PH 240C Neural Networks

Jingshen Wang

November 17, 2021

In this section, we introduce feedforward neural networks (Svozil et al., 1997), or multilayer perceptrons. According to wikipedia, a feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. The feedforward neural network is the first and simplest type of artificial neural network devised.

1 A single-layer of neurons

The simplest kind of neural network is a single-layer perceptron network, which consists of a single layer of output nodes; the inputs (recorded in $x \in \mathbb{R}^d$) are fed directly to the outputs via a series of weights:

$$\mathbf{1}(\mathrm{w}'x + b > t) \in \{0, 1\}$$

The sum of the products of the weights w and the inputs x is calculated in each node, and if the value is above some threshold t (typically 0) the neuron fires and takes the activated value (equals 1 in the above activation function); otherwise it takes the deactivated value (equals 0 in our activation function). Neurons with this kind of activation function are also called **artificial neurons** or linear threshold units. In the literature the term perceptron often refers to networks consisting of just one of these units.

Note that what differentiates the outputs of different artificial neurons are their parameters (also referred to as their weights). Say we have access to m neurons in total, and each neuron has weight w_j and bias b_j. The output of the neuron j with an indicator activation function is thus

$$\mathbf{1}(\mathrm{w}_j'x + b_j > t), \quad j = 1, \ldots, m.$$

We can also instead use other form of the activation function instead of the step function. For example, sigmoid activation function is also frequently used due to its clear benefit in optimization (we will see this in a minute)

$$\frac{1}{1 + \exp\big(-(\mathrm{w}_j'x + b_j)\big)}, \quad j = 1, \ldots, m.$$

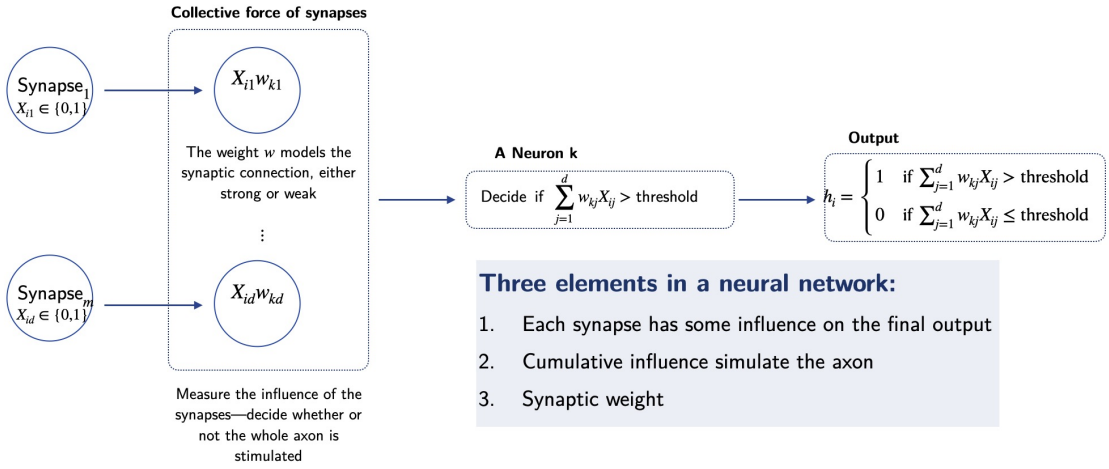

Figure 1: Illustration of how one neuron k stimulated by the synaptic changes $X_i = (X_{i1}, \ldots, X_{id})'$ and then produces an output h_i.

So what do these activations really tell us? Well, one can think of these activations as indicators of the presence of some weighted combination of features. We can then use a combination of these activations to perform classification tasks.

2 Feed-forward Computation

Similar to our problem setup in the supervised learning section, the learner receives a training sample $S = \{(X_i, Y_i)\}_{i=1}^n$ of size n i.i.d from $\mathcal{X}$ according to some unknown distribution $F(\cdot)$, with $Y_i = f(X_i)$ and $X_i \in \mathbb{R}^d$, $Y_i \in \{0, 1\}$. Instead of building a linear classifier from the set:

$$\mathcal{H} = \{x \to \mathbf{1}(x'\beta > 0) : \beta \in \mathbb{R}^d\},$$

we search a classifier from a class of non-linear functions:

$$\mathcal{H} = \{x \to \mathbf{1}(\phi(x)'\beta > 0) : \beta \in \mathbb{R}^d\}.$$

The question is then how to choose the mapping $\phi(\cdot)$:

1. One option is to use a very generic ϕ, such as the infinite-dimensional ϕ used by kernel machines based on the gaussian kernel. However, such a method often faces difficulties when generalizing the prediction model to test data.

2. Another option is to manually engineer $\phi(\cdot)$. Until the advent of deep learning, this was the dominant approach. It requires a decade of human effort for each separate task, with practitioners specializing in different domains.

