Loss Functions and Their Gradients

Compared to activation functions, loss functions have a wildly different purpose. Loss functions are how the model measure's an objective, and they are the means by which a model determines its performance. Essentially, loss functions are the object that the model will try and optimize. From a human perspective, loss functions guide the shape of the decision surface that the model will try and learn, and different loss functions have different geometrical properties. From the model's perspective, the loss function is the way to prune hypothesis from the hypothesis space into the version space, and are the only way the model measures itself with respect to the ground truth.

Loss functions, like activation functions, require both a function reference and a reference to a gradient function. Just like activation functions, we will implement them as functors, both having __call__ and gradient methods. The loss function is the end of information flow in a supervised learning model, where information feeds through the net, is collected as predictions, and then measured against the ground truth using the loss function, where error is fed back into the network and used to update model parameters.

Lets talk about some of the most popular loss functions available, although please create your own! The good news is that all loss functions here are element-wise independent, so we don't need to worry about computing an entire Jacobian matrix since it is a diagonal matrix.

Mean Squared Error (MSE)

MSE is perhaps the simplest loss function on this list. What MSE does is it takes two objects of the same shape (vectors, scalars, matrices, etc.), the predictions $\mathbf{Y}_{hat}$ and the ground truth $\mathbf{Y}$ where $\mathbf{Y}_{hat}$ is an estimator of $\mathbf{Y}$ both containing k examples. MSE computes:

$$MSE(\mathbf{Y}_{hat}, \mathbf{Y}) = \frac{1}{2k}\sum_{i=1}^{k}(\mathbf{Y}_{hat_i} - \mathbf{Y}_i)^2$$

This is the average of the squared differences between the estimated values and the ground truth multipled by $\frac{1}{2}$. This function measures the expected loss (i.e. the loss of the expected value of the squared error). MSE is always positive, and the closer to zero, the better. MSE comprises both the variance and bias of the model, and if the model is unbiased, measures an unbiased loss.

If we have k examples with m dimensions (categorical outputs) per output (so our output matrices have shape $k \times m$, then we can view examples in both the ground truth and the predictions as points in this m dimensional space. The decision surface, also known as the decision function are regions of space that correspond to different categorical values of out outputs. We cannot observe the "true" surface completely as we only have a subset of points from this space, but we can make our best guess from the current model parameters, and measure success from the loss function. MSE draws a k dimensional sphere around each point in the ground truth that has a radius measured from the corresponding prediction point. MSE then reports half of the measured average of squared radii of these spheres, and the model seeks to minimize these radii.

MSE does not measure directionality of the radii, for instance all prediction points that are on the sphere with radius r from a ground truth point are considered equally bad. Therefore, MSE in incapable of differentiating between misclassification priorities, i.e. MSE cannot comprehend that it is more costly to misclassify a point as class A versus class B if class A and class B both share a boundary with the true class C.

So why the extra $\frac{1}{2}$? This is just a preference, but it makes the gradient nicer to compute. Lets go ahead and compute the gradient of MSE by computing the partial derivative of a single term (along the diagonal since it is element-wise independent):

$$\begin{aligned}\frac{\partial MSE_i(\mathbf{Y}_hat, \mathbf{Y})}{\partial \mathbf{Y}_{hat_{ij}}} &= \frac{\partial \frac{1}{2k}\sum_{i=1}^{k}(\mathbf{Y}_{hat_i} - \mathbf{Y}_i)^2}{\partial \mathbf{Y}_{hat_{ij}}} \\ &= \frac{1}{2k}\sum_{i=1}^{k}\frac{\partial(\mathbf{Y}_{hat_i} - \mathbf{Y}_i)^2}{\partial \mathbf{Y}_{hat_{ij}}} \\ &= \frac{2(\mathbf{Y}_{hat_{ij}} - \mathbf{Y}_{ij})}{2k} \\ &= \frac{\mathbf{Y}_{hat_{ij}} - \mathbf{Y}_{ij}}{k}\end{aligned}$$

When differentiating, the $\frac{1}{2}$ cancels out when we bring down the 2 from the exponent of the differences between the two quantities. This means that each element of the gradient becomes the difference of the predicted and expected value divided by the number of examples. This makes implementation easy, we just have to force everything to be 2D:

```
class MSE(object):
    def __init__(self):
        pass

    def __call__(self, Y_hat, Y):
        assert(Y_hat.shape == Y.shape)
        Y_hat_ = np.atleast_2d(Y_hat)
        Y_ = np.atleast_2d(Y)

        Y_hat_ = Y_hat_.reshape(-1, Y_hat_.shape[-1])
        Y_ = Y_.reshape(-1, Y_.shape[-1])

        return 0.5*np.sum((Y_hat_-Y_)**2)/Y_hat_.shape[0]

    def gradient(self, Y_hat, Y):
        assert(Y_hat.shape == Y.shape)
        return (Y_hat-Y)/np.prod(Y_hat.shape[:-1])
```

Cross Entropy

Cross entropy is a loss function that measures the difference between two probability distributions. Note, they $\mathbf{must}$ be probability distributions that are defined over the same domain for this loss to make sense. We will talk about this more after we see the definition, which we will define for k discrete probability distributions that have m possible values:

$$f(\mathbf{Y}_{hat}, \mathbf{Y}) = -\sum_{i=1}^{k}\sum_{j=1}^{m}\mathbf{Y}_{ij}log(\mathbf{Y}_{hat_{ij}})$$

Binary Cross Entropy

Previous Forward