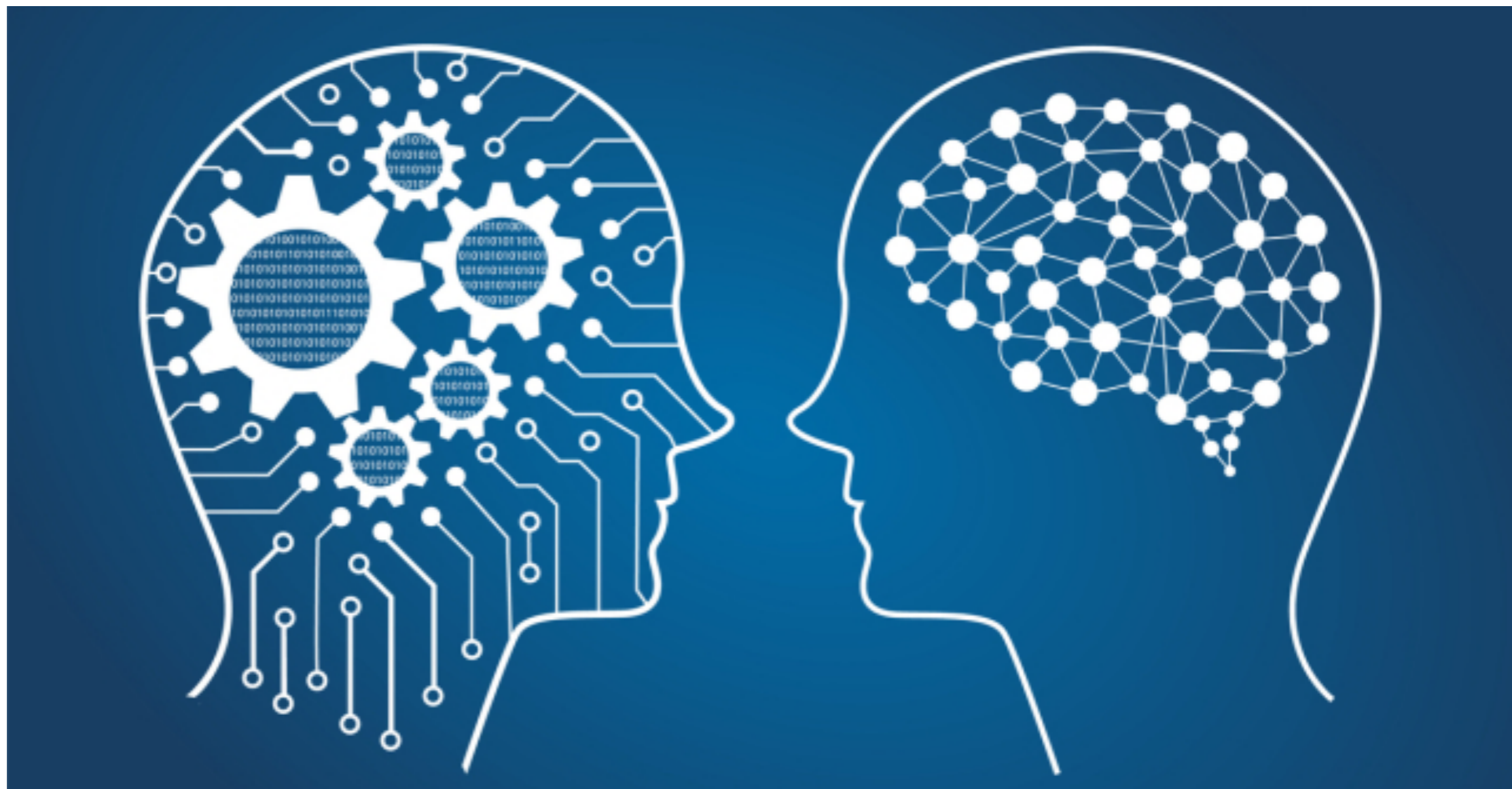




iit

ISTITUTO ITALIANO
DI TECNOLOGIA



An introduction to machine learning

Tor Vergata
10-11 October 2018

Edoardo Milanetti

edoardo.milanetti@uniroma1.it

References

- i. **Machine Learning - IBM** - Judith Hurwitz and Daniel Kirsch
- ii. **Introduction to Machine Learning - 2th edition** - Ethem Alpaydin
- iii. **Machine Learning** - Tom M. Mitchell
- iv. **Understanding machine learning - theory and algorithms** - Shai Shalev-Shwartz and Shai Ben-David
- v. **Neural Networks and Statistical Learning** - K.-L. Du and M. N. S. Swamy
- vi. **Introduction to Machine Learning** - Nils J. Nilsson
- vii. **A high-bias, low-variance introduction to Machine Learning for physicists** - Pankaj Mehta, Ching-Hao Wang, Alexandre G. R. Day, and Clint Richardson
- viii. **Data Clustering Theory, Algorithms, and Applications** - Guojun Gan, Chaoqun Ma , Jianhong Wu
- ix. **Analysis of Recurrent Neural Networks with Application to Speaker Independent** - Prof. Dr.-Ing. O.E. Herrmann Ir. L.P.J. Veelenturf Dr. Ir. S.H. Gerez

References

- i. **Machine Learning - IBM** - Judith Hurwitz and Daniel Kirsch
- ii. **Introduction to Machine Learning - 2th edition** - Ethem Alpaydin
- iii. **Machine Learning** - Tom M. Mitchell
- iv. **Understanding machine learning - theory and algorithms** - Shai Shalev-Shwartz and Shai Ben-David
- v. **Neural Networks and Statistical Learning** - K.-L. Du and M. N. S. Swamy
- vi. **Introduction to Machine Learning** - Nils J. Nilsson
- vii. **A high-bias, low-variance introduction to Machine Learning for physicists** - Pankaj Mehta, Ching-Hao Wang, Alexandre G. R. Day, and Clint Richardson
- viii. **Data Clustering Theory, Algorithms, and Applications** - Guojun Gan, Chaoqun Ma , Jianhong Wu
- ix. **Analysis of Recurrent Neural Networks with Application to Speaker Independent** - Prof. Dr.-Ing. O.E. Herrmann Ir. L.P.J. Veelenturf Dr. Ir. S.H. Gerez

Lecture 1 | Machine Learning (Stanford)

<https://www.youtube.com/watch?v=UzxYIbK2c7E>

<https://www.coursera.org/learn/machine-learning>

11. Introduction to Machine Learning

<https://www.youtube.com/watch?v=h0e2HAPTGF4>

Deep Learning

<http://cs231n.stanford.edu>

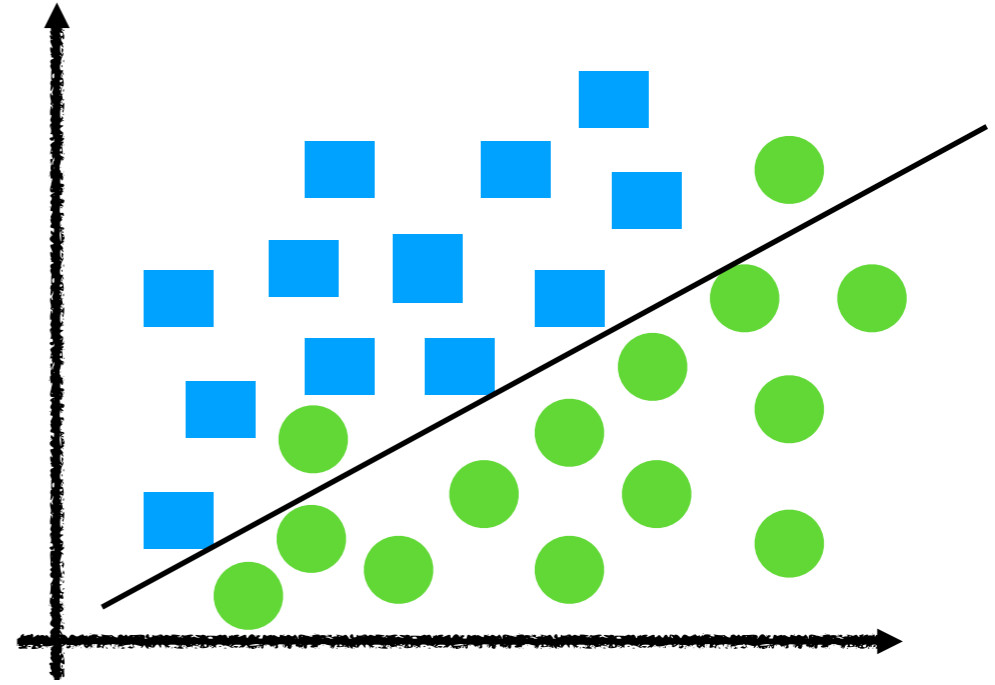
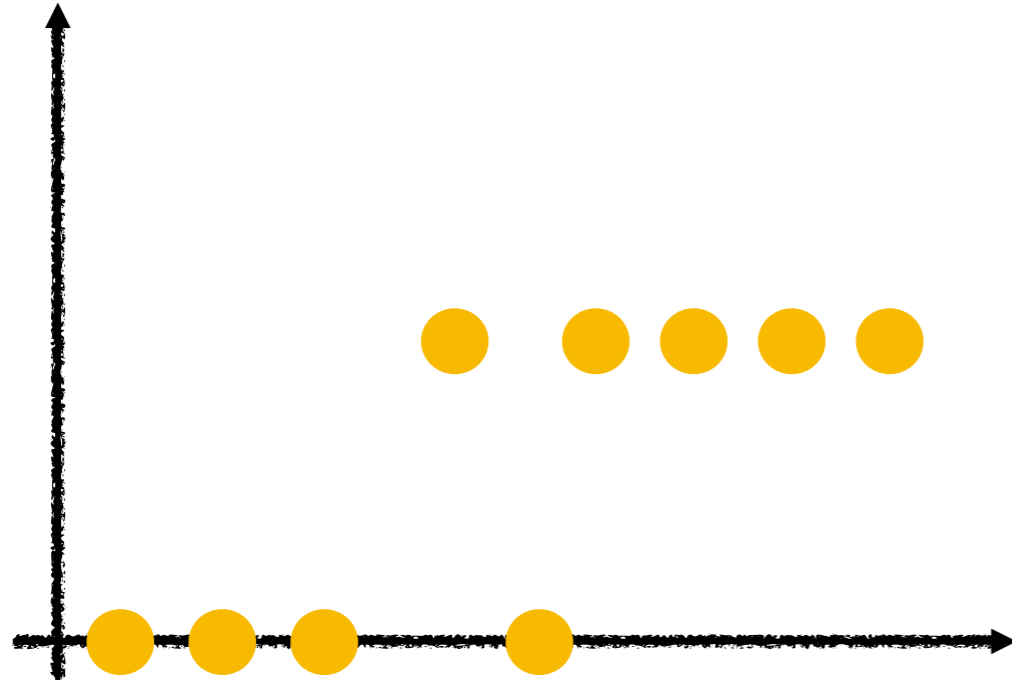
<https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050>

History

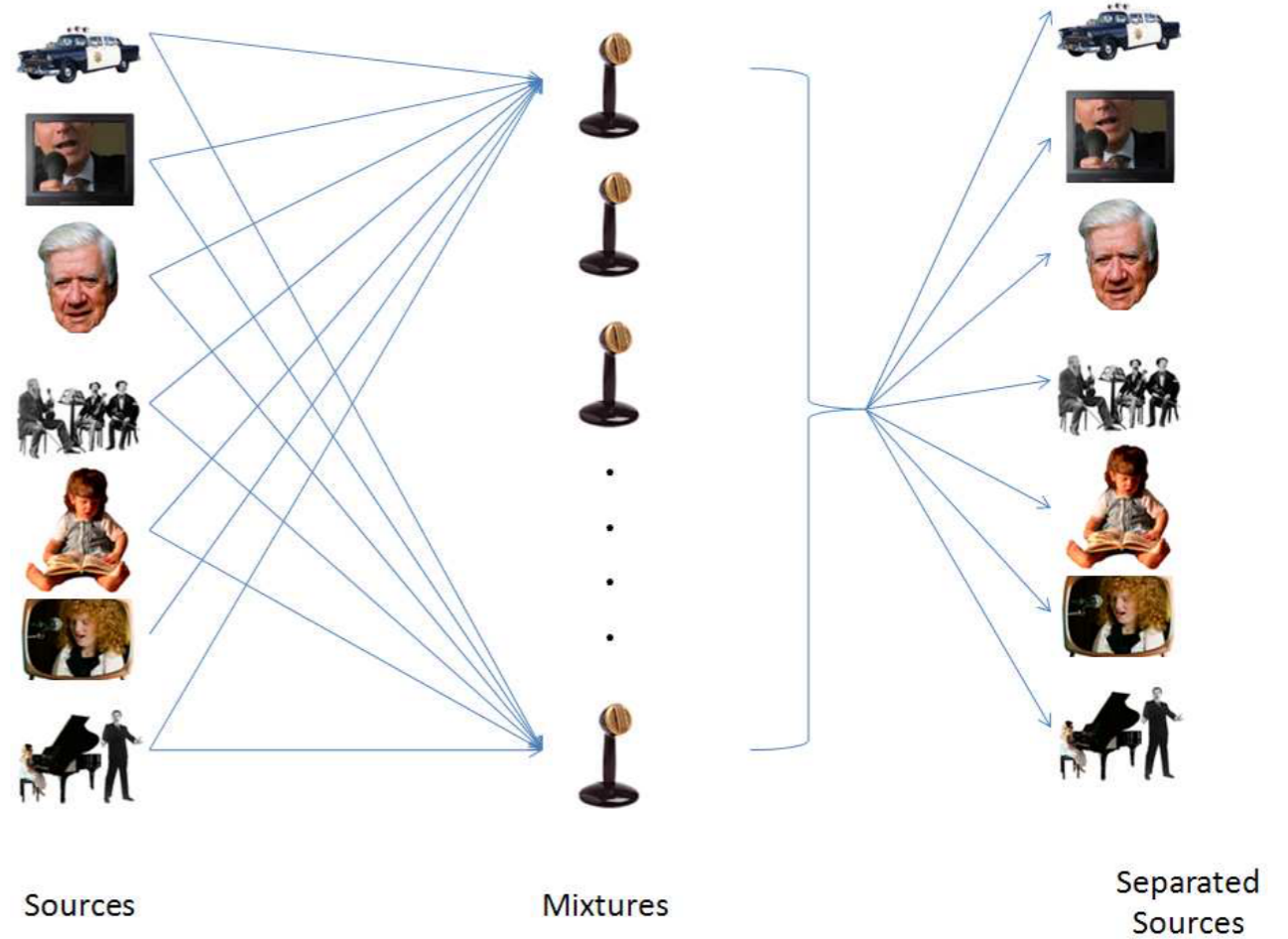
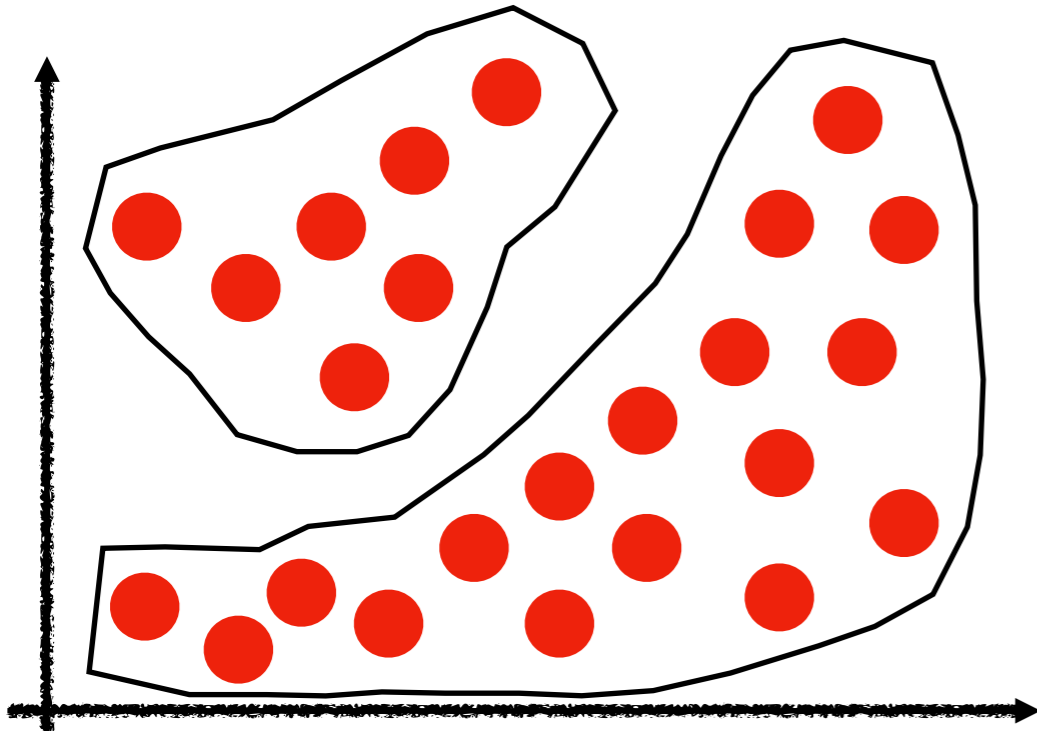
1950		Turing's Learning Machine
1951		First Neural Network Machine
1952		Machines Playing Checkers
1957	Discovery	Perceptron
1963	Achievement	Machines Playing Tic-Tac-Toe
1967		Nearest Neighbor
1969		Limitations of Neural Networks
1970		Automatic Differentiation (Backpropagation)
1972	Discovery	Term frequency–inverse document frequency (TF-IDF)
1979		Stanford Cart
1980	Discovery	Neocognitron
1981		Explanation Based Learning
1982	Discovery	Recurrent Neural Network
1985		NetTalk
1986	Discovery	Backpropagation
1989	Discovery	Reinforcement Learning
1992	Achievement	Machines Playing Backgammon
1995	Discovery	Random Forest Algorithm
1995	Discovery	Support Vector Machines
1997	Achievement	IBM Deep Blue Beats Kasparov
1997	Discovery	LSTM
1998		MNIST database
2002		Torch Machine Learning Library
2006		The Netflix Prize
2009	Achievement	ImageNet
2010		Kaggle Competition
2011	Achievement	Beating Humans in Jeopardy
2012	Achievement	Recognizing Cats on YouTube
2014		Leap in Face Recognition
2014		Sibyl
2016	Achievement	Beating Humans in Go



supervised



unsupervised



Definition

Defining learning broadly, to include any computer program that improves its performance at some task through experience.

Definition:

A computer program is said to **learn** from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.

You have to follow the data...

- 1) Machine learning uses a variety of algorithms that **iteratively learn from data** to improve, describe data, and predict outcomes.
- 2) As data is constantly added, the machine learning models ensure that the solution is constantly **updated**.
- 3) If you use the most appropriate and constantly changing data sources in the context of machine learning, you have the opportunity to predict the future!

You have to follow the data...

The ability to distribute compute processing across clusters of computers has dramatically improved the ability to analyze complex data in record time

Defining Big Data

Big data is any kind of data source that has at least one of four shared characteristics:

- » Extremely large Volumes of data
- » The ability to move that data at a high Velocity of speed
 - » An ever-expanding Variety of data sources
 - » Veracity so that data sources truly represent truth

When Do We Need Machine Learning?

The problem's complexity and the need for adaptivity.

Tasks That Are Too **Complex** to Program.

Tasks Performed by Animals/Humans

Examples of such tasks include driving, speech recognition, and image understanding.

