Predictive Analytics (ISE529)

Artificial Neural Network (I)

Dr. Tao Ma
ma.tao@usc.edu

Tue/Thu, May 22 - July 1, 2025, Summer

USC Viterbi
School of Engineering
Daniel J. Epstein Department of Industrial and Systems Engineering

USC University of Southern California

Neural Network Applications

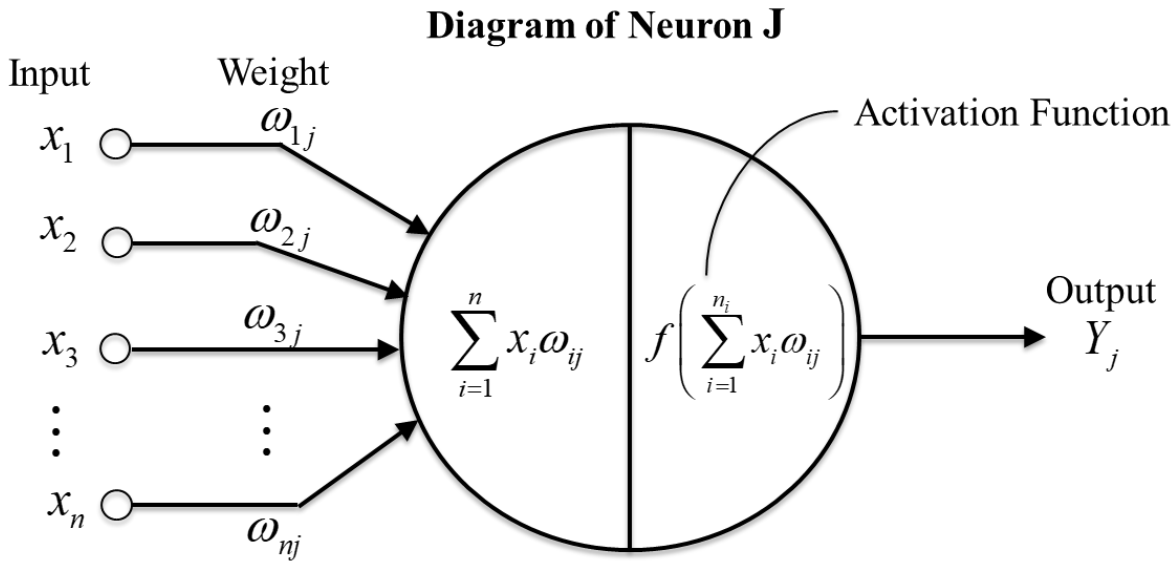
Neural Network is one of the most important tools in predictive analytics and is the cornerstone of deep learning.

- Forecast, meteorology, road network traffic…
- Classification (image, video, etc.)
- Clustering
- Pattern recognition, computer vision…
- Large-scale Language inference, generative AI, ChatGPT
- Autonomous driving
- Facial recognition…

What is the working mechanism that enables Neural Networks to implement those tasks?

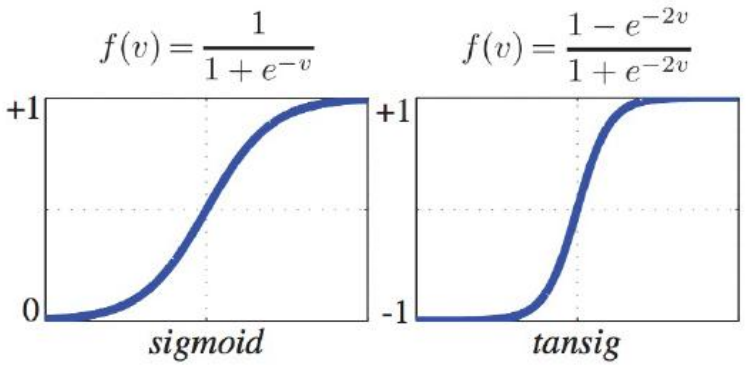
A Single Neuron

- A single neuron is the basic **operator** in a Neural Network.
- The structure of a single neuron is schematically as follows:



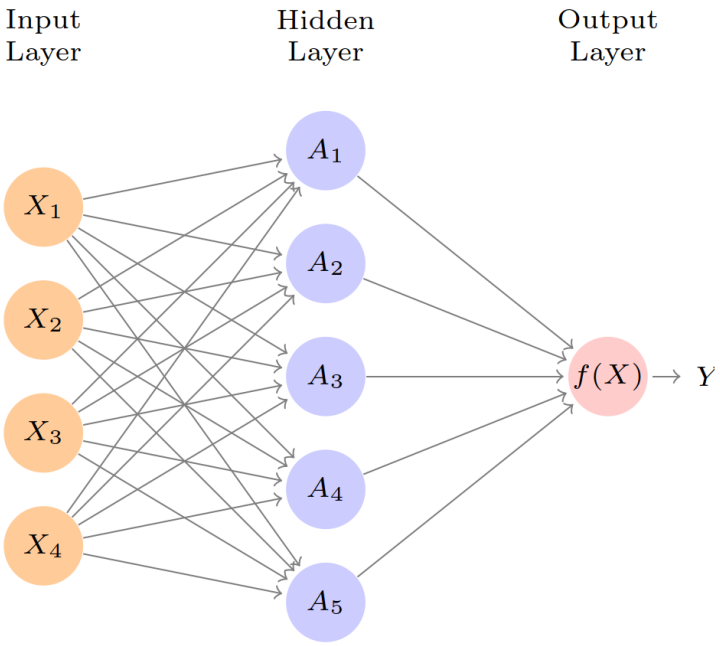
Activation functions $f\left(\sum \omega x\right)$:

- Sigmoid function
- Hyperbolic tangent sigmoid
- ReLU function
- Softmax function



Architecture of Neural Networks

A typical architecture of Neural Networks consist of neurons, layers, and connections :



A neural network takes an input vector of p variables $X = (X_1, X_2, \ldots, X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y . Figure above shows a simple ***feed-forward neural network*** for modeling a quantitative response using $p = 4$ predictors. In the terminology of neural networks, the four features $X_1, \ldots, X_4$ make up the units in the **input layer**. The arrows indicate that each of the inputs from the input layer feeds into each of the K hidden input layer units.

Single Layer Neural Networks

The mathematical form of a typical neural network model can be written as

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j) \end{aligned}$$

The model builds in two steps for feed-forward computation:

1. the K **activations** A_k, $k = 1, \ldots, K$, in the **hidden layer** are computed as functions of the input features $X_1, X_2, \ldots, X_p$,

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

where $g(z)$ is a nonlinear activation function that is specified in advance. Each A_k can be seen as a different transformation of the original function features.

2. These K **activations** from the hidden layer then feed into the **output layer**, resulting in

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k.$$

Parameters and Activation Function

All the parameters $\beta_0, \beta_1, \ldots, \beta_K$ and $w_{k0}, w_{k1}, \ldots, w_{kp}$ need to be estimated from data, the process of parameter estimation is called network training.

