Artificial Neural Network

A neural network is a network of neurons. A neuron is a graph representation of some (typically) nonlinearized linear function.

Let $f(x) = x^T \beta$ be a linear function, and $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a nonlinear function. Then a neuron has the form $\sigma(x^T \beta)$, where $\sigma$ is an activation function, $\beta_j$ is weight and $x_i$ is (typically) output of another neuron. Weights are parameters learned from data. The choice of activation function is a hyper-parameter. There are many choices\(^1\). A common one is logistic sigmoid function

$$\sigma(x^T \beta) = \frac{1}{1 + \exp(-x^T \beta)}.$$ \hspace{1cm} (1)

A network has many connected layers, typically divided into input layer, hidden layers and output layer. The number of layers and numbers of neurons in those layers are hyper-parameters.

A neural network model is more complex if it has more layers, more neurons per layer or more complex activation functions.

[Discussion] What if all activation functions are linear?

There are many network architectures. A common one is feed-forward neural network (Fig 1).

![Figure 1: Neural Network Architecture [PRML, Figure 5.1]](source)

In the output layer, we can construct one neuron for each class. For example, if there are three classes (e.g., cat, dog, fish), then we can construct three output neurons $y_1, y_2, y_3$, where $y_k$ is the probability that $x$ comes from class $k$ (e.g., cat). To make sure $y_k$’s can be interpreted as probabilities, e.g., $\sum_k y_k = 1$, we can apply the softmax function on $y_k$’s, i.e.

$$\text{softmax}[y_k] = \frac{\exp(y_k)}{\sum_{j=1}^3 \exp(y_j)}.$$ \hspace{1cm} (2)

\(^1\)https://en.wikipedia.org/wiki/Activation_function
A common approach to learn a neural network is back-propagation, which is essentially stochastic gradient descent. If the network can be represented by a (very complex) function $f$, then its loss over $n$ training examples is

$$J_n(f) = \sum_{i=1}^{n} (f(x_i) - y_i)^2,$$

which is often used as the objective function to minimize. We can apply chain rule to compute the gradient at any weight $w$ and update it by

$$w = w - \eta \cdot \frac{\partial}{\partial w} \text{loss}_n(f),$$

where $\eta$ is learning rate (hyper-parameter). Smaller $\eta$ can lead to better (local) optimum but more slowly convergence.

Neural network is likely to overfit due to its big model complexity. There are several ways to avoid it. One is weight decay, which adds a regularization term to the objective function

$$J_n(f) = \sum_{i=1}^{n} (f(x_i) - y_i)^2 + \lambda \sum w^2.$$  

Another approach is early stop, which stops updating the network before it converges. A recent approach is dropout, which only updates a sub-network in every interaction (or, epoch).

[Exercise] Compute gradients at $w_{11}$ and $\beta_1$ in Fig . Let $\sigma$ be the activation function.

[Reading] ELS Chapter 11.3, 11.5 (11.5.1, 11.5.2, 11.5.4.)

[Other Reference] PRML, Chapter 5.

First, we have

$$\hat{y} = \sigma(\beta_1 z_1 + \beta_2 z_2),$$

$$z_1 = \sigma(w_{11} x_1 + w_{21} x_2),$$

$$z_2 = \sigma(w_{12} x_1 + w_{22} x_2).$$

The loss over $n$ training examples is

$$J_n(f) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2.$$  

Take derivative w.r.t. $\beta_1$, we have

$$J'_n(f) = \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \cdot (\hat{y}_i)' = \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \cdot \sigma'(\beta_1 z_{i1} + \beta_2 z_{i2}) \cdot z_{i1}.$$
Take derivative w.r.t. $w_{11}$, we have

$$J'_n(f) = \sum_{i=1}^n 2(\hat{y}_i - y_i) \cdot (\hat{y}_i)'$$

$$= \sum_{i=1}^n 2(\hat{y}_i - y_i) \cdot \sigma'(\beta_1 z_{i1} + \beta_2 z_{i2}) \cdot (\beta_1' z_{i1}')$$

$$= \sum_{i=1}^n 2(\hat{y}_i - y_i) \cdot \sigma'(\beta_1 z_{i1} + \beta_2 z_{i2}) \cdot (\beta_1' z_{i1}')$$

$$= \sum_{i=1}^n 2(\hat{y}_i - y_i) \cdot \sigma'(\beta_1 z_{i1} + \beta_2 z_{i2}) \cdot \beta_1' \sigma'(w_{11} x_{i1} + w_{21} x_{i2}) \cdot (x_{i1}).$$

(11)

The above network architecture is known as multi-layer perceptron (MLP). Modern deep learning uses more sophisticated architectures. We will briefly introduce CNN and RNN.

Convolutional Neural Network (CNN) is good at processing image data. The convolution layers run convolution to extract local features of image. The pooling layers down-sample image.

![Convolutional Neural Network](image1.png)

**Figure 2: Convolutional Neural Network**

Recurrent Neural Network (RNN) is good at processing sequential data. Historical sequence information is encoded by hidden states $s$. Each cell predicts the next element $y$ using the current element $x$ and hidden state $s$, typically through a nonlinearized linear function.

![Recurrent Neural Network](image2.png)

**Figure 3: Convolutional Neural Network**