Will Monroe
CS 109

Lecture Notes #23
August 16, 2017

Neural Networks

Based on a handout by Chris Piech

Neural networks (or, as they are fashionably called now, *deep learning*) are, on a practical level, a rather simple extension of logistic regression. This simple idea has yielded powerful results, however. Neural networks have shown success at voice recognition, computer vision (e.g., Facebook's ability to recognize a photo of you), and natural language processing, and they have been the power behind splashy AI successes like Google's AlphaGo and Deep Dream. You are about to learn math that has had a big impact on everyday life and will likely continue to revolutionize many fields in the future.

Let's start with intuition gained from a simple analogy. You can think of a logistic regression function, $\sigma(\theta^T \mathbf{x})$, as a cartoon model of a single neuron inside your brain. A neural network (specifically, a *feedforward* neural network) is the result of putting many layers of logistic regression functions together.

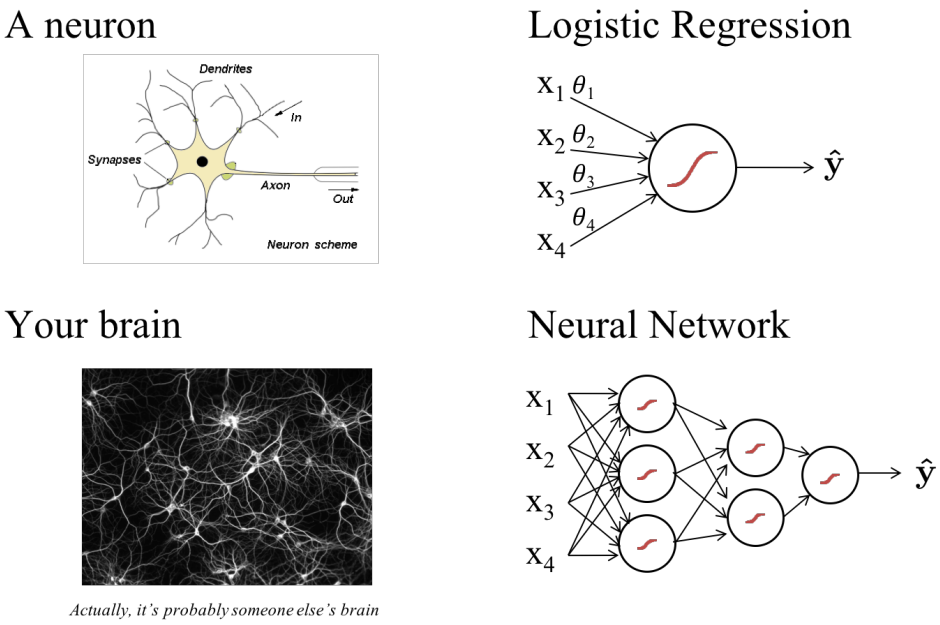

Figure 1: Logistic regression is a cartoon model of a single neuron. Neural networks are a cartoon of a brain.

This simple idea allows for models which can represent complex functions from input features ($\mathbf{x}$) to outputs ($\hat{y}$). In CS 109 we are going to interpret the output of a neural network as the *probability* that the (binary) class label is 1. Note that this notation is slightly different from the one we used in logistic regression; $\hat{y}$ is now a probability, rather than a discrete 0/1 value.