In the early instances of neural networks, the *sigmoid activation function* was favored, which is the same function used in logistic regression

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

The preferred choice in modern neural networks is the *ReLU* (rectified linear unit) *activation function*, which takes the form

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

A ReLU activation can be computed and stored more efficiently than a sigmoid activation.

Example for feed-forward computation

The name ***neural network*** originally derived from thinking of these hidden units as analogous to **neurons in the brain** — values of the activations A_k close to one are *firing*, while those close to zero are *silent* (using the sigmoid activation function).

The nonlinearity in the activation function g(·) is essential and allows the model to capture complex nonlinearities and interaction effects.

Consider a very simple example with $p = 2$ input variables $X = (X_1, X_2)$, and $K = 2$ hidden units with $\boldsymbol{g(z) = z^2}$. We specify the other parameters as

$$\begin{aligned} \beta_0 = 0, \quad \beta_1 = \tfrac{1}{4}, \quad \beta_2 = -\tfrac{1}{4}, \\ w_{10} = 0, \quad w_{11} = 1, \quad w_{12} = \ \ 1, \\ w_{20} = 0, \quad w_{21} = 1, \quad w_{22} = -1. \end{aligned}$$

Example for feed-forward computation

1. We get activations:

$$\begin{aligned} h_1(X) &= (0 + X_1 + X_2)^2, \\ h_2(X) &= (0 + X_1 - X_2)^2. \end{aligned}$$

2. We get the model output:

$$\begin{aligned} f(X) &= 0 + \tfrac{1}{4} \cdot (0 + X_1 + X_2)^2 - \tfrac{1}{4} \cdot (0 + X_1 - X_2)^2 \\ &= \tfrac{1}{4} \left[(X_1 + X_2)^2 - (X_1 - X_2)^2 \right] \\ &= X_1 X_2. \end{aligned}$$

Hence, the sum of two nonlinear transformations of linear functions can give us an interaction!

Objective Function

Fitting a neural network requires estimating the unknown parameters in neural network model $\beta_0, \beta_1, \ldots, \beta_K$ and $w_{k0}, w_{k1}, \ldots, w_{kp}$

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j) \end{aligned}$$

For a **quantitative response** (regression), typically **squared-error loss** is used as the objective function, so that the parameters are chosen to minimize

$$L(\theta \mid X) = \sum_{i=1}^{n} \left(y_i - f_\theta(x_i) \right)^2$$

For a **qualitative response** (classification), typically multinomial log-likelihood is used as the objective function

$$L(\theta \mid X) = -\sum_{i=1}^{n} \sum_{m=1}^{M} y_{im} \log\left(f_m(x_i) \right)$$

Multilayer Neural Networks

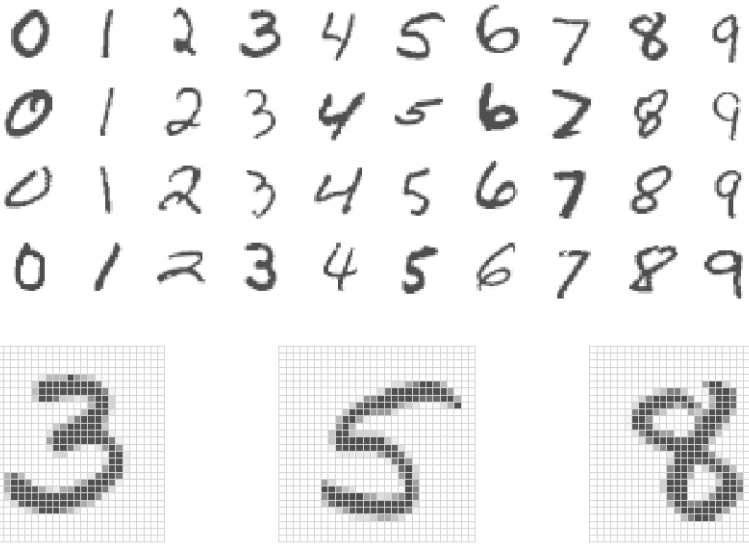
In theory a single hidden layer with a large number of units has the ability to approximate most functions.

However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

Modern neural networks typically have more than one hidden layer, and often many units per layer.

Example:
A large dense neural network was built for the famous and publicly available ***MNIST*** handwritten digit dataset. Figure shows examples of these digits.

The goal is to build a model to classify the images into their correct digit class 0–9.

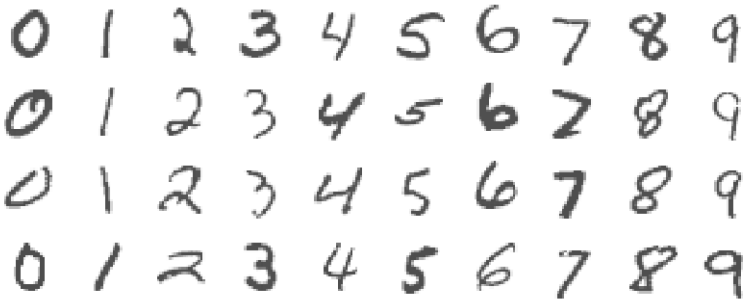


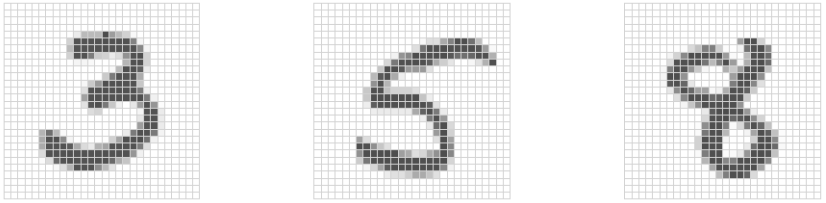
MNIST Data Set

Every image has $p = 28 \times 28 = 784$ pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is.

These pixels are stored in the **input vector** X. The output is the class label, represented by a vector $Y = (Y_0, Y_1, \ldots, Y_9)$ of 10 dummy variables, with a **one** ("1") in the position corresponding to the label, and zeros elsewhere, known as *one-hot encoding*.

There are 60,000 training images, and 10,000 test images.