Tasks beyond Human Capabilities:

Analysis of very large and complex data sets: astronomical data, turning medical archives into medical knowledge, weather prediction, analysis of genomic data, Web search engines, and electronic commerce.

Adaptivity. One limiting feature of programmed tools is their rigidity – once the program has been written down and installed, it stays unchanged. However, many tasks change over time or from one user to another. **Machine learning tools – programs whose behavior adapts to their input data – offer a solution to such issues; they are, by nature, adaptive to changes in the environment they interact with.**

Artificial Intelligence

Reasoning

Natural
Language
Processing
(NLP)

Planning

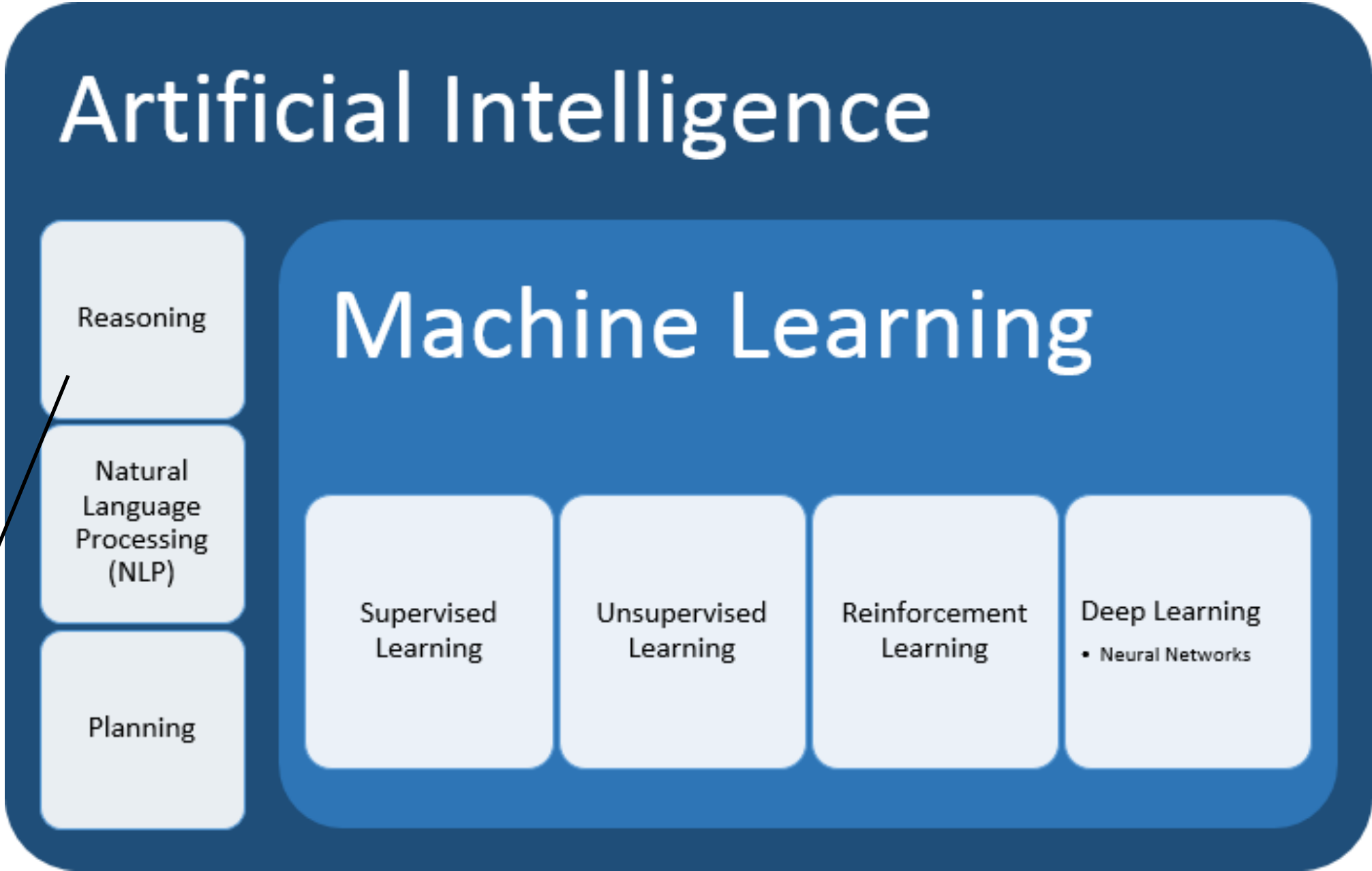
Machine Learning

Supervised
Learning

Unsupervised
Learning

Reinforcement
Learning

Deep Learning
• Neural Networks



Reasoning helps fill in the blanks when there is incomplete data. Machine reasoning helps make sense of connected data.

Artificial Intelligence

Reasoning

Natural
Language
Processing
(NLP)

Planning

Machine Learning

Supervised
Learning

Unsupervised
Learning

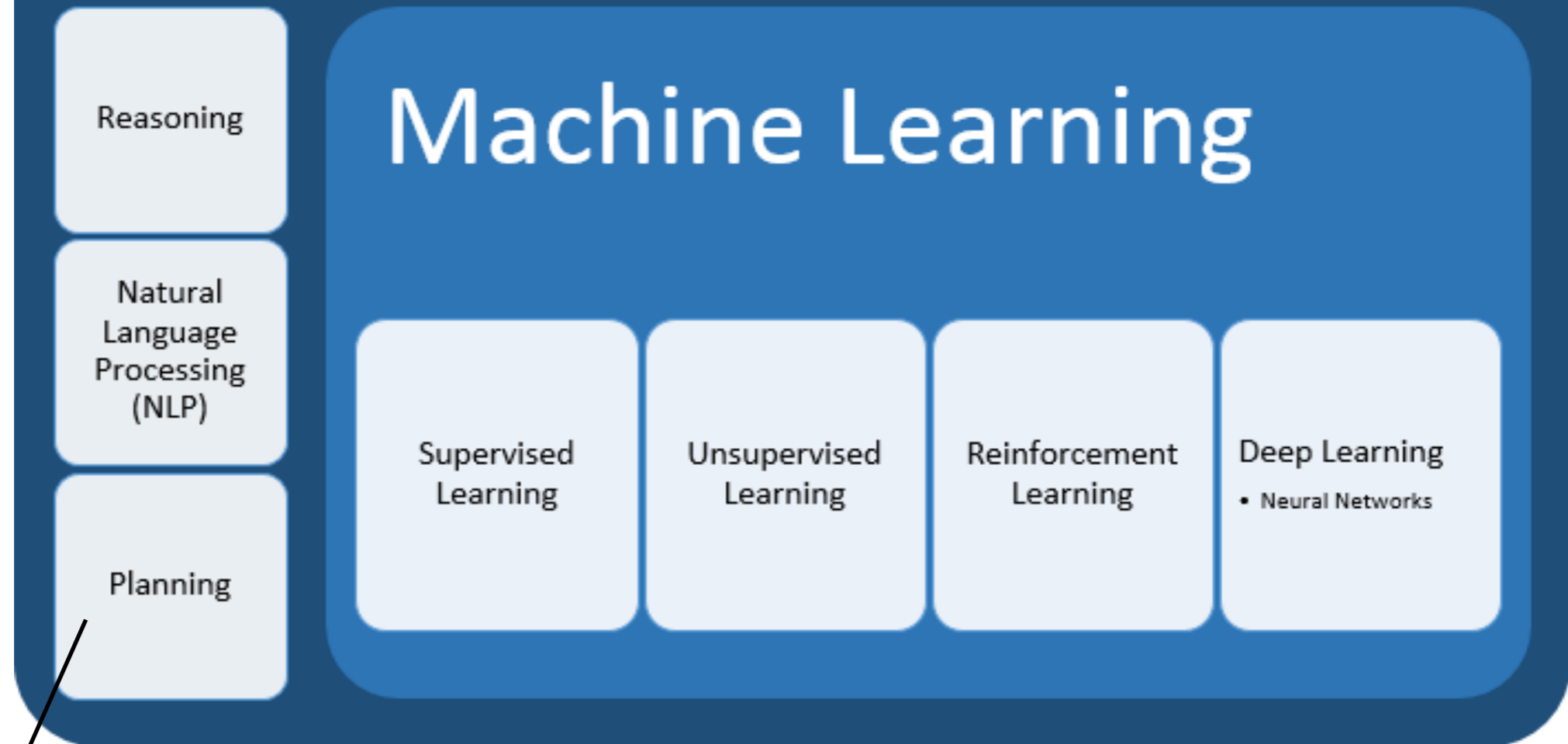
Reinforcement
Learning

Deep Learning
• Neural Networks

Reasoning helps fill in the blanks when there is incomplete data. Machine reasoning helps make sense of connected data.

NLP is the ability to train computers to understand both written text and human speech. NLP techniques are needed to capture the meaning of unstructured text from documents or communication from the user.

Artificial Intelligence

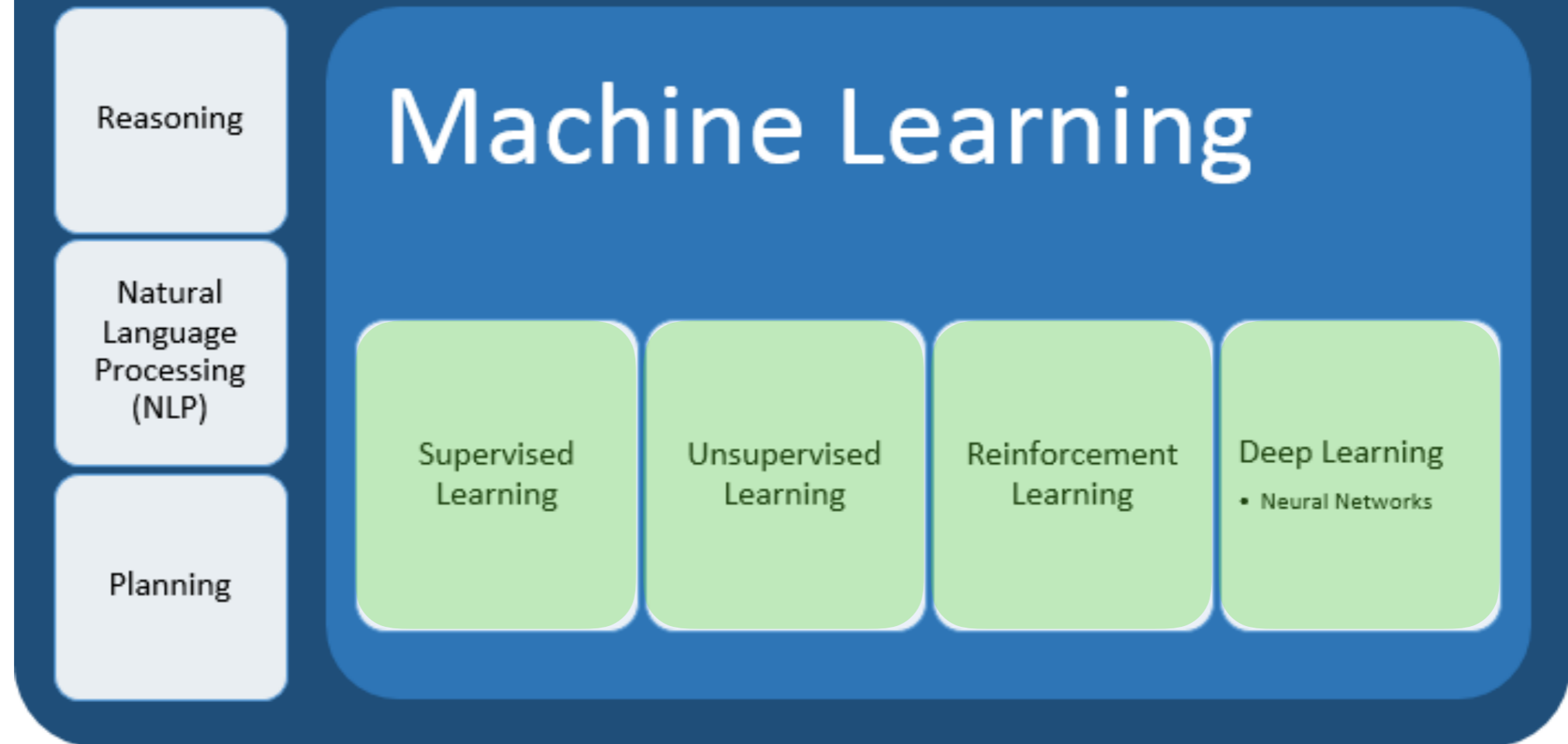


Reasoning helps fill in the blanks when there is incomplete data. Machine reasoning helps make sense of connected data.

NLP is the ability to train computers to understand both written text and human speech. NLP techniques are needed to capture the meaning of unstructured text from documents or communication from the user.

Planning: Automated planning is the ability for an intelligent system to act autonomously and to construct a sequence of actions to reach a goal.

Artificial Intelligence



Supervised learning typically begins with an established set of data and a certain understanding of how that data is classified.

Unsupervised learning is best suited when the problem requires a massive amount of data that is unlabeled.

Reinforcement learning is a behavioral learning model. The algorithm receives feedback from the analysis of the data so the user is guided to the best outcome. Reinforcement learning differs from other types of supervised learning because the system isn't trained with the sample data set but through trial and error.

Deep learning is a specific method of machine learning that incorporates neural networks in successive layers in order to learn from data in an iterative manner. Deep learning is especially useful when you're trying to learn patterns from unstructured data.

(a)

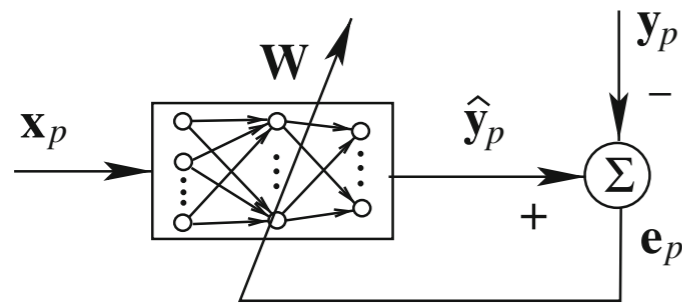


Fig. Learning methods. **a** Supervised learning. $e_p = \hat{y}_p - y_p$. **b** Unsupervised learning. **c** Reinforcement learning

- **Supervised learning** is widely used in classification, approximation, control, modeling and identification, signal processing, and optimization.

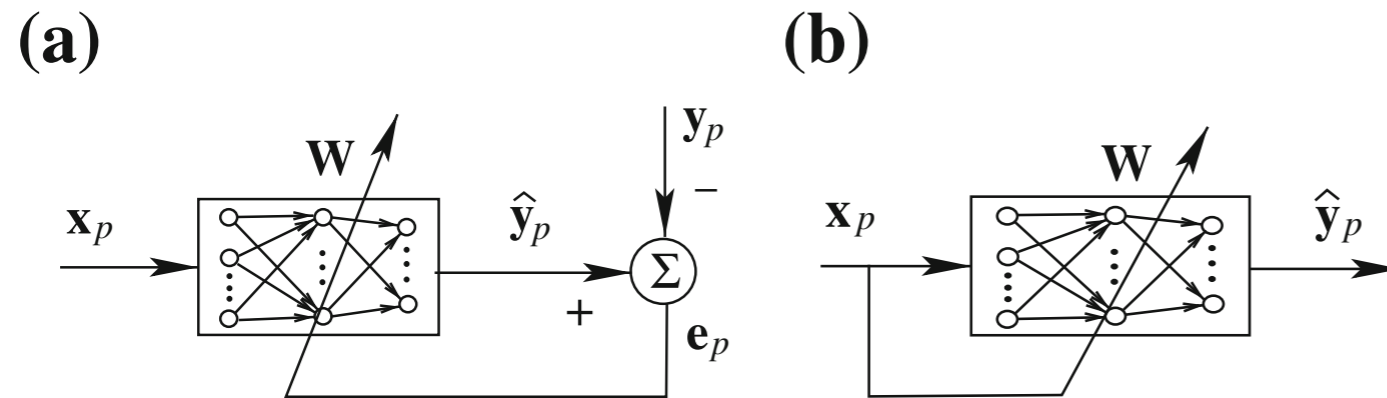


Fig. Learning methods. **a** Supervised learning. $e_p = \hat{y}_p - y_p$. **b** Unsupervised learning. **c** Reinforcement learning

- **Supervised learning** is widely used in classification, approximation, control, modeling and identification, signal processing, and optimization.
- **Unsupervised learning** schemes are mainly used for clustering, vector quantization, feature extraction, signal coding, and data analysis.

- **Supervised learning**

- **Unsupervised learning**

- **Reinforcement learning**

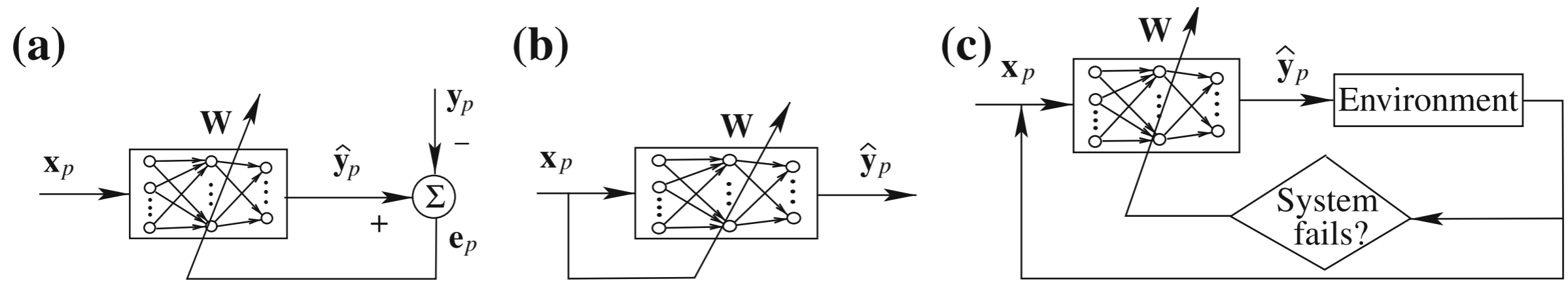


Fig. Learning methods. **a** Supervised learning. $e_p = \hat{y}_p - y_p$. **b** Unsupervised learning. **c** Reinforcement learning

- **Supervised learning** is widely used in classification, approximation, control, modeling and identification, signal processing, and optimization.
- **Unsupervised learning** schemes are mainly used for clustering, vector quantization, feature extraction, signal coding, and data analysis.
- **Reinforcement learning** is usually used in control and artificial intelligence.

- Supervised learning

- Unsupervised learning

- Reinforcement learning

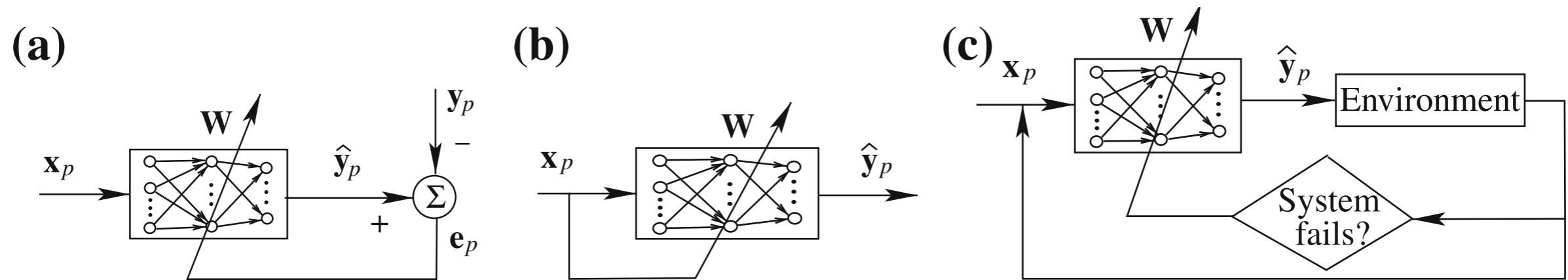


Fig. Learning methods. **a** Supervised learning. $e_p = \hat{y}_p - y_p$. **b** Unsupervised learning. **c** Reinforcement learning

- **Supervised learning** is widely used in classification, approximation, control, modeling and identification, signal processing, and optimization.
- **Unsupervised learning** schemes are mainly used for clustering, vector quantization, feature extraction, signal coding, and data analysis.
- **Reinforcement learning** is usually used in control and artificial intelligence.

