# **Deep Learning**
10/29/2025

Deep learning is a machine learning algorithm that uses neural networks as the predictive model which is trained using known input and output (target) data. Similar to statistical models, a trained neural network model must be validated with the out-of-sample test data (a separate dataset from the training set). The prediction accuracy of the model is then judged by the error between the predicted values and the true values. Unlike the statistical models, neural network models are capable of predicting complex non-linear input-output relations. However, training a neural network model can be time consuming and computationally expensive.

## Universal Approximation Theorem

The promise that a neural network accurately mimics a complex function is based on the universal approximation theorem which has various expressions of similar ideas. One of them is the Cyhenko approximation by superposition of sigmoidal function.

Let $C([0,1]^n)$ denote the set of all continuous function $[0,1]^n \to \mathbb{R}$, and let $\sigma$ be a sigmoidal activation function then function $f(x)$ as expressed below covers $C([0,1]^n)$.

$$f(x) = \sum_{i=1}^{N} \alpha_i \sigma\left(w_i^T x + b_i\right)$$

This theorem suggests that for any $g(x) \subset C([0,1]^n)$ and any $\epsilon \in \mathbb{R} > 0$, there exists:

$$f : x \to \sum_{i=1}^{N} \alpha_i \sigma(w_i^T x + b_i)$$

as an approximation of $g(x)$ such that

$$|f(x) - g(x)| \leq \epsilon \qquad \forall x \subset [0,1]^n$$

Function $f(x)$ is in fact a single hidden layer feed froward network with sigmoidal activations. Therefore, a neural network as "shallow" as a single hidden layer network is capable of approximating any continuous function, $g(x)$, to an arbitrary accuracy as long as the number of neurons (width of the network) in the hidden layer, $N$, is adequately large. Likewise, we have universal approximation theorem for width-bounded $ReLU$ networks.

For any Lebesgue-integratable function $g : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully-connected $ReLU$ network with width $d_m \leq n + 4$, such that the function $f$ represented by the this network satisfies:

$$\int_{\mathbb{R}^n} |f(x) - g(x)|\, dx \leq \epsilon$$

This theorem implies that any continuous function $g : \mathbb{R}^n \to \mathbb{R}$ can be approximated by a deep $ReLU$ network with width $d_m \leq n + 4$. Further, this conclusion has been extended to any continuous discriminatory activation function other than $ReLU$. A discriminatory function is the function whose integral over any non-zero measure is not zero. Thus, the broad applicability of the universal approximation theorem establishes the use case for any width-bound deep neural network, *aka*, deep learning to construct such approximation for most real world functions.

Gradient Descent

Finding the approximation function, $f$, is an optimization problem with the objective of minimizing the error or "loss function" between the approximation function predictions and the true function:

$$\min_w \mathcal{L}(w)$$

where $\mathcal{L}(w)$ is the loss function and $w$ the weight matrix of the deep neural network.

Gradient descent method uses the gradient of the loss function to iteratively move the function closer to its minimum. The concept is that the gradient of $\mathcal{L}(w)$ points to the steepest direction in increase of the function value, whereas the opposite direction of the gradient guides towards the function minimum. The algorithm starts with an arbitrary weight matrix ($w^{(0)}$) and the subsequent updates to the weight matrix are done by letting:

$$w^{(k+1)} = w^{(k)} - \eta \nabla_w \mathcal{L}(w^{(k)})$$

where $\eta$ is the step size, or learning rate, and $\nabla_w \mathcal{L}$ the gradient of the loss function with respect to $w$ (a Jacobian matrix). The iteration continues until the stopping criteria are meet. The criteria could be the relative changes in the loss function, a specific value of the loss function, or a maximum number of the iterations. This algorithm assumes that the step size, $\eta$, is a constant throughout all the steps. Alternatively, an adaptive step size can be established for each step:

$$\eta_k = arg \min_\eta \mathcal{L} \left( w^{(k)} - \eta_k \nabla_w \mathcal{L}(w^{(k)}) \right)$$