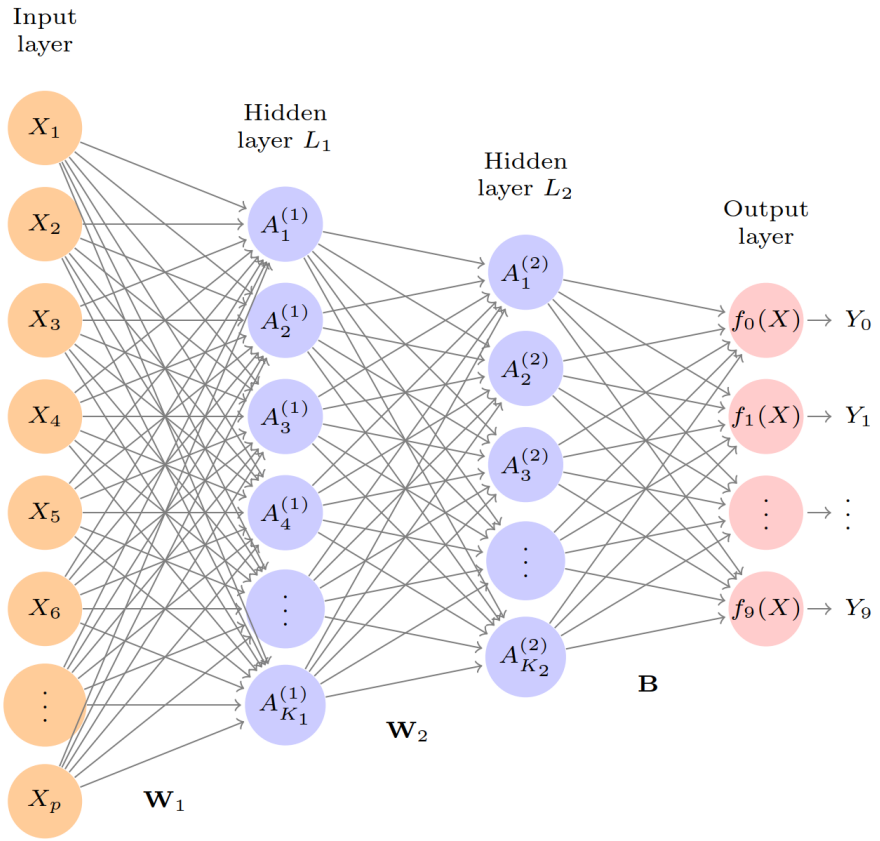
Multilayer Neural Networks

A multilayer network architecture works well for solving the digit classification task. The input layer has p = 784 units, two hidden layers L_1 (256 units) and L_2 (128 units). It has **ten** output variables. The loss function used for training the network is tailored for the multiclass classification task. This network has 235,146 parameters (referred to as weights).

The notation W_1 represents the entire matrix of weights that feed from the input layer to the first hidden layer L_1. This matrix will have 785×256 = 200,960 elements;

Each element $A_k^{(1)}$ feeds to the second hidden layer L_2 via the matrix of weights W_2 of dimension 257 × 128 = 32,896.

The matrix B stores all 129 × 10 = 1,290 of these weights.



Multilayer Neural Networks

The first hidden layer is as

$$\begin{aligned} A_k^{(1)} &= h_k^{(1)}(X) \\ &= g(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j) \end{aligned}$$

for $k = 1, \ldots, K_1$ (=256). The second hidden layer treats the activations $A_k^{(1)}$ of the first hidden layer as inputs and computes new activations

$$\begin{aligned} A_\ell^{(2)} &= h_\ell^{(2)}(X) \\ &= g(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}) \end{aligned}$$

for $\ell = 1, \ldots, K_2$ (=128).

Thus, through **a chain of transformations**, the network is able to build up fairly complex transformations of X that ultimately feed into the output layer as features.

The **superscript** notation indicates to which layer the activations and weights (coefficients) belong.

Multilayer Neural Networks

We now get to the **output** layer, where we now have 10 responses

$$\begin{aligned} Z_m &= \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} h_\ell^{(2)}(X) \\ &= \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}, \end{aligned}$$

for $m = 0, 1, \ldots, 9$. However, we would like our estimates to represent class probabilities $f_m(X) = \Pr(Y = m \mid X)$ which is ***softmax*** function.

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{9} e^{Z_\ell}}$$

for $m = 0, 1, \ldots, 9$. Even though the goal is to build a classifier, the model actually estimates a **probability** for each of the 10 classes. The classifier then assigns the image to the class with the highest probability. To train this network, we look for coefficient estimates that minimize the negative multinomial log-likelihood known as the *cross-entropy*.

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im} \log(f_m(x_i))$$

Training Neural Network

- Any neural network must be trained with big data before it has the capacity to implement specific tasks.

- The essence of training neural networks is to find **optimal weight matrices** such that the neural networks can generate the output as close as possible to the target values. This is analogy to model estimation process for a regression model. It is also called **learning process**.

- The capacity of neural networks lies in the way it is trained with big data.
  - Supervised learning
  - Unsupervised learning

- Its capacity also depends on the data set used in training.
  - Traffic data
  - Image data
  - Language data
  - …

Training Neural Network

Given observations (x_i, y_i), $i = 1, \ldots, n$, we could fit the model by solving a nonlinear least squares problem

$$\underset{\{w_k\}_1^K,\, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2,$$

where

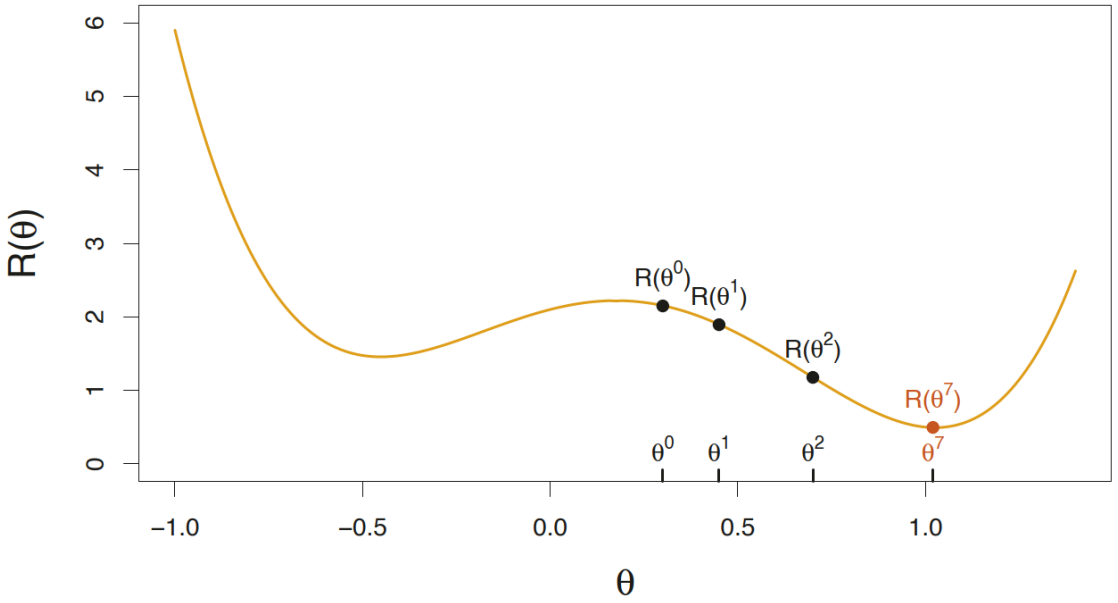
$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right)$$

We can rewrite the objective function as

$$R(\theta) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f_\theta(x_i))^2,$$

Example of Nonconvex Solutions

Because of the **nested** arrangement of the parameters and the symmetry of the hidden units, the problem is **nonconvex** in the parameters, and hence there are multiple solutions.