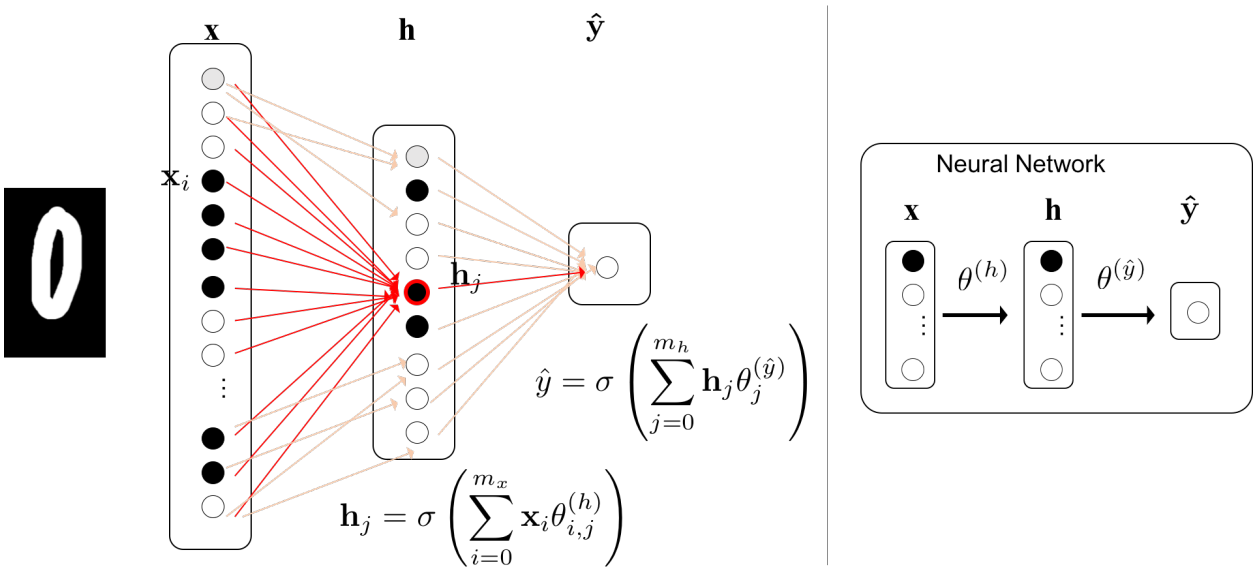

Figure 2: Two diagrams of the same neural network.

Simple Neural Network

As a motivating example, we are going to build a simple neural network that can learn to classify hand-written digits as either the number "0" or the number "1". Above is a diagram of a neural network that we will use. It has three "layers" of neurons. The input layer ($\mathbf{x}$) is a vector of pixel values (0 for dark, 1 for light) in the hand-drawn number. The hidden layer ($\mathbf{h}$) is a vector of logistic regression cells which each take all the elements of $\mathbf{x}$ as input. The output layer is a single logistic regression cell that takes all of the elements of the *hidden layer* $\mathbf{h}$ as input. In the figure on the left a single hidden neuron is highlighted. It's only possible to draw so many arrows from $\mathbf{x}$ to $\mathbf{h}$ without it becoming too messy, so imagine that every hidden neuron has arrows from every $\mathbf{x}$. Formally:

$$\hat{y} = \sigma\left(\sum_{j=0}^{m_h} h_j \theta_j^{(\hat{y})}\right) = P(Y = 1|\mathbf{x}) \tag{1}$$

$$h_j = \sigma\left(\sum_{i=0}^{m_x} \mathbf{x}_i \theta_{i,j}^{(h)}\right) \tag{2}$$

These equations introduce a few new pieces of notation. Let's spell out what each term means. The parameters of the equations are all of the symbols θ. There are two groups of parameters: the weights for the logistic cells in the hidden unit ($\theta^{(h)}$) and weights for the output logistic cell ($\theta^{(\hat{y})}$). These are collections of parameters. There is a value $\theta_{i,j}^{(h)}$ for every pair of input i and hidden unit j and there is a $\theta_j^{(\hat{y})}$ for every hidden unit j. There are $m_x = |\mathbf{x}|$ number of inputs and there are $m_h = |\mathbf{h}|$ number of hidden units. Familiarize yourself with the notation. The math of neural networks is not as hard as the notation!

For a given image (and its corresponding $\mathbf{x}$) the neural network will produce a single value $\hat{y}$. Because it is the result of a sigmoid function, it will have a value in the range $[0, 1]$. We are going

to interpret this value as the probability that the hand-written digit is the number "1". This is the same classification assumption made by logistic regression.

Once you understand the notation and how a value $\hat{y}$ is computed given θ and $\mathbf{x}$ (called the **forward pass**), you are most of the way there. The only step left is to choose the values of θ that maximize the likelihood of our training data. Recall that the process for MLE is to (1) write the log-likelihood function and then (2) find the values of θ that maximize the log-likelihood. Just like in logistic regression, we are going to use gradient ascent to choose our θ's. This means we need the partial derivative of the log likelihood with respect to each parameter.

Log Likelihood

We start with the same assumption as logistic regression. For one data point with true output y and predicted output $\hat{y}$, the likelihood of that data is:

$$P(Y = y|X = \mathbf{x}) = (\hat{y})^y(1 - \hat{y})^{1-y}$$

If you treat $\mathbf{h}$ as the input, you get the logistic regression assumption:

$$P(Y = y|X = \mathbf{x}) = \sigma(\theta^T\mathbf{h})^y \cdot \left[1 - \sigma(\theta^T\mathbf{h})\right]^{(1-y)}$$

If we extend this idea to write the likelihood of n independent datapoints $(\mathbf{x}^{(i)}, \hat{y}^{(i)})$we get:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{n} P(Y = y^{(i)}|X = \mathbf{x}^{(i)}) \\ &= \prod_{i=1}^{n} \sigma(\theta^T\mathbf{h}^{(i)})^{y^{(i)}} \cdot \left[1 - \sigma(\theta^T\mathbf{h}^{(i)})\right]^{(1-y^{(i)})} \end{aligned}$$

Taking the log of this gives us the following log likelihood function for the neural network:

$$LL(\theta) = \sum_{i=0}^{n} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log[1 - \hat{y}^{(i)}] \tag{3}$$

Though this doesn't look like it is an equation in terms of θ, it is. Plugging in the definition for $\hat{y}$ would expose the θ's. However, we will leave this expression alone for now, because as we saw with logistic regression, keeping around temporary variables makes taking derivatives easier.

