Apply Loss Functions

Neural networks are versatile models that can be used to do both regression and classification. The difference between those two types of networks will be the final transition function and the output vector.

Recall that a neural network with $l-1$ hidden layers has the following generic form:

☆ **Key Points**

Neural networks can be used for regression and classification.

The output of a regression network is a real number or vector of real numbers.

The output of the neural network for classification is a normalized probability vector.

$$\mathbf{h}(\mathbf{x}) = \mathbf{W}_l \sigma\left(\mathbf{W}_{l-1} \sigma\left(\cdots \mathbf{W}_2 \sigma\left(\mathbf{W}_1 \mathbf{x}\right) \cdots\right)\right)$$

Regression

If we want to do a one dimensional regression (i.e., our label y is a real scalar), then our neural network must have a one dimensional output. This can be trivially done by setting the dimension of $\mathbf{W}_l$ to $1 \times d_{l-1}$, where d_{l-1} is the dimension of the output of the last hidden layer. With the size of the output set, all we need to do now is decide a loss function. Because we are performing regression, we will use the more popular loss function, squared loss:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{x}_i, y_i) = \frac{1}{n} \sum_{i=1}^{n} \left(\mathbf{h}(\mathbf{x}_i) - y_i\right)^2$$

With this loss function, we can calculate the gradient with respect to all weights $\mathbf{W}_l, \mathbf{W}_{l-1}, \ldots, \mathbf{W}_1$ and do stochastic gradient descent to learn those weights. Of course nothing is stopping us from regressing higher dimensional labels.

Classification

Classification problems output belief vectors (also known as logits). Every element in the belief vector corresponds to the model's belief that the input example belongs to a certain class. For example, imagine we have a k-class classification problem with labels $\{0, 1, \ldots, k-1\}$. Our model will output a k dimensional belief vector. If the j^{th} element has the highest value, then the model believes that that input belongs to class j.

To output a k dimensional belief vector, we set the dimension of $\mathbf{W}_l$ to $k \times d_{l-1}$, where d_{l-1} is the dimension of the output of the last hidden layer. However, we are not done yet. The output of a neural network is a real vector that is not easy to work with, as each element could be any value on different scales. As such, we usually apply the **softmax** transition function. Often, the last layer is thus called the softmax layer.

The softmax layer can be written as follows:

$$[\mathbf{softmax}\left(\mathbf{h}(x)\right)]_i = \frac{\exp\{[h(x)]_i\}}{\sum_{j=1}^{k}\exp\{[h(x)]_j\}}$$

All the softmax is doing is that it first exponentiates each dimension $z \rightarrow \mathbf{exp}(z)$ and then normalizes these values across classes so that the output values sum to 1. The exponentiation has two effects: 1. all values become positive; 2. whatever output is largest will be highly dominant. The normalization then turns all outputs into well-defined probabilities.

Note: Convince yourself that the sum of all entries of the softmax output is 1 and each entry is in the range of 0 to 1.

The softmax layer essentially produces a vector of probabilities where the j-th dimension represents the probabilities of $\mathbf{x}$ belonging to class j, i.e., $[\mathbf{softmax}(\mathbf{h}(\mathbf{x}))]_j = P(y = j|\mathbf{x})$.

With this probabilistic representation, we can use our MLE principle — minimizing the negative log likelihood — to get the following loss function for classification:

$$\mathbf{max}\ \frac{1}{n}\prod_{i=1}^{n} P\left(y = y_i|\mathbf{x}_i\right) = \mathbf{min}\ \frac{1}{n}\sum_{i=1}^{n} -\log P\left(y = y_i|\mathbf{x}_i\right) = \mathbf{min}\ \frac{1}{n}\sum_{i=1}^{n} -\log\left[\mathbf{softmax}(\mathbf{h}\right.$$