As an example, the figure shows a simple nonconvex function of a single variable θ; there are two solutions: one is a local minimum at $\theta = -0.46$, and the other is a global minimum at $\theta = 1.02$. In general, we can hope to end up at a (good) local minimum.

Two Step Operations

Neural Networks training is implemented via two-step operations:

- Feed forward propagation performs two tasks:
    1. feed data to neural network, through the network computation, to generate output.
    2. calculate the **discrepancy** between the output and the target value, i.e., the total error.

The goal of training is to minimize the discrepancy between the model output and the target value (observed value).

- Backward propagation performs two tasks:
    1. calculate gradient
    2. adjust the weight matrix

Gradient Descent Approach

In the 2nd step backward propagation, the solution can be found using **gradient descent** approach. The idea of gradient descent is very simple.

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.

2. Iterate until the objective function fails to decrease:

   (a) Find a vector δ that reflects a small change in θ, such that

   $\theta_{t+1} = \theta_t + \delta$ reduces the objective; i.e., such that $R(\theta_{t+1}) < R(\theta_t)$.

   (b) Set $t \leftarrow t + 1$.

Backpropagation

The gradient of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of the first-order partial derivatives at that point:

$$\nabla R(\theta^m) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta^m}$$

This gives the direction in θ-space in which $R(\theta)$ *increases* most rapidly. **The idea of gradient descent** is to move θ a little in the **opposite** direction (since we wish to go downhill), hence, the θ is updated by the following formula:

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m).$$

where ρ is the learning rate. This step will decrease the objective $R(\theta)$; i.e., $R(\theta^{m+1}) \leq R(\theta^m)$ with a small enough value of the learning rate. If the gradient vector is zero, then we may have arrived at a minimum of the objective.

Backpropagation

To get gradient for neural network model, we apply the **chain rule of differentiation** to the objective function

$$R_i(\theta) = \frac{1}{2}\Big(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g\big(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\big)\Big)^2$$

First, we take the derivative with respect to β_k:

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} \\ &= -(y_i - f_\theta(x_i)) \cdot g(z_{ik}). \end{aligned}$$

Then, we take the derivative with respect to w_{kj}:

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \\ &= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}. \end{aligned}$$

Where $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$. The act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule — a process known as **backpropagation** in the neural network literature, i.e., redistribute total errors.

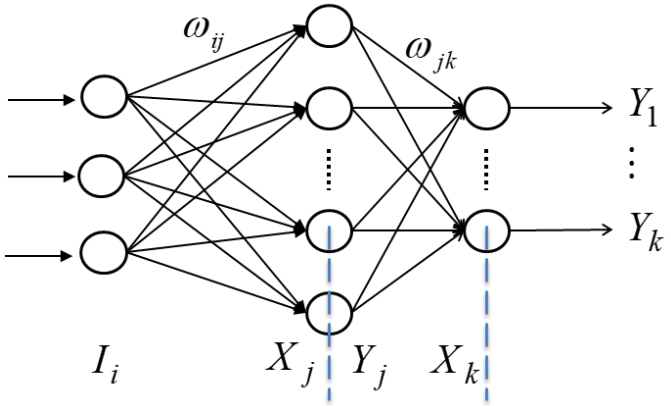
STEP BY STEP EXAMPLE

Feed Forward Propagation

From **input layer** to **hidden layer**:

$$X_j = \sum_{i=1}^{n_i} \omega_{ij} I_i \qquad (1)$$

$$Y_j = f\left(X_j\right) = \frac{1}{1+e^{-X_j}} \qquad (2)$$

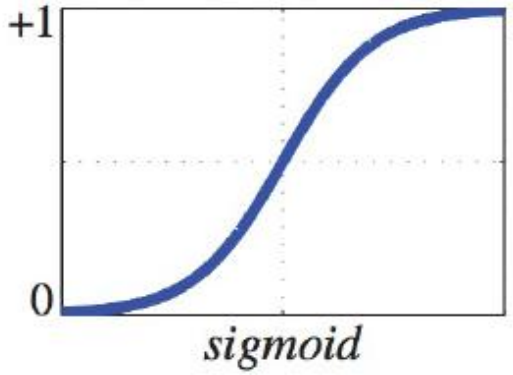


From **hidden layer** to **output layer**:

$$X_k = \sum_{j=1}^{n_j} \omega_{jk} Y_j \qquad (3)$$

$$Y_k = f\left(X_k\right) = \frac{1}{1+e^{-X_k}} \qquad (4)$$

$$f(v) = \frac{1}{1+e^{-v}}$$



Feed Forward Propagation (cont'd)

- Our goal is to match the neural network output to the target values as close as possible.
- **Measure the discrepancy** between the output from the neural network model and the target values as follows:

$$E = \frac{1}{2}\sum_{k}\left(Y_k - O_k\right)^2 \qquad (5)$$

- How can we reduce the total error E ?
  - Use Gradient Descent Method
  - Take the 1st order partial derivative of E with respect to weights
  - Adjust the weights ω_{ij} and ω_{jk}

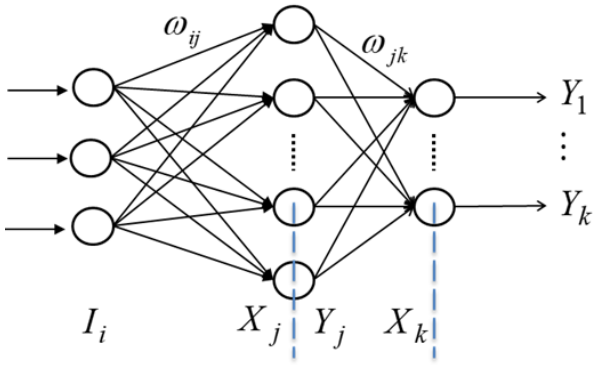
Backward Propagation

Use gradient descent of the error function E with respect to ω_{jk} to adjust weights between **output** layer and **hidden** layer:

$$\frac{\partial E}{\partial Y_k} = 2 * \frac{1}{2}\left(Y_k - O_k\right) = Y_k - O_k$$

$$\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial Y_k}\frac{\partial Y_k}{\partial X_k} = \frac{\partial E}{\partial Y_k} Y_k\left(1-Y_k\right) = \left(Y_k - O_k\right) Y_k\left(1-Y_k\right)$$

