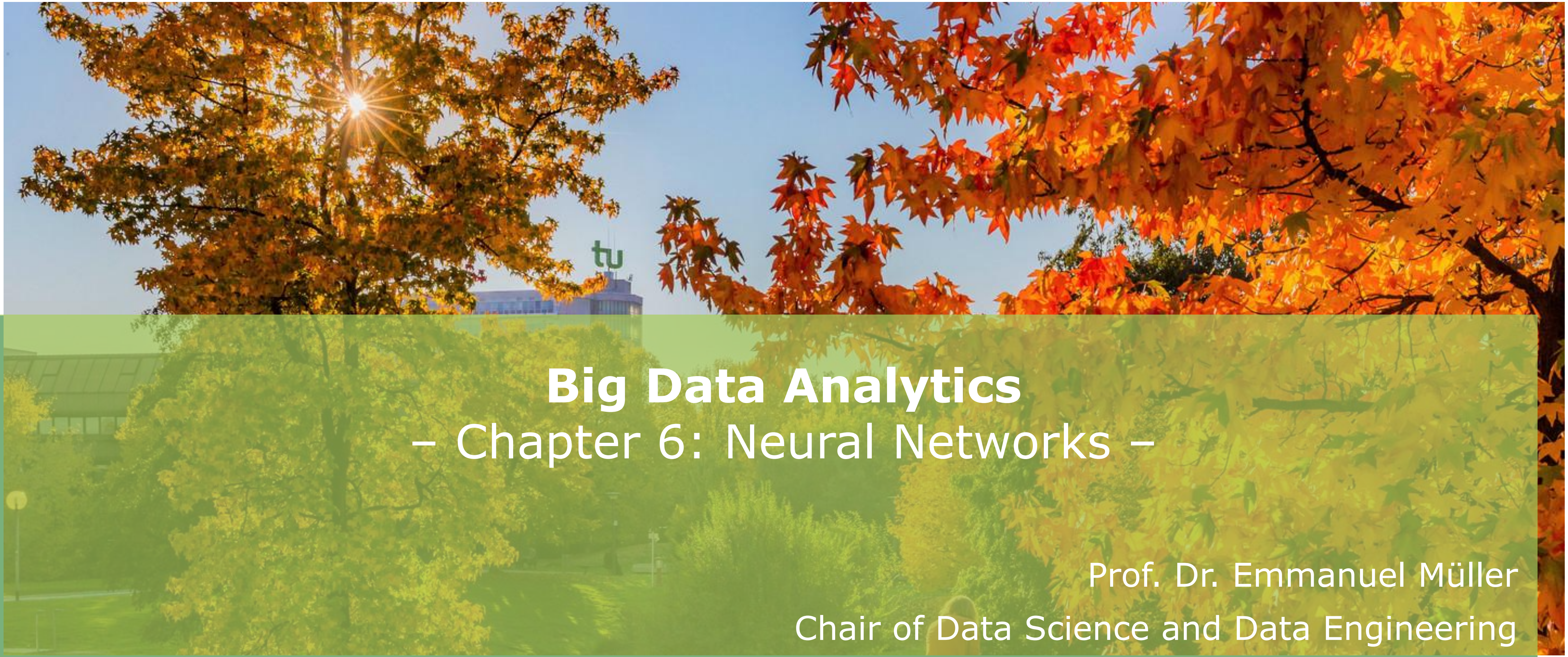
technische universität dortmund

Chair of Data Science and Data Engineering
Prof. Dr. Emmanuel Müller



Big Data Analytics
– Chapter 6: Neural Networks –

Prof. Dr. Emmanuel Müller
Chair of Data Science and Data Engineering

Additional Literature for this Chapter

© Jiawei Han, Micheline Kamber, Jian Pei: Data Mining – Concepts and Techniques, 4th ed., Morgan Kaufmann Publishers, 2023.

Slides credit: http://hanj.cs.illinois.edu/bk4/

Content Overview

- Neural Networks
  - Basic concepts
  - Improve Training of Deep Learning Models
- Applications of Neural Networks
  - Convolutional Neural Networks
  - Recurrent Neural Networks
  - Graph Neural Networks
- Interpretable Machine Learning

Chapter 10. Deep Learning

- Basic Concepts
- Improve Training of Deep Learning Models
- Convolutional Neural Networks
- Recurrent Neural Networks
- Graph Neural Networks
- Summary