Backpropagation

We are going to choose values of θ using our old friend MLE (maximum likelihood estimation). MLE applied to deep networks gets the special name **backpropagation**. To choose the optimal values of theta we are going to use gradient ascent, continually updating our thetas in a way that leads to increases in likelihood. To apply gradient ascent, we will need to know the partial derivatives of log-likelihood with respect to each of the parameters.

Since the log likelihood of all the data is a sum of the log likelihood of each data point, we can calculate the derivative of the log likelihood with respect to a single data instance $(\mathbf{x}, y)$. The

derivative with respect to all the data will simply be the sum of the derivatives with respect to each instance (by derivative of summation).

The one great idea that makes MLE simple for deep networks is that by using the chain rule from calculus we can decompose the calculation of gradients in a deep network. Let's work it out. The values that we need to calculate are the partial derivatives of the log likelihood with respect to each parameter. The chain rule can let us calculate gradients one layer at a time. We can decompose the calculation of the gradient with respect to the output parameters as such:

$$\frac{\partial LL(\theta)}{\partial \theta_j^{(\hat{y})}} = \frac{\partial LL}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \theta_j^{(\hat{y})}} \tag{4}$$

Similarly, we can decompose the calculation of the gradient with respect to the hidden layer parameters as:

$$\frac{\partial LL(\theta)}{\partial \theta_{i,j}^{(h)}} = \frac{\partial LL}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_j} \cdot \frac{\partial h_j}{\partial \theta_{i,j}^{(h)}} \tag{5}$$

Each of those terms is reasonable to calculate. Here are their closed form equations:

$$\frac{\partial LL(\theta)}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})} \qquad \frac{\partial \hat{y}}{\partial \theta_j^{(\hat{y})}} = \hat{y}[1-\hat{y}] \cdot h_j$$

$$\frac{\partial \hat{y}}{\partial h_j} = \hat{y}[1-\hat{y}]\theta_j^{(\hat{y})} \qquad \frac{\partial h_j}{\partial \theta_{i,j}^{(h)}} = h_j[1-h_j]x_j$$

In this simple model, we only have one layer of hidden neurons. If we added more we could keep using the chain rule to calculate derivatives with respect to parameters deeper in the network.

Derivative of LL with respect to the output

The expression for the partial derivative $\frac{\partial LL(\theta)}{\partial \hat{y}}$ is a basic exercise in differentiation:

$$LL = y \log \hat{y} + (1-y) \log[1-\hat{y}]$$
$$\frac{\partial LL(\theta)}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})}$$

Derivative of the output with respect to the output parameters

The partial derivative of $\hat{y}$ with respect to an output parameter $\theta_j^{(\hat{y})}$ uses formulas we previously learned for logistic regression:

$$\hat{y} = \sigma(z) \qquad \text{where } z = \sum_{i=0}^{m_h} h_i \theta_i^{(\hat{y})}$$

$$\frac{\partial \hat{y}}{\partial \theta_j^{(\hat{y})}} = \sigma(z)[1-\sigma(z)]\frac{\partial z}{\partial \theta_j^{(\hat{y})}} \qquad \text{derivative of logistic}$$

$$= \hat{y}[1-\hat{y}] \cdot h_j \qquad \text{because } \hat{y} = \sigma(z)$$

Where to Go from Here

This has been just a taste of one kind of neural network, which can be used for binary classification. As a problem to be solved by deep learning, binary classification is quite boring! To go beyond binary classification, we need more interesting structures:

- **Other log-likelihoods**: softmax for handling more than two classes, least-squares and others for predicting continuous outputs.
- **Convolutions and pooling**: for sequential, 2D, and 3D inputs.
- **Recurrent connections**: for sequential inputs and outputs.

There are also many tricks for making neural networks work well that we don't have time to go into:

- **Initialization**: if you initialize parameters to zero, they will never become different! Random initialization is important, and how you do it can make a big difference.
- **Regularization** (parameter priors, dropout, and normalization): for preventing the neural network from latching onto random noise in the training data.
- **Stochastic gradient ascent** (and other fancy optimization methods): for substantially speeding up training.

If these previews make you excited to explore more, take a look at CS 224N (natural language processing), CS 231N (computer vision), or CS 273B (genomics and biomedicine).

Deep learning is a growing field. There is a lot of room for new innovations. Can we develop structures that do a better job of incorporating prior beliefs? Can we make a neural network that can have a natural conversation with us, or teach us, or write code for us? What kinds of networks would work best for these tasks remains an open question. It is worth knowing the math of deep learning well, because you may one day have to invent the next iteration.