$$\frac{\partial E}{\partial \omega_{jk}} = \frac{\partial E}{\partial X_k}\frac{\partial X_k}{\partial \omega_{jk}} = \frac{\partial E}{\partial X_k} Y_j = \left(Y_k - O_k\right) Y_k\left(1-Y_k\right) Y_j \qquad (6)$$



$$E = \frac{1}{2}\sum_k \left(Y_k - O_k\right)^2$$

$$\omega_{jk,n} = \omega_{jk,n-1} - \rho * \frac{\partial E}{\partial \omega_{jk,n-1}} \qquad (7)$$

where ρ is learning rate

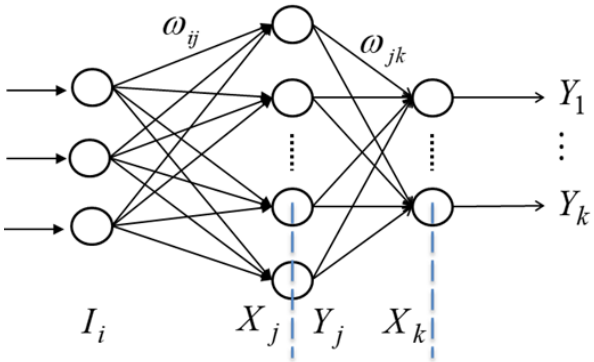
Side Note:
The 1st order derivative of Sigmoid function:

$$y = \frac{1}{1+e^{-x}} \Rightarrow \frac{dy}{dx} = \left(1-y\right) y$$

Backward Propagation (cont'd)

Use gradient descent of the error function E with respect to ω_{ij} to adjust weights between **hidden layer** and **input layer**:

$$\frac{\partial E}{\partial Y_j} = \sum_{k=1}^{n_k} \frac{\partial E}{\partial X_k} \frac{\partial X_k}{\partial Y_j} = \sum_{k=1}^{n_k} \frac{\partial E}{\partial X_k} \omega_{jk} = \sum_{k=1}^{n_k} (Y_k - O_k) Y_k (1 - Y_k) \omega_{jk}$$



$$\frac{\partial E}{\partial X_j} = \frac{\partial E}{\partial Y_j} \frac{\partial Y_j}{\partial X_j} = \frac{\partial E}{\partial Y_j} Y_j (1 - Y_j) = Y_j (1 - Y_j) \sum_{k=1}^{n_k} (Y_k - O_k) Y_k (1 - Y_k) \omega_{jk}$$

$$E = \frac{1}{2} \sum_k (Y_k - O_k)^2$$

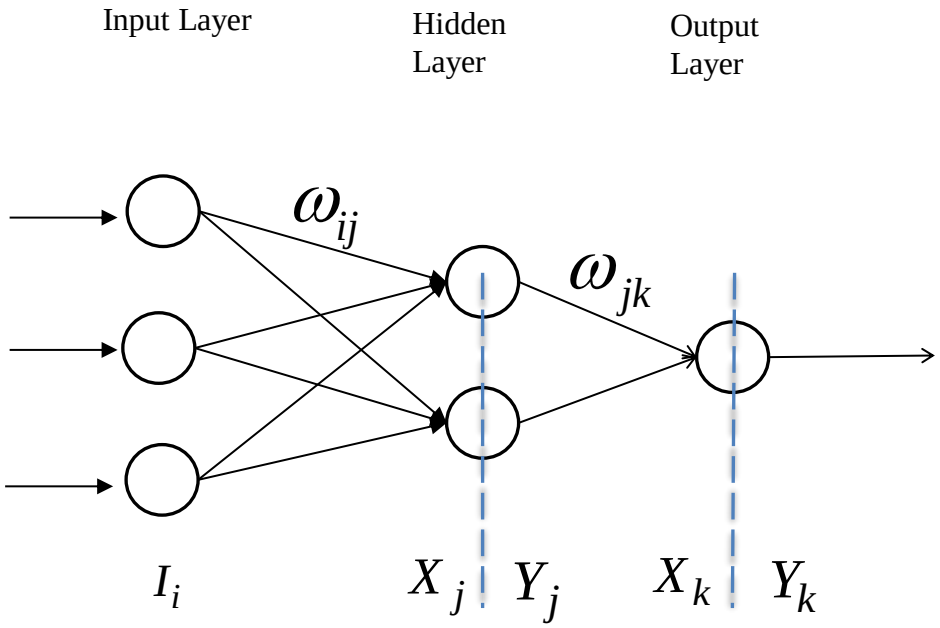
$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial X_j} \frac{\partial X_j}{\partial \omega_{ij}} = \frac{\partial E}{\partial X_j} I_i = I_i Y_j (1 - Y_j) \sum_{k=1}^{n_k} (Y_k - O_k) Y_k (1 - Y_k) \omega_{jk} \qquad (8)$$

$$\omega_{ij,n} = \omega_{ij,n-1} - \rho * \frac{\partial E}{\partial \omega_{ij,n-1}} \qquad (9)$$

where ρ is learning rate

Numerical Example

We use a simple neural network as follows to show Forward and Backward propagation operations with numerical data.



Input layer:	3 neurons
Hidden layer:	2 neurons
Output layer:	1 neuron
Activation Function:	Sigmoid
Learning rate:	5.0

Example: Iteration 1

USC Viterbi School of Engineering

Initial Weight Matrix

Initial

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.1000	0.1000	
I_2	0.1000	0.1000	
I_3	0.1000	0.1000	
H_1			0.5000
H_2			0.5000

Feed Forward Propagation

$$X_j = \sum_{i=1}^{n_i} \omega_{ij} I_i \;;\; Y_j = f(X_j) = \frac{1}{1+e^{-X_j}} \;;\; X_k = \sum_{j=1}^{n_j} \omega_{jk} Y_j \;;\; Y_k = f(X_k) = \frac{1}{1+e^{-X_k}}$$

iteration 1

Input		Hidden		Output		Desired Output	Error
in	out	in	out	in	out		
X_i	I_i	X_j	Y_j	X_k	Y_k	O_k	E
5.0000	5.0000	0.7500	0.6792	0.6792	0.6636	1.0000	0.0566
0.5000	0.5000	0.7500	0.6792				
2.0000	2.0000						

Backward Propagation

$$\frac{\partial E}{\partial \omega_{ij}} = I_i Y_j (1-Y_j) \sum_{k=1}^{n_k} (Y_k - O_k) Y_k (1-Y_k) \omega_{jk}$$

$$\frac{\partial E}{\partial \omega_{jk}} = (Y_k - O_k) Y_k (1-Y_k) Y_j$$

$$\omega_n = \omega_{n-1} - \rho * \frac{\partial E}{\partial \omega_{n-1}}$$

where ρ is learning rate

Initial

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.1000	0.1000	
I_2	0.1000	0.1000	
I_3	0.1000	0.1000	
H_1			0.5000
H_2			0.5000