Inspired by a biological neuron

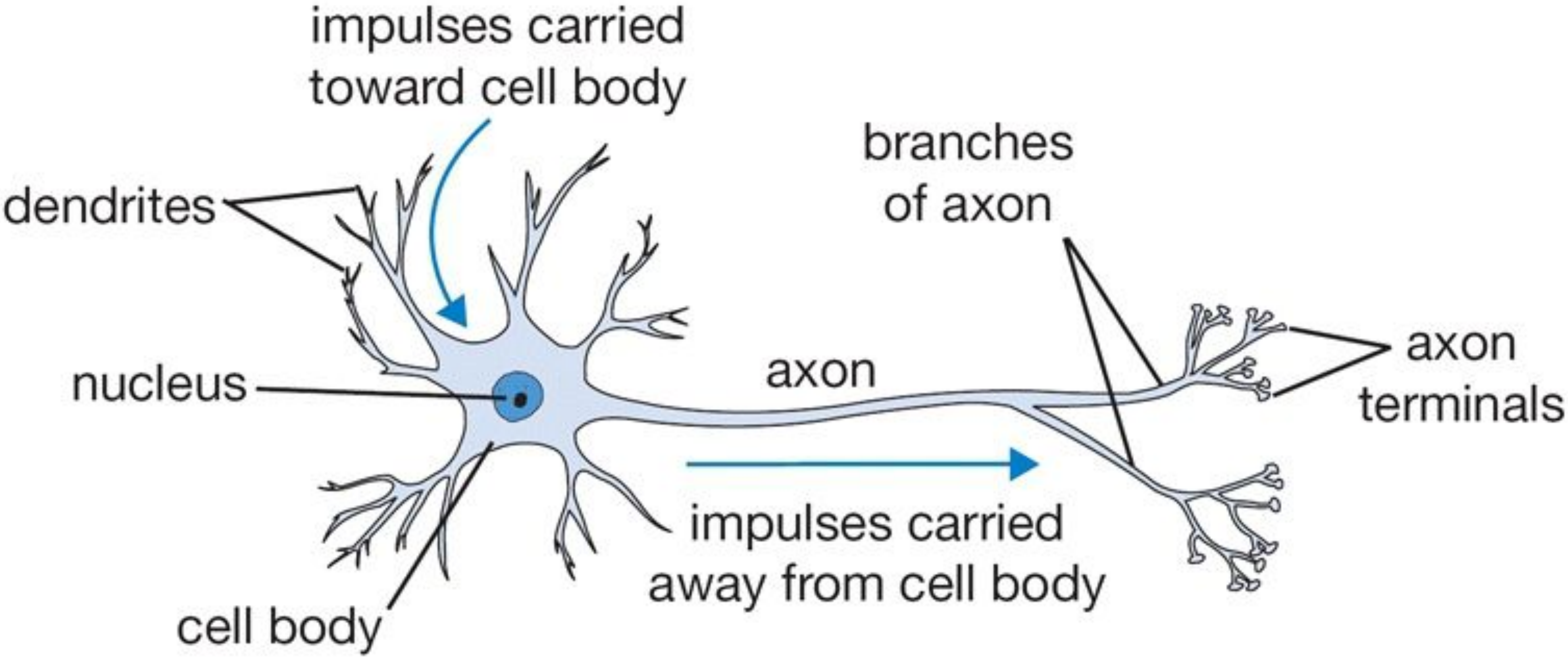
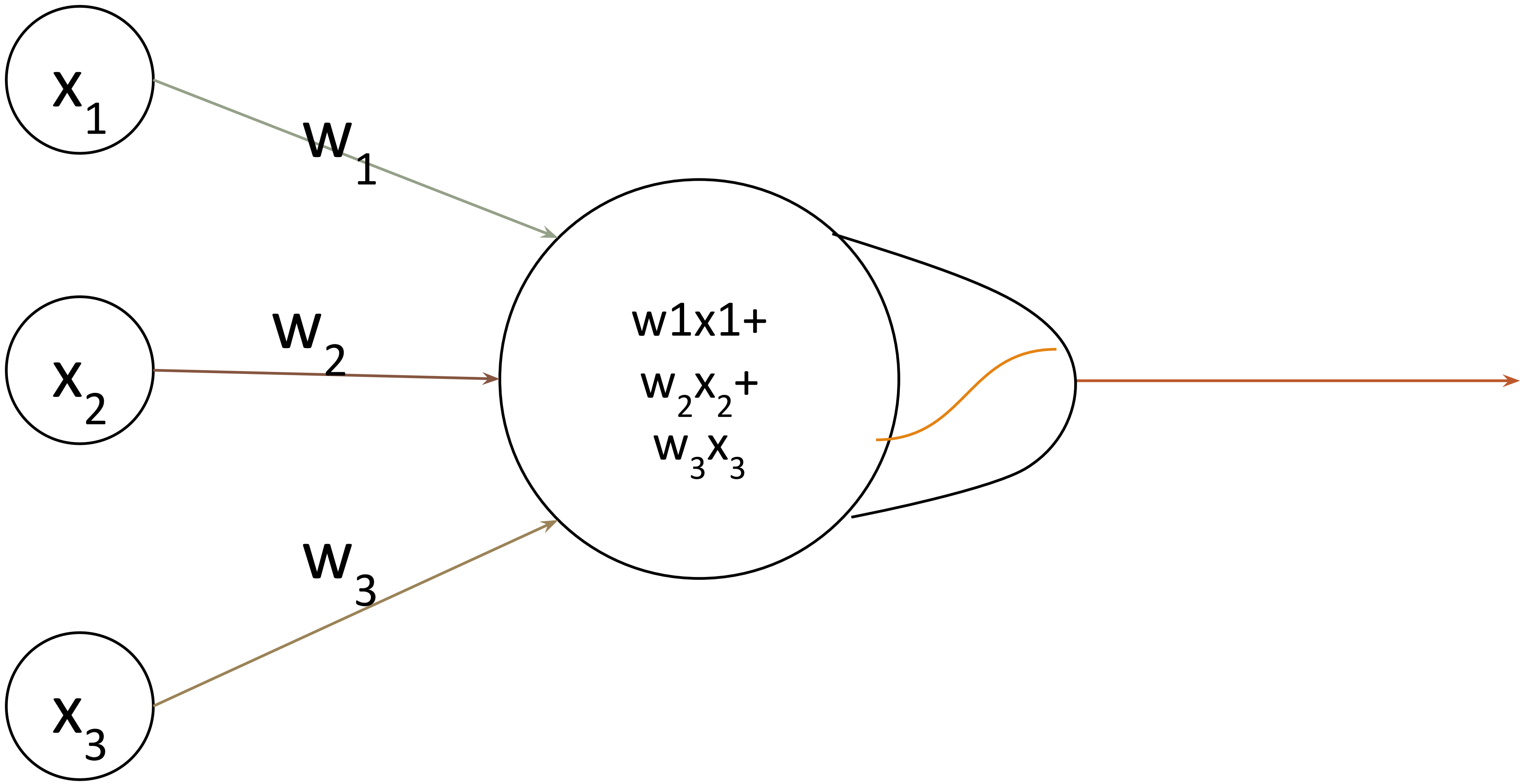
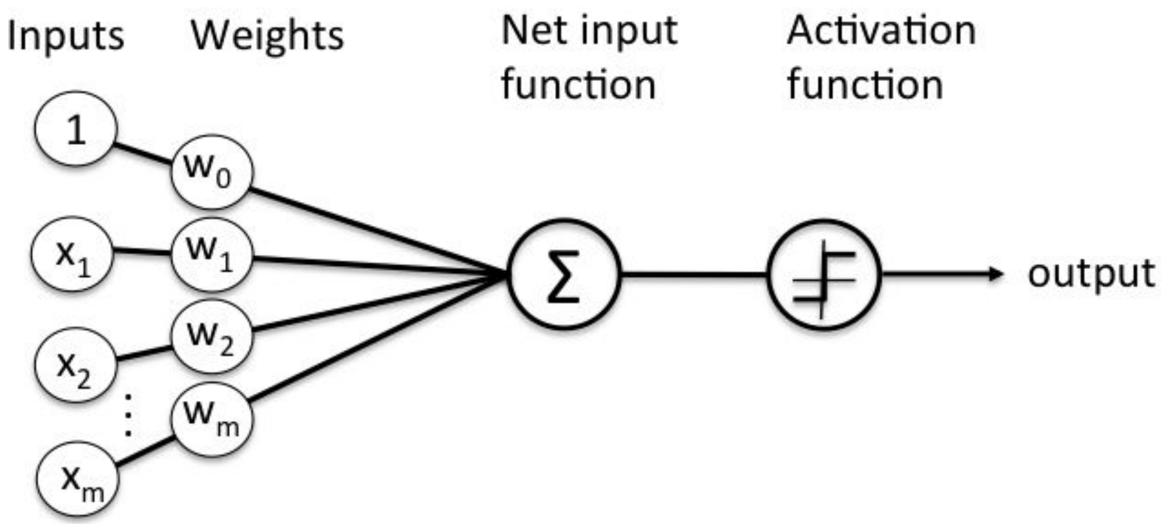


Image credit:
http://cs231n.github.io/neural-networks-1/

How to model a single neuron?



Perceptron: Predecessor of a Neural Network



$$\hat{y} = f(\boldsymbol{W}^T\boldsymbol{x}) = f(\sum W_i x_i + b)$$

Activation Function
Adding non-linearity to the model

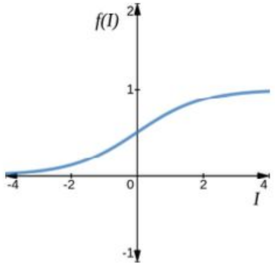
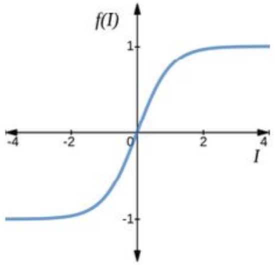
Weights

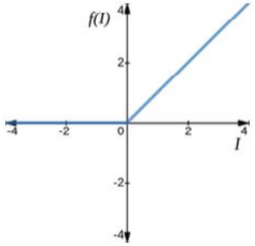
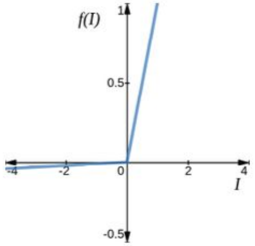
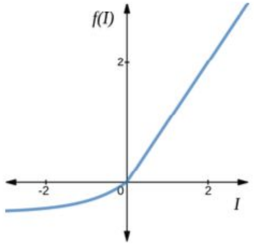
Bias
A measure of how easy it is to get the perceptron to output a 1

- Computes a weighted sum of inputs
- Invented in 1957 by Frank Rosenblatt. The original perceptron model does not have a non-linear activation function

