SIMPLE NEURAL NETWORK MATH

JAMES SAXON

1. Defining the Network

Consider an artificial neural network of the form shown in Figure 1. We have D input variables and K classes, and our unique hidden layer has H nodes. We'll call inputs $\boldsymbol{x}$ and outputs $\boldsymbol{f}$, activation functions g, and weights $\boldsymbol{w}$. (We denote both vectors and matrices in bold.) The entire net may then be expressed:

$$f_k = g_k\left(\sum_j^H w_{kj}^{(2)} \times g_j\left(\sum_i^D w_{ji}^{(1)} x_i\right)\right).$$

Note that we are not explicitly writing the bias nodes; instead we're incorporating them into $\boldsymbol{x}$ and $\boldsymbol{h}$ as constants of 1. Their values are thus incorporated into the weights, which are accordingly extended by 1. That means that the weights $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are matrices of size $(D+1) \times H$ and $(H+1) \times K$ respsectively. This condenses the math, at the cost of having to track 'ones.'

For this example we'll take g_k as the identity $(g_k(x) \equiv x)$ and use the rectified linear unit function for g_j:

$$g_j(x) \equiv g(x) = \mathrm{ReLU}(x) = \max(0, x).$$

We seek to minimize the total loss of our network by tuning the weights. To do this, we naturally want $\partial L/\partial w$. We do this through back-propogation, but it perhaps helps first to lay it out with the chain rule. Intuititively, each weight multiplies *an* input and forwards to *all* outputs. The math must reflect this, which will help write down the expressions.

Date: November 23, 2016.

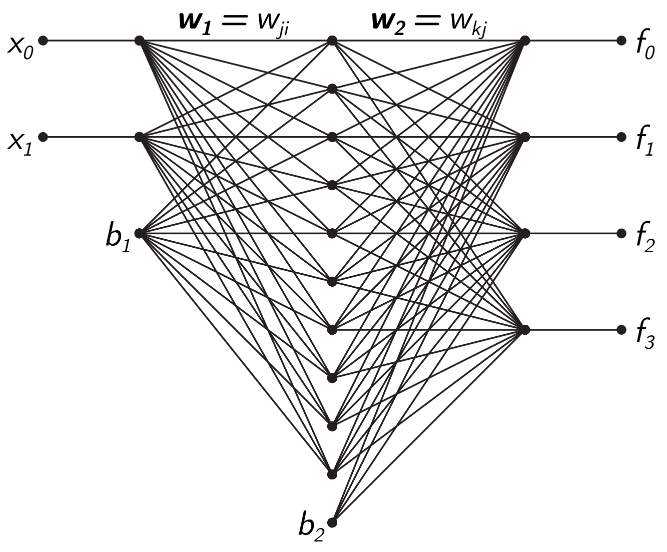

Figure 1. Schematic diagram of a simple neural network.

2. Gradient of the Loss (Steepest Descent)

Calling misclassification loss (data loss) L_d, the number of training elements N, and the regularization loss L_r, we have $L = L_d/N + L_r$. The regularization term is $L_r = \frac{\alpha}{2}\sum w^2$, so its derivative with respect to the w's is trivial: αw. (Apologies for sloppy notation.)

The data loss is $L_d = \sum_i L_i$; per term, we use the softmax definition:

$$L_i = -\log\left(e^{f_{y_i}} \Big/ \sum_j e^{f_j}\right).$$

Let's calculate the gradient with respect to the first and second layer weights. In the first, that's

$$\frac{\partial L_d}{\partial w_1} = \frac{\partial L}{\partial f_j}\frac{\partial f_j}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial w_1}.$$

The terms are pretty easy to understand:

- $\partial L/\partial f_j$ is the loss with respect to (WRT) the scores. This is the only 'tricky' derivative and is calculated below.
- $\partial f_j/dg$ is the scores WRT the activation function. This just turns off the derivative when h is positive; it is the heaviside function $\Theta(h)$.
- $\partial g/dh$ is the activation function with respect to the hidden layer products. That's just the $\boldsymbol{w}_2$ weights!
- $\partial h/\partial w_1$ is the hidden layer products WRT the weights – just the inputs $\boldsymbol{x}$!

We also need the second layer weights; these are easier, just $\partial L/d\boldsymbol{w}_2 = (\partial L/\partial f_j)(\partial f_j/\partial \boldsymbol{w}_2)$. The first term we've already seen and the second is clearly just the hidden layer *values*. Note that the hidden layer values are needed several times, so we should hold on to these!

Let's solve out $\partial L_d/\partial f_j$. Calling the correct classification for an object y_i, this is

$$\frac{\partial L_i}{\partial f_k} = \frac{\partial}{\partial f_k}\left[-\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)\right]$$

$$= \frac{\partial}{\partial f_k}\left[\log\left(\sum e^{f_j}\right) - \log\left(e^{f_j}\right)\right] = \frac{\partial}{\partial f_k}\log\left(\sum e^{f_j}\right) - \mathbb{1}\left(y_i = k\right)$$

The remaining unevaluated term is easy with the chain rule: $(\partial \log/d\mathrm{arg})(\partial \mathrm{arg}/\partial f_k)$, which comes down to $e^{f_k}/\sum_j e^{f_j} = p_k$. The full term is thus

$$\frac{\partial L_i}{\partial f_k} = p_k - \mathbb{1}\left(y_i = k\right).$$

3. The terms for our program.

We're now all done! Calling our hidden layer $\boldsymbol{h}$ and including the regularization we can write:

- $d\boldsymbol{f} \equiv p_k - \mathbb{1}\left(y_i - k\right)$
- $d\boldsymbol{w}_2 \equiv \partial L_i/\partial \boldsymbol{w}_2 = \boldsymbol{h} \cdot d\boldsymbol{f} + \alpha \boldsymbol{w}_2$
- $d\boldsymbol{h} \equiv \partial L_i/dh = d\boldsymbol{f} \cdot \boldsymbol{w}_2$
- $d\boldsymbol{w}_1 \equiv \partial L_i/d\boldsymbol{w}_1 = \boldsymbol{x} \cdot d\boldsymbol{h} + \alpha \boldsymbol{w}_1$

This matches our intuitions about the gradients. The larger the values on the inputs, the more the first layer weights change. The larger the values coming out of the hidden layer, the larger the gradients on the second layer weights. The $d\boldsymbol{f}$ and $d\boldsymbol{h}$ are just intermediate steps; they do not update anything.

Now we just need to code it up!! We will do this entirely with matrix multiplication, because `numpy` is highy optimized for this. With `for` loops, this would just be too slow.

3.1. **Initialize.** Our initialize function will initialize our parameters and hyperparameters:

- The number of classes, K.
- The dimensions of our input variables $D\,(= 2)$.
- The number of hidden layers, H, defaulting to 100.
- Very small initialization for our weights (0.01 times a Guassian).
  - For this, we'll use `np.random.randn(x, y)`.
- The step size and regularization (α).

3.2. **Evaluate.** The evaluate function must perform the dot products and ReLU activation. Additionally, we're going to have to deal with the 'bias trick,' by extending (`np.append()`) the column of 'ones' to our inputs and hidden layer.

3.3. **Classify.** This will simply choose the maximum score, with `np.argmax()`.

3.4. **Train.** This is the big kahuna class; it's just a loop over iterations (say, 5000), over which we will calculate the gradients and update the parameters, as already discussed *ad nauseum.*