− 5.0 ×

Gradient

dE/dω	$dE/d\omega_{ij}$		$dE/d\omega_{jk}$
	H_1	H_2	O_1
I_1	-0.0409	-0.0409	
I_2	-0.0041	-0.0041	
I_3	-0.0164	-0.0164	
H_1			-0.0510
H_2			-0.0510

=

Updated

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.3046	0.3046	
I_2	0.1205	0.1205	
I_3	0.1818	0.1818	
H_1			0.7551
H_2			0.7551

Example: Iteration 2

USC Viterbi School of Engineering

Updated Weight Matrix

Updated

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.3046	0.3046	
I_2	0.1205	0.1205	
I_3	0.1818	0.1818	
H_1			0.7551
H_2			0.7551

Feed Forward Propagation

$$X_j = \sum_{i=1}^{n_i} \omega_{ij} I_i \;;\; Y_j = f(X_j) = \frac{1}{1+e^{-X_j}} \;;\; X_k = \sum_{j=1}^{n_j} \omega_{jk} Y_j \;;\; Y_k = f(X_k) = \frac{1}{1+e^{-X_k}}$$

iteration 2

Input		Hidden		Output		Desired Output	Error
in	out	in	out	in	out		
X_i	I_i	X_j	Y_j	X_k	Y_k	O_k	E
5.0000	5.0000	1.9468	0.8751	1.3215	0.7894	1.0000	0.0222
0.5000	0.5000	1.9468	0.8751				
2.0000	2.0000						

Backward Propagation

$$\frac{\partial E}{\partial \omega_{ij}} = I_i Y_j (1-Y_j) \sum_{k=1}^{n_k} (Y_k - O_k) Y_k (1-Y_k) \omega_{jk}$$

$$\frac{\partial E}{\partial \omega_{jk}} = (Y_k - O_k) Y_k (1-Y_k) Y_j$$

$$\omega_n = \omega_{n-1} - g * \frac{\partial E}{\partial \omega_{n-1}}$$

where g is learning rate

Updated

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.3046	0.3046	
I_2	0.1205	0.1205	
I_3	0.1818	0.1818	
H_1			0.7551
H_2			0.7551

$-$ 5.0 $\times$

Gradient

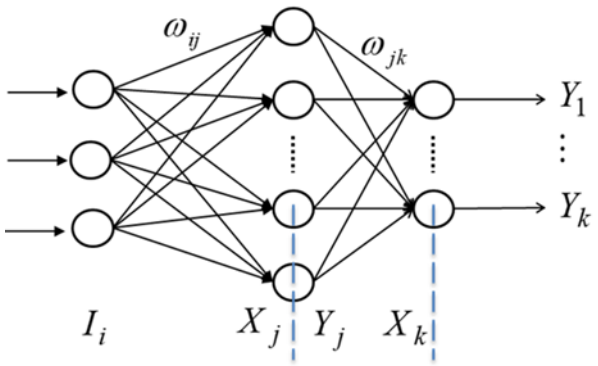
dE/dω	$dE/d\omega_{ij}$		$dE/d\omega_{jk}$
	H_1	H_2	O_1
I_1	-0.0144	-0.0144	
I_2	-0.0014	-0.0014	
I_3	-0.0058	-0.0058	
H_1			-0.0306
H_2			-0.0306

$=$

Updated

Weights	ω_{ij}		ω_{jk}
	H_1	H_2	O_1
I_1	0.3768	0.3768	
I_2	0.1277	0.1277	
I_3	0.2107	0.2107	
H_1			0.9082
H_2			0.9082

Summary of Training Process



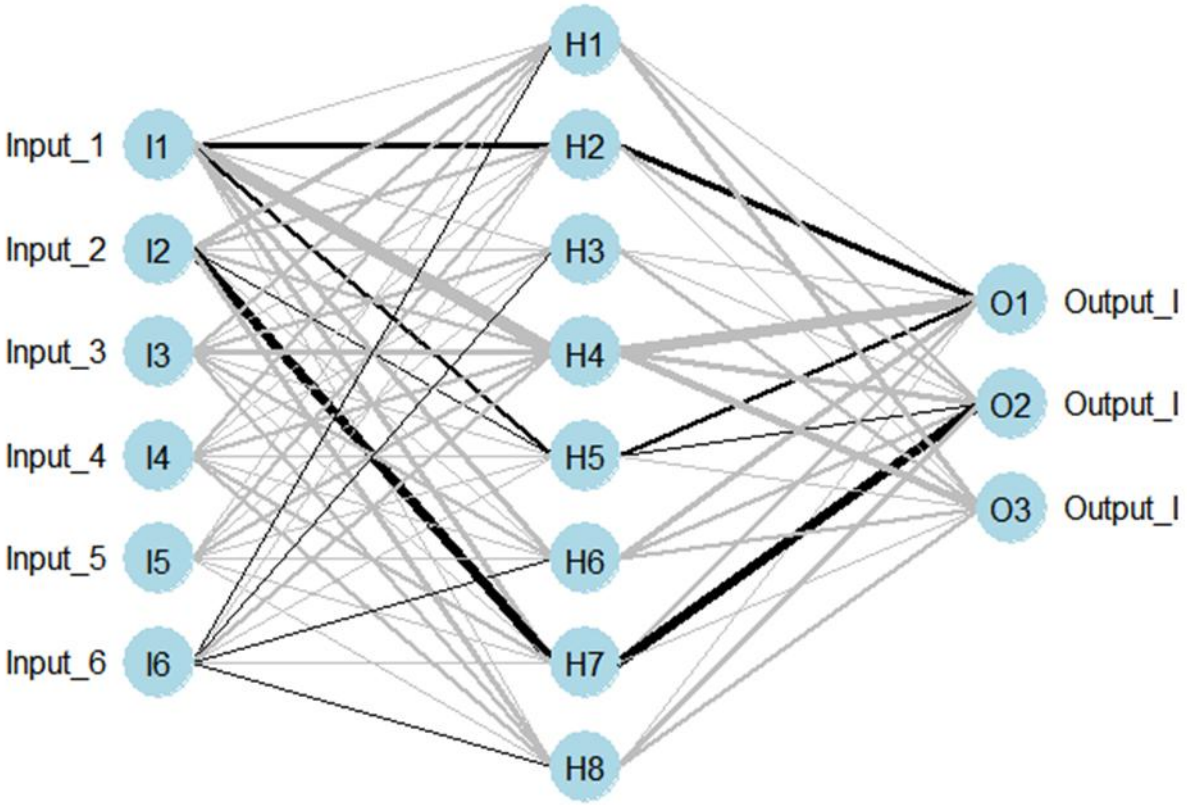
$$E = \frac{1}{2}\sum_k \left(Y_k - O_k\right)^2$$

- Send an input data I_i to the neurons in the input layer
- Feed Forward propagation:
  - Calculate the output Y_k
  - Given target output O_k, calculate total error E
- Backward propagation:
  - Adjust each weight ω_{jk} between output layer and hidden layer
  - Adjust each weight ω_{ij} between hidden layer and input layer
- Repeat with a new input data

Demo of Neural Network Training

USC Viterbi School of Engineering

Multilayer Perceptrons Neural Network



- The thickness of each connection between two neurons is proportional to the magnitude of the connection.
- The training process is the matter of adjusting the weight matrix.