This is called a line search for the optimal step size at step $k$. The recursive function calls by this search method is computationally expensive. The convergence of gradient descent is ensured by the following theorem. If the loss function is Lipschitz continuous with Lipschitz constant $L$ and the step size is chosen $\eta \leq 1/L$, then

$$\mathcal{L}(w^{(k+1)}) - \mathcal{L}(w^{(k)}) \leq \frac{\eta}{2} \left\| \nabla_w \mathcal{L}(w^{(k)}) \right\|^2$$

If the loss function gradient is bounded and step size sufficiently small, each iteration will lead to a smaller loss function in value along the way.

Stochastic Gradient Descent

The loss function in gradient descent is typically an average loss over the entire training dataset ($m$):

$$\mathcal{L}(w) = \frac{1}{m} \sum_{i=1}^{m} l_i(w)$$

where $l_i$ is the loss function for training data ($i$).

Stochastic gradient descent is to select a subset of the training dataset as $\mathcal{B}$ (called a batch), and perform the following iteration to update the weights:

$$w^{(k+1)} = w^{(k)} - \eta \frac{1}{\|\mathcal{B}\|} \sum_{i \in \mathcal{B}} \nabla_w l_i(w^{(k)})$$

If $\mathcal{B}$ is randomly generated so that every data point in the training set has the same probability of being part of it, then stochastic gradient descent is unbiased and computationally more efficient than the original gradient descent method.

Newton’s Method

In addition to gradient descent which searches for the function minimum by its gradient but on the opposite direction, Newton’s method is also very popular in solving the optimization problem. Newton’s method minimizes the loss function by iteratively minimizing not the function itself but the quadratic approximation of the loss function.

$$\mathcal{L}(w) = \mathcal{L}(w_0) + \nabla\mathcal{L}(w_0)(w - w_0) + \frac{1}{2}(w - w_0)^T \nabla^2 \mathcal{L}(w_0)(w - w_0)$$

If the loss function has a positive definite Hessian, the iteration of the weights is as the following:

$$w^{(k+1)} = w^{(k)} - \frac{\nabla\mathcal{L}(w^{(k)})}{\nabla^2\mathcal{L}(w^{(k)})}$$

Newton’s method not only requires evaluating the gradient of the loss function, but the Hessian (a second order derivative matrix) of the loss function. Therefore, this algorithm is much more computationally expensive than gradient descent and not used in neural network training.

Back Propagation

The evaluation of the loss function gradient can be made more efficiently using a method called back propagation. First let’s define the output of $l^{th}$ layer in network before activation as:

$$z_j^l = \sum_i w_{j,i}^l a_i^{l-1}$$

Note that the bias $b_i$ term is now absorbed into the weight matrix and after activation:

$$a_j^l = \sigma(z_j^l)$$

Let’s also define

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l}$$

Therefore use the chain rule,

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^l} = \frac{\partial \mathcal{L}}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{i,j}^l} = \delta_j^l a_i^{l-1}$$

since $a_i^{l-1}$ is a known quantity in the network, if one can find $\delta_j^l$ for each network layer, one can solve for $\nabla_w \mathcal{L}$. To find $\delta_j^l$ for each network layer, one needs to start with the last layer ($L$) in the network then solve for each previous layer backwards successively. As such, the algorithm is called back propagation.

