Neural Network (Notes from Hinton's Class)

Ninghui Li

September 2017

1 Lecture 3

1.1 Linear Neuron and Its Gradient Descent Learning Rule

Recall that a linear neuron has input $\mathbf{x} = \langle x_1, x_2, \ldots, x_k \rangle$ and weight $\langle w_1, w_2, \ldots, w_k \rangle$. The output is

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

Deriving the delta rule. We use squared error as the cost function (which we want to minimize). We use t^n to denote the desired output at node n.

$$\begin{aligned}
E &= \frac{1}{2} \sum_n (t^n - y^n)^2 && \text{Error function} \quad (1) \\
\frac{\partial E}{\partial w_i} &= \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{\mathrm{d}E^n}{\mathrm{d}y^n} && \text{Derivative of } E \\
&= \frac{1}{2} \sum_n x_i^n 2(t^n - y^n) \\
&= \sum_n x_i^n (t^n - y^n) && (2)
\end{aligned}$$

The Gradient Descent method is to update each weight by its the negation of the partial derivative of the cost function with respect to the weight.

1.2 Logistic Neuron and Its Gradient Descent Learning Rule

Recall that a logistic neuron (aka sigmoid neuron) has input $\mathbf{x} = \langle x_1, x_2, \ldots, x_k \rangle$ and weight $\langle w_1, w_2, \ldots, w_k \rangle$. The output is

$$\begin{aligned}
y &= \frac{1}{1 + e^{-z}} && \text{This is the logistic function.} \quad (3) \\
z &= b + \sum_i x_i w_i && z \text{ is called the logit} \quad (4)
\end{aligned}$$

The derivative of the logistic neuron is computed as follows:

$$
\begin{aligned}
\frac{\partial y}{\partial w_i} &= \frac{\partial z}{\partial w_i}\frac{\mathrm{d}y}{\mathrm{d}z} && \text{Chaining rule.} \\
&= x_i \cdot (-1)\frac{\frac{\mathrm{d}(1+e^{-z})}{\mathrm{d}z}}{(1+e^{-z})^2} \\
&= x_i \frac{e^{-z}}{(1+e^{-z})^2} \\
&= x_i \frac{e^{-z}}{1+e^{-z}}\frac{1}{1+e^{-z}} \\
&= x_i y(1-y) && (5)
\end{aligned}
$$

The learning rule can be computed using the above derivative:

$$
\begin{aligned}
E &= \frac{1}{2}\sum_n (t^n - y^n)^2 && \text{Error function} \\
\frac{\partial E}{\partial w_i} &= \frac{1}{2}\sum_n \frac{\partial y^n}{\partial w_i}\frac{\mathrm{d}E^n}{\mathrm{d}y^n} \\
&= \frac{1}{2}\sum_n x_i^n y^n (1-y^n) 2 (t^n - y^n) \\
&= \sum_n x_i^n y^n (1-y^n)(t^n - y^n) && (6)
\end{aligned}
$$

Note that the extra term in (6) when compared with (2) is $y^n(1-y^n)$, which is the slope of the logistic function.

1.3 The Backpropagation Algorithm

So far, we have learning rules for NN with a single layer, where we have inputs and *desired output*. Adding adding hidden layers, it is more difficult to learn, because we do not know what the desired output for the hidden neurons (nodes) should be. However, we can still compute the partial derivative of the error of the final output with respect to each weight, by using the backpropagation algorithm.