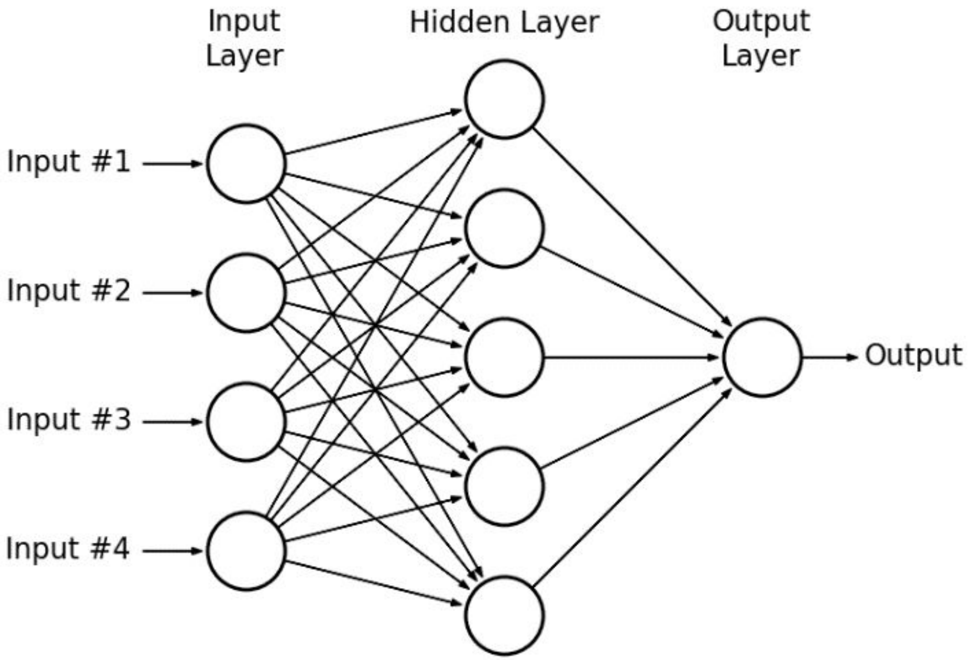
Perceptron: Predecessor of a Neural Network

- Examples of activation functions

Sigmoid	$\frac{1}{1+e^{-I}}$	
Tanh	$\frac{e^{I}-e^{-I}}{e^{I}+e^{-I}}$	

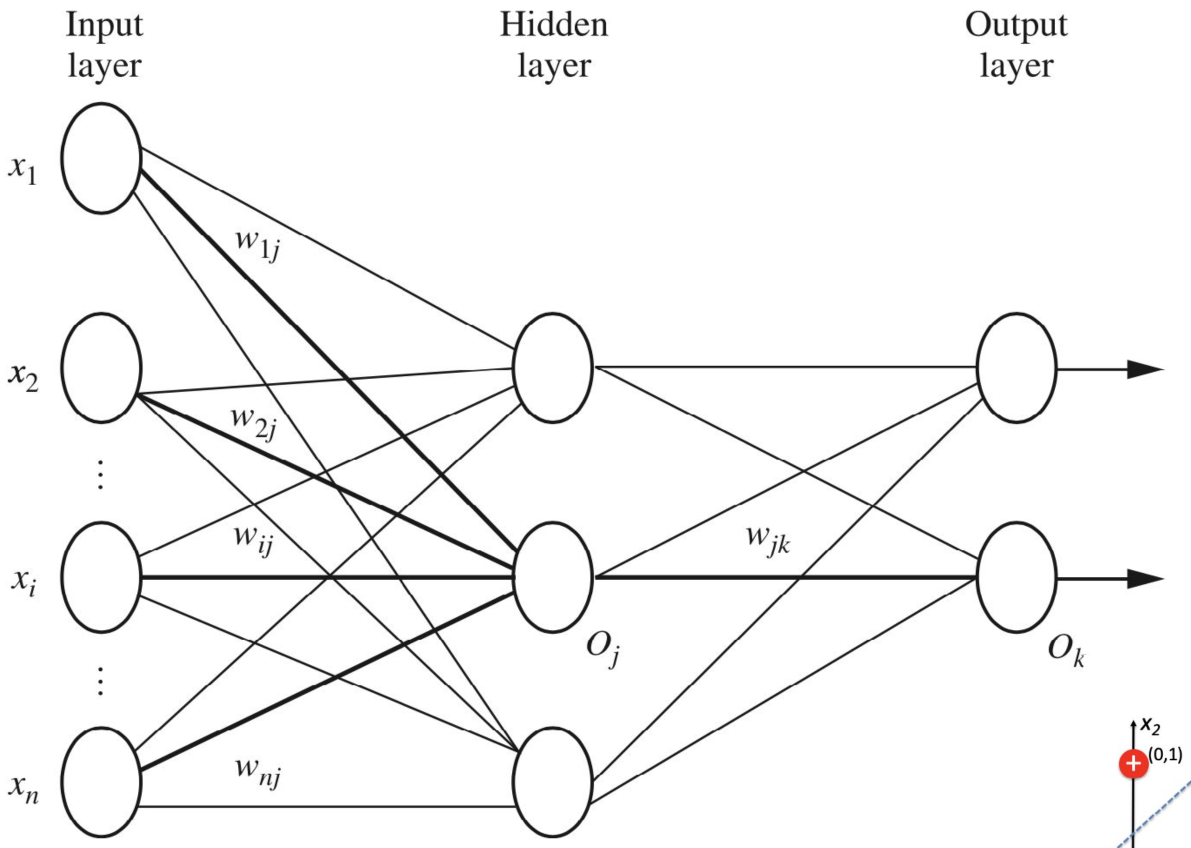
ReLU	$\begin{cases} 0, I \leq 0 \\ I, I > 0 \end{cases}$	
Leaky ReLU	$\begin{cases} 0.01 \times I, I < 0 \\ I, I \geq 0 \end{cases}$	
ELU	$\begin{cases} \alpha(e^{I} - 1), I \leq 0 \\ I, I > 0 \end{cases}$	

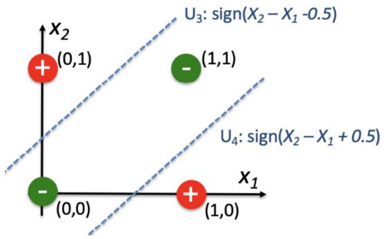
Multilayer Perceptron (MLP)



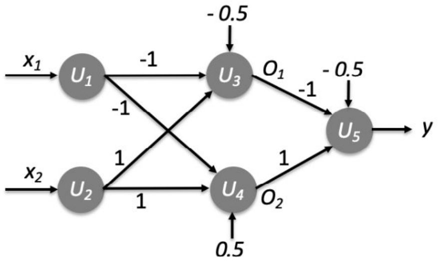
- Stacking multiple layers of perceptrons (adding hidden layers) makes a multilayer perceptron (MLP)
- MLP can engage in sophisticated decision making, where perceptrons fail
  - E.g. XOR problem
- Try it: http://playground.tensorflow.org

Feed-forward Neural Networks

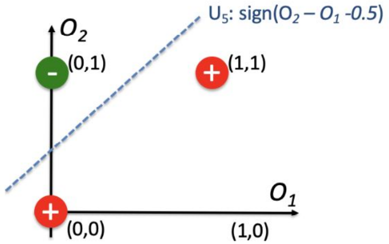




(a) Input 4 tuples



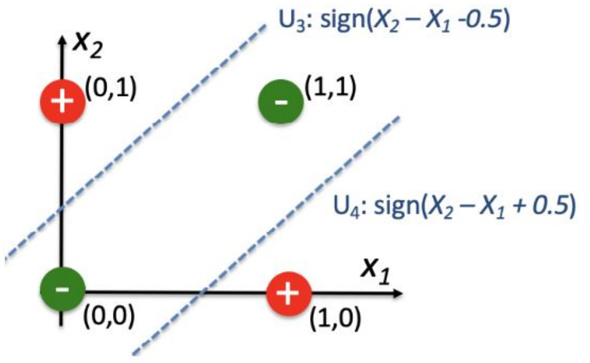
(b) MLP



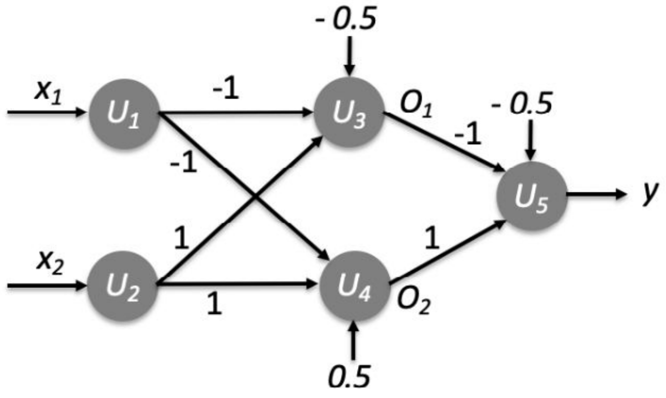
(c) Outputs of two hidden units

- Why multiple layers
  - Automatic feature/representation learning
  - Learn complicate (nonlinear) mapping function

