*Xiaoxu Qi*

# Note On Neural Network

Dec. 27th, 2017

# Goals

- How to choose activation function
- How to balance training time and precision: dropout
- Some practical papers

# Why sigmoid neutron

- To avoid completely flip from small change in weights or bias. (Bias: how hard it is for a neutron to fire.)
- Change in easy-to-control way: gradually modify the weights and biases so that the network gets closer to behave more in the way we desire.
- Small changes in weights and bias cause only a small change in their output

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

# Just like a perceptron

- algebraic form:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

- In calculus view: delta(output) is a linear function of delta(w) and delta(b), easy to predict how change of input perform on change of output

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

# Activation Function

- Different activation function, different form for the partial derivatives.
- Ref: http://neuralnetworksanddeeplearning.com/chap1.html

# Other topics

- Feedforward
- Recurrent neural networks

# Gradient Descent

- Cost Function:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

- Smooth function of weights and biases.

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

- To compute derivatives to find extremum.

$$\Delta C \approx \nabla C \cdot \Delta v.$$

$$\Delta v = -\eta \nabla C,$$

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

- delta(c) will always decrease to minimise cost value.

$$v \rightarrow v' = v - \eta \nabla C.$$

# Stochastic Gradient Descent

- estimate the gradient $\nabla C$ by computing $\nabla C_x$ for a small sample of randomly chosen training inputs
- mini-batch:
- epoch of training

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

# Back propagation

- Fast algorithm for computing gradients.

**Summary: the equations of backpropagation**

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

1. **Input** $x$**:** Set the corresponding activation $a^1$ for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error** $\delta^L$**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

- Ref: http://neuralnet
chap2.html

# Cross entropy cost function

- a generalization of the cross-entropy for probability distribution

$$C = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)],$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j}$$
$$= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j.$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y).$$

- the greater the initial error, the faster the neuron learns

# Softmax

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}},$$

# Overfitting

- Early Stopping: Once the classification accuracy on the validation_data has saturated, we stop training
- Finally evaluation with test_data

# Regularization to reduce overfitting

- weight decay
- L2/L1 regularization
- Dropout

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w$$
$$= \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}.$$

- Ref: Dropout: A simple way to prevent neural networks from overfitting
- The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there

# Dropout

- averaging different neural network
- voting
- This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

# Weight Initialisation

- saturation
- learning slowdown
- weight = random(0 - 1) * sqrt(1/Nin)

# Hyper Parameter

- Broad strategy
- Learning rate