A Gentle Start

Imagine you have just arrived in some small Pacific island. You soon find out that papayas are a significant ingredient in the local diet. However, you have never before tasted papayas. You have to learn how to predict whether a papaya you see in the market is tasty or not.



First, you need to decide which features of a papaya your prediction should be based on

papaya's color

papaya's softness

A Formal Model – The Statistical Learning Framework

Domain set: An arbitrary set, X . This is the set of objects that we may wish to label. For example, in the papaya learning problem mentioned before, the domain set will be the set **of all papayas**. Usually, these domain points will be represented by a vector of features (like the papaya's color and softness).

Label set: For our current discussion, we will restrict the label set to be a two-element set, usually $\{0,1\}$ or $\{-1,+1\}$. Let Y denote our set of possible labels. For our papayas example, let Y be $\{0, 1\}$, where 1 represents being tasty and 0 stands for being not-tasty.

Training data: $S=((x_1,y_1)\dots(x_m,y_m))$ is a finite sequence of pairs in $X \times Y$: that is, a sequence of labeled domain points.

The learner's output: The learner is requested to output a prediction rule,

$$h : X \rightarrow Y$$

This function is also called a predictor, a hypothesis, or a classifier. We use the notation $A(S)$ to denote the hypothesis that a learning algorithm, A , returns upon receiving the training sequence S .

The choice of which functions to include in H usually depends on our intuition about the problem of interest

A simple data-generation model we assume that there is some “correct” labeling function,

$$f : X \rightarrow Y, \text{ and that } y_i = f(x_i) \text{ for all } i$$

The labeling function is unknown to the learner. In fact, this is just what the learner is trying to figure out.

A Formal Model – The Statistical Learning Framework

Measures of success: the error of h is the probability to draw a random instance \mathbf{x} , according to the distribution \mathbf{D} , such that $\mathbf{h}(\mathbf{x})$ does not equal $\mathbf{f}(\mathbf{x})$

Formally, given a domain subset, $A \subset X$, the probability distribution, \mathbf{D} , assigns a number, $\mathbf{D}(A)$, which determines how likely it is to observe a point $\mathbf{x} \in A$.

We refer to A as an event and express it using a function $\pi: X \rightarrow \{0,1\}$, namely, $A = \{\mathbf{x} \in X: \pi(\mathbf{x}) = 1\}$.

In that case, we also use the notation $\mathbb{P}_{\mathbf{x} \sim \mathbf{D}}[\pi(\mathbf{x})]$ to express $\mathbf{D}(A)$.

We define the error of a prediction rule, $h: X \rightarrow Y$, to be

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathbb{P}_{x \sim \mathcal{D}}[h(x) \neq f(x)] \stackrel{\text{def}}{=} \mathcal{D}(\{x : h(x) \neq f(x)\}).$$

A learning algorithm receives as input a training set S , sampled from an unknown distribution D and labeled by some target function f , and should output a predictor $h_S: X \rightarrow Y$ (the subscript S emphasizes the fact that the output predictor depends on S).

The goal of the algorithm is to find h_S that minimizes the error with respect to the unknown D and f .

A Formal Model – The Statistical Learning Framework

Empirical Risk Minimization: Since the learner does not know what \mathbf{D} and \mathbf{f} are, the true error is not directly available to the learner.

A useful notion of error that can be calculated by the learner is the training error – the error the classifier incurs over the training sample:

$$L_S(h) \stackrel{\text{def}}{=} \frac{|\{i \in [m] : h(x_i) \neq y_i\}|}{m}$$

where $[m] = \{1, \dots, m\}$.

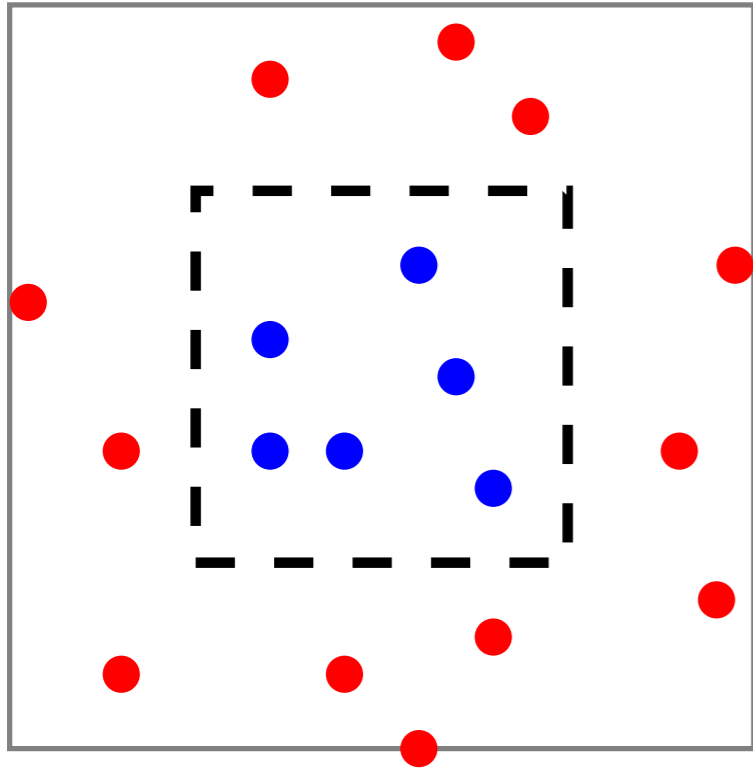
The terms empirical error and empirical risk are often used interchangeably for this error.

The empirical risk is the average loss of an estimator for a finite set of data

Since the training sample is the snapshot of the world that is available to the learner, it makes sense to search for a solution that works well on that data. This learning paradigm – coming up with a predictor h that minimizes $L_S(h)$ – is called **Empirical Risk Minimization** or ERM for short.

The goal of learning is to minimize the risk function

Overfitting



Assume that the probability distribution \mathbf{D} is such that instances are distributed uniformly within the gray square and the labeling function, \mathbf{f} , determines the label to be 1 if the instance is within the inner blue square, and 0 otherwise.

The area of the gray square in the picture is 2 and the area of the blue square is 1.

Consider the following predictor:

$$h_{\mathcal{S}}(x) = \begin{cases} y_i & \text{if } \exists i \in [m] \text{ s.t. } x_i = x \\ 0 & \text{otherwise.} \end{cases}$$

$\text{LD}(h_{\mathcal{S}}) = 1/2$. We have found a predictor whose performance on the training set is excellent, yet its performance on the true “world” is very poor. **This phenomenon is called overfitting.**

Goal: search for conditions under which there is a guarantee that ERM does not overfit



conditions under which when the ERM predictor has good performance with respect to the training data, it is also highly likely to perform well over the underlying data distribution

Empirical Risk Minimization with Inductive Bias

The idea of risk minimization is not only measure the performance of an estimator by its risk, but to actually search for the estimator that minimizes risk over distribution

$$\text{ERM}_{\mathcal{H}}(S) \in \underset{h \in \mathcal{H}}{\text{argmin}} L_S(h),$$

where argmin stands for the set of hypotheses in \mathcal{H} that achieve the minimum value of $L_S(h)$ over \mathcal{H}

By restricting the learner to choosing a predictor from \mathcal{H} , we bias it toward a particular set of predictors. Such restrictions are often called an inductive bias.

Since the choice of such a restriction is determined before the learner sees the training data, it should ideally be based on some prior knowledge about the problem to be learned. For example, for the papaya taste prediction problem we may choose the class \mathcal{H} to be the set of predictors that are determined by axis aligned rectangles (in the space determined by the color and softness coordinates).

A fundamental question in learning theory is, over which hypothesis classes $\text{ERM}_{\mathcal{H}}$ learning will not result in overfitting

Intuitively, choosing a more restricted hypothesis class better protects us against overfitting but at the same time might cause us a stronger inductive bias

Ingredients

Almost every problem in ML and data science starts with the same ingredients.

- The first ingredient is the dataset X .
- The second is the model $g(w)$, which is a function of the parameters w .
- The final ingredient is the cost function $C(X, g(w))$ that allows us to judge how well the model $g(w)$ explains, or in general performs on, the observations X .

The model is fit by finding the value of w that minimizes the cost function. For example, one commonly used cost function is the squared error. Minimizing the squared error cost function is known as the method of least squares, and is typically appropriate for experiments with Gaussian measurement errors.

Typically, the majority of the data are partitioned into the training set (e.g., 90%) with the remainder going into the test set. The model is fit by minimizing the cost function using only the data in the training set $\hat{w} = \operatorname{argmin}_w \{C(X_{\text{train}}, g(w))\}$. Finally, the performance of the model is evaluated by computing the cost function using the test set $C(X_{\text{test}}, g(\hat{w}))$. The value of the cost function for the best fit model on the training set is called the in-sample error $E_{\text{in}} = C(X_{\text{train}}, g(\hat{w}))$ and the value of the cost function on the test set is called the out-of-sample error $E_{\text{out}} = C(X_{\text{test}}, g(\hat{w}))$

Classification

Decision Trees

Bayesian Networks

Gaussian Mixture Models

Random forest

Association Rule Learning

encoder-decoder model

Regression

Dimensionality Reduction

Hebbian Learning

Stochastic Gradient Descent

Recurrent Neural Networks

Support Vector Machines

Neural Network

Neural Network

Inductive Logic Programming

Kernel Methods

Genetic Algorithms

Gradient Descent

**Stochastic
Gradient Descent**

Decision Trees

Clustering

Convex Learning Problems

Decision Tree Learning

classification

Maximum Likelihood Estimation

Gaussian Mixture Models

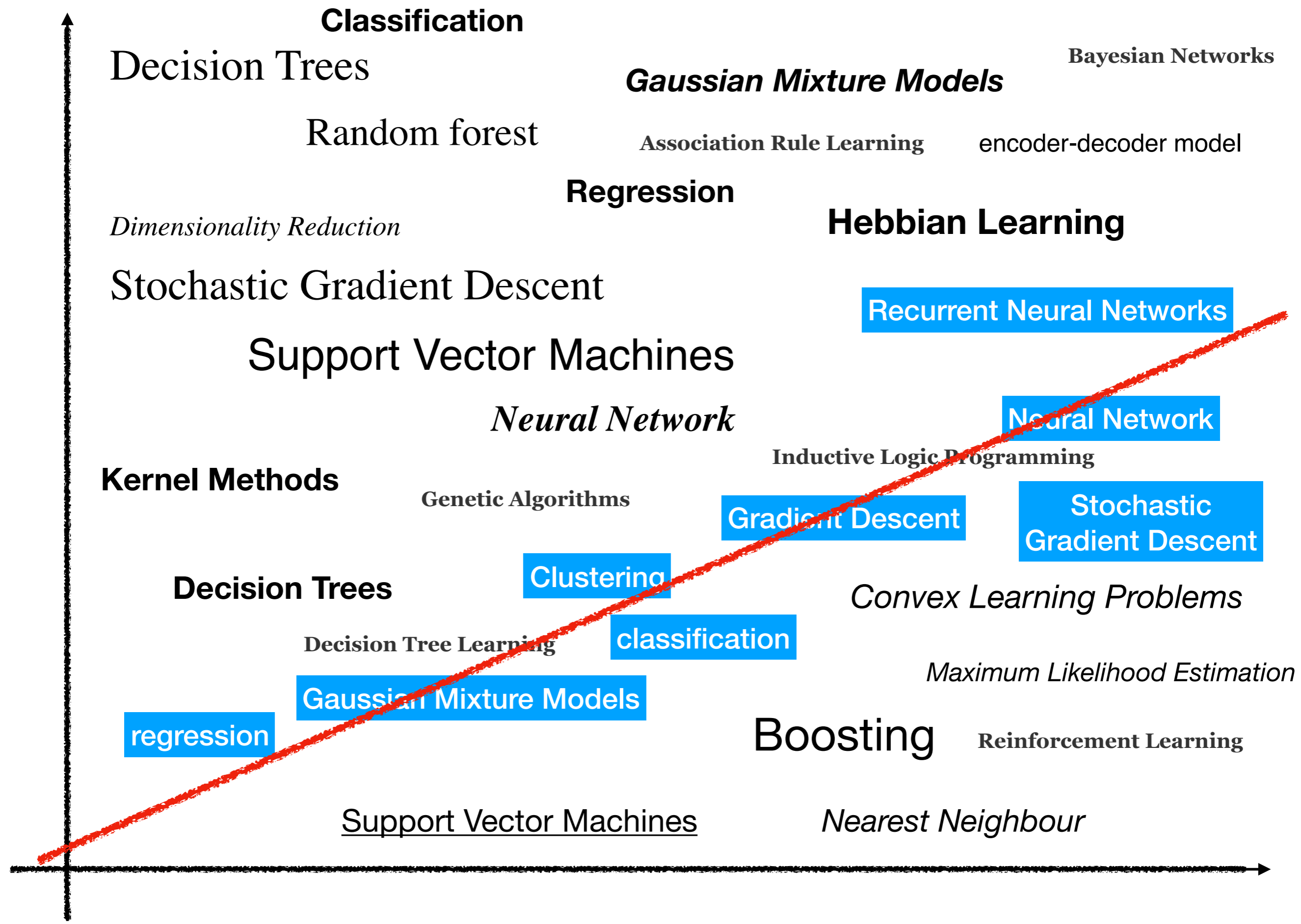
regression

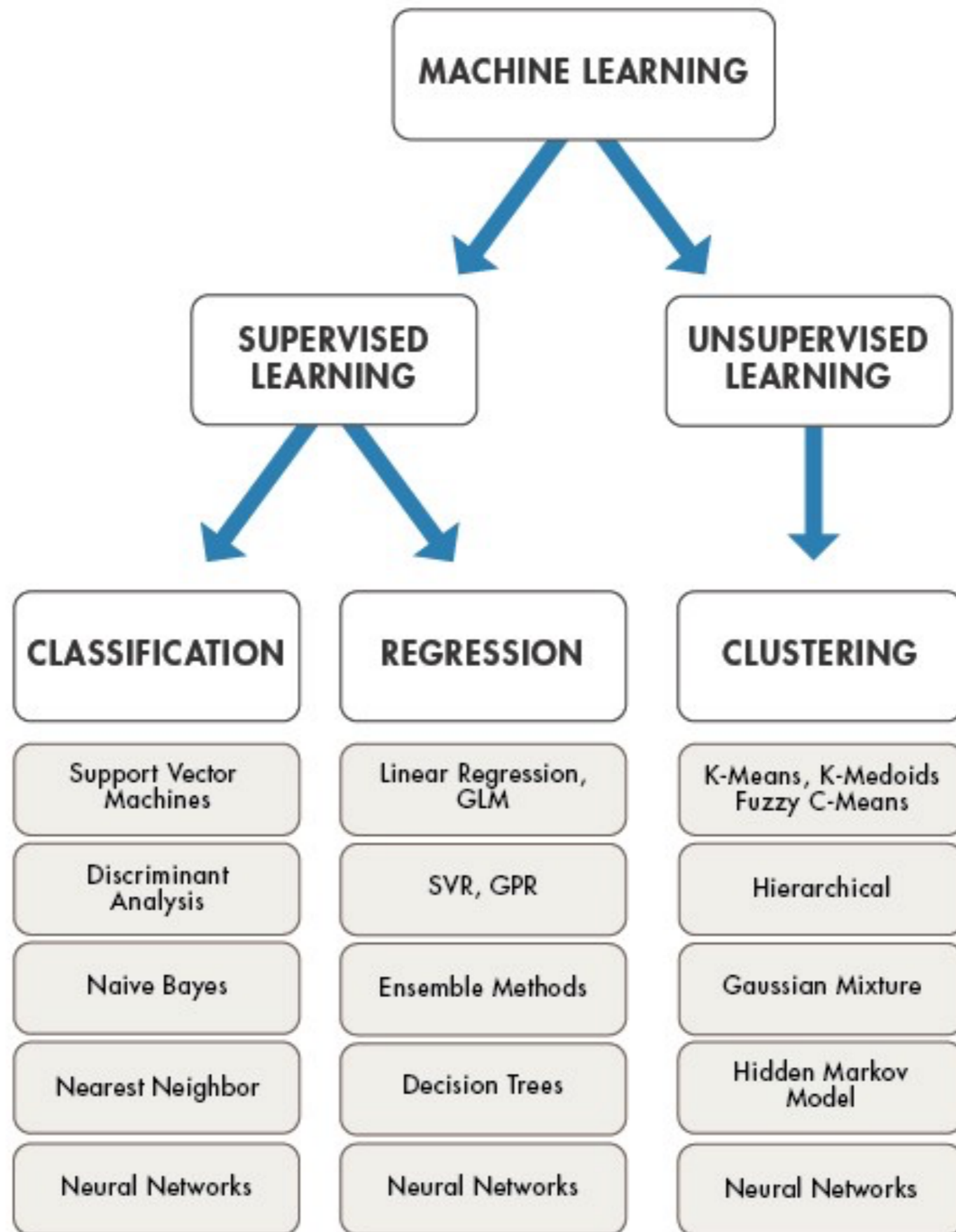
Boosting

Reinforcement Learning

Support Vector Machines

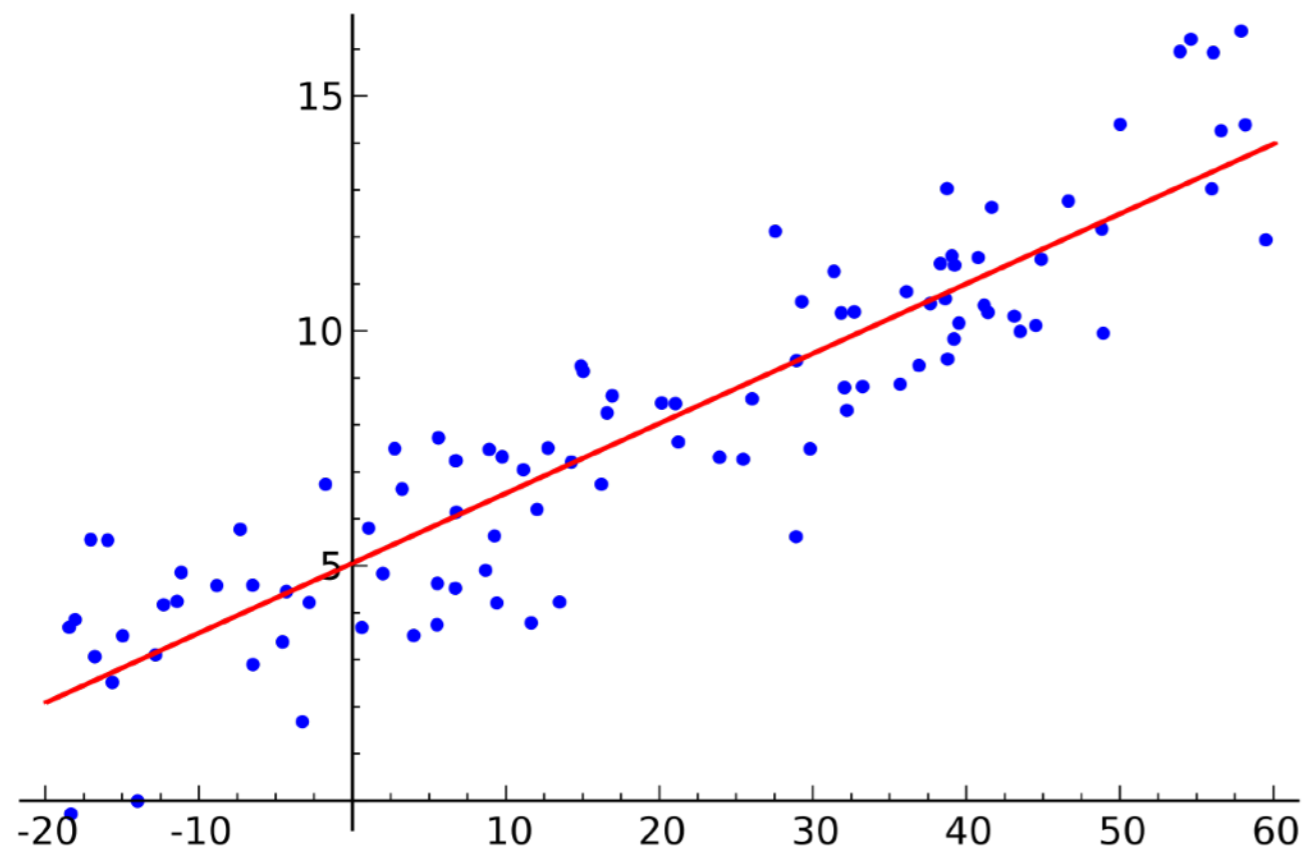
Nearest Neighbour





Regression

Regression is a method of modelling a target value based on independent predictors.



