A Lurid Account X crossed Smigh: Calculating textbook softmax derivatives

Jesse Waite

February 2020

1 Introduction

The softmax function is a vector-in, vector-out function. This makes its derivatives a bit messy, but there are a number of tricks for simplifying its expression and various gradient calculations.

2 The Likelihood Function Loss and Gradient Derivation (easy version)

For the multiclass prediction problem of k classes and n examples, the likelihood function is defined below. The exponent $y_{n,k}$ within the inner $\prod$ is defined as 1.0 for the correct class/label, and zero otherwise. All the incorrect labels for an example thus go to 1.0, and only the predicted probability of the correct label contributes to the overall likelihood. The likelihood is a multi-class generalization of the Bernoulli likelihood function of a set of experiments resulting in success/failure.

$$L(W, X) = \prod_n \prod_k \left(\frac{e^{w_k \cdot x_n}}{\sum_j^k e^{w_j \cdot x_n}} \right)^{y_{n,k}}$$

We wish to maximize the likelihood function, but the math works out better if we try to *minimize* the negative log-likelihood instead. This is valid since $log()$ is a monotonically increasing function. We'll use $J(W, X)$ for the negative log-likelihood.

$$J(W, X) = -1 * log(L(W, X))$$

$$J(W, X) = -\sum_n \sum_k y_{n,k} \Big(log(e^{w_k \cdot x_n}) - log\big(\sum_j^k e^{w_j \cdot x_n} \big) \Big)$$

$$J(W,X) = -\sum_n \sum_k y_{n,k}\Big(w_k \cdot x_n - log\big(\sum_j^k e^{w_j \cdot x_n}\big)\Big)$$

$$\boxed{J(W,X) = -\sum_n \sum_k y_{n,k}\big(w_k \cdot x_n\big) + \sum_n \sum_k y_{n,k} log\big(\sum_j^k e^{w_j \cdot x_n}\big)}$$

Notice that in the right-hand sum (over n and k) that $\sum_k y_{n,k}$ is 1.0, since it is a one-hot encoded vector. Likewise, nothing inside the $log()$ depends on the index k. Based on this, the log-likelihood simplifies to:

$$\boxed{J(W,X) = -\sum_n \sum_k y_{n,k}\big(w_k \cdot x_n\big) + \sum_n log\big(\sum_j^k e^{w_j \cdot x_n}\big)}$$

3 Gradient calculation $\frac{\delta J}{\delta W}$

The gradient we desire is $\frac{\delta J}{\delta W}$, the derivative of $J(W,X)$ with respect to the weights W. In these expressions, w and x are vectors, whereas y and s are scalars. To start, take the derivative w.r.t. w_j, the weight vector for class j.

$$\frac{\delta J}{\delta w_j} = -\sum_n y_{n,j} x_n + \sum_n \frac{1}{\sum_i^k e^{w_i \cdot x_n}} * e^{w_j \cdot x_n} * x_n$$

$$\frac{\delta J}{\delta w_j} = -\sum_n y_{n,j} x_n + \sum_n x_n \frac{e^{w_j \cdot x_n}}{\sum_i^k e^{w_i \cdot x_n}}$$

$$\frac{\delta J}{\delta w_j} = -\sum_n y_{n,j} x_n + \sum_n \frac{e^{w_j \cdot x_n}}{\sum_i^k e^{w_i \cdot x_n}} x_n$$

$$\frac{\delta J}{\delta w_j} = \sum_n x_n \big(\frac{e^{w_j \cdot x_n}}{\sum_i^k e^{w_i \cdot x_n}} - y_{n,j}\big)$$

$$\frac{\delta J}{\delta w_j} = \sum_n x_n \big(s_j - y_{n,j}\big)$$

Where s_j (a scalar) is the softmax function $p(y_j|x)$ for class j, and $y_{n,j}$ (a scalar) is 1 if $k == j$, and 0 otherwise. $\frac{\delta J}{\delta w_j}$ is for a single class' weight vector w_j. Admittedly, this is poor notation, since I've mixed scalars with vectors. In cleaner vector form, where x, s, y are all vectors, the loss for the single i^{th} training example is:

$$\frac{\delta J_i}{\delta W} = x_i \otimes \big(s_i - y_i\big)$$

Read it mnemonically as "x crossed smigh" for 'x crossed s minus y', and you'll still remember it by lunch. X crossed smigh–that treacherous . Y took from s, then x crossed them both. X crossed smigh. Got it? IMHO the cheesier the mnemonic, the easier to remember.