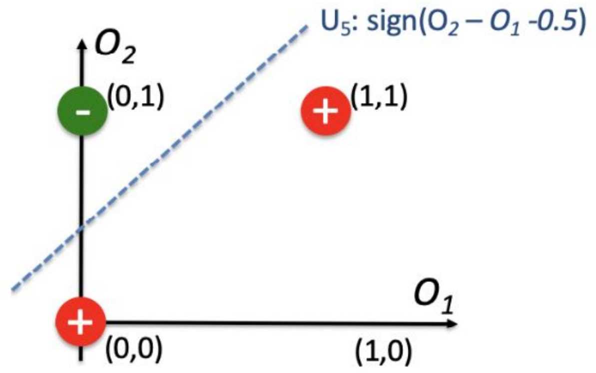
Feed-forward Neural Networks



(a) Input 4 tuples



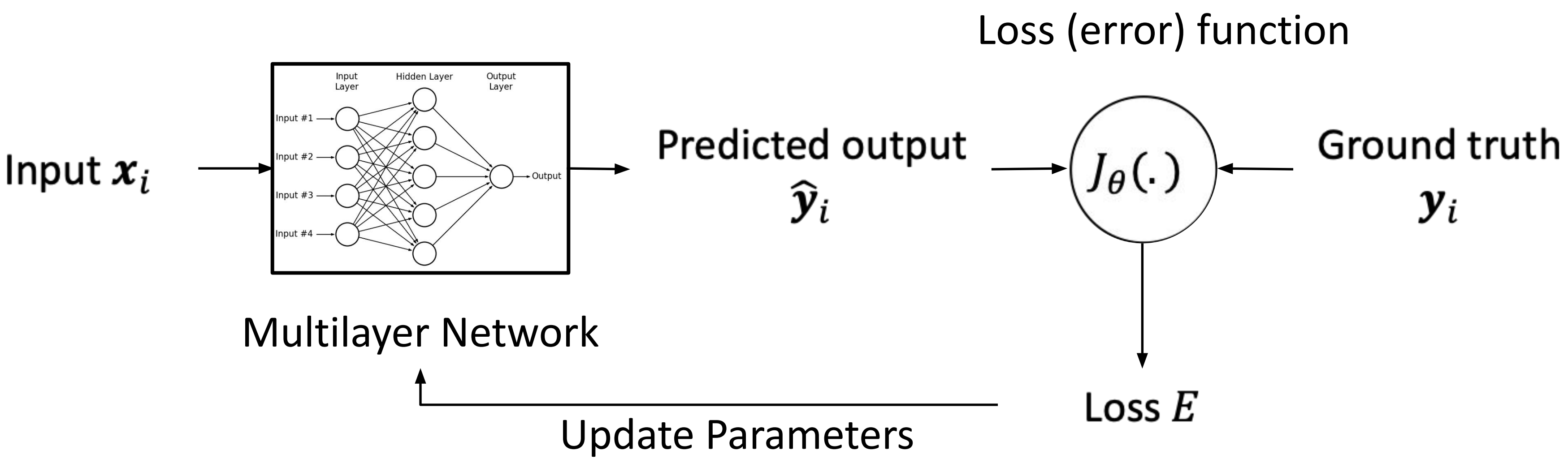
(b) MLP



(c) Outputs of two hidden units

X_1	X_2	O_1 = sign ($X_2 - X_1$-0.5)	O_2 = sign ($X_2 - X_1$ + 0.5)
0	0	0	+1
1	1	0	+1
1	0	0	0
0	1	1	1

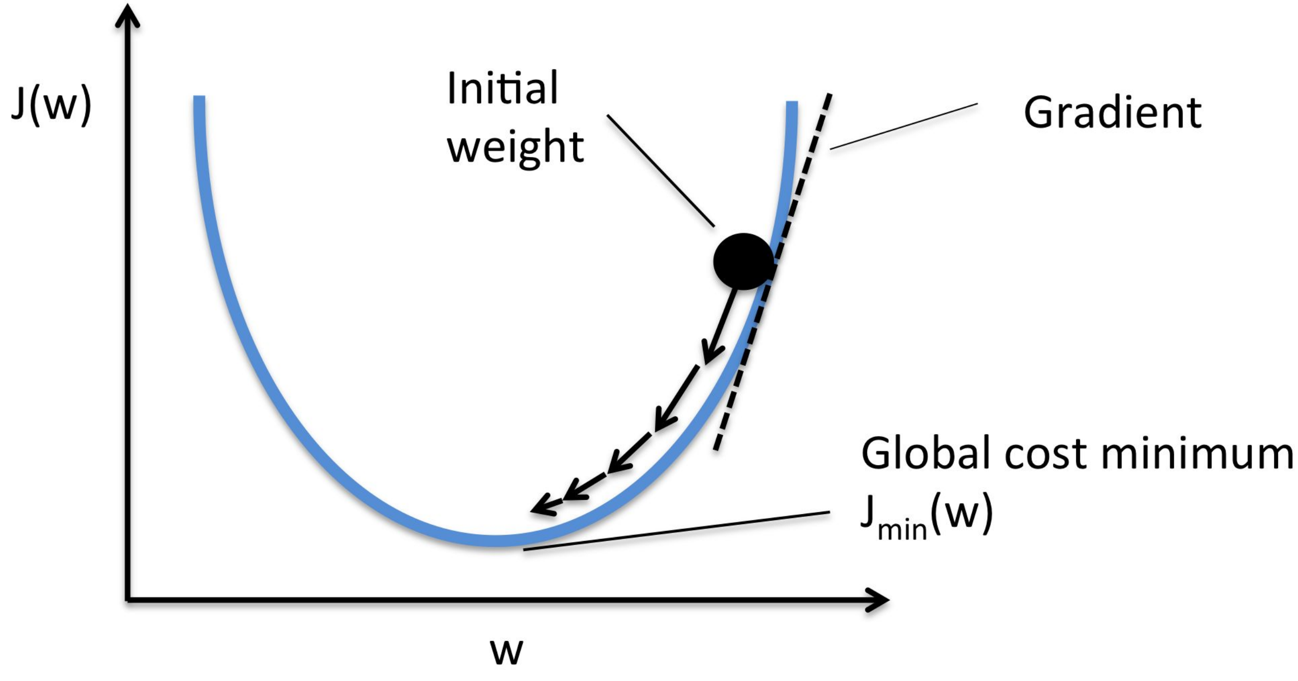
Learning NN Parameters



- **Gradient Descent Algorithm**
- **Input**: Training sample $\boldsymbol{x}_i$ and its label $\boldsymbol{y}_i$

1. **Feed Forward**: Get prediction $\hat{\boldsymbol{y}}_i = \mathrm{MLP}(x_i)$, and loss $E = J(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$
2. **Compute Gradient**: For each parameter θ_j (weights, bias), compute its gradient $\frac{\partial}{\partial \theta_j} J_\theta$
3. **Update Parameter**: $\theta_j = \theta_j - \eta \cdot \frac{\partial}{\partial \theta_j} J_\theta$