Since $f(x) = (a_1^L, a_2^L, \ldots, a_N^L)$ and $g(x) = (y_1, y_2, \ldots, y_N)$,

$$\delta_j^L = \frac{\partial \mathcal{L}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L}\sigma'(z_j^L) = \left(a_j^L - y_j\right)\sigma'(z_j^L)$$

Now $\delta_j^{L-1}$ can be found using $\delta_j^L$ through back propagation.

$$\delta_j^{L-1} = \frac{\partial \mathcal{L}}{\partial z_j^{L-1}} = \sum_i^N \frac{\partial \mathcal{L}}{\partial z_i^L}\frac{\partial z_i^L}{\partial z_j^{L-1}} = \sum_i^N \delta_i^L \frac{\partial z_i^L}{\partial z_j^{L-1}}$$

The second derivative in above equation can be evaluated using the chain rule:

$$\frac{\partial z_i^L}{\partial z_j^{L-1}} = \frac{\partial \sum_k w_{i,k}^L \sigma(z_k^{L-1})}{\partial z_j^{L-1}} = \frac{\partial\left(w_{i,j}^L \sigma(z_j^{L-1})\right)}{\partial z_j^{L-1}} = w_{i,j}^L \sigma'(z_j^{L-1})$$

Therefore, $\delta_j^l$ and the gradient of the loss function with respect to the weights for a specific layer can be found using its next layer $\delta_j^{l+1}$ in the network via back propagation:

$$\delta_j^l = \sum_i^N \delta_i^{l+1} w_{i,j}^{l+1} \sigma'(z_j^l)$$

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^l} = a_i^{l-1}\delta_j^l = a_i^{l-1}\sum_i^N \delta_i^{l+1} w_{i,j}^{l+1}\sigma'(z_j^l)$$

## Training Data

Collecting actual training data can often be challenging. One could however generate simulated data for learning classification tasks. An example of the generated training dataset is shown in Figure 1 and the R script that produces the data is listed in Figure 2.

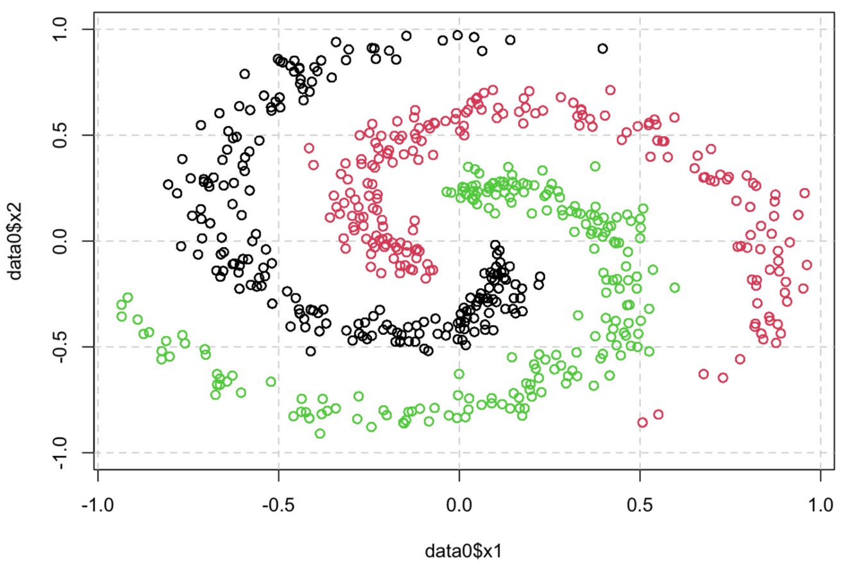

*Figure 1. Simulated Training Dataset for Classification*

```
t=seq(from=0,to=1,by=0.005)
theta=function(t,c){2*pi*((2*t+c)/3)+0.3*rnorm(length(t))}
c1=theta(t,1)
c2=theta(t,2)
c3=theta(t,3)
y1=c1*0+1
y2=c2*0+2
y3=c3*0+3
data0=data.frame(cbind(c=c(c1,c2,c3),y=c(y1,y2,y3)))
data0$x1=(0.9*t+0.1)*sin(data0$c)
data0$x2=(0.8*t+0.2)*cos(data0$c)
plot(data0$x1,data0$x2,type="n", ylim=c(-1,1)); grid(lty=2)
for (i in t){points(data0$x1, data0$x2, col=data0$y)}
```

*Figure 2. Script to Generate the Training Data in Figure 1*