Let y_j denote output of node j in the output layer, y_i denote output of a node in the previous hidden layer. Thus we have

$$
y_j = \frac{1}{1+e^{-z_j}} \tag{7}
$$

$$
z_j = \sum_i y_i w_{ij} \tag{8}
$$

We can compute the error derivative:

$$\begin{aligned}
E &= \frac{1}{2}\sum_j (t^j - y^j)^2 && \text{Error function} \quad (9)\\
\frac{\partial E}{\partial y_j} &= -(t^j - y^j) && \text{Derivative w.r.t. last layer output} \quad (10)\\
\frac{\partial E}{\partial z_j} &= \frac{\mathrm{d}y_j}{\mathrm{d}z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j} && \text{See the steps for (5).} \quad (11)\\
\frac{\partial E}{\partial w_{ij}} &= \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j} && (12)\\
\frac{\partial E}{\partial y_i} &= \sum_j \frac{\mathrm{d}z_j}{\mathrm{d}y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j} && \text{Derivative w.s.t. second to last layer output} \quad (13)
\end{aligned}$$

Two key observations are

1. From (11) and (12), knowing the partial derivative w.r.t. one layer's output (i.e., $\frac{\partial E}{\partial z_j}$) and the output of the previous layer (i.e., y_i), we can compute the partial derivative w.r.t. to the weight, (i.e., $\frac{\partial E}{\partial w_{ij}}$). This enables us to update the w_{ij} by gradient descent.

2. From (11) and (13), Knowing the the partial derivative w.r.t. one layer's output and the NN state, we can compute the partial derivative w.r.t. the previous layer's output. That is, we are able to back-propagate partial derivatives.

2 Lecture 4

2.1 Softmax

The motivation behind Softmax is to deal with two things:

1. We want to predict one out of n possible output. That is, the output values should sum up to 1.

2. Using the logistic neuron, the partial derivative is $sum_n x_i^n y^n(1-y^n)(t^n-y^n)$ (See (6)). This can be very small when y approaches 0 or 1. Suppose that y is very close to 0, whereas t is 1. That is, the error is large, yet the update will be very small.

Softmax output function. To solve this problem, we change both the output function of each node, and the error function. We group some output neurons into a softmax group, and define the output to be

$$\begin{aligned}
y_j &= \frac{e^{z_j}}{\sum_{i\in\text{group}} e^{z_i}} && (14)\\
z_i &= b + \mathbf{x}^T\mathbf{w} && z \text{ is called the logit} \quad (15)
\end{aligned}$$

Note that one output y_j is dependent on all the z_i's in the same softmax group. We compute

$\frac{\partial y_j}{\partial z_i}$ as follows.

$$\text{Let } \Sigma = \sum_{i \in \text{group}} e^{z_i} \qquad \text{Introduce } \Sigma \text{ to simplify the formulas} \tag{16}$$

$$\text{Then } y_j = e^{z_j} \cdot \frac{1}{\Sigma} \tag{17}$$

$$\text{Thus } \frac{\partial y_j}{\partial z_i} = \frac{\partial e^{z_j}}{\partial z_i} \cdot \frac{1}{\Sigma} + e^{z_j} \cdot \frac{\partial (1/\Sigma)}{\partial z_i} \tag{18}$$

$$= \frac{\partial e^{z_j}}{\partial z_i} \cdot \frac{1}{\Sigma} + e^{z_j} \cdot (-1) \frac{1}{\Sigma^2} \cdot \frac{\partial \Sigma}{\partial z_i} \tag{19}$$

There are two cases, depending on whether $j = i$. Case one, when $j = i$, we have

$$\begin{aligned} \frac{\partial y_j}{\partial z_j} &= \frac{e^{z_j}}{\Sigma} - \frac{(e^{z_j})^2}{\Sigma^2} = \frac{e^{z_j}\Sigma - (e^{z_j})^2}{\Sigma^2} \\ &= \frac{e^{z_j}(\Sigma - e^{z_j})}{\Sigma^2} = \frac{e^{z_j}}{\Sigma} \frac{(\Sigma - e^{z_j})}{\Sigma} \\ &= y_j(1 - y_j) \end{aligned} \tag{20}$$