Comments

- Layers are combined to describe the architecture of a neural network
- Modifications to network architecture impact its capacity and performance
- The output of each layer must fit the input of the next layer
- Advantages:
  - nonlinear mapping from input to output space
  - Strong anti-noise capability
- Disadvantages:
  - No statistical diagnosis, inference, confidence interval for each weight (compare to coefficients of a regression model)
  - Lack of systematic way and theoretical basis to determine an optimal structure of network

Specification of Neural Networks

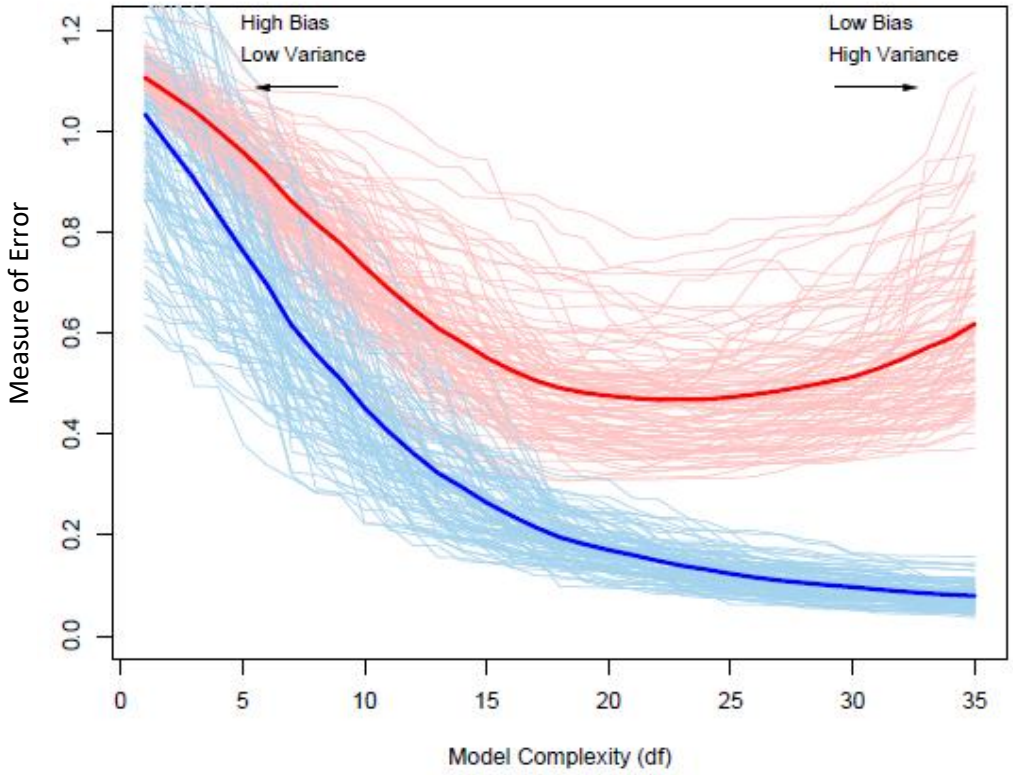
Determine the specification of a Neural Network via **hyperparameters** that decide the time and computational cost, define the structure of the neural network model, affect the model's prediction accuracy.

- # of Layers
  - 1 input layer, 1 hidden layer, 1 output layer is good enough for most of problems
  - If hidden layers ≥ 2, it is a deep neural network
- # of Neurons in each layer
  - # neurons in input and output layers are determined by the data
  - There is no clear rules to determine # neurons in hidden layers. A Rule of Thumb is 2/3 of sum of # neurons in input and output for a three-layer MLP model.
- Learning rate
- Stopping policy – cross validation

Cross Validation

A common issue about training a predictive neural network is the **overfitting problem** shown in the graph below.

- Generalization (Training) error – blue curve
- Prediction error with test data set – red curve



Overfitting problem

Cross Validation (cont'd)

Cross validation (*k*-fold) procedure:

- Randomly divide the dataset into three parts:
  - a training set,
  - a validation set, and
  - a test set

- The training set is used to fit/train the models
- The validation set is used to estimate prediction error for model selection, determine the hyperparameters
- The test set is used for assessment of final model
- A typical split might be 50% for training, and 25% each for validation and testing

Overfitting Issue

To overcome overfitting issue, two general strategies are employed when fitting neural networks.

- Slow Learning: the model is fit in a somewhat slow iterative fashion, using gradient descent with very small value of $\boldsymbol{\rho}$. The fitting process is then stopped when overfitting is detected.

- Regularization: penalties are imposed on the parameters, usually lasso or ridge method.

Regularization and SGD

Gradient descent usually takes many steps to reach a local minimum. In practice, there are a number of approaches for accelerating the process.

- **Stochastic Gradient Descent** (SGD)

When *n* is large, instead of summing gradients over all *n* observations, we can sample a small fraction or minibatch of them each time we compute a gradient step. This process is known as SGD and is the state of the art for learning deep neural networks.

- **Regularization** is essential to avoid overfitting by augmenting the objective function with a penalty term:

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)) + \lambda\sum_{j}\theta_j^2.$$

This is **ridge regularization** using the cross-validation approach to determine λ. **Lasso regularization** is also popular as an additional form of regularization, or as an alternative to ridge.

Regularization and SGD

- We can also use different values of λ for the groups of weights from different layers.

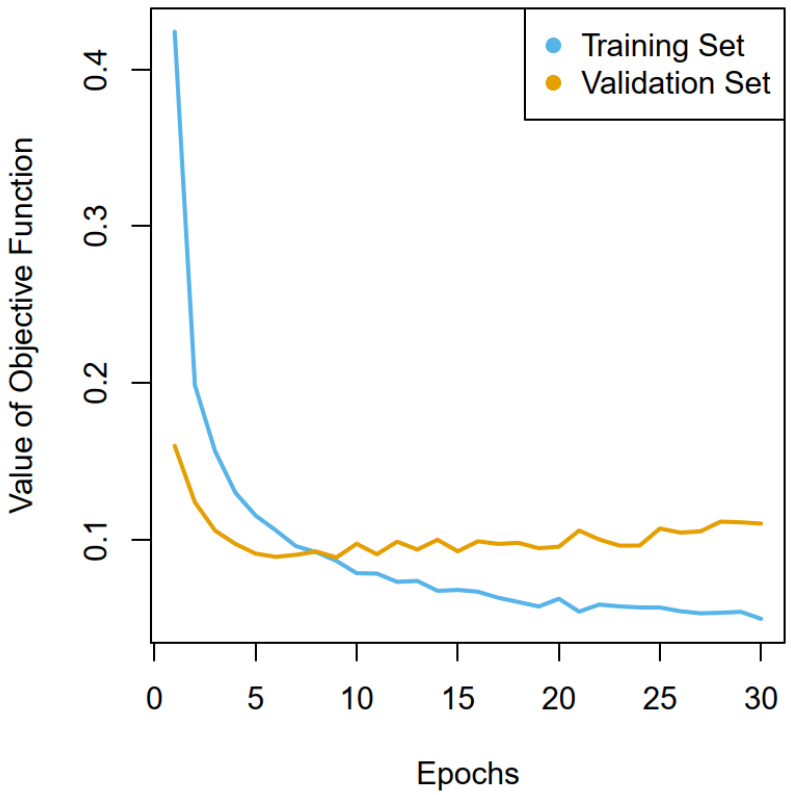
- The term **epochs** counts the number of times that the full training set has been processed.