Empirical Explanation of Gradient Descent

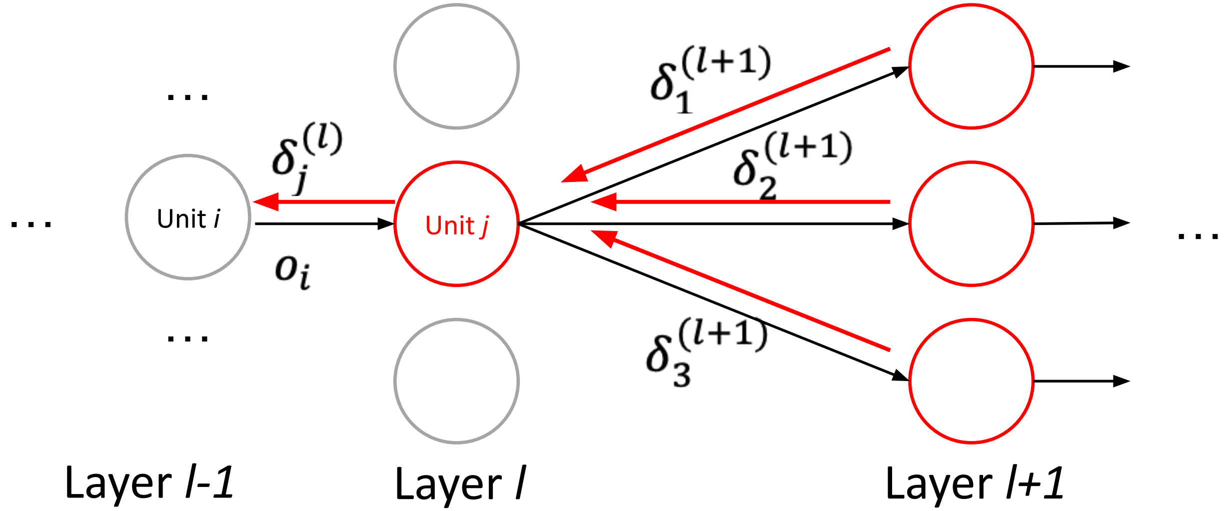


$$\theta_j = \theta_j - \eta \cdot \frac{\partial}{\partial \theta_j} J_\theta$$

η is the ***learning rate***, which controls the 'step size' of the optimization

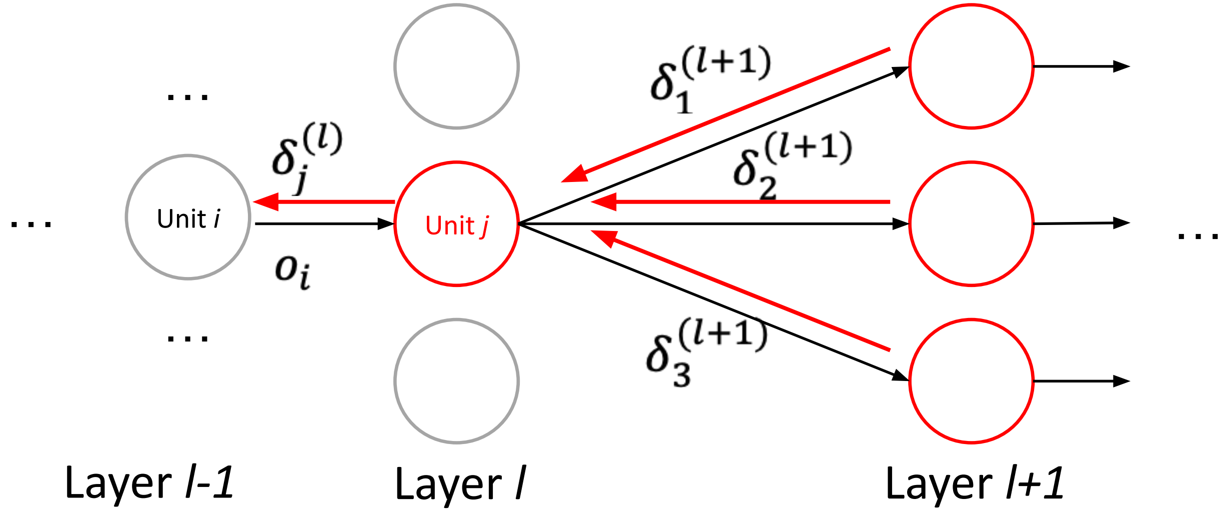
- Our objective is to minimize the loss function J, which is a function of the model parameters
- A gradient measures how much the output of a function changes if you change the inputs a little bit
- We update the parameters, based on their gradients, so that the loss function is going 'downhill'

Gradient Computation: Backpropagation



- The gradient of w_{ij} in the lth layer (corresponding to unit j in layer l, connected to unit i in layer l-1) is a function of
  - All 'error' terms from layer l+1 $\delta_k^{(l+1)}$ -- An auxiliary term for computation, not to be confused with gradients
  - Output from unit i in layer l-1 (input to unit j in layer l) -- Can be stored at the feed forward phase of computation