Simple linear regression is a type of regression analysis where there is a linear relationship between the independent (x) and dependent(y) variable

$$y = a_0 + a_1 * x$$

The motive of the linear regression algorithm is to find the best values for **a_0** and **a_1**

Since we want the best values for **a_0** and **a_1**, we convert this search problem into a minimization problem where we would like to minimize the error between the predicted value and the actual value

Mean Squared Error(MSE) function

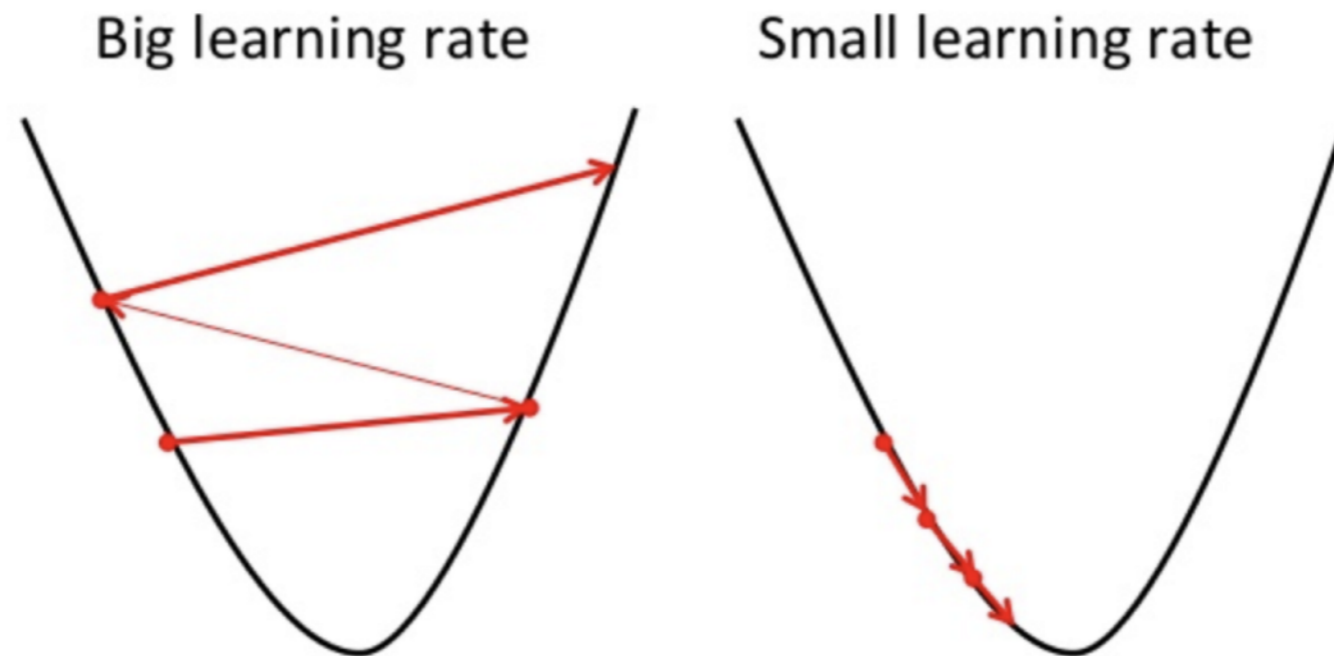
This provides the average squared error over all the data points

$$J = \frac{1}{n} \sum_{i=1}^n (pred_i - y_i)^2$$

Regression

Gradient descent is a method of updating a_0 and a_1 to reduce the cost function(MSE)

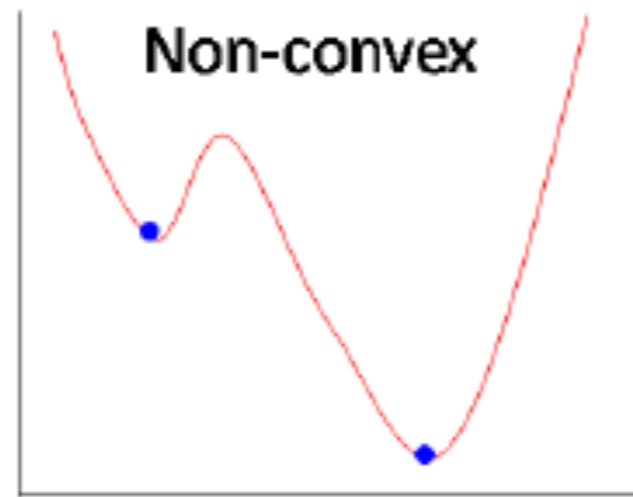
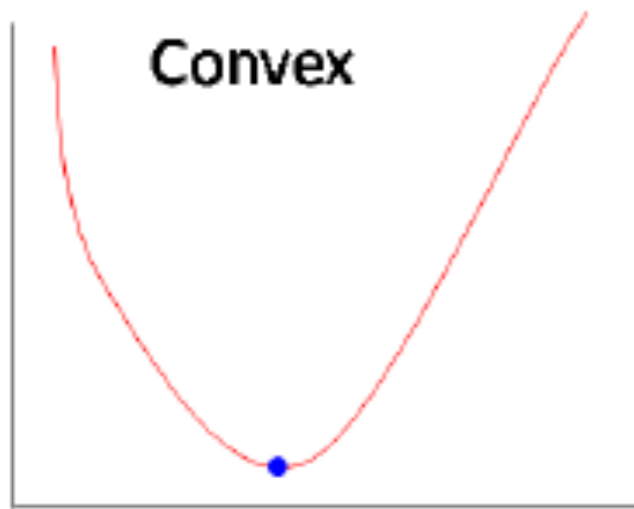
The idea is that we start with some values for a_0 and a_1 and then we change these values iteratively to reduce the cost



If you decide to take one step at a time you would eventually reach the bottom of the pit but this would take a longer time.

If you choose to take longer steps each time, you would reach sooner but, there is a chance that you could overshoot the bottom of the pit and not exactly at the bottom.

Regression



Sometimes the cost function can be a non-convex function where you could settle at a local minima but for linear regression, it is always a convex function

The partial derivatives are the gradients and they are used to update the values of a_0 and a_1

$$J = \frac{1}{n} \sum_{i=1}^n (\text{pred}_i - y_i)^2$$

$$J = \frac{1}{n} \sum_{i=1}^n (a_0 + a_1 \cdot x_i - y_i)^2$$

Alpha is the learning rate which is a hyperparameter that you must specify

$$\frac{\partial J}{\partial a_0} = \frac{2}{n} \sum_{i=1}^n (a_0 + a_1 \cdot x_i - y_i) \implies \frac{\partial J}{\partial a_0} = \frac{2}{n} \sum_{i=1}^n (\text{pred}_i - y_i)$$

$$\frac{\partial J}{\partial a_1} = \frac{2}{n} \sum_{i=1}^n (a_0 + a_1 \cdot x_i - y_i) \cdot x_i \implies \frac{\partial J}{\partial a_1} = \frac{2}{n} \sum_{i=1}^n (\text{pred}_i - y_i) \cdot x_i$$

$$a_0 = a_0 - \alpha \cdot \frac{2}{n} \sum_{i=1}^n (\text{pred}_i - y_i)$$

$$a_1 = a_1 - \alpha \cdot \frac{2}{n} \sum_{i=1}^n (\text{pred}_i - y_i) \cdot x_i$$

Polynomial Regression

Consider a probabilistic process that assigns a label \mathbf{y}_i to an observation \mathbf{x}_i . The data are generated by drawing samples from the equation

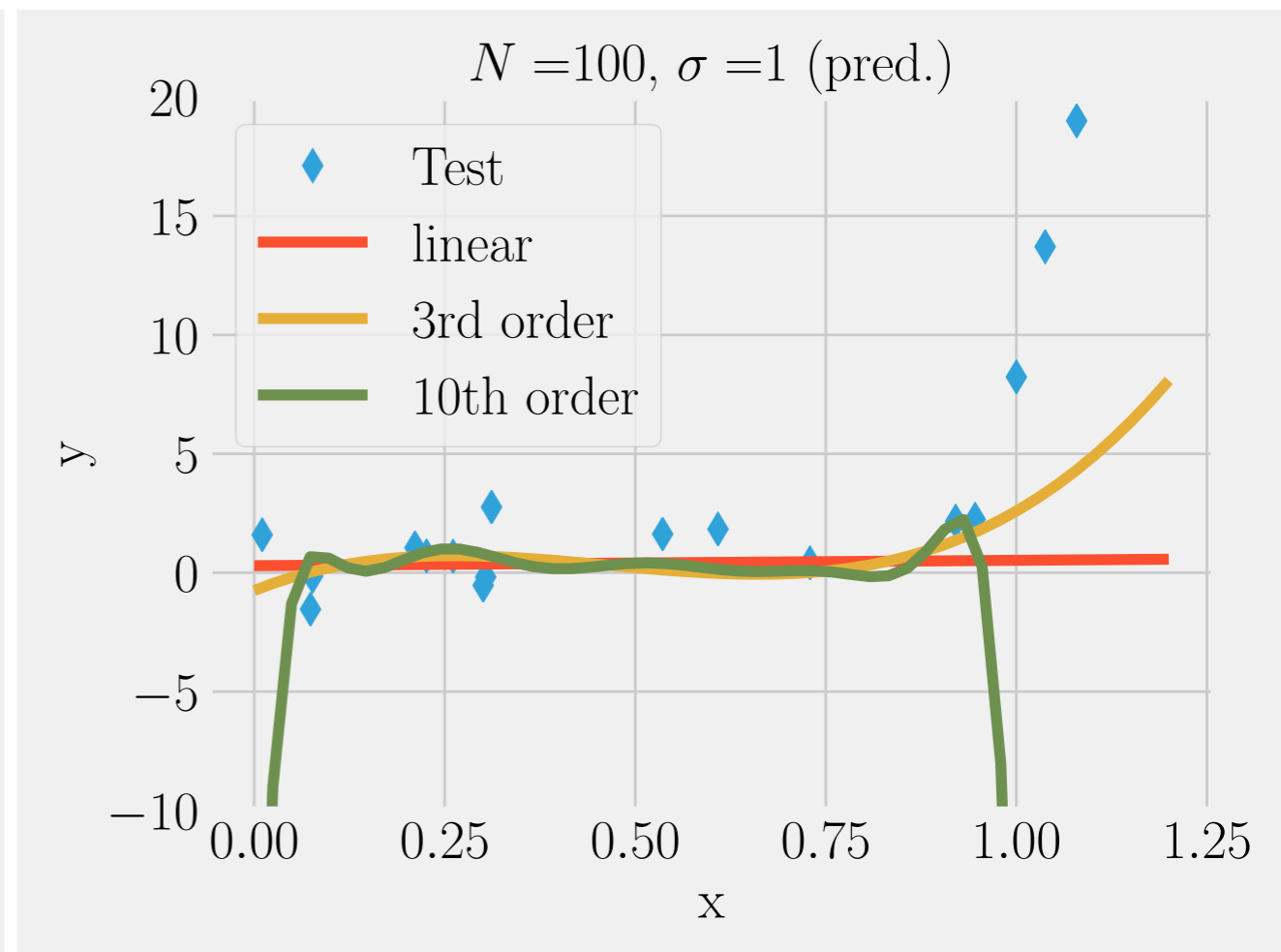
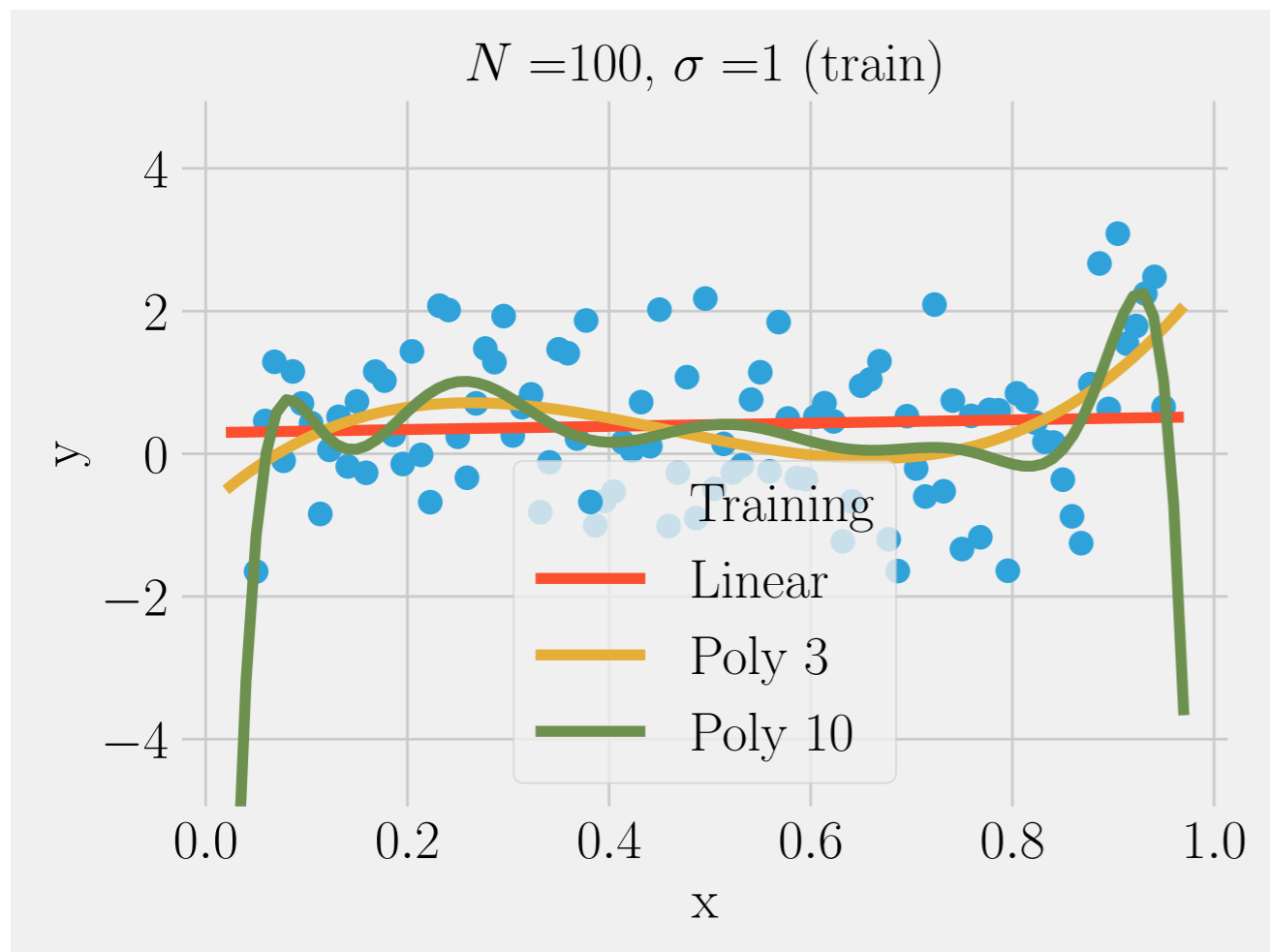
$$y_i = f(x_i) + \eta_i,$$

where $\mathbf{f}(\mathbf{x}_i)$ is some fixed (but possibly unknown) function, and $\boldsymbol{\eta}_i$ is a Gaussian, uncorrelated noise variable, such that

$$\begin{aligned}\langle \eta_i \rangle &= 0, \\ \langle \eta_i \eta_j \rangle &= \delta_{ij} \sigma^2.\end{aligned}$$

We will refer to the $\mathbf{f}(\mathbf{x}_i)$ as the function used to generate the data, and $\boldsymbol{\sigma}$ as the noise strength. The larger $\boldsymbol{\sigma}$ is the noisier the data; $\boldsymbol{\sigma} = \mathbf{0}$ corresponds to the noiseless case.

To make predictions, we will consider a family of functions $\mathbf{g}_\alpha(\mathbf{x}; \mathbf{w}_\alpha)$ that depend on some parameters \mathbf{w}_α . These functions represent the model class that we are using to model the data and make predictions. Note that we choose the model class without knowing the function $f(\mathbf{x})$.



