation in f_{W_1,w_2} is used to emphasize that W_1 and w_2 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 14, since we do not have a bias term.

The neural network f_{W_1,w_2} can be trained to predict a real-valued output given an m-dimensional input (a regression problem). Given a data set 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 W_1 and the vector w_2) that minimize the squared error loss over the data set:

$$\operatorname*{argmin}_{W_1,w_2} \sum_{i=1}^{n} (y_i - f_{W_1,w_2}(x_i))^2 = \operatorname*{argmin}_{W_1,w_2} \sum_{i=1}^{n} (y_i - w_2^T\sigma(W_1^T x_i))^2$$

To perform this minimization, gradient descent is conducted on the loss with respect to the parameters W_1, w_2.

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

$$\mathcal{L}(W_1, w_2) = (y - f_{W_1,w_2}(x))^2 = (y - w_2^T\sigma(W_1^T x))^2 \tag{1}$$

Backpropagation leverages the chain rule, along with dynamic programming, to compute the required partial derivatives $\frac{\partial \mathcal{L}(W_1,w_2)}{\partial W_1}$ and $\frac{\partial \mathcal{L}(W_1,w_2)}{\partial w_2}$ 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 $\mathcal{L}(W_1, w_2)$ by computing the quantities $z_1 = W_1^T x$, $z_2 = \sigma(z_1)$, $z_3 = w_2^T z_2$, the error $z_4 = y - z_3$, then finally the loss $\mathcal{L}(W_1, w_2) = z_4^2$. Backpropagation then performs a "backward pass" to compute the partial derivatives, starting with $\frac{\partial \mathcal{L}(W_1,w_2)}{\partial w_2}$.

(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 w_2*

Using the chain rule, write down an expression for $\frac{\partial \mathcal{L}(W_1, w_2)}{\partial w_2}$. 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 w_2 is an h-dimensional vector, so the partial derivative will be an h-dimensional vector. The expression $w_2^T \sigma(W_1^T x) = w_2^T z_2$ is a dot product between the vector w_2 and the vector z_2. Recall that for a dot product between two vectors $x^T y$, we have $\frac{\partial x^T y}{\partial x} = y$.

(c) *Updating W_1*

Using the chain rule, write down an expression for $\frac{\partial \mathcal{L}(W_1, w_2)}{\partial W_1}$. Once again, use the intermediate quantities from the forward pass wherever possible.

Hint: W_1 is a $m \times h$-dimensional matrix, so the partial derivative will be an $m \times h$-dimensional matrix. You can approach this problem by noting that

$$\frac{\partial \mathcal{L}(W_1, w_2)}{\partial W_1} = 2(y - w_2^T \sigma(W_1^T x)) \cdot - \frac{\partial w_2^T \sigma(W_1^T x)}{\partial W_1}$$

and finding the partial derivative of $w_2^T \sigma(W_1^T x)$ with respect to each element $[W_1]_{ij}$ of W_1. Use this result to write the partial derivative of W_1 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 W_1 and w_2 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)?

3. **Activation Functions for Neural Networks**

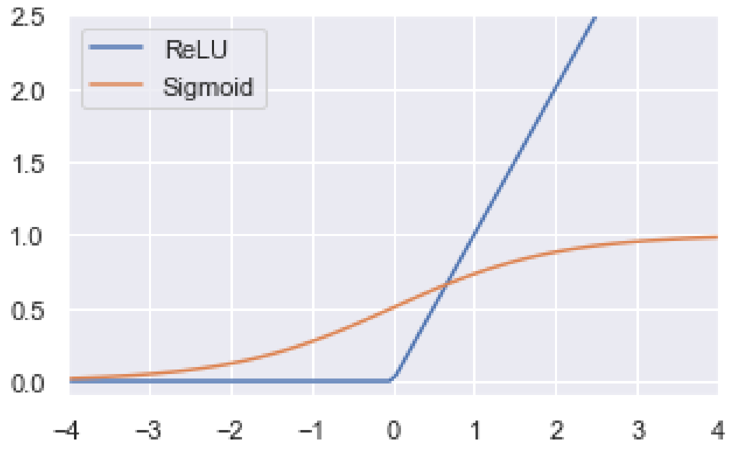
We saw in lecture that neural networks can learn complex, non-linear patterns from data using activation functions. One simple activation function which you've seen in this class before is the *sigmoid* function used in Logistic Regression, defined as:

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

However, most modern-day implementations of neural networks avoid using sigmoid functions for the nonlinearity, and instead favor functions like the *Rectified 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) *Vanishing Gradients*

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