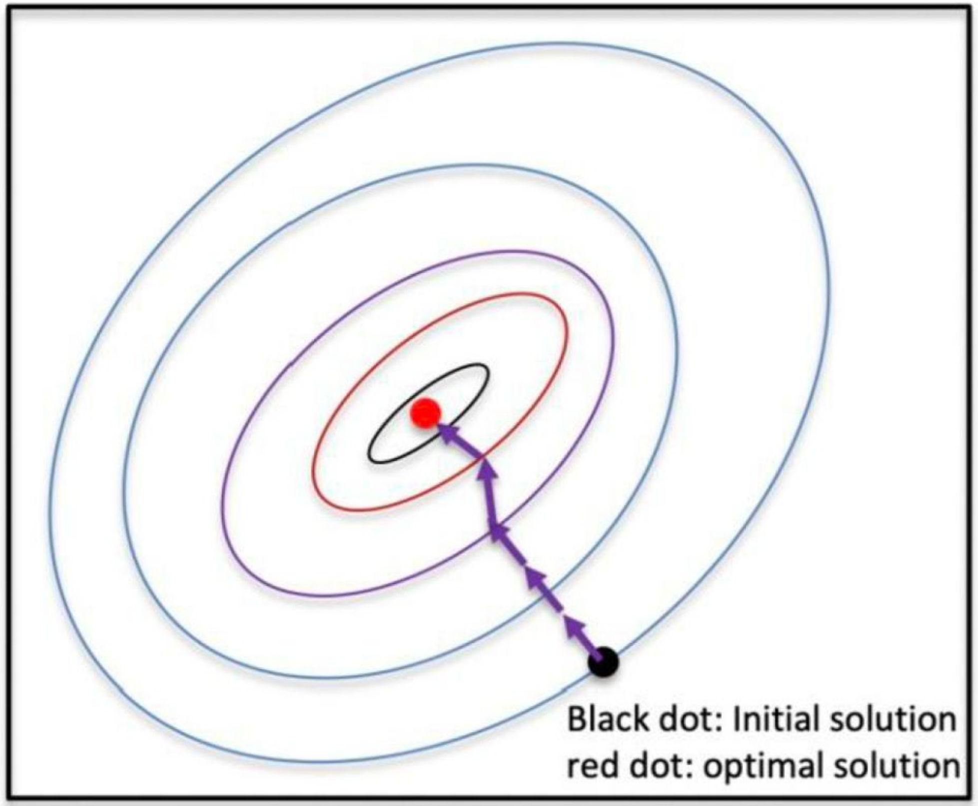
Gradient Computation: Backpropagation



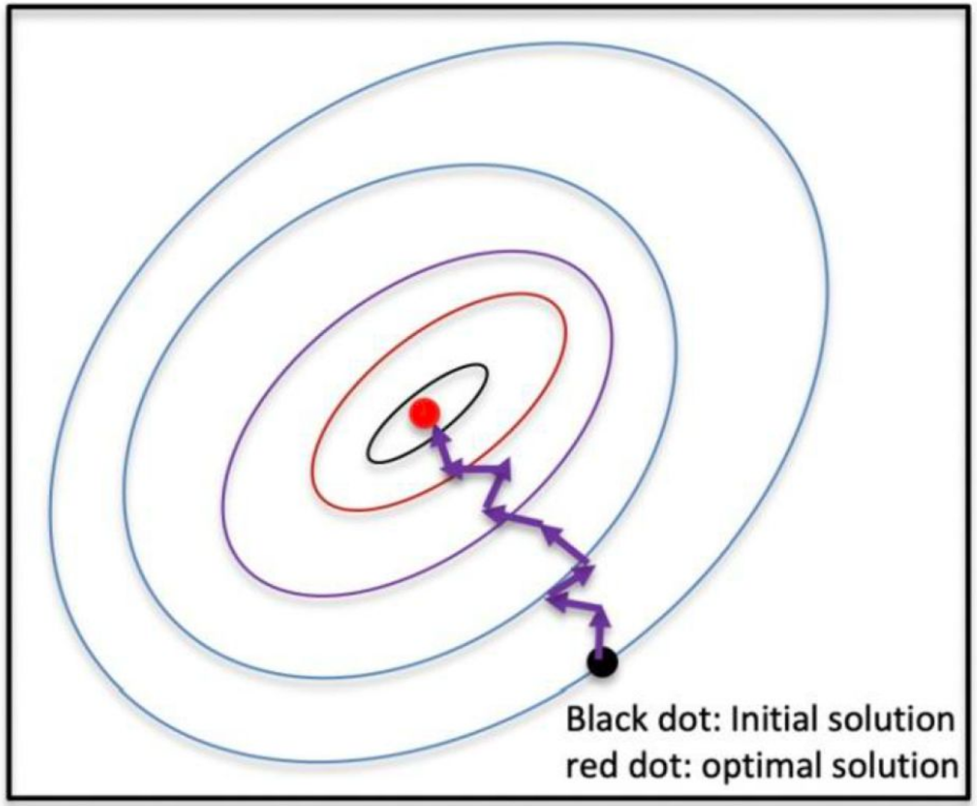
- The 'error' terms $\delta_j^{(l)}$ is a function of
  - All $\delta_k^{(l+1)}$ in the layer l+1, if layer l is a hidden layer
  - The overall loss function value, if layer l is the output layer
- We can compute the error at the output, and distribute backwards throughout the network's layers (backpropagation)

Each forward pass and backpropagation over all training samples is called one **epoch**.

Gradient descent



(a) Gradient descent

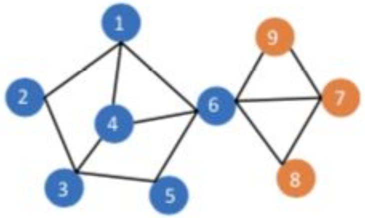
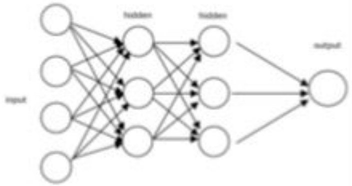
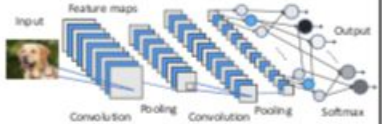
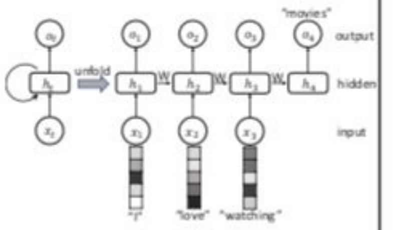
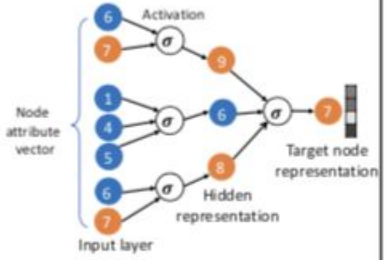


(b) Stochastic gradient descent

From Neural Networks to Deep Learning

- **Deep Learning** refers to training (deep) neural networks with many layers
  - More neurons, more layers
  - More complex ways to connect layers
- Deep Learning has some major advantages, making it popular
  - Tremendous improvement of performance in many tasks
    - Image recognition, natural language processing, AI game playing…
  - Requires no (or less) feature engineering, making end-to-end models possible
- Several factors lead to deep learning's success
  - Very large data sets
  - Massive amounts of computation power (GPU acceleration)
  - Advanced neural network structures and tricks
    - Convolutional neural networks, recurrent neural networks, graph convolutional networks
    - Dropout, ReLU, residual connection, …

Overview of Typical Deep Learning Architectures

Data type	Multi-dimensional Features: credit rating, account balance $x = (4.5, 500, 3, 5)$ #deposits, #withdraws	Grid 	Sequence x = "I love watching movies."	Graph 
DL Archi-tecture	Feed-forward Network 	CNN 	RNN 	GNN 

Chapter 10. Deep Learning

- Basic Concepts
- Improve Training of Deep Learning Models
- Convolutional Neural Networks
- Recurrent Neural Networks
- Graph Neural Networks
- Summary

Techniques to Improve Deep Learning Training

- Key Challenges for Training Deep Learning Models
- Responsive Activation Functions
- Adaptive Learning Rate
- Dropout
- Pretraining
- Cross Entropy

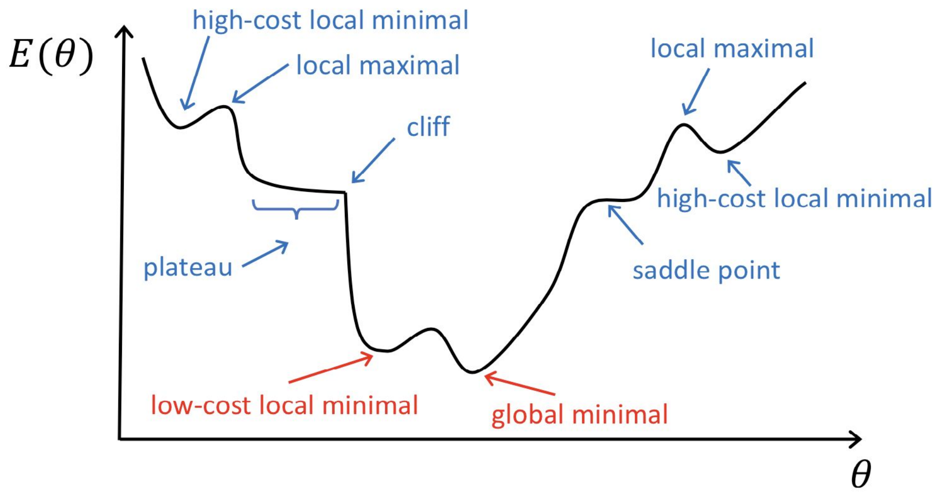
Key Challenges

- **Optimization Problem in Deep Learning**
  - Minimize the (approximated) training error *E*
  - (Stochastic) gradient descent to find the model parameter

$$E(\boldsymbol{\theta}) = \frac{1}{m} \sum_{l=1}^{m} \mathrm{Loss}(\hat{T}(X^l, \boldsymbol{\theta}), T^l)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_t$$

- **Challenge 1: Optimization**
  - *E* is non-convex in general
  - How to find a high-quality local optima



- **Challenge 2: Generalization**
  - What we do: minimize (approximated) training error
  - What we really want: minimize the generalization error
  - How to mitigate over-fitting

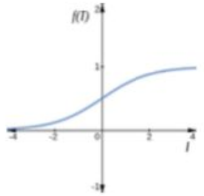
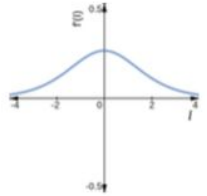
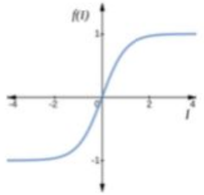
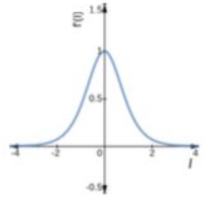
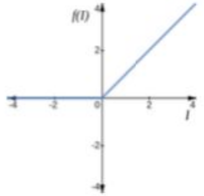
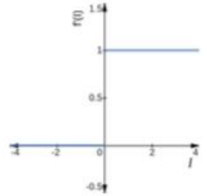
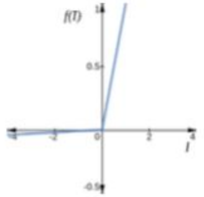
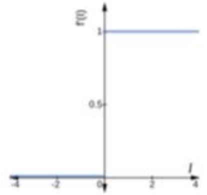
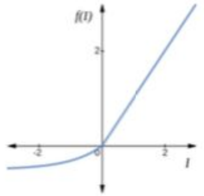
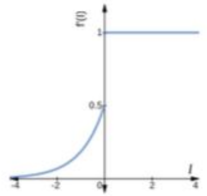
Responsive Activation Function

