CS 520: Neural Networks and Error Propagation

16:198:520

Instructor: Wes Cowan

As seen in class, simple linear models such as linear regression, logistic regression, or hard perceptron classifiers, can be computationally limited in what they can model. Instead of moving to more complex, higher degree models, we can instead consider *stacking* these simple models, treating the outputs of some as the inputs of other, and building up to some final cumulative output. Frequently, in models like these, adequately trained models have the property that lower level models / 'neurons' serve as basic feature detectors or feature processors - for instance detecting horizontal or vertical lines - which then pass these results as inputs to higher level feature detectors, building up in complexity, until the highest level nodes or neurons can detect very complex properties, such as dogs or faces. This is in fact observed as well in actual neural networks such as the optical processing of the retina / optic nerve / visual cortex arrangement in animal brains. But how to actually train these models?

In a sense, a neural network is simply a high dimensional parametric model, in particular one that takes repeated, nested non-linear functions of linear functions. A simple one might look like:

$$f(x) = g(ag(bx + c) + dg(ex + f) + h), \tag{1}$$

where the network has one input node (taking in the value of x), two hidden nodes (which compute $g(bx + c)$ and $g(ex + f)$ respectively), and a single output node which computes g of a linear function of its two inputs (from the hidden layer). Training this model, or learning from a data set, is simply a matter of taking x values and their corresponding y values, and trying to determine the appropriate values of (a, b, c, d, e, f, h) to try to minimize the total loss of the model.

If f depends on a vector of parameters $\underline{w}$, we have the following training scheme based on stochastic gradient descent (SGD) to update the value of all the parameters:

0) Initialize $\underline{w}(0)$ to be a reasonable (potentially randomized) starting point. Select a reasonable stepsize $\alpha > 0$.
1) At time t, select a point $(\underline{x}, \underline{y})$ from your data set.
2) Update the parameters according to

$$\underline{w}(t + 1) = \underline{w}(t) - \alpha \nabla_{\underline{w}} \text{Loss}(\underline{x}). \tag{2}$$

3) Repeat until termination.

This was the basis for linear regression (squared error loss) and logistic regression, and had strong parallels to the perceptron learning rule. Computing these gradients can be difficult for arbitrary models and arbitrary loss functions - but in the case of the stacked models we're dealing with here, the gradient has a simple structure that lends itself to a very natural algorithm known as error back-propagation.

We typically arrange a network of nodes or 'neurons' in layers: the input layer takes the input data $\underline{x}$, one node for each component of $\underline{x}$; hidden neurons takes as input a weighted combination of the neurons below them, compute a non-linear function of this input, and pass it up to the next layer as output; the last layer of neurons, the output neurons, simply report their output as the output of the whole model. In general, for node i, we have the following,

$$\begin{aligned} \text{in}_i &= \sum_{\text{prev}j} w_{j,i} \text{out}_j \\ \text{out}_i &= g(\text{in}_i). \end{aligned} \tag{3}$$

The output of the whole model, for a given set of weights, is then $f_{\underline{w}}(\underline{x})$, the vector of the final values of the output nodes. We can then define the square loss function as

$$\text{Loss}(\underline{x}, \underline{y}) = ||f_{\underline{w}}(\underline{x}) - \underline{y}||^2 = \sum_{\text{output}k} (\text{out}_k - y_k)^2 . \tag{4}$$

It is useful to present the algorithm and then justify mathematically rather than derive it from scratch. Hence the following,

Stochastic Gradient Descent and Error Back Propagation for Neural Networks:

0) Initialize $\underline{w}(0)$ to be a reasonable (potentially randomized) starting point. Select a reasonable stepsize $\alpha > 0$.

1) At time t, select a point $(\underline{x}, \underline{y})$ from your data set.

