Notes about Neural Networks

Justin Johnson

September 1, 2014

1 Backpropogation

A neural network is a function $f : \mathbb{R}^{n_0} \to \mathbb{R}^{n_k}$ which can be written as a series of function compositions:

$$f = f_k \circ f_{k-1} \circ \cdots \circ f_1$$

where each function $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ is a **layer**. Each layer f_i may also be a function of some **parameters** $w_i \in \mathbb{R}^{d_i}$. If we treat these parameters as explicit inputs to the functions f_i, then each f_i is viewed as a function $\mathbb{R}^{d_i} \times \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$. If we group the parameters for all layers into a single variable w, then the entire network computes its output as:

$$f(w, x) = f_k(w_k, f_{k-1}(w_{k-1}, \cdots f_2(w_2, f_1(w_1, x)) \cdots))$$

This computation is known as the **forward** pass of the network. It is traditional to denote the output of the ith layer as $a_i \in \mathbb{R}^{n_i}$; this is the ith **activation** of the network. Given an input x to the network, we can then compute the output y as:

$$a_0 = x \qquad a_{i+1} = f_i(w_i, a_{i-1}) \qquad y = a_k$$

In order to **train** a network, we need a **loss function** $\ell : \mathbb{R}^{n_k} \times \mathbb{R}^{n_k} \to \mathbb{R}$, a set of **training data** $(x_1, y_1), \ldots, (x_m, y_m)$, and a **regularizer** $R : \mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_k}$. Training a network then amounts to solving the optimization problem

$$J(w) = \sum_{j=1}^{m} \ell(y_j, f(w, x_j)) + R(w)$$
$$w^* = \arg\min_w J(w)$$

This is typically a nonconvex optimization problem, and is traditionally solved using stochastic subgradient descent. In order to do this, we need to be able to compute the subgradients $\frac{\partial J}{\partial w_i}$ of the objetive with respect to the parameters of each layer.

2 Softmax

Softmax is the function $f : \mathbb{R}^n \to \mathbb{R}^n$ given by

$$f_i(x) = \frac{e^{x_i}}{\sum_{k=1}^n e^{x_k}}$$

A bit of algebra shows that its derivatives are given by

$$\frac{\partial f_i}{\partial x_j} = \begin{cases} -\dfrac{e^{x_i} e^{x_j}}{\left(\sum_{k=1}^n e^{x_k}\right)^2} & i \neq j \\ \left(\dfrac{e^{x_i}}{\sum_{k=1}^n e^{x_k}}\right)\left(1 - \dfrac{e^{x_i}}{\sum_{k=1}^n e^{x_k}}\right) & i = j \end{cases}$$

Note that this is symmetric. If we let $y = f(x)$ then this can be rewritten in vectorized form as

$$\frac{\partial f}{\partial x} = \text{diag}(y) - yy^T$$

Computing the product $(\frac{\partial f}{\partial x})z$ for a vector $z \in \mathbb{R}^n$ is given by

$$\left(\frac{\partial f}{\partial x}\right) z = y \circ z - yy^T z = y \circ (z - (y^T z)\mathbf{1})$$

where $\circ$ is an elementwise product and $\mathbf{1} \in \mathbb{R}^n$ is a constant vector of ones.

Now suppose that $X \in \mathbb{R}^{n \times m}$ is a matrix of m inputs stored in columns, and $Y \in \mathbb{R}^{n \times m}$ is the matrix of outputs obtained by applying f to each column of X. Given a matrix $dY \in \mathbb{R}^{n \times m}$ of upstream derivatives, in the backprop step we must compute the matrix $dX \in \mathbb{R}^{n \times m}$ whose ith column is equal to $\frac{\partial f}{\partial x}$ evaluted at the ith column of X, multiplied by the ith column of Y. Using the above results, it is clear that

$$dX = Y \circ (dY - \mathbf{1}\mathbf{1}^T (Y \circ dY))$$

This can be efficiently implemented in numpy using broadcasting as:

```
dX = Y * (dY - np.sum(Y * dY, axis=0))
```

3 Cross-Entropy Loss

The cross-entropy between two probability distributions p and q over discrete classes $\{1, 2, \ldots, k\}$ is given by

$$H(q, p) = -\sum_{i=1}^{k} q_i \log(p_i)$$

Note that if p is given by a softmax function and $q_i = 1$ iff $y_i = i$ then picking the softmax weights via maximum likelihood estimation is equivalent to minimizing the sum of the cross-entropy between the predicted class probabilities p and the true class probabilities q over the training set. The derivatives are given by

$$\frac{\partial H}{\partial q_i} = -\log(p_i) \qquad\qquad \frac{\partial H}{\partial p_i} = -\frac{q_i}{p_i}$$