The gradient of the loss function with respect to the full weight matrix $\frac{\delta J}{\delta W}$ is then:

$$\boxed{\frac{\delta J}{\delta W} = \sum_n x_n \otimes \big(s - y^*\big)}$$

Where s is our vector of output probabilities and y^* is a one-hot encoding of true class labels. One epoch of gradient descent is given by subtracting this value (to proceed in the direction opposite of the gradient) multiplied by our learning rate α:

$$W = W - \alpha \tfrac{\delta J}{\delta W}$$

4 Useful and Some Less Useful Tedium

A concrete example does a better job of explaining the mechanics of the gradient calculation. Let $x = [2, 2, 2]$ and $s = [0.1, 0.7, 0.2]$ and the true one-hot encoding $y = [0, 0, 1]$. For zero-based indexing the correct label is $j = 2$, but our model is over-estimating $j = 1$: $s_1 = 0.7$.

$$x \otimes \big(s - y\big) = [2, 2, 2] \otimes ([0.1, 0.7, 0.3] - [0, 0, 1])$$

$$x \otimes \big(s - y\big) = [2, 2, 2] \begin{bmatrix} (0.1 - 0) \\ (0.7 - 0) \\ (0.3 - 1) \end{bmatrix}$$

$$x \otimes \big(s - y\big) = [2, 2, 2] \begin{bmatrix} 0.1 \\ 0.7 \\ -0.7 \end{bmatrix}$$

$$x \otimes \big(s - y\big) = \begin{bmatrix} [2, 2, 2] \odot 0.1 \\ [2, 2, 2] \odot 0.7 \\ [2, 2, 2] \odot -0.7 \end{bmatrix}$$

$$x \otimes \big(s - y\big) = \begin{bmatrix} 0.2 & 0.2 & 0.2 \\ 1.4 & 1.4 & 1.4 \\ -1.4 & -1.4 & -1.4 \end{bmatrix}$$

Note how the input (all 2's) is scaled by the magnitudes of the predictions: incorrect predictions are large and positive, and correct ones are large and negative.

At this point you read too quickly, as I would, and some tenured author handwaves breezily about how simple or obvious the math is, belying the hours they spent studying and preparing them between lackadaisacal trips to the university coffee shop. Needless to say, this is not our sort *friendo*, and while compactness is practical to oneself, it is usually confounding to an audience grasping it for the first time.

So for a quick review, the outer product of two vectors $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ is a matrix of dimension $n \times m$ whose rows are the vector-scalar products of a and each (scalar) element b_i of b:

$$C_{n\times m} = \begin{bmatrix} a \odot b_1 \\ a \odot b_2 \\ \vdots \\ a \odot b_m \end{bmatrix}$$

Therefore, the outer product in $\frac{\delta J}{\delta W}$ is:

$$C_{n\times n} = \begin{bmatrix} x \odot (s_1 - y_1) \\ x \odot (s_2 - y_2) \\ \vdots \\ x \odot (s_n - y_n) \end{bmatrix}$$

The point of belaboring this expression is that we can observe important properties by being this explicit. Only one of y_i is 1, and all the rest are 0. So for the incorrect labels, the gradient of the loss with respect to that weight vector is the input multiplied by the prediction for that (incorrect) class, $x_n \odot s_j$. Therefore, the larger our prediction for an incorrect label, the larger and more positive the weight update for that example. However for the correct class, we get the opposite: $x_n \odot (s_j - 1.0)$, meaning that if our prediction for the correct label is small, then the update for this example is larger and more negative. So the closer any of our predictions are to their target label values of either 0.0 or 1.0, the smaller the magnitude of the update. The positivity/negativity relationship is reversed in gradient descent since we proceed in the *opposite* direction of these updates. It might actually be clearer to include that negation above to make the weight-update behavior clearer, but if you've read this far, you grasp it anyway.

Outlier effects: recall that the softmax function tends to exaggerate the probability of the max value of some vector. This is desirable discrimination behavior for any classifier, however, note that the weight corrections will be exaggerated via these scalar probabilities s because they result from multiplication with them. I don't know what this means in depth, but it means that incorrect predictions will have very large contribution to the gradient. Knowing this, you might even exploit input data clustering in initial stages of gradient descent to mitigate outliers, but its just hacking. The point is just that the value of each example to gradient is highly uneven, and you could design training methods

to exploit this behavior. It also helps inform regularization strategies, most notably because if the data are linearly separable, then the gradient updates will proceed forever, and the magnitude of the weights will grow without bound!