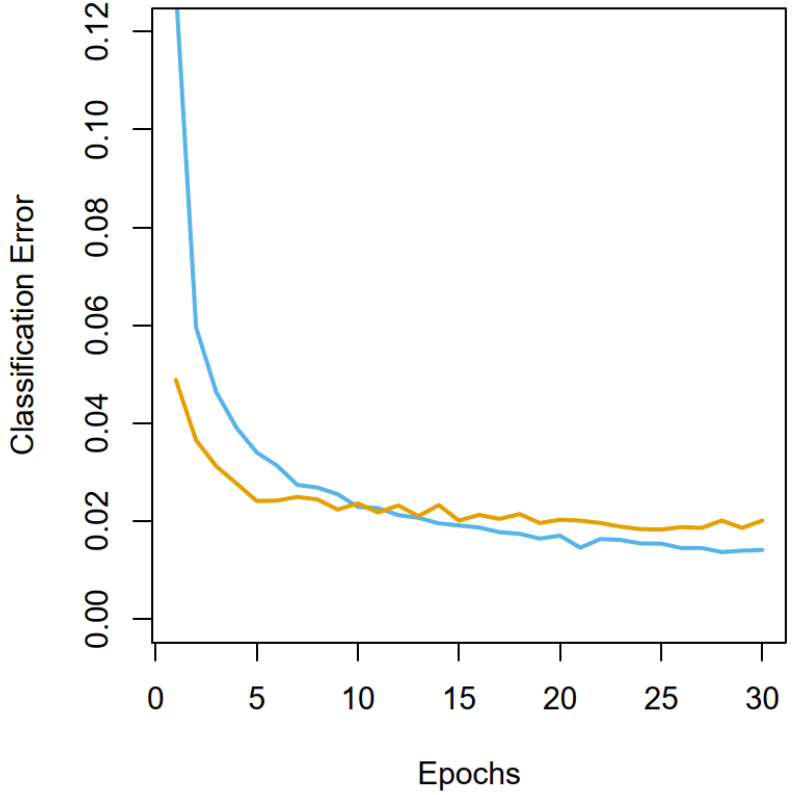
Example:

The multilayer network used in the digit recognition problem (MNIST) has over 235,000 weights, which is around four times the number of training examples. For this network, the **minibatch** size was 128 observations per gradient update. 20% of the 60,000 training observations were used as a validation set in order to determine when training should stop. So, in fact 48,000 observations were used for training, and hence there are $48{,}000/128 \approx 375$ minibatch **gradient updates per epoch**.

Regularization and SGD

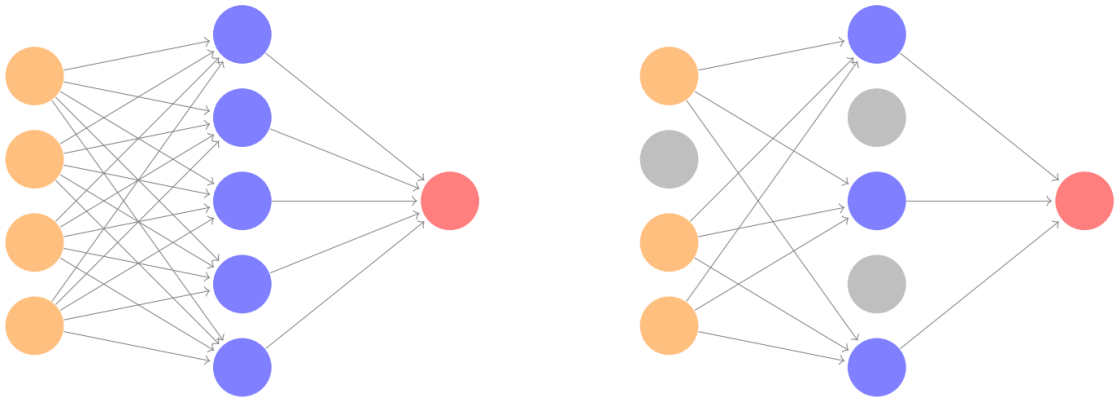
We see that the value of the validation objective actually starts to increase by 30 epochs, so **early stopping** can also be used as an additional form of regularization.





Dropout Learning

The idea is to randomly remove a fraction ϕ of the neurons in a layer when fitting the model. This is done separately each time a training observation is processed. This prevents nodes from becoming over-specialized and can be seen as a form of regularization. In practice dropout is achieved by randomly setting the activations for the "dropped out" neurons to zero, while keeping the architecture intact. (similar idea as random forest)



Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random and ignored in an instance of training.

Network Tuning

A Neural Network requires tuning some hyperparameters that all have an effect on the performance:

- The **number of hidden layers**, and the **number of units per layer**. Modern thinking is that the number of units per hidden layer can be large, and overfitting can be controlled via the various forms of regularization.
- **Regularization** tuning parameters. These include the **dropout rate** ϕ and the strength λ of lasso and ridge regularization and are typically set separately at each layer.
- Details of **stochastic gradient descent**. These include the **batch size**, the **number of epochs**, and if used, details of data augmentation.

Implementation of Neural Network

Use existing Python libraries to code neural networks

- SKlearn
- Tensorflow
- Keras
- PyTorch, etc.

The following example uses Sklearn library to train a Perceptron neural network with built-in data set “iris” and perform prediction.

Example with *Python*

Implementation of neural network with existing library

```
from sklearn import datasets
import numpy as np
iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
print('Class labels:', np.unique(y))
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1,
stratify=y)
```

Example with *Python* (cont'd)

```
sc = StandardScaler()
sc.fit(X_train)
sc.fit(X_test)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score


ppn = Perceptron(n_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)    ##This is training the model
y_pred = ppn.predict(X_test_std)    ##Test/Validating the model

print('Misclassified samples: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy: %.2f' % ppn.score(X_test_std, y_test))
```