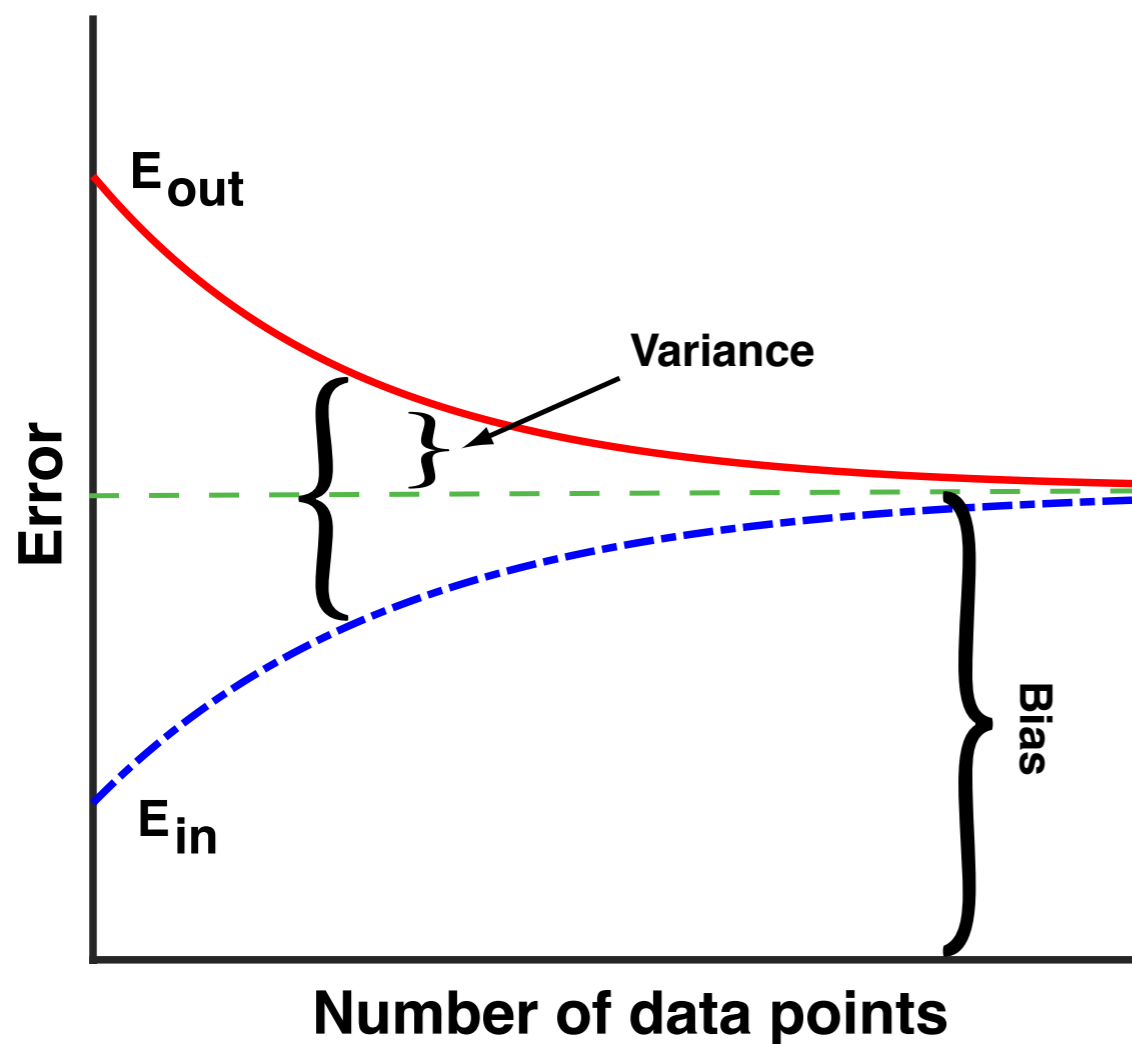
Obviously, more data and less noise leads to better predictions

Complex models with many parameters, such as the tenth order polynomial in this example, can capture both the global trends and noise-generates patterns at the same time.

In this case, the model can be tricked into thinking that the noise encodes real information. This problem is called “**overfitting**” and leads to a steep drop-off in predictive performance.

Number of data

The model is fit by minimizing the cost function using only the data in the training set $\mathbf{w} = \operatorname{argmin}_{\mathbf{w}} \{C(\mathbf{X}_{\text{train}}, g(\mathbf{w}))\}$.



Models with a large difference between the in-sample and out-of-sample errors are said to “overfit” the data.

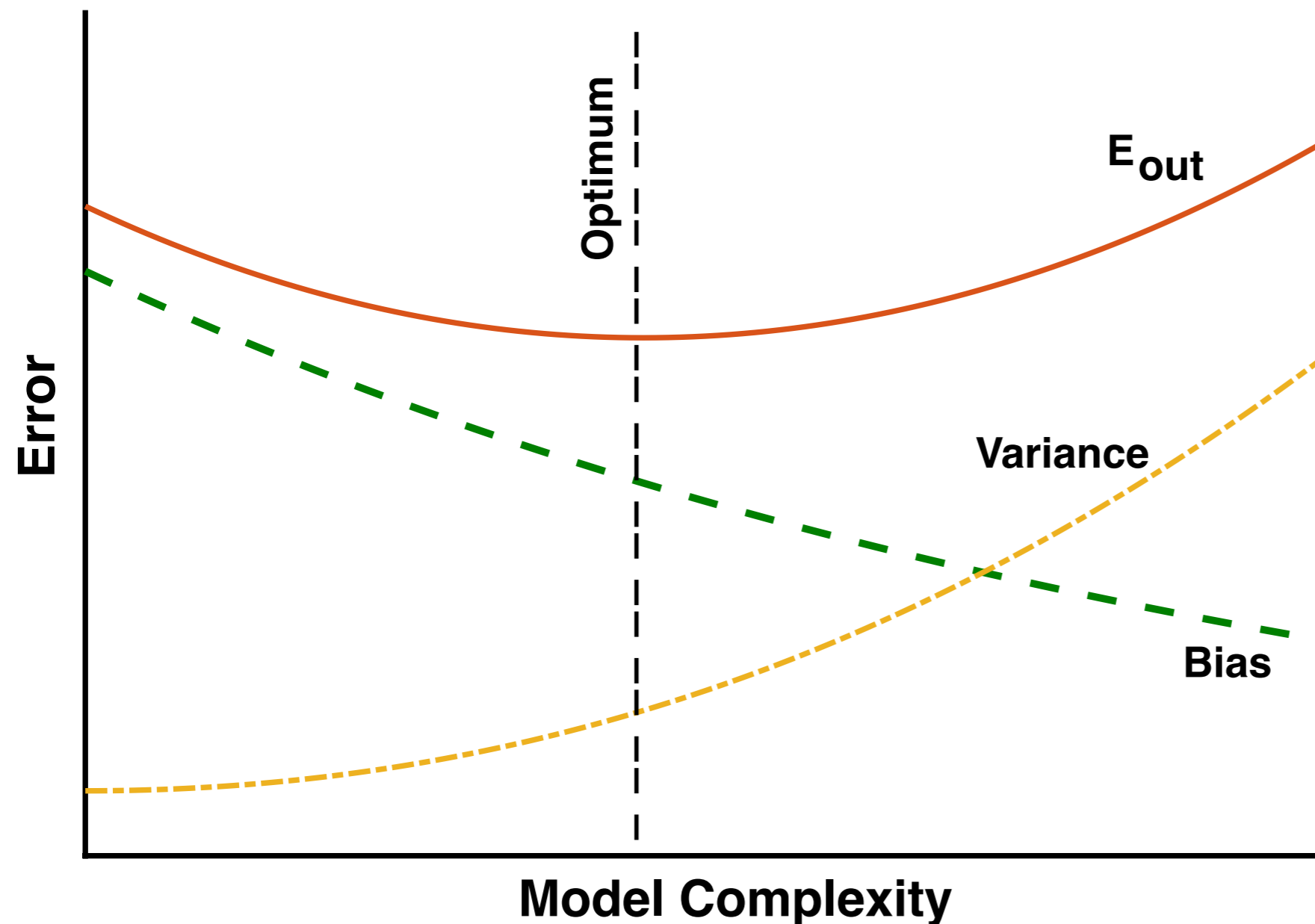
One of the lessons of statistical learning theory is that it is not enough to simply minimize the training error, since the out-of-sample error can still be large.

$$E_{\text{in}} = \mathcal{C}(\mathbf{X}_{\text{train}}, g(\hat{\mathbf{w}}))$$

$$E_{\text{out}} = \mathcal{C}(\mathbf{X}_{\text{test}}, g(\hat{\mathbf{w}}))$$

One of the most important observations we can make is that the out-of-sample error is almost always greater than the in-sample error, i.e. $E_{\text{out}} \geq E_{\text{in}}$.

Model complexity



This schematic shows the typical out-of-sample error E_{out} as function of the model complexity for a training dataset of fixed size. Notice how the bias always decreases with model complexity, but the variance, i.e. fluctuation in performance due to finite size sampling effects, increases with model complexity. Thus, **optimal performance is achieved at intermediate levels of model complexity.**

Even though using a more complicated model always reduces the bias, at some point the model becomes too complex for the amount of training data and the generalization error becomes large due to high variance.



to minimize E_{out} and maximize our predictive power, it may be more suitable to use a more biased model with small variance than a less-biased model with large variance.

This important concept is commonly called the **bias-variance tradeoff** and gets at the heart of why machine learning is difficult.

The bias-variance tradeoff

We will discuss the bias-variance tradeoff in the context of continuous predictions such as regression

$$\mathbf{X}_{\mathcal{L}} = \{(y_j, \mathbf{x}_j), j = 1 \dots N\} \qquad y = f(\mathbf{x}) + \epsilon$$

Assume that we have a statistical procedure (e.g. least-squares regression) for forming a predictor $\hat{g}_{\mathcal{L}}(\mathbf{x})$ that gives the prediction of our model for a new data point \mathbf{x} .

This estimator is chosen by minimizing a cost function which we take to be the squared error

$$\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x})) = \sum_i (y_i - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2$$

we can view $\hat{g}_{\mathcal{L}}$ as a stochastic functional that depends on the dataset \mathbf{L} and we can think of $\mathbb{E}\mathcal{L}$ as the expected value of the functional if we drew an infinite number of datasets $\{\mathbf{L}_1, \mathbf{L}_2, \dots\}$.

The bias-variance tradeoff

We would also like to average over different instances of the “noise” ϵ and we denote the expectation value over the noise by E_ϵ . Thus, we can decompose the expected generalization error as

$$\begin{aligned} E_{\mathcal{L},\epsilon}[\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x}))] &= E_{\mathcal{L},\epsilon} \left[\sum_i (y_i - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2 \right] \\ &= E_{\mathcal{L},\epsilon} \left[\sum_i (y_i - f(\mathbf{x}_i) + f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2 \right] \\ &= \sum_i E_\epsilon [(y_i - f(\mathbf{x}_i))^2] + E_{\mathcal{L},\epsilon} [(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] + 2E_\epsilon [y_i - f(\mathbf{x}_i)] E_{\mathcal{L}} [f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i)] \\ &= \sum_i \sigma_\epsilon^2 + E_{\mathcal{L}} [(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2], \end{aligned}$$

$$\begin{aligned} E_{\mathcal{L}} [(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] &= E_{\mathcal{L}} [(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)] + E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)] - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] \\ &= E_{\mathcal{L}} [(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] + E_{\mathcal{L}} [(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] \\ &\quad + 2E_{\mathcal{L}} [(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])] \\ &= (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 + E_{\mathcal{L}} [(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2]. \end{aligned}$$

The bias-variance tradeoff

$$E_{\text{out}} = E_{\mathcal{L}, \epsilon}[\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x}))] = \text{Bias}^2 + \text{Var} + \text{Noise}.$$

$$\text{Bias}^2 = \sum_i (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2$$

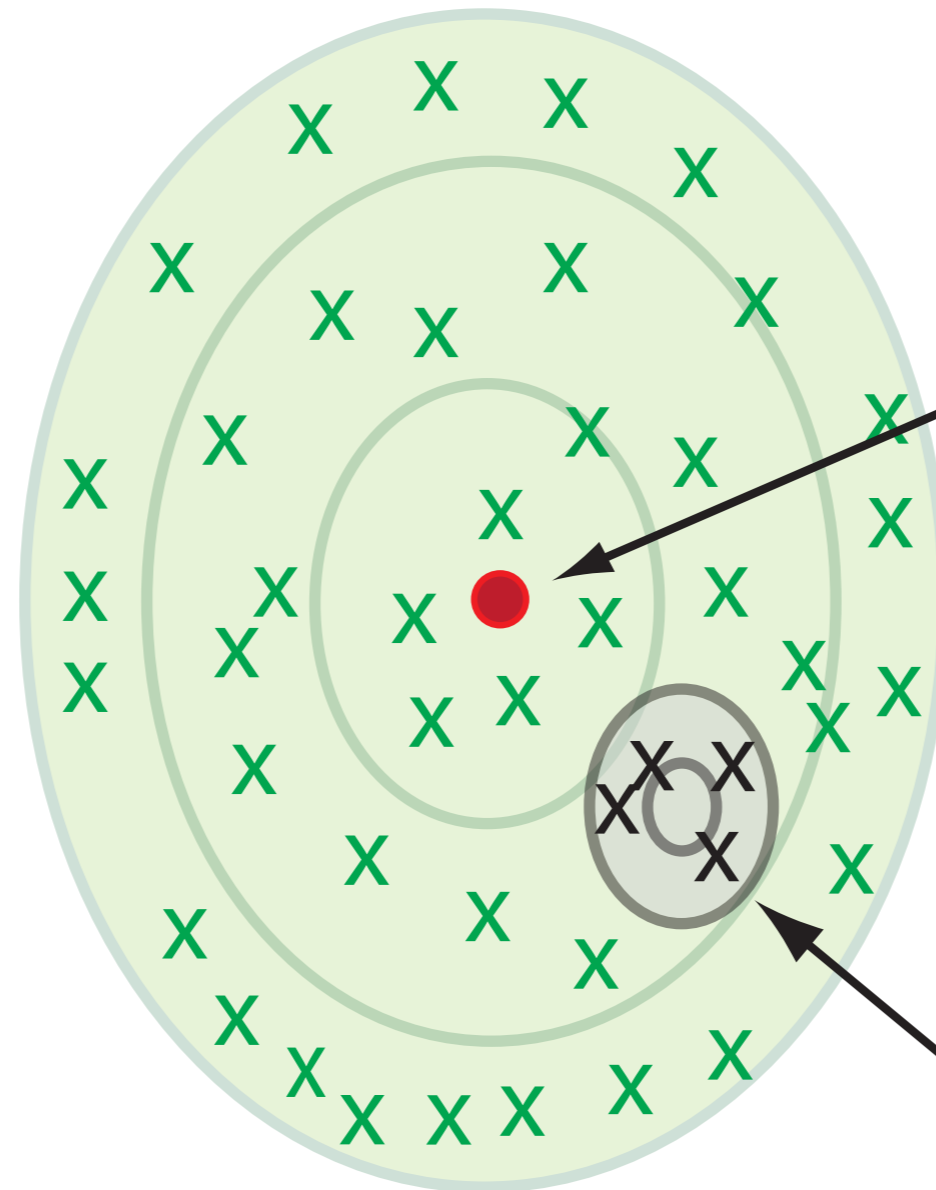
$$\text{Var} = \sum_i E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2],$$

The **bias-variance tradeoff** summarizes the fundamental tension in machine learning, particularly supervised learning, between the complexity of a model and the amount of training data needed to train it.

Since data is often limited, in practice it is often useful to use a less-complex model with higher bias – a model whose asymptotic performance is worse than another model – because it is easier to train and less sensitive to sampling noise arising from having a finite-sized training dataset (smaller variance).

The bias-variance tradeoff

**High variance,
low-bias model**



True model

**Low variance,
high-bias model**

Thus, depending on the amount of training data, it may be more favorable to use a less complex, high-bias model to make predictions.

Gaussian Mixture Models

A Gaussian Mixture Model (GMM) is a parametric probability density function represented as a weighted sum of Gaussian component densities.

GMM parameters are estimated from training data using the iterative Expectation-Maximization (EM)

$$p(\mathbf{x}|\lambda) = \sum_{i=1}^M w_i g(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i), \quad \sum_{i=1}^M w_i = 1.$$

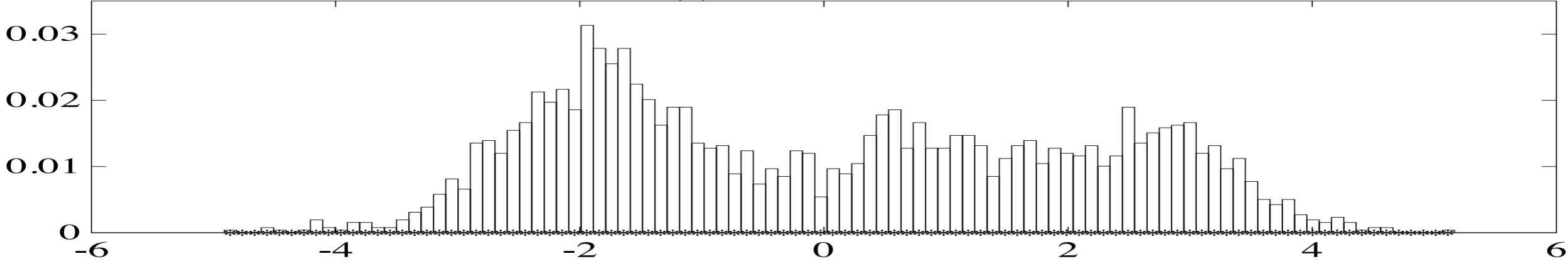
$$g(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) = \frac{1}{(2\pi)^{D/2} |\boldsymbol{\Sigma}_i|^{1/2}} \exp \left\{ -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_i)' \boldsymbol{\Sigma}_i^{-1} (\mathbf{x} - \boldsymbol{\mu}_i) \right\},$$

The complete Gaussian mixture model is parameterized by the mean vectors, covariance matrices and mixture weights from all component densities

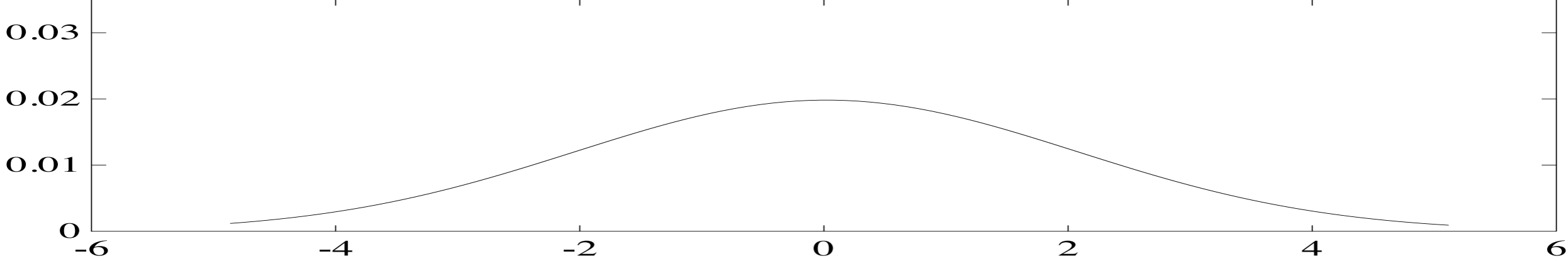
$$\lambda = \{w_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\} \quad i = 1, \dots, M.$$

Gaussian Mixture Models

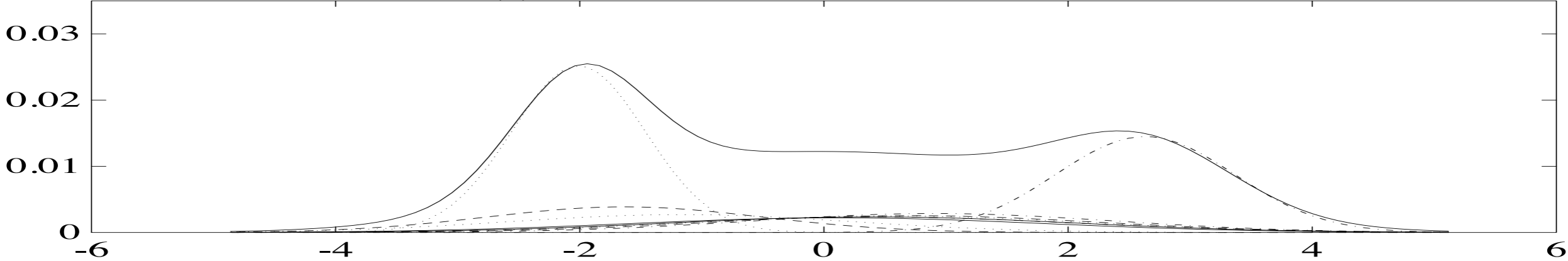
(a) HISTOGRAM



(b) UNIMODAL GAUSSIAN



(c) GAUSSIAN MIXTURE DENSITY



Maximum Likelihood Parameter Estimation

Given training vectors and a GMM configuration, we wish to estimate the parameters of the GMM, λ , which in some sense best matches the distribution of the training feature vectors.