- **Saturation of Sigmoid Activation Function**
  - The output $O = \sigma(I) = \frac{1}{1+e^{-I}} \in (0,1)$
  - The derivative $\frac{\partial O}{\partial I} = O(1-O)$
  - The error of an output unit $\delta_j = \boxed{O_j(1-O_j)}(O_j - T_j)$

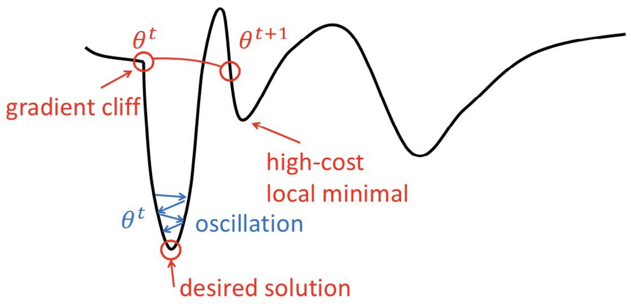
decaying

- **Saturation of Sigmoid Activation Function**
  - If $O_j \approx 1$ or $O_j \approx 0$, both derivative and error will be close to 0
  - Further exacerbated due to backpropagation -> gradient vanishing -> B.P. is stuck or takes long time to terminate
- **A More Responsive Activation Function: Rectified Linear Unit (RELU)**
  - The output *O=f(I) = I* if *I>0, O=0* otherwise
  - The gradient: $\frac{\partial O}{\partial I} = 1$ if $I > 0$ and $\frac{\partial O}{\partial I} = 0$
  - The error: 0 if the unit is inactive (*I<0*), otherwise, aggregate all error terms from the units in the next higher layer the unit is connected to, w/o decaying -> avoid gradient vanishing

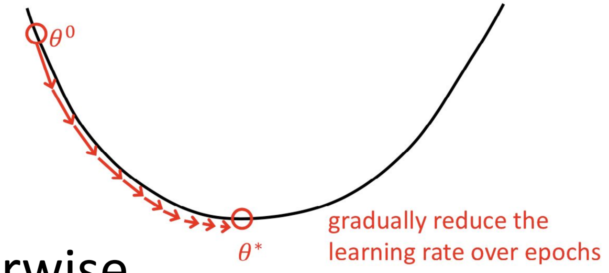
Activation Functions

Name	Definition $(f(I))$	Plot	Derivative of $f(I)$	Plot
Sigmoid	$\frac{1}{1+e^{-I}}$		$f(I)(1-f(I))$	
Tanh	$\frac{e^{I}-e^{-I}}{e^{I}+e^{-I}}$		$1-f(I)^2$	
ReLU	$\begin{cases} 0, I \leq 0 \\ I, I > 0 \end{cases}$		$\begin{cases} 0, I \leq 0 \\ 1, I \geq 0 \end{cases}$	
Leaky ReLU	$\begin{cases} 0.01 \times I, I < 0 \\ I, I \geq 0 \end{cases}$		$\begin{cases} 0.01, I < 0 \\ 1, I \geq 0 \end{cases}$	
ELU	$\begin{cases} \alpha(e^{I}-1), I \leq 0 \\ I, I > 0 \end{cases}$		$\begin{cases} \alpha e^{I}, I \leq 0 \\ 1, I > 0 \end{cases}$	

Adaptive Learning Rate



- **Stochastic gradient descent to find a local optima**
  - Default choice for η: a fixed, small positive constant $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_t$
  - Problems: slow progress, or jump over 'gradient cliff', or oscillation

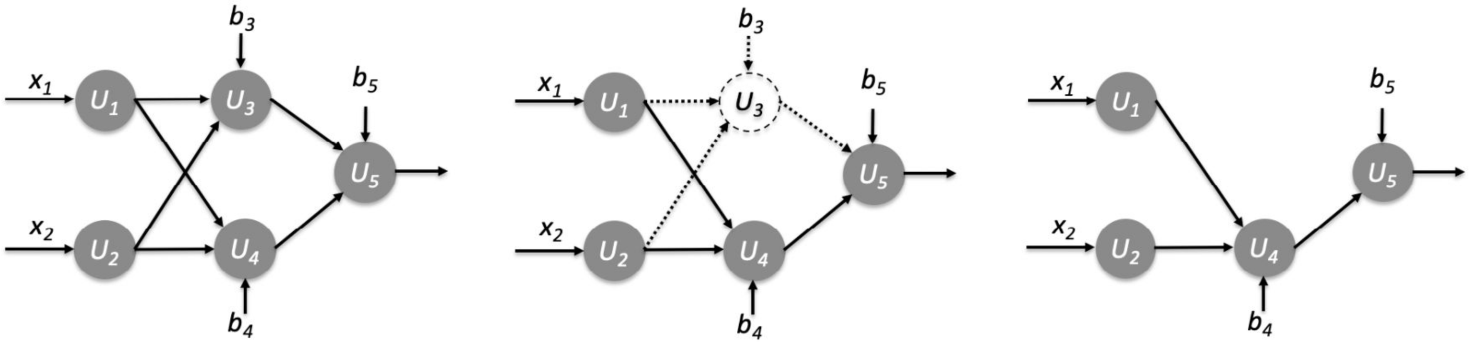


- **Adaptive learning rate: let η change over epoch**
  - Strategy #1: $\eta_t = \frac{1}{t}\eta_0$
  - Strategy #2: $\eta_t = (1 - \frac{t}{T})\eta_0 + \frac{t}{T}\eta_\infty$ if $t \leq T$ and $\eta_t = \eta_\infty$ otherwise
    - Intuitions: smaller adjustment as algorithm progresses
    - $e.g.,\ \ \eta_0 = 0.9;\ \eta_\infty = 10^{-9}$
  - Strategy #3 (AdaGrad): $\eta_t = \frac{1}{\rho + r_i}\eta_0 \quad r_i = \sqrt{\sum_{k=1}^{t-1} \boldsymbol{g}_{i,k}^2}$
    - Intuition: The magnitude of gradient g_t : indicator of the overall progress
    - $e.g., \rho = 10^{-8}$
  - Strategy #4 (RMSProp): exponential decaying weighted sum of squared historical gradients

Dropout

- **The Purpose of Dropout: to Prevent Overfitting**
- **How does it works?**
  - At each epoch (ρ dropout rate, e.g., $\rho = 0.5$)
    - randomly dropout some non-output units
    - Perform B.P. on the dropout network
  - Scale the final model parameters

$$\boldsymbol{\theta}^* \leftarrow \rho \cdot \boldsymbol{\theta}^*$$

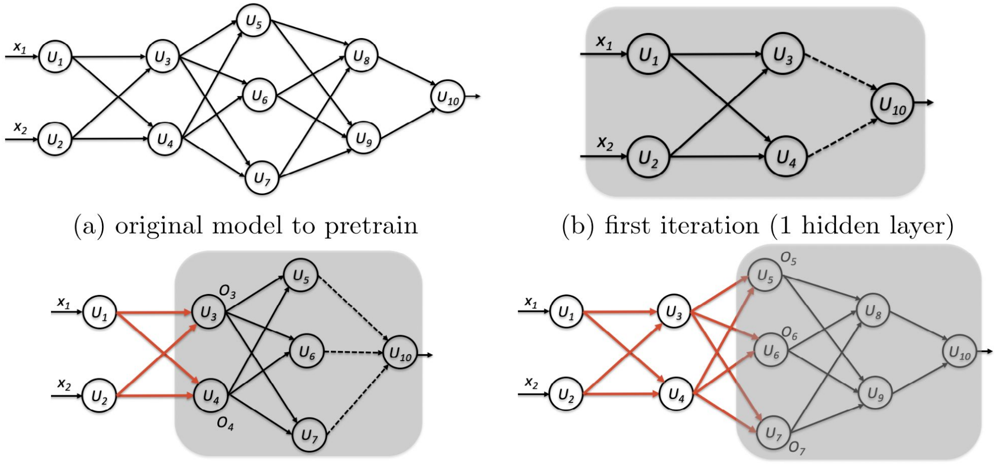


(a) Original network (b) Dropout U_3 (c) Dropout network

- **Why does Dropout Work?**
  - Dropout can be viewed as a regularization technique
    - Force the model to be more robust to noise, and to learn more generalizable features
  - Dropout vs. Bagging
    - Bagging: Each base model is trained *independently* on a bootstrap sample
    - Dropout: the model parameters of the current dropout network are updated based on that of the previous dropout network(s)

