Artificial Neural Networks

Pramook Khungurn

December 11, 2019

1 Neural Network Structures

- Neural networks are composed of **nodes** or **units**.
- Nodes are connected by **directed links**.
- The direct link from Node i to Node j serves to propagate the **activation** a_i from Node i to Node j. The direct link (i, j) also has weight $w_{i,j}$ associated with it.
- Each node has a dummy input $a_0 = 1$ with weight $w_{0,j}$ associated with it.
- Each Node j computes a weight sum $in_j = \sum_{i=0}^{n} w_{i,j} a_i$, passes the sum to an **activation function** $g(\cdot)$, and takes the output as the activation of the node:

$$a_j = g(in_j) = g\bigg(\sum_{i=0}^{n} w_{i,j} a_i \bigg).$$

- If g is a step function, then we call the node a **perceptron**. If g is the logistic function, then the node is a **sigmoid perception**.
- There are two ways to connect nodes together: **feed forward network** or **recurrent network**
- Units can be arranged in **layers**.

2 Single-Layer Feed-Forward Neural Network

- In a single-layer network, each perceptron is a network in itself. So, the capability is not different from a collection of linear classifiers.
- Linear classifiers can only classify linearly separable. However, most boolean functions are not linearly separable.
- The **majority function** is linearly separable, and can be captured effectively by perceptrons. Decision tree have a hard time capturing it.

3 Multi-Layer Feed-Forward Network

- Boolean functions AND, OR, and NOT can be captured by a single unit. Thus, we can represent any boolean functions by connecting a lot of units together.
- With a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy.

- With two layers, functions that are not continuous can be represented.
- However, with a particular network structure, it is difficult to characterize which functions are representable.

4 Learning in Multilayer Network

- We represent the neural network as a vector function $\mathbf{h_w}$ rather than a scalar function $h_{\mathbf{w}}$.
- For the L_2 loss, for any weight w, we have

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w} \|\mathbf{y} - \mathbf{h_w}(\mathbf{x})\|^2 = \frac{\partial}{\partial w} \sigma_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2$$

  where the index k ranges over the nodes in the output layer.
- If there are m outputs, the above equation allows us to decompose the learning problem into m learning problems, provided that we remember to add up the gradient contributions from each of them when updating the weights.
- To calculate the gradient, we need to **back-propagate** the error from the output layer to the hidden layers and then the input layer.
- Let Err_k be the kth component of the error vector $\mathbf{y} - \mathbf{h_w}$. Define the modified error $\Delta_k = Err_k g'(in_k)$. The weight update rule becomes

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot a_j \cdot \Delta_k$$

- The modified error Δ_j can also be defined for node j connecting to the output nodes as follows:

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k.$$

  The update rule for weight $w_{i,j}$ is then:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j.$$

- The back propagation process can be summarized as follows:
  - Compute the Δ values for output units, using the observed error.
  - Staring with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
    * Propagate the Δ values back to the previous layer.
    * Update the weights between the two layers.
- When try to construct a neural network, there is no theory telling us what the topology of the network should be. So, we find a good network topology for a problem by cross-validation.