The aim of ML estimation is to find the model parameters which maximize the likelihood of the GMM given the training data. For a sequence of T training vectors $X = \{x_1, \dots, x_T\}$, the GMM likelihood, assuming independence between the vectors, can be written as,

$$p(X|\lambda) = \prod_{t=1}^T p(\mathbf{x}_t|\lambda).$$

Unfortunately, this expression is a non-linear function of the parameters λ and direct maximization is not possible. However, ML parameter estimates can be obtained iteratively using a special case of the **expectation-maximization (EM)** algorithm .

The basic idea of the EM algorithm is, beginning with an initial model λ , to estimate a new model λ^- , such that $p(X|\lambda^-) \geq p(X|\lambda)$

$$\bar{w}_i = \frac{1}{T} \sum_{t=1}^T \text{Pr}(i|\mathbf{x}_t, \lambda). \quad \bar{\mu}_i = \frac{\sum_{t=1}^T \text{Pr}(i|\mathbf{x}_t, \lambda) \mathbf{x}_t}{\sum_{t=1}^T \text{Pr}(i|\mathbf{x}_t, \lambda)}. \quad \bar{\sigma}_i^2 = \frac{\sum_{t=1}^T \text{Pr}(i|\mathbf{x}_t, \lambda) x_t^2}{\sum_{t=1}^T \text{Pr}(i|\mathbf{x}_t, \lambda)} - \bar{\mu}_i^2,$$

Classification

Classification algorithms are used when the desired output is a **discrete label**.

Many use cases, such as determining whether an email is spam or not, have only two possible outcomes. This is called binary classification (on the other hand, regression is useful for predicting outputs that are continuous)

Types of classification algorithms in Machine Learning:

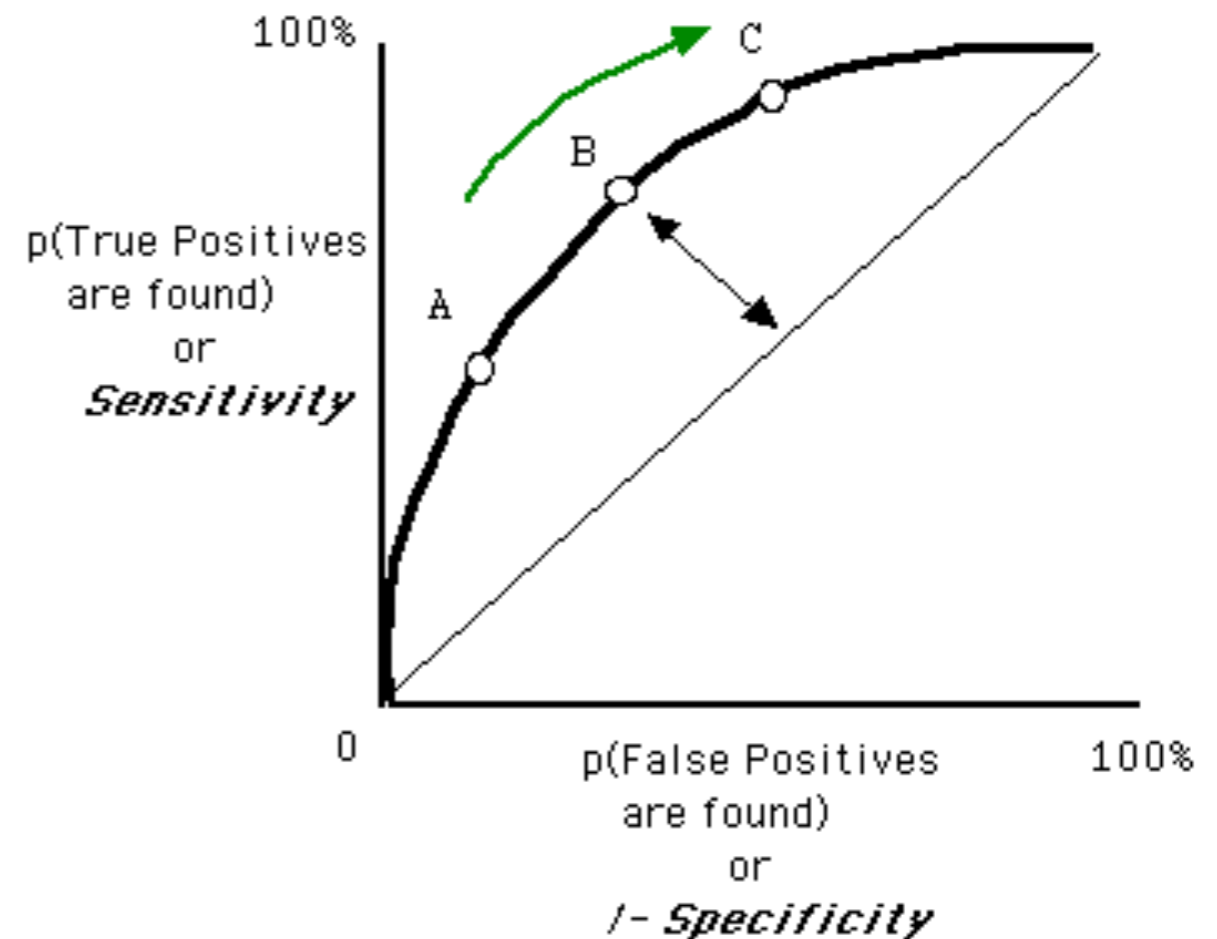
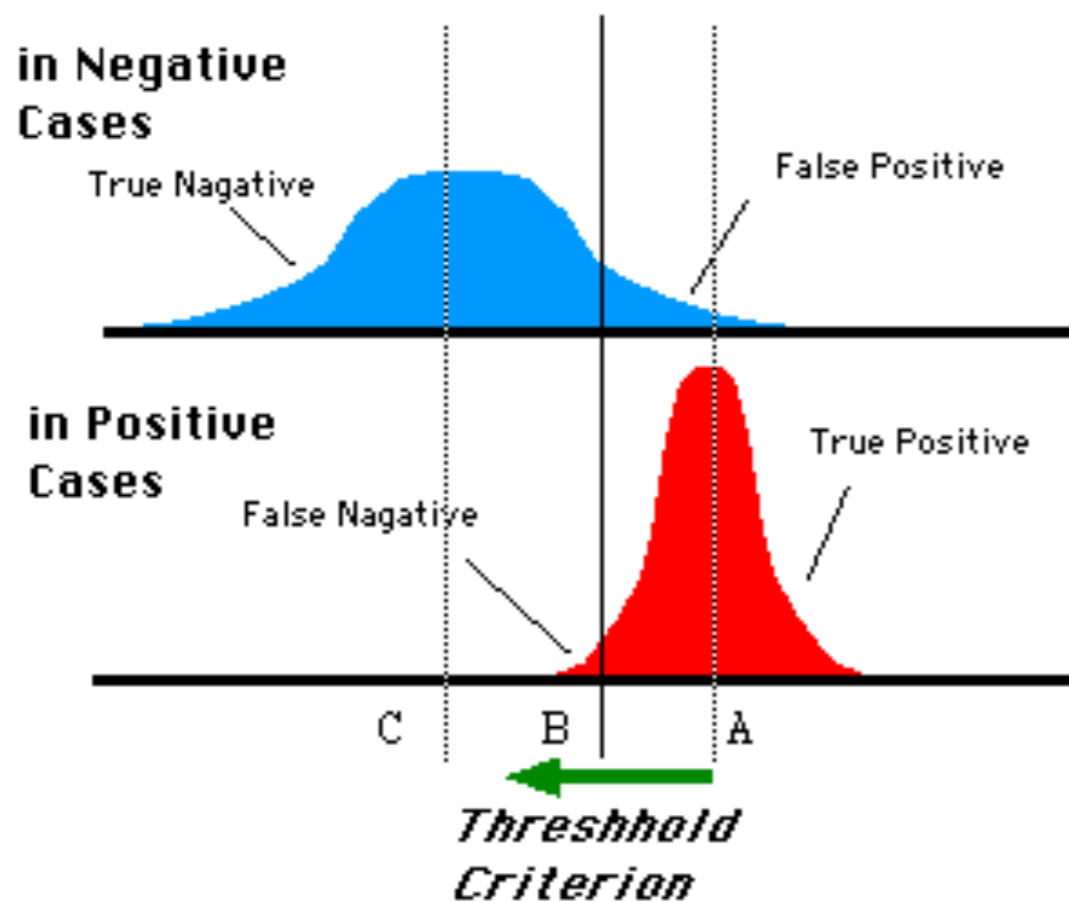
- Linear Classifiers: Logistic Regression, Naive Bayes Classifier
- Support Vector Machines
- Decision Trees
- Boosted Trees
- Random Forest
- Neural Networks
- Nearest Neighbor

Classification

ROC curve (Receiver Operating Characteristics)

ROC curve is used for visual comparison of classification models which shows the trade-off between the true positive rate and the false positive rate. **The area under the ROC curve is a measure of the accuracy of the model.** When a model is closer to the diagonal, it is less accurate and the model with perfect accuracy will have an area of 1.0

Distributions of the Observed signal strength



Linkage-Based Clustering Algorithms

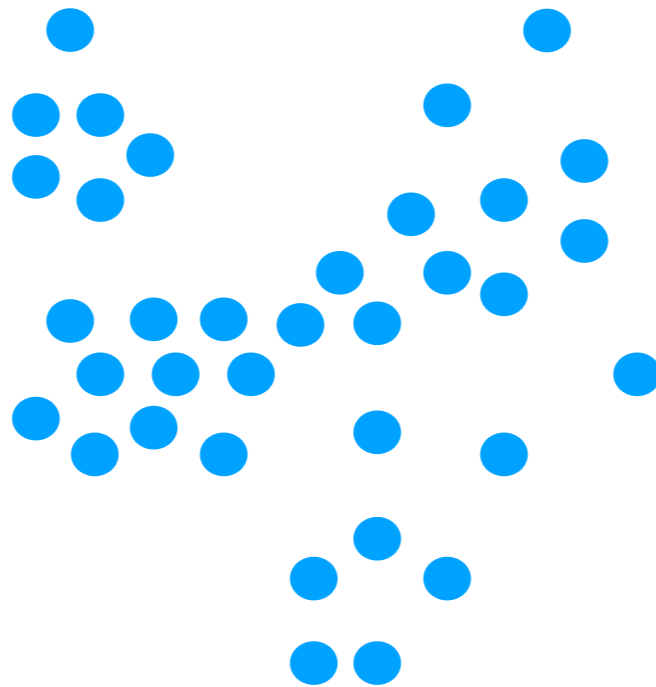
Linkage-based clustering is probably the simplest and most straightforward paradigm of clustering.

They start from the trivial clustering that has each data point as a single-point cluster. Then, repeatedly, these algorithms merge the “closest” clusters of the previous clustering.

Two parameters, then, need to be determined to define such an algorithm clearly.

First, we have to decide how to measure (or define) the distance between clusters, and, second, we have to determine when to stop merging.

There are many ways of extending d to a measure of distance between domain subsets (or clusters). The most common ways are



Linkage-Based Clustering Algorithms

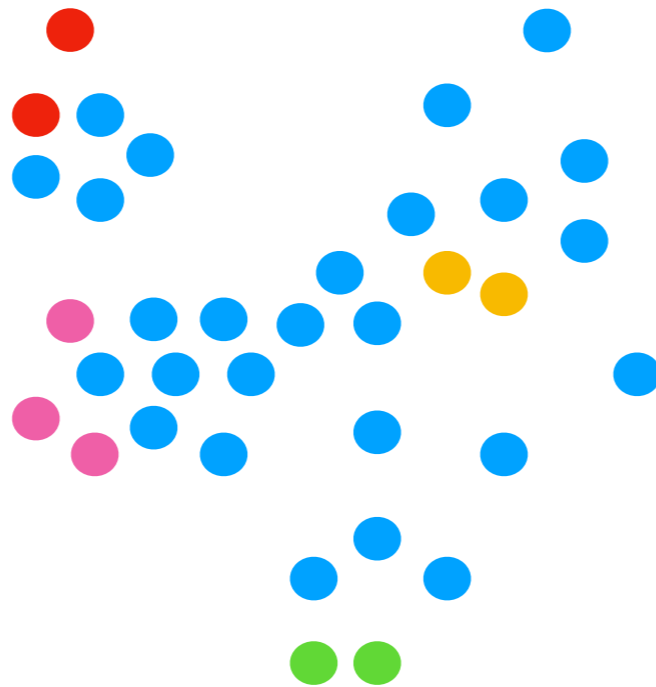
Linkage-based clustering is probably the simplest and most straightforward paradigm of clustering.

They start from the trivial clustering that has each data point as a single-point cluster. Then, repeatedly, these algorithms merge the “closest” clusters of the previous clustering.

Two parameters, then, need to be determined to define such an algorithm clearly.

First, we have to decide how to measure (or define) the distance between clusters, and, second, we have to determine when to stop merging.

There are many ways of extending d to a measure of distance between domain subsets (or clusters). The most common ways are



Linkage-Based Clustering Algorithms

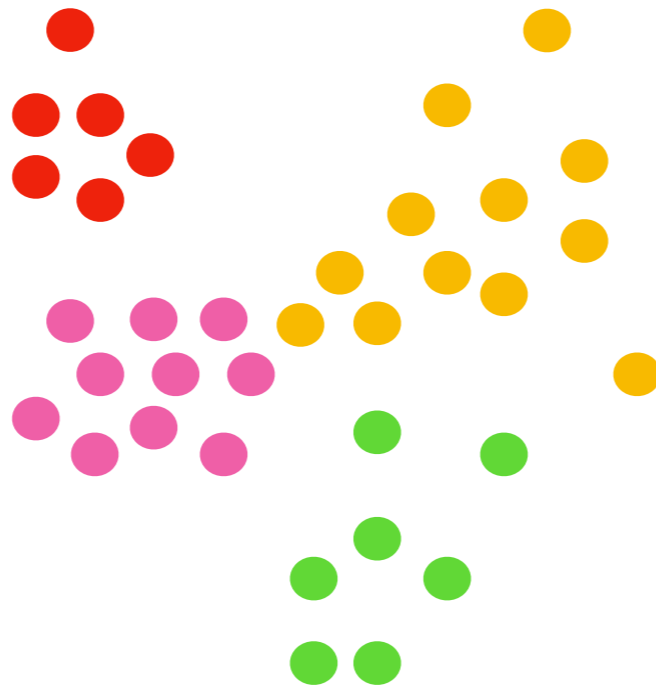
Linkage-based clustering is probably the simplest and most straightforward paradigm of clustering.

They start from the trivial clustering that has each data point as a single-point cluster. Then, repeatedly, these algorithms merge the “closest” clusters of the previous clustering.

Two parameters, then, need to be determined to define such an algorithm clearly.

First, we have to decide how to measure (or define) the distance between clusters, and, second, we have to determine when to stop merging.

There are many ways of extending d to a measure of distance between domain subsets (or clusters). The most common ways are



Linkage-Based Clustering Algorithms

Linkage-based clustering is probably the simplest and most straightforward paradigm of clustering.

They start from the trivial clustering that has each data point as a single-point cluster. Then, repeatedly, these algorithms merge the “closest” clusters of the previous clustering.

Two parameters, then, need to be determined to define such an algorithm clearly.

First, we have to decide how to measure (or define) the distance between clusters, and, second, we have to determine when to stop merging.

There are many ways of extending d to a measure of distance between domain subsets (or clusters). The most common ways are

1. **Single Linkage** clustering, in which the between-clusters distance is defined by the minimum distance between members of the two clusters, namely,

$$D(A, B) \stackrel{\text{def}}{=} \min\{d(x, y) : x \in A, y \in B\}$$

2. **Average Linkage** clustering, in which the distance between two clusters is defined to be the average distance between a point in one of the clusters and a point in the other, namely,

$$D(A, B) \stackrel{\text{def}}{=} \frac{1}{|A||B|} \sum_{x \in A, y \in B} d(x, y)$$

3. **Max Linkage** clustering, in which the distance between two clusters is defined as the maximum distance between their elements, namely,

$$D(A, B) \stackrel{\text{def}}{=} \max\{d(x, y) : x \in A, y \in B\}.$$

K-means clustering is a type of **unsupervised** learning, which is used when you have **unlabeled** data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K .

The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity.

The results of the K-means clustering algorithm are:

1. The centroids of the K clusters, which can be used to label new data
2. Labels for the training data (each data point is assigned to a single cluster)

Algorithm

The K-means clustering algorithm uses iterative **refinement** to produce a final result. The algorithm inputs are the number of clusters K and the data set. The data set is a collection of features for each data point. The algorithm starts with initial estimates for the K centroids, which can either be randomly generated or randomly selected from the data set.

The algorithm then iterates between two steps:

1. Data assignment step:

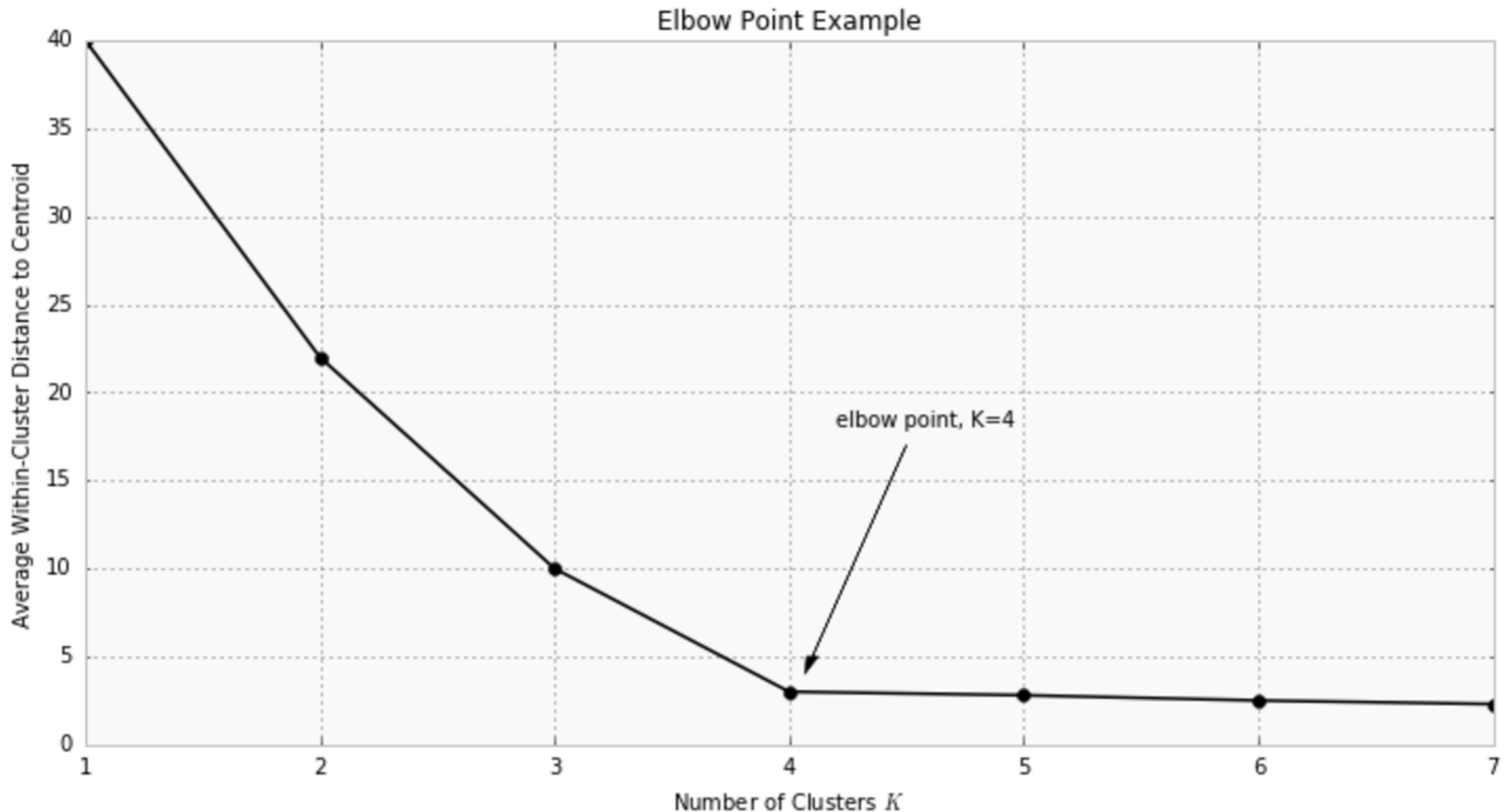
Each centroid defines one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance.