3. The strategy of deep learning is to learn ϕ. In this approach, we essentially model the outcome

with

$$Y_i = \mathbf{1}\big(\phi(X_i;\theta)'\beta > 0\big),$$

where the parameter θ needed to learn from a broad class of functions, and β maps from $\phi(x)$ to desired outputs. This is an example of a deep feedforward network with ϕ defining a hidden layer. This approach is the only one of the three

Suppose for now we can stack these activation functions $g(\cdot)$ together as

$$h = (h_1, \ldots, h_m)' \triangleq g(W'X_i + b > t) \in \mathbb{R}^{m\times 1}, \quad W = (\mathrm{w}_1, \ldots, \mathrm{w}_m)',\ b = (b_1, \ldots, b_m)',$$

and the activation function $g(\cdot)$ is applied component-wise. Then our complete network function is defined as

$$f(\beta, W, b) \triangleq \phi(X_i;\theta)'\beta = g(W'X_i + b > t)'\beta$$

Our goal is to learn the relationship between an outcome y and the input vector x: $y = f(x)$ based on our training data. In other words, given a set of input vectors and the outcome, we aim to learn $f(\cdot)$ based on $\{(X_i, Y_i)\}_{i=1}^n$. Like most machine learning models, neural networks also need an optimization objective, a measure of error or goodness which we want to minimize or maximize respectively. We may chose the mean squared error loss function to simplify the mathematical derivation as much as possible. The MSE loss function is

$$l(\beta, W, b) = -\frac{1}{2}\sum_{i=1}^{n}\big(Y_i - f(X_i;\beta, W, b)\big)^2.$$

Training a Other loss function we have introduced in the empirical risk minimization section can be applied as well.

Multilayer neurons work in a similar fashion, see Figure 2 for illustration.

3 Training with backpropagation

Training a feedforward neural network is not much different from training any other machine leanring model with gradient descent. Suppose θ contains all unknown parameters including β, W and b, we typically need the gradient information for any parameters as required in the update equation:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \cdot \partial\frac{l(\theta)}{\partial\theta}.$$

Backpropagation is technique that allows us to efficiently use the chain rule of differentiation to calculate loss gradients for any parameter used in the feed-forward computation on the model. To

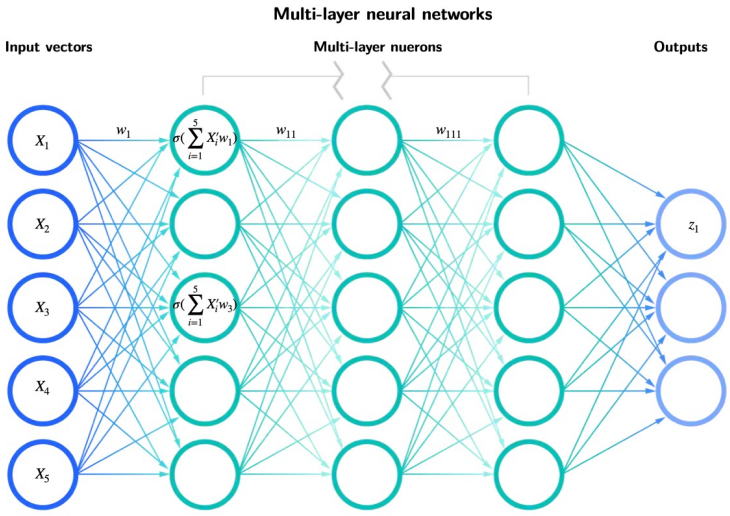

Figure 2: Illustration of a multilayer neural network.

understand this further, let us understand a toy network with two neurons for which we perform backpropagation. Note that the term "backpropagation" strictly refers only to the algorithm for computing the gradient, not how the gradient is used.

Suppose we have one input vector x. When it passes the first neuron, the input is weighted by w_1 and produce some product function $P_1 = x'w_1$. The performance function then goes through some activation function, say sigmoid activation function $\sigma(P_1)$, resulting in an outcome y. The outcome y then goes through another neuron, weighted by w_2, resulting in a product function $P_2 = y'\mathrm{w}_2$. In the last step, we go though another sigmoid function $\sigma(P_2)$, which is the final outcome, denoted as z. Therefore the loss function is of the form:

$$l(\mathrm{w}_1, \mathrm{w}_2) = -\frac{1}{2}(d - z)^2,$$

where d is the desired outcome. The unknown parameters are thus $\theta = (\mathrm{w}_1, \mathrm{w}_2)$.

Now we want to evaluate the change of the loss function with respect to w_2 (gradient),

$$\frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial \mathrm{w}_2} = \frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial z} \cdot \frac{\partial z}{\partial \mathrm{w}_2},$$

where the second term can be again replaced through chain rule:

$$\frac{\partial z}{\partial \mathrm{w}_2} = \frac{\partial z}{\partial P_2} \cdot \frac{\partial P_2}{\partial \mathrm{w}_2}.$$

This means:

$$\frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial \mathrm{w}_2} = \frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial z} \cdot \frac{\partial z}{\partial P_2} \cdot \frac{\partial P_2}{\partial \mathrm{w}_2}.$$

Note that for a given smooth sigmoid function, all above derivatives can be calculated. Benefiting from the sigmoid activation function we have adopted $\sigma(x) = \frac{1}{1+e^{-x}}$, we thus have

$$\frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial z} = d - z, \quad \frac{\partial P_2}{\partial \mathrm{w}_2} = y, \quad \frac{\partial z}{\partial P_2} = \frac{\partial \sigma(P_2)}{\partial P_2} = z(1-z).$$

Similarly for the partial derivative with respect to w_1, we have

$$\frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial \mathrm{w}_1} = \frac{\partial l(\mathrm{w}_1, \mathrm{w}_2)}{\partial z} \cdot \frac{\partial z}{\partial P_2} \cdot \frac{\partial P_2}{\partial y} \cdot \frac{\partial y}{\partial P_1} \cdot \frac{\partial P_1}{\partial \mathrm{w}_1}.$$

Now the gradients are computed, and we can continue to train the neural networks with gradient-based algorithms.

An algorithm usually contains two interactive passes: (1) forward pass that gives some input values and learns the final outcome, and (2) backward pass that learns the gradient of the objective function.

References

Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.