Lisa Yan
CS109

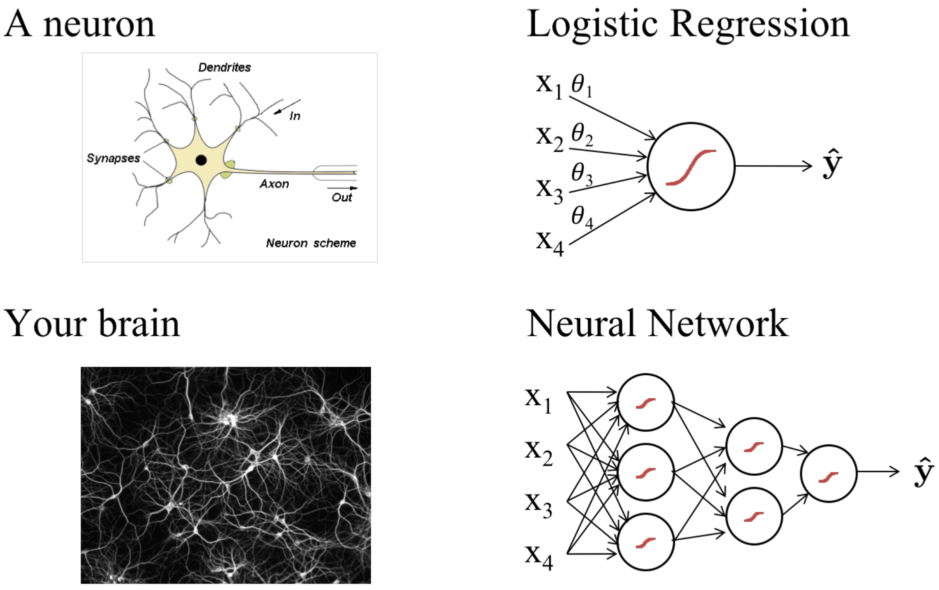
Lecture Notes #26
November 20, 2019

Deep Learning

Based on a chapter by Chris Piech

Deep Learning (the new term to refer to Neural Networks) is one of the greatest ideas in computer science that I have been exposed to. On a practical level they are a rather simple extension of Logistic Regression. But the simple idea has had powerful results. Deep Learning is the core idea behind dramatic improvements in Artificial Intelligence. It is the learning algorithm behind Alpha Go, Voice Recognition, Computer Vision (think facebook's ability to recognize a photo of you), Google's Deep Dream, Educational Knowledge Tracing and modern Natural Language Processing. You are about to learn math that has had a big impact on every day life and will likely continue to revolutionize many disciplines and sub-disciplines.

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. Neural Networks (aka Deep Learning) is the result of putting many layers of Logistic Regression functions together.


A neuron

Logistic Regression

Your brain

Neural Network

Actually, it's probably someone else's brain

Figure 1: Logistic Regression is a cartoon model of a single neuron. Neural Networks model a brain.

This simple idea allows for models which can represent complex functions from input features ($\mathbf{x}$) to outputs ($\hat{y}$). In CS109 we are going to interpret the output of a neural network in the same was as we interpreted the output of logistic regression: as a prediction of the probability of a class label.

Simple Deep Network

As a motivating example we are going to build a simple Deep Network that can learn to classify hand written digits as either the number "1" or the number "2". Here 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 darkness in the hand drawn number. The hidden layer ($\mathbf{h}$) is a vector of logistic regression cells which are 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. We are going to interpret the output value $\hat{y}$ in the same way that we interpreted the output of vanilla logistic regression: as an estimation of $P(Y = 1|\mathbf{x})$. Formally:

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

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

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 isn't particularly difficult. The notation is!

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. Here are two diagrams of the same network with one layer of hidden neurons. In the figure on the left a single hidden neuron is highlighted. Keep in mind that all hidden neurons take all values $\mathbf{x}$ as input. I could only draw so many arrows between $\mathbf{x}$ and $\mathbf{h}$ without it becoming too messy.

Once you understand the notation, and think through how you would compute a value $\hat{y}$ given θ and $\mathbf{x}$ (called the "Forward Pass") you are most of the way there. The only step left is to think through how to chose 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 theta that maximize the log-likelihood. Just like in logistic regression we are going to use gradient ascent to chose our thetas. Thus we simply 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 datapoint with true output y and predicted output $\hat{y} = P(Y = y|\mathbf{X} = \mathbf{x})$, the likelihood of that data is:

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

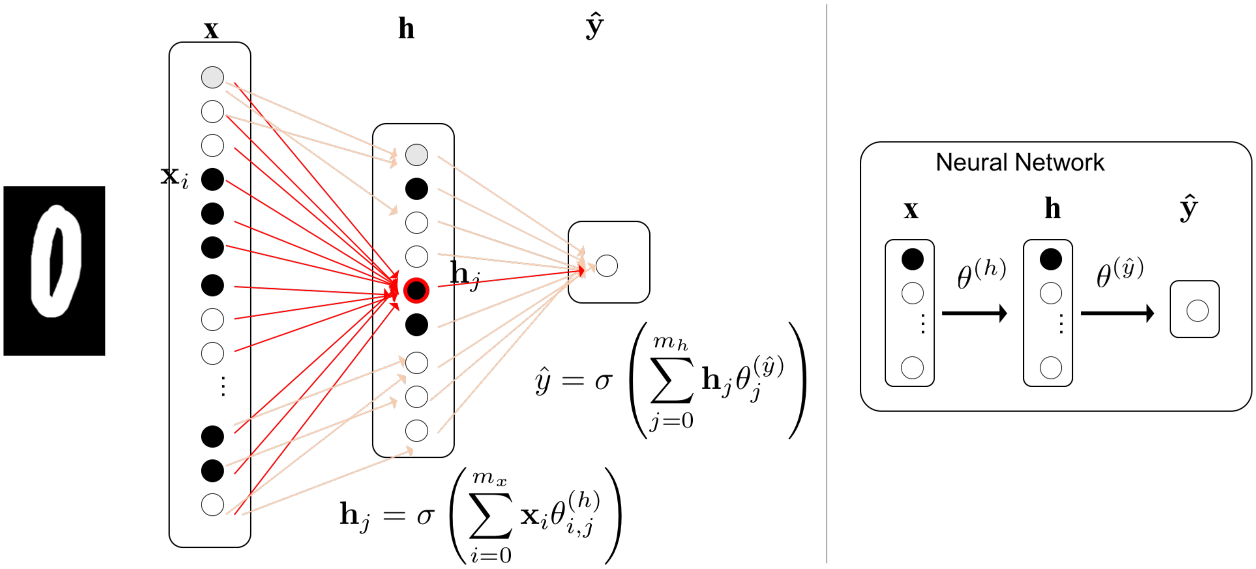

$\mathbf{x}$
$\mathbf{h}$
$\hat{\mathbf{y}}$
$\mathbf{x}_i$
$\mathbf{h}_j$
$\hat{y} = \sigma\left(\sum_{j=0}^{m_h} \mathbf{h}_j \theta_j^{(\hat{y})}\right)$
$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{m_x} \mathbf{x}_i \theta_{i,j}^{(h)}\right)$
Neural Network
$\mathbf{x}$
$\mathbf{h}$
$\hat{\mathbf{y}}$
$\theta^{(h)}$
$\theta^{(\hat{y})}$

Figure 2: Two diagrams of the same neural network.

If you plug in 0 or 1 in for y you get the logistic regression assumption (try it)! If we extend this idea to write the likelihood of n independent datapoints $(\mathbf{x}^{(i)}, \hat{y}^{(i)})$we get:

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

If you take the log of the likelihood you get 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. You could plug in the definition for $\hat{y}$. Note that $\hat{y}$ is a probability; during prediction, we report the argument that has maximal probability $\hat{y} = P(Y = y|\mathbf{X} = \mathbf{x})$.

Backpropagation

We are going to chose values of θ using our old friend MLE (maximum likelihood estimation). MLE applied to deep networks gets a special name “Backpropagation". To chose the optimal values of theta we are going to use gradient ascent where we continually update our thetas in a way that leads to a step up with respect to likelihood. In order 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 big idea that is really worth wrapping your head around, is that chain rule can let us calculate gradients one layer at a time. By the chain rule 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:

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

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

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

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

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.

Example 1

As an example, consider evaluating the partial derivative $\frac{\partial LL(\theta)}{\partial \hat{y}}$. To do so first write out the function for LL in terms of $\hat{y}$ then differentiate:

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

$$\frac{\partial LL(\theta)}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})}$$

It is that simple! Let's try another example.

Example 2

Let's calculate the partial derivative of $\hat{y}$ with respect to an output parameter $\theta_j^{(\hat{y})}$:

$$\hat{y} = \sigma(z) \qquad \text{Where } z = \sum_{i=0}^{m_h} \mathbf{h}_i \theta_i^{(\hat{y})}$$

$$\frac{\partial \hat{y}}{\partial \theta_j^{(\hat{y})}} = \sigma(z)[1-\sigma(z)]\frac{\partial z}{\partial \theta_j^{(\hat{y})}} \qquad \text{Using the formulate for derivative of sigmoid}$$

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

The Future

Deep learning is a growing field. There is a lot of room for improvement. Can we think of better networks? Can we develop structures that do a better job of incorporating prior beliefs? These problems (and many others) are open. It is worth knowing the math of deep learning well because you may one day have to invent the next iteration.