2. Centroid update step:

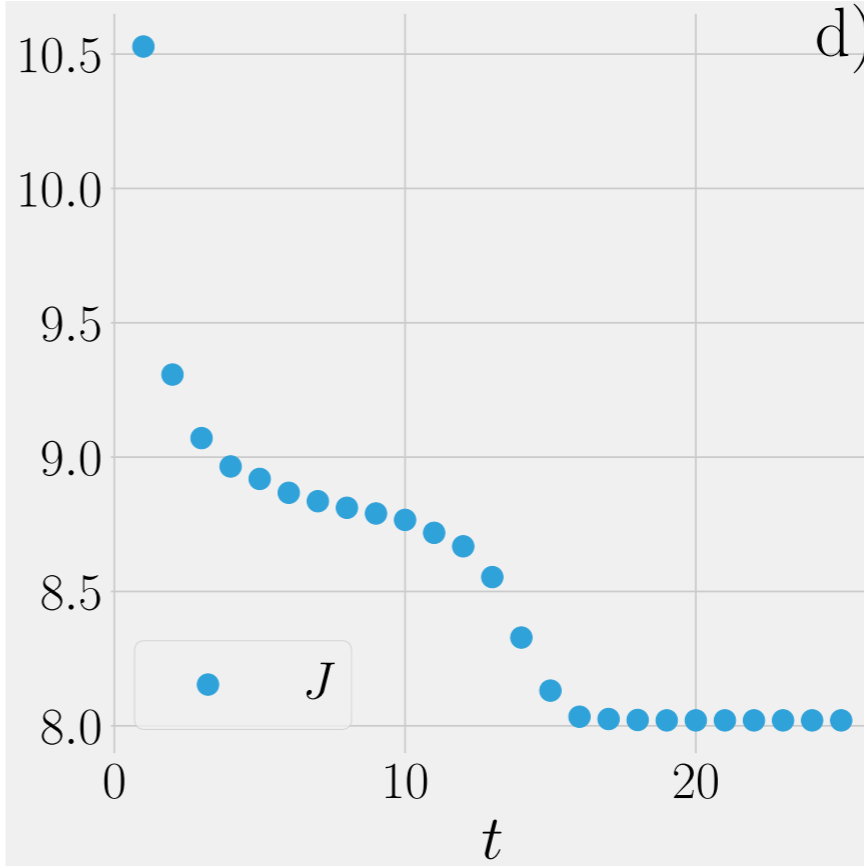
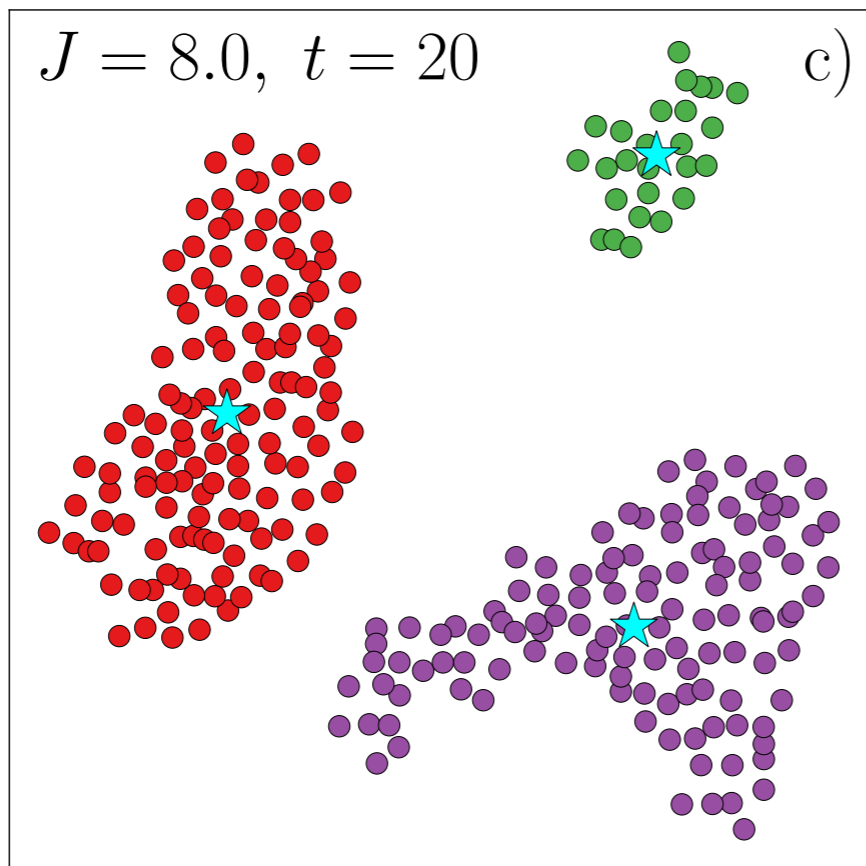
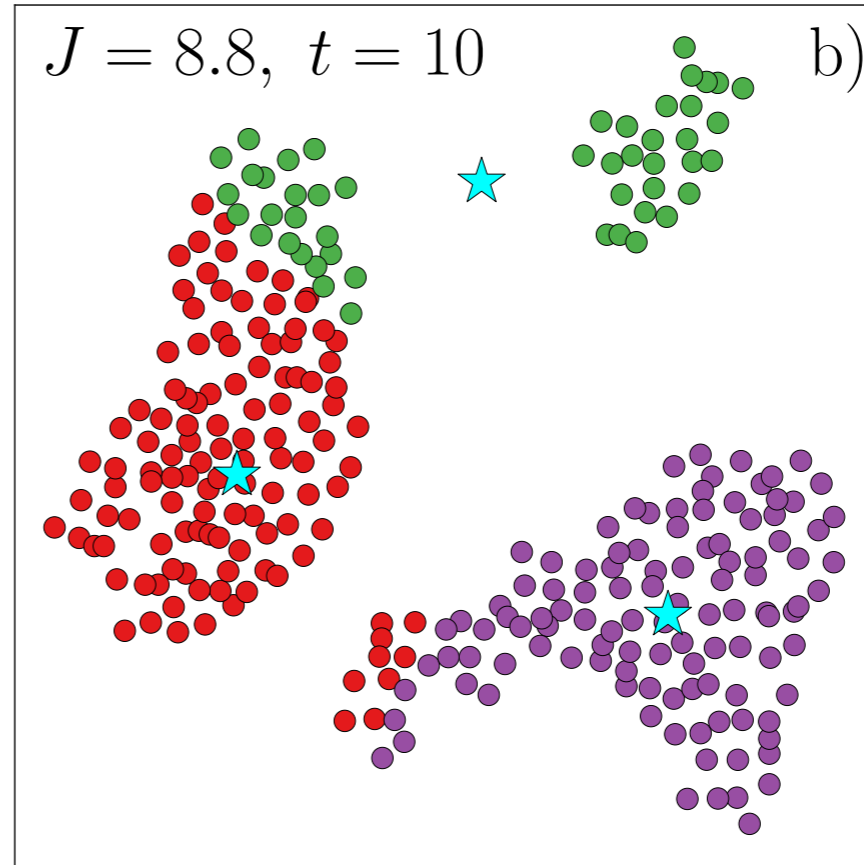
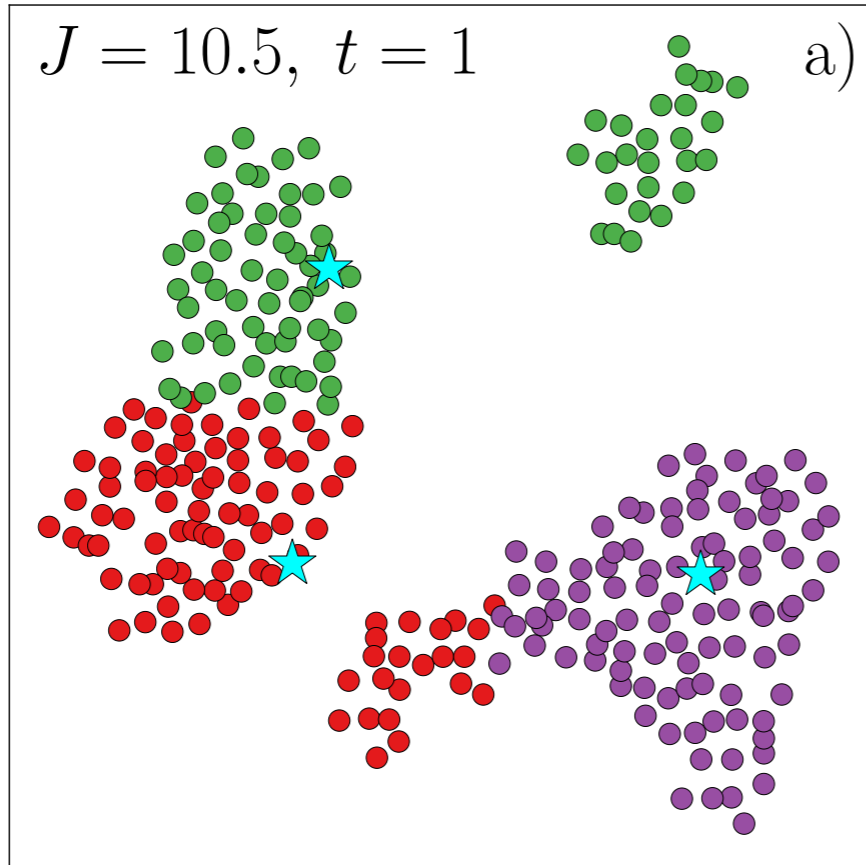
In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid's cluster.

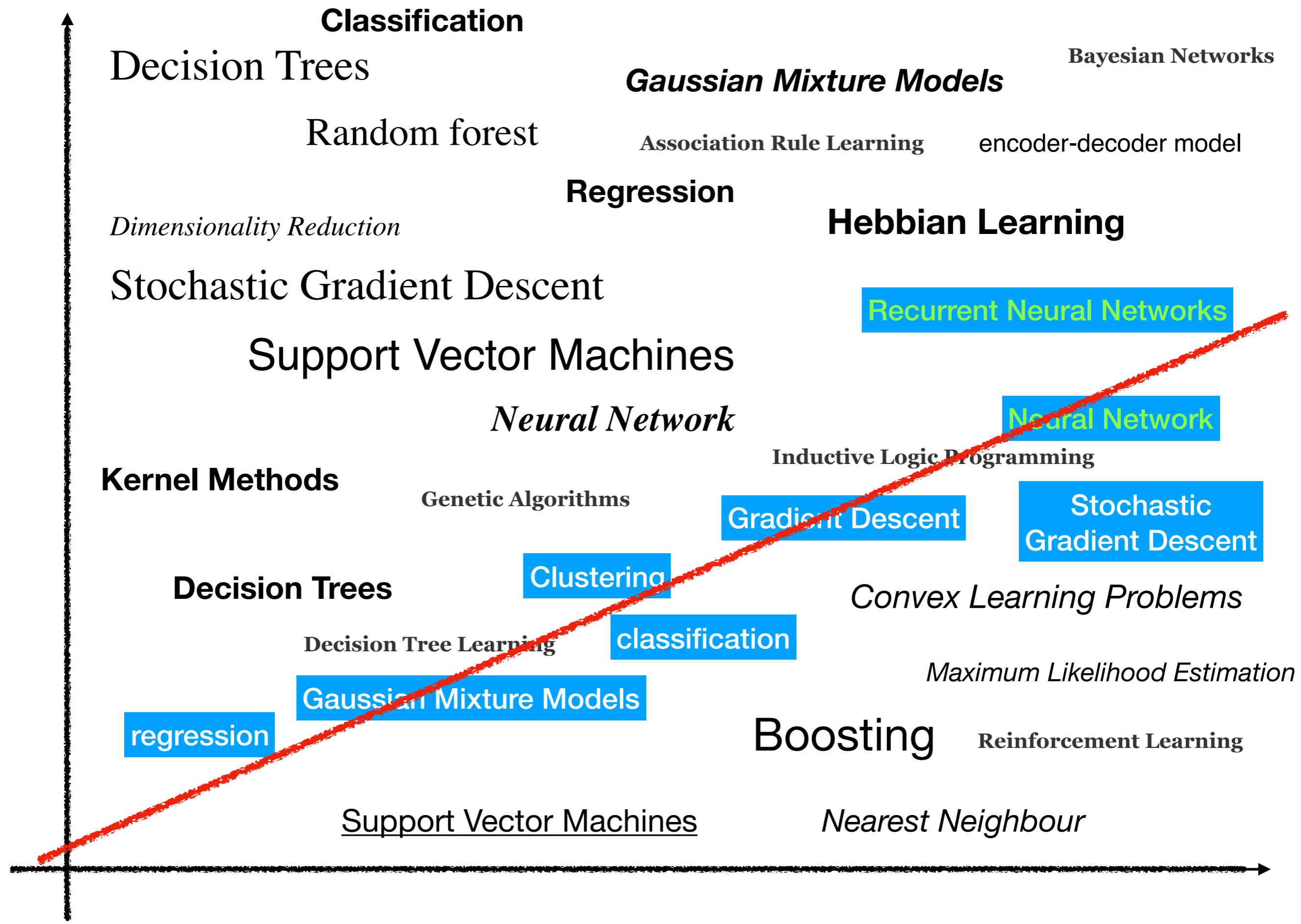
K-means clustering

One of the metrics that is commonly used to compare results across different values of K is the mean distance between data points and their cluster centroid. Since increasing the number of clusters will always reduce the distance to data points, increasing K will always decrease this metric, to the extreme of reaching zero when K is the same as the number of data points



K-means clustering





Stochastic Gradient Descent

Neural Network Model

Backpropagation algorithm

Supervised Learning

Supervised learning adjusts network parameters by a direct comparison between the actual network output and the desired output. Supervised learning is a closed-loop feedback system, where the error is the feedback signal. The error measure is usually defined by the mean squared error (MSE)

$$E = \frac{1}{N} \sum_{p=1}^N \| \mathbf{y}_p - \hat{\mathbf{y}}_p \|^2 ,$$

Unsupervised Learning

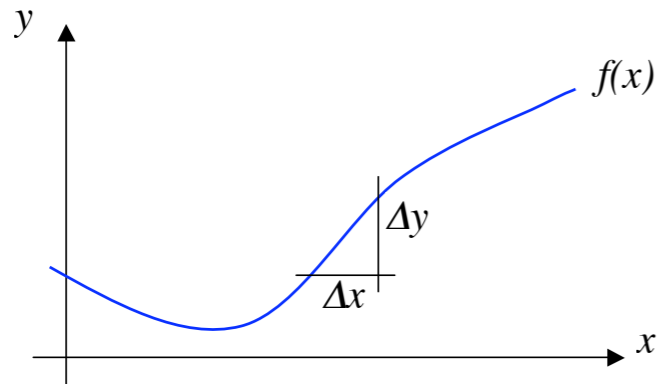
Unsupervised learning involves no target values. It is solely based on the correlations among the input data, and is used to find the significant patterns or features in the input data without the help of a teacher

Reinforcement learning

is a special case of supervised learning, where the exact desired output is unknown. The teacher supplies only feedback about success or failure of an answer. This is cognitively more plausible than supervised learning since a fully specified correct answer might not always be available to the learner or even the teacher. It is based only on the information as to whether or not the actual output is close to the estimate.

Stochastic Gradient Descent (SGD)

The branch of mathematics concerned with computing gradients is called Differential Calculus. The relevant general idea is straightforward. Consider a function $y = f(x)$



$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

If we want to change the value of x to minimise a function $f(x)$, what we need to do depends on the gradient of $f(x)$ at the current value of x . There are three cases:

If $\frac{\partial f}{\partial x} > 0$ then $f(x)$ increases as x increases so we should decrease x

If $\frac{\partial f}{\partial x} < 0$ then $f(x)$ decreases as x increases so we should increase x

If $\frac{\partial f}{\partial x} = 0$ then $f(x)$ is at a maximum or minimum so we should not change x

In summary, we can decrease $f(x)$ by changing x by the amount:

$$\Delta x = x_{new} - x_{old} = -\eta \frac{\partial f}{\partial x}$$

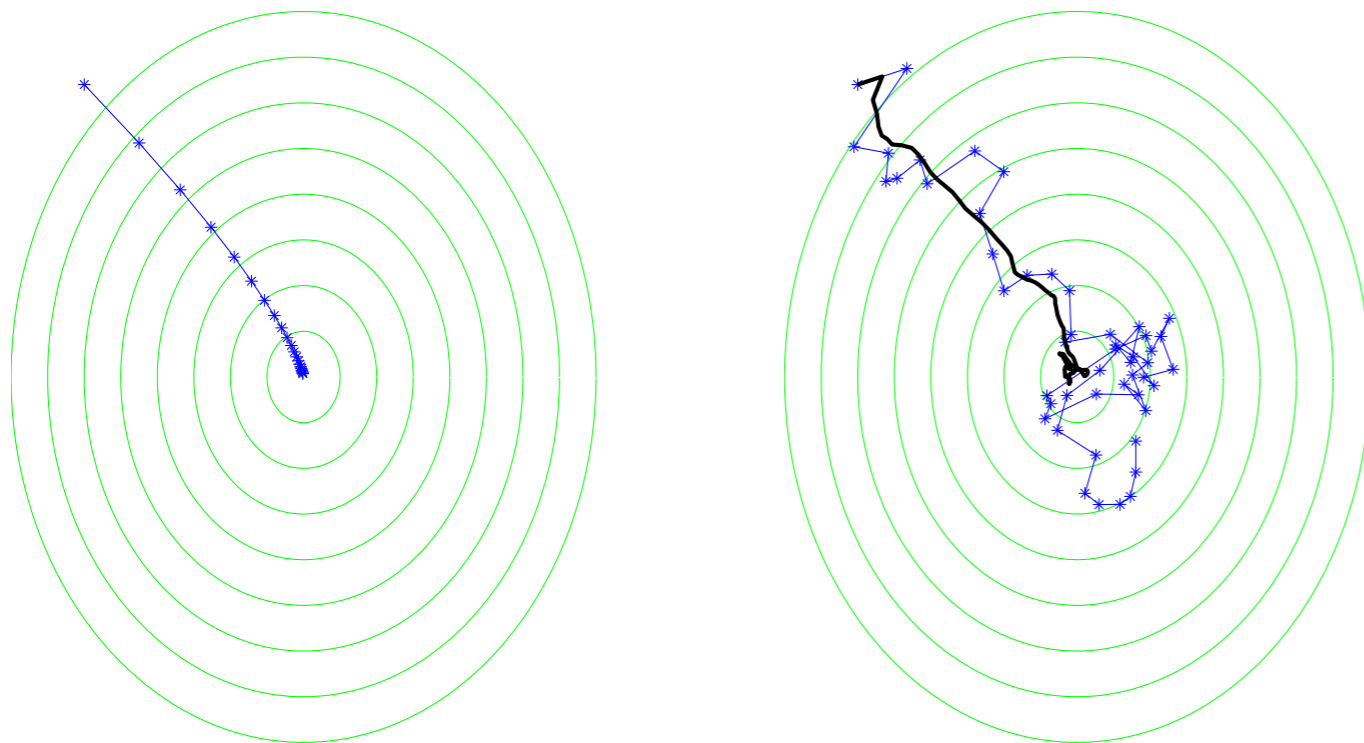
where η is a small positive constant specifying how much we change x by, and the derivative $\partial f / \partial x$ tells us which direction to go in. If we repeatedly use this equation, $f(x)$ will (assuming η is sufficiently small) keep descending towards a minimum, and hence this procedure is known as **gradient descent minimisation**.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent: we are minimizing the risk function, and since we do not know D we also do not know the gradient of $LD(\mathbf{w})$. SGD circumvents this problem by allowing the optimization procedure to take a step along a random direction, as long as the expected value of the direction is the negative of the gradient.