2) Update the weights, layer by layer, in the following way:

   2.a) For each output node k, compute the modified error,

   $$\Delta_k = (\text{out}_k - y_k)\, g'(\text{in}_k), \tag{5}$$

   and update weights according to, for each prior node j,

   $$w_{j,k}(t+1) = w_{j,k}(t) - \alpha \Delta_k \text{out}_j. \tag{6}$$

   2.b) For each hidden node j, compute the modified error,

   $$\Delta_j = \left(\sum_{\text{next}k} \Delta_k w_{j,k}(t) \right) g'(\text{in}_j)., \tag{7}$$

   and update weights according to, for each prior node i,

   $$w_{i,j}(t+1) = w_{i,j}(t) - \alpha \Delta_j \text{out}_i. \tag{8}$$

3) Repeat until termination.

This notion of modified error appears to stem from nothing, but in fact can be seen in the following way: the output nodes have error based on how much their output differs from what their output ought to be, scaled by the sensitivity to changes in the input. For any hidden node, since there is no notion of what the error for that node 'ought' to be, we base the error of that node based on the error of the nodes it contributes to - for every 'next' node k, we consider the error on that k, weighted by how much j contributes to that node. And again, the whole thing is scaled by this notion of sensitivity.

But where does this come from? This is in fact stochastic gradient descent applied to this layered model and the squared loss function above. Observe that for any output node k and hidden node j, we have

$$\begin{aligned}
\frac{\partial \text{Loss}}{\partial w_{j,k}} &= \sum_{\text{output}k'} \frac{\partial}{\partial w_{j,k}} \left(\text{out}_{k'} - y_{k'}\right)^2 \\
&= \frac{\partial}{\partial w_{j,k}} \left(\text{out}_k - y_k\right)^2 \\
&= 2\left(\text{out}_k - y_k\right) \frac{\partial}{\partial w_{j,k}} \text{out}_k \\
&= 2\left(\text{out}_k - y_k\right) \frac{\partial}{\partial w_{j,k}} g(\text{in}_k) \\
&= 2\left(\text{out}_k - y_k\right) g'(\text{in}_k) \frac{\partial}{\partial w_{j,k}} \text{in}_k \\
&= 2\left(\text{out}_k - y_k\right) g'(\text{in}_k) \frac{\partial}{\partial w_{j,k}} \left[\sum_{\text{prev } j'} \text{out}_{j'} w_{j',k}\right] \\
&= 2\left(\text{out}_k - y_k\right) g'(\text{in}_k) \text{out}_j \\
&= 2\Delta_k \text{out}_j
\end{aligned} \tag{9}$$

Using this form of the partial derivative, we can recover the output layer update equations as given in the above algorithm.

But what about the hidden layers? Consider any hidden node j, directly below the output layer, and the possible signal coming into it from some node i. By the chain rule

$$\begin{aligned}
\frac{\partial \text{Loss}}{\partial w_{i,j}} &= \sum_{\text{out } k} \frac{\partial \text{Loss}_k}{\partial \text{out}_k} \frac{\partial \text{out}_k}{\partial \text{out}_j} \frac{\partial \text{out}_j}{\partial w_{i,j}} \\
&= \sum_{\text{out } k} 2(\text{out}_k - y_k) g'(\text{in}_k) \frac{\partial \text{in}_k}{\partial \text{out}_j} \frac{\partial \text{out}_j}{\partial w_{i,j}} \\
&= \sum_{\text{out } k} 2(\text{out}_k - y_k) g'(\text{in}_k) w_{j,k} \frac{\partial \text{out}_j}{\partial w_{i,j}} \\
&= 2\left[\sum_{\text{out } k} \Delta_k w_{j,k}\right] g'(\text{in}_j) \frac{\partial \text{in}_j}{\partial w_{i,j}} \\
&= 2\left[\sum_{\text{out } k} \Delta_k w_{j,k}\right] g'(\text{in}_j) \text{out}_i \\
&= 2\Delta_j \text{out}_i
\end{aligned} \tag{10}$$

Hence we recover the same update equations via gradient descent for the layer beneath the output layer as well. This can be continued inductively, generating the update equations for the nodes at every level.