Case two, when $j \neq i$, we have

$$\begin{aligned} \frac{\partial y_j}{\partial z_i} &= -\frac{e^{z_j} e^{z_i}}{\Sigma^2} = -\frac{e^{z_j}}{\Sigma} \frac{e^{z_i}}{\Sigma} \\ &= -y_j y_i \end{aligned} \tag{21}$$

Softmax cost function. We define the cost function to be

$$C = -\sum_j t_j \log y_j \qquad t_j \text{ is the target value} \tag{22}$$

Let us see what the partial derivative looks like.

$$\begin{aligned}
\frac{\partial C}{\partial z_i} &= -\sum_j \frac{\partial (t_j \log y_j)}{\partial y_j} \frac{\partial y_j}{\partial z_i} \\
&= -\sum_j \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_i} \\
&= -\left(\frac{t_i}{y_i} \frac{\partial y_i}{\partial z_i} + \sum_{j \neq i} \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_i} \right) \\
&= -\left(\frac{t_i}{y_i} y_i (1 - y_i) + \sum_{j \neq i} \frac{t_j}{y_j} (-y_j y_i) \right) \\
&= -\left(t_i (1 - y_i) - \sum_{j \neq i} t_j y_i \right) \\
&= -\left(t_i - t_i y_i - \sum_{j \neq i} t_j y_i \right) \\
&= -\left(t_i - y_i \left(\sum_j t_i \right) \right) \\
&= -(t_i - y_i) \\
&= y_i - t_i
\end{aligned}$$

3 Lecture 6

Understand quadratic error function. Consider the following error function

$$E = 1000x^2 + y^2$$

, and the point $x = 1$, $y = 100$. To converge to the minimum $x = 0, y = 0$, we want to move y 100 times as fast as x.

However, the derivative values are

$$\frac{\partial E}{\partial x} = 2000x = 2000; \frac{\partial E}{\partial y} = 2y = 200$$

That is, the step size we change x will be 10 times as large the step we change y.

The desired vector of change direction is $(-1, -100)$. The gradient vector is $(2000, 200)$. They are almost orthogonal.

4 Lecture 7: Recurrent Neural Networks

Sequential data includes language data, time series data (such as stock prices), credit-card transactions, etc. There are several learning tasks for sequential data.

- In *sequential supervised learning*, each training example is a pair (x^i, y^i) of sequences; the goal is to learn to output the correct output sequences. For example, in part-of-speech tagging, one (x^i, y^i) pair might consist of $x_i = \langle$do you want fries with that$\rangle$ and $y_i = \langle$verb pronoun verb noun prep pronoun$\rangle$. The goal is to correctly predict a new label sequence $y = h(x)$ given an input sequence x.

- In the *time-series prediction problem*, one is given many sequences and tries to predict the next element after having seen the beginning part of a sequence.

Recurrent neural networks, or RNNs, are a family of neural networks for processing sequential data. Basic RNNs are a network of neuron-like nodes, each with a directed (one-way) connection to every other node. Each node (neuron) has a **time-varying** real-valued activation. Each connection (synapse) has a modifiable real-valued weight. Nodes are either input (receiving data from outside the network), output nodes (yielding results) or hidden nodes (neither input or output).

Input vectors arrive at the input nodes in discrete time. At any given time step, each non-input unit computes its current activation (result) as a nonlinear function of the weighted sum of the activations of all units that connect to it. In supervised learning, target activations can be supplied for some output units at certain time steps. For example, if the input sequence is a speech signal corresponding to a spoken digit, the final target output at the end of the sequence may be a label classifying the digit.

In reinforcement learning, a fitness function or reward function is occasionally used to evaluate the RNN's performance. For example, when training to play a game in which progress is measured with the number of points won.

Recurrent neural networks can be viewed as providing a function definition of the form:

$$h^t = f(h^{t-1}, x^t, \theta),$$

where θ consists of all parameters (such as weights in a RNN), x_n gives the input at the n'th

See the *http://karpathy.github.io/2015/05/21/rnn-effectiveness/*