Pretraining

- **Pretraining:** the process of initializing the model in a suitable region
- **Greedy supervised pretraining**
  - pre-set the model parameters layer-by-layer in a greedy way
  - Start with simple model, add one additional layer at a time,
  - pretrain the parameters of the newly added layer, while fixing for other layers
  - Each time, equivalent to training a two-layered network
  - Followed up a fine—tuning process
- **Other Pretraining Strategies**
  - Unsupervised pretraining: based on auto-encoder
  - Hybrid strategy
- **Pretraining for transfer learning**

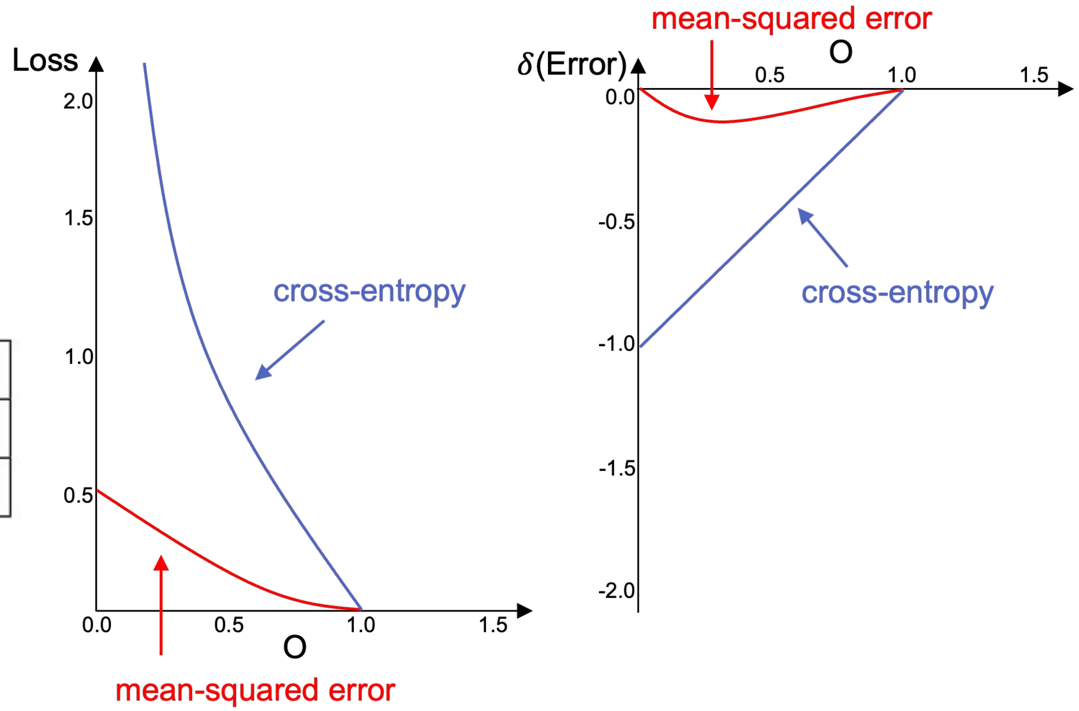


(a) original model to pretrain (b) first iteration (1 hidden layer)
(c) second iteration (2 hidden layers) (d) third iteration (3 hidden layers)

Cross Entropy

- **Measure the disagreement between the actual (T) and predicted (O) target values**
  - For regression: mean-squared error
  - For (binary) classification: cross-entropy
- **Mean-squared error vs. Cross-entropy**

	Mean-squared error	Cross entropy
Loss	$\frac{1}{2}(T-O)^2$	$-T\log O-(1-T)\log(1-O)$
Error δ	$O(1-O)(O-T)$	$O-T$



Assume positive example (*T=1*)

- **Cross-entropy for Multiclass Problem**
  - Actual target $T=(T_1,T_2,...,T_C)$
  - Predicted output $O=(O_1,O_2,...,O_C)$

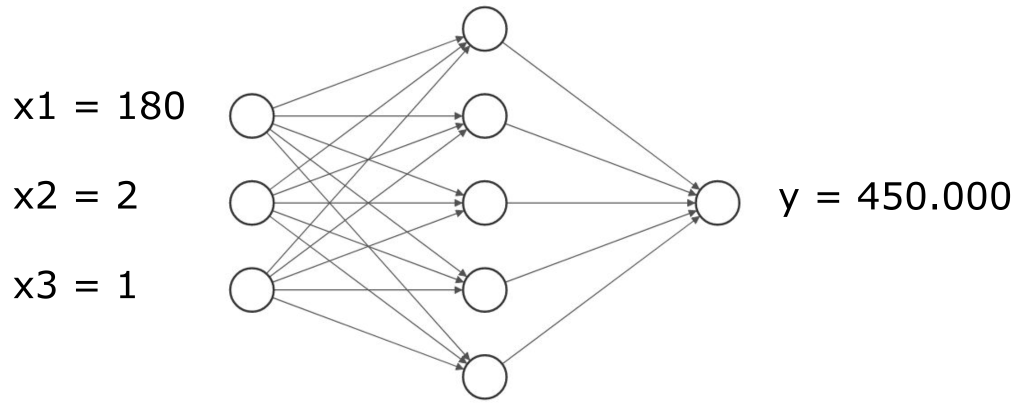
$$\mathrm{Loss}(T,O)=-\sum_{j=1}^{C}T_j\log O_j$$

Example: House Price Prediction

Task: predicting house prices with an MLP (optimize Mean-squared error)

- Input layer (3 neurons): each represents a feature in the input data
- Hidden layer (5 neurons): learned weights capture relationships within the data
- Output layer (1 neuron): produces continuous output prediction based on learned patterns

X_1 = Area	X_2 = # Bedrooms	X_3 = Air-conditioning	y = House Price
80	1	No (0)	250.000
200	3	Yes (1)	600.000
…	…	…	…
180	2	Yes (1)	?



Example: House Price Classification

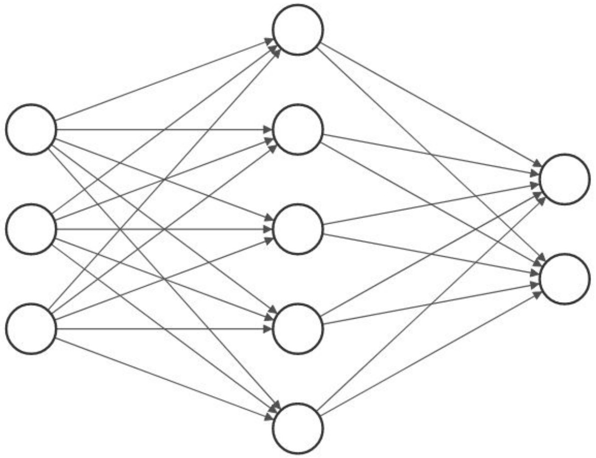
Transformation into binary classification problem:

- Discretize house prices into C1 = Affordable, C2 = Too Expensive (e.g., use fixed threshold)
- Adjust MLP architecture:
  - One output neuron for each class, providing raw class scores (logits)
  - Softmax function turns logits into class probabilities
  - Optimize Cross-entropy w.r.t. class probabilities

x1 = 180

x2 = 2

x3 = 1



Logits

z1= 1.3

z2 = 2.6

Softmax

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Class probabilities

p1 = 0.214

p2 = 0.786

Prediction as too expensive with probability 78.6%!