Gradient descent is an iterative algorithm. We start with an initial value of \mathbf{w} (say, $\mathbf{w}^{(1)} = 0$). Then, at each iteration, we take a step in the direction of the negative of the gradient at the current point.

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla f(\mathbf{w}^{(t)}), \quad \eta > 0$$



In **stochastic gradient descent** we do not require the update direction to be based exactly on the gradient. Instead, we allow the direction to be a random vector and only require that its expected value at each iteration will equal the gradient direction.

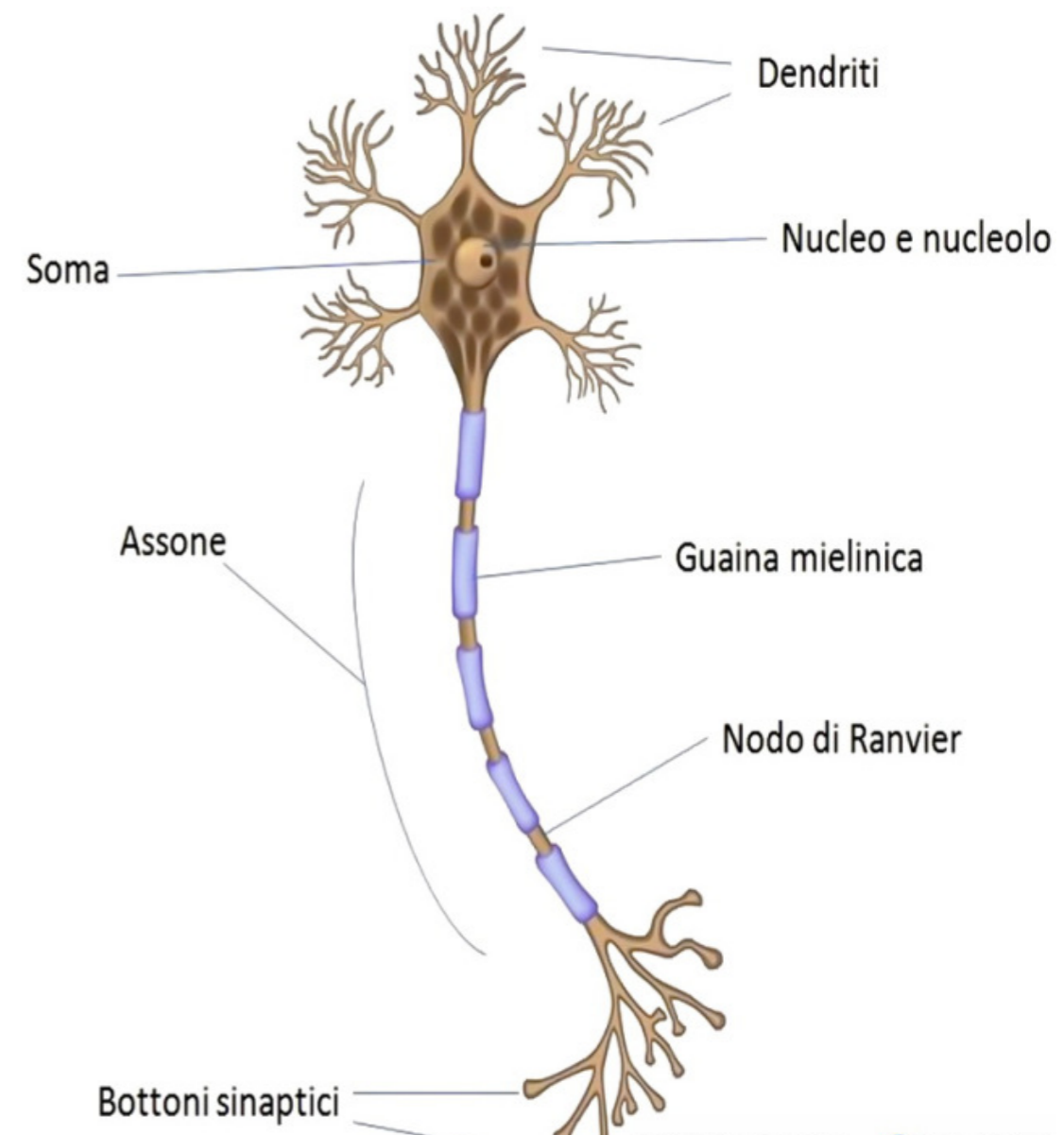
Stochastic Gradient Descent (SGD)

The basic idea behind these methods is straightforward: iteratively adjust the parameters in the direction where the gradient of the cost function is large and negative.

Neural Networks

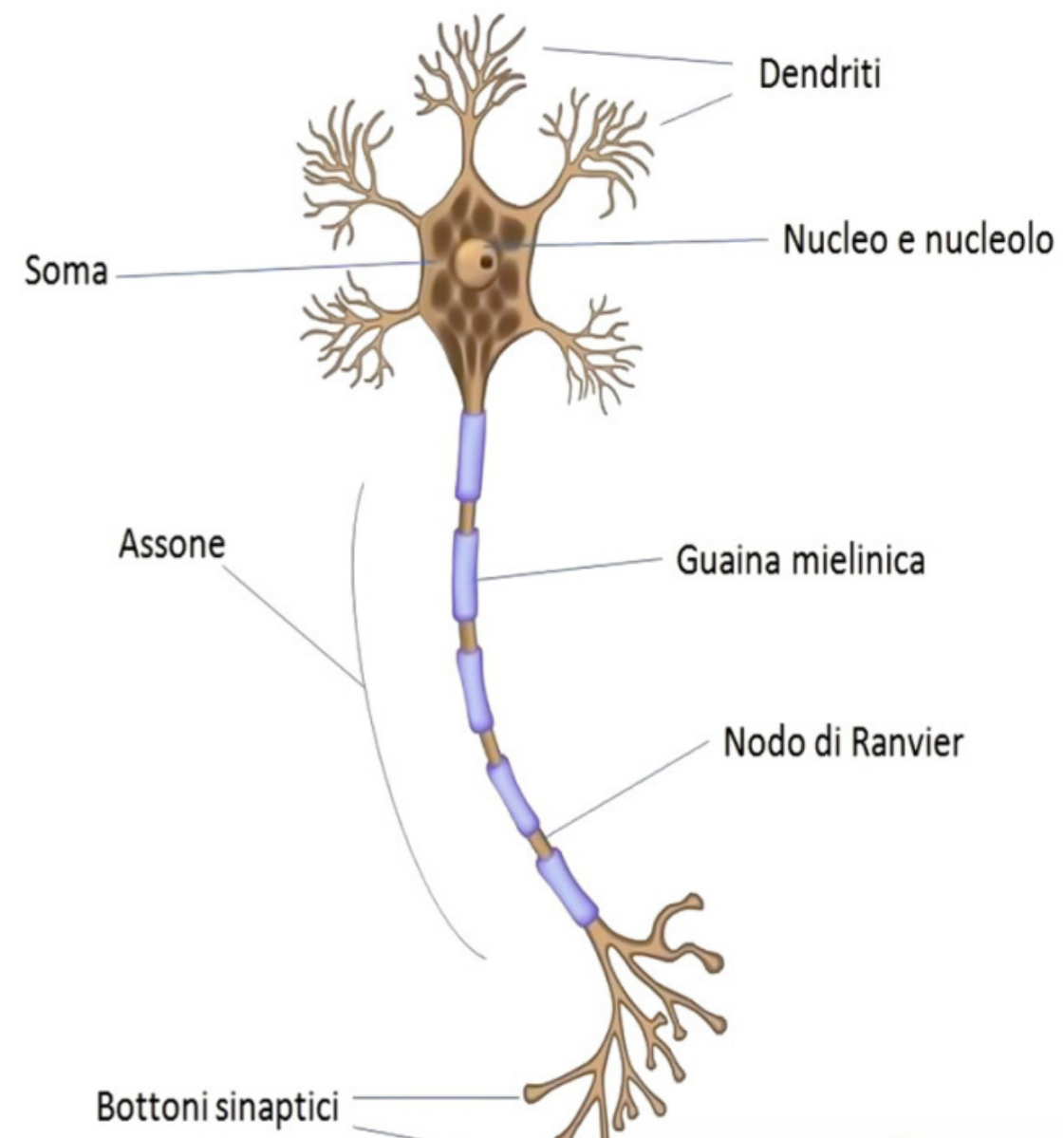
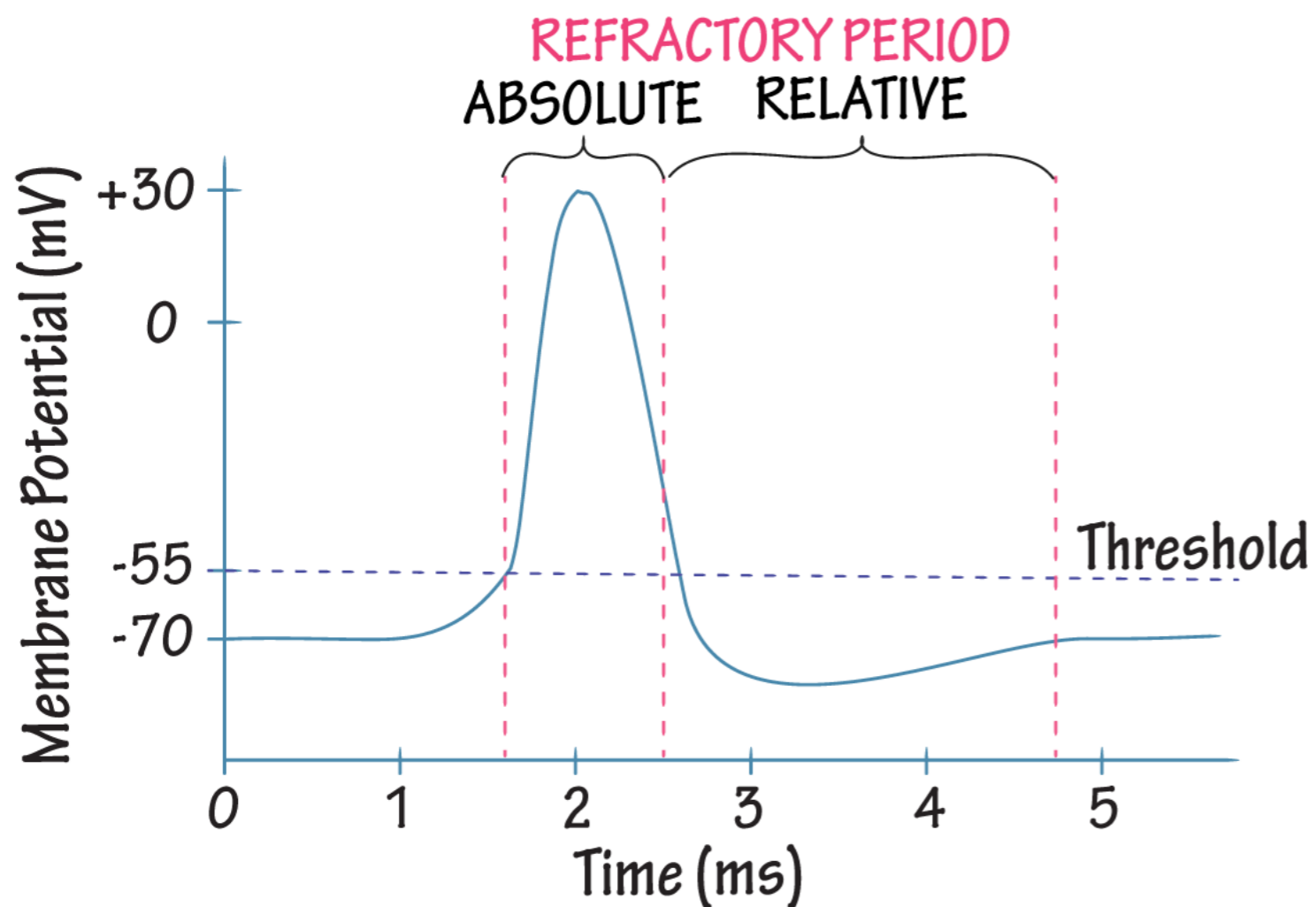
An artificial neural network learning algorithm is a learning algorithm that is inspired by the structure and functional aspects of **biological neural networks**.

Computations are structured in terms of an interconnected group of artificial neurons, processing information using a connectionist approach to computation. Modern neural networks are **non-linear statistical data modeling tools**.



We will use them to model complex relationships between inputs and outputs, to find **patterns in data**.

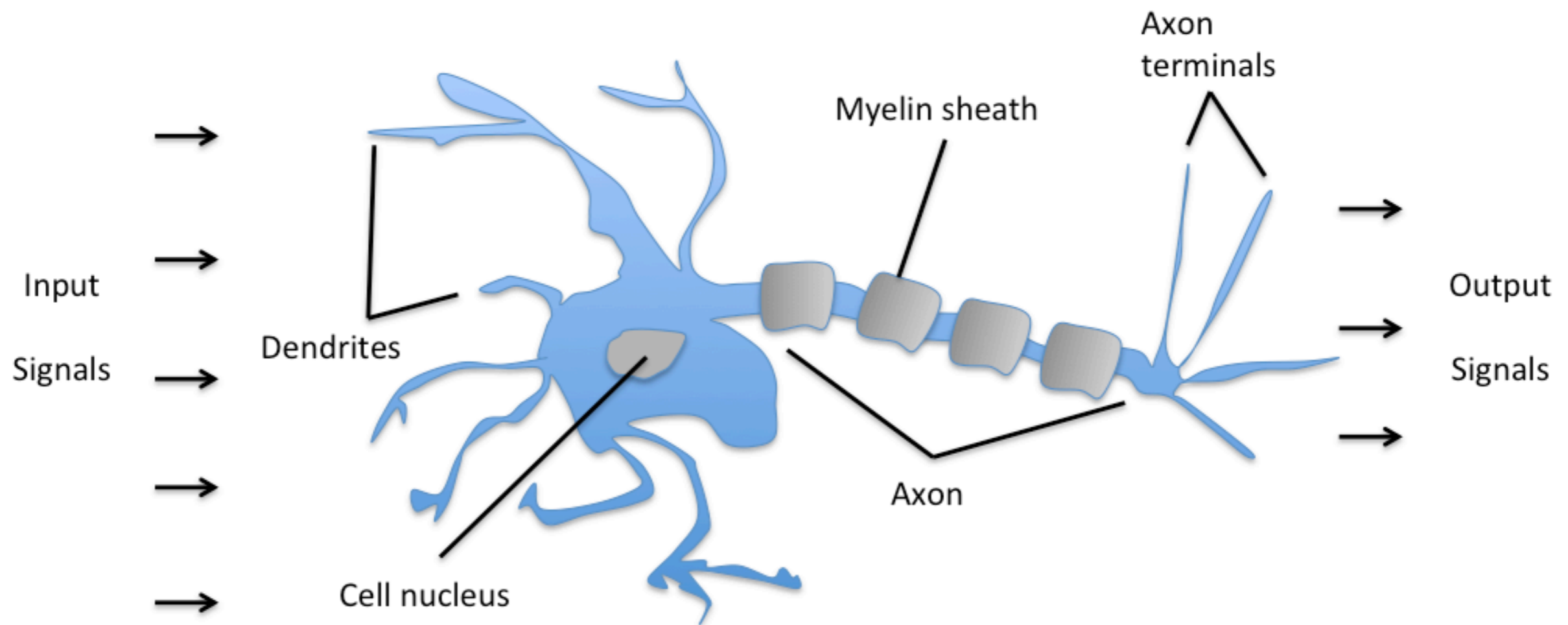
Neural Networks



An **action potential** occurs when a neuron sends information down an axon, away from the cell body. Neuroscientists use other words, such as a "spike" or an "impulse" for the action potential. Some event (a stimulus) causes the resting potential to move toward 0 mV. When the depolarization reaches about -55 mV a neuron will fire an action potential. This is the **threshold**. If the neuron does not reach this critical threshold level, then no action potential will fire.

Neural Networks

The signals of variable magnitudes arrive at the dendrites. Those input signals are then accumulated in the cell body of the neuron, and if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.



Schematic of a biological neuron.

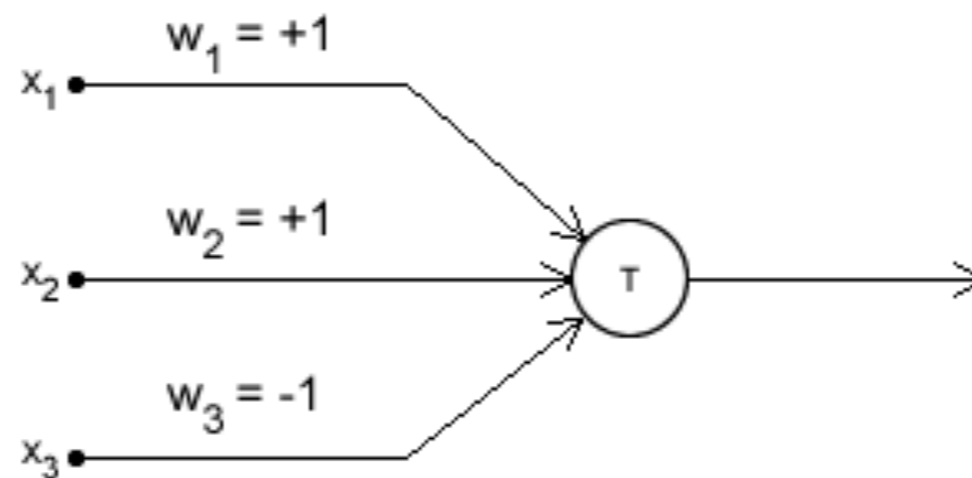
Conceptually, it is helpful to divide neural networks into four categories:

- (i) general purpose neural networks for **supervised learning**,
- (ii) neural networks designed specifically for **image processing**, the most prominent ex- ample of this class being Convolutional Neural Networks (CNNs),
- (iii) neural networks for **sequential data** such as Recurrent Neural Networks (RNNs),
- (iv) neural networks for **unsupervised learning** such as Deep Boltzmann Machines.

First artificial neurons: The McCulloch-Pitts model

Warren McCulloch and Walter Pitts, 1943

The McCulloch-Pitts model was an extremely simple artificial neuron. The inputs could be either a **zero** or a **one**. And the output was a zero or a one. And each input could be either excitatory or inhibitory.



The variables **w1**, **w2** and **w3** indicate which input is **excitatory**, and which one is **inhibitory**.

If a weight is 1, it is an excitatory input.

If it is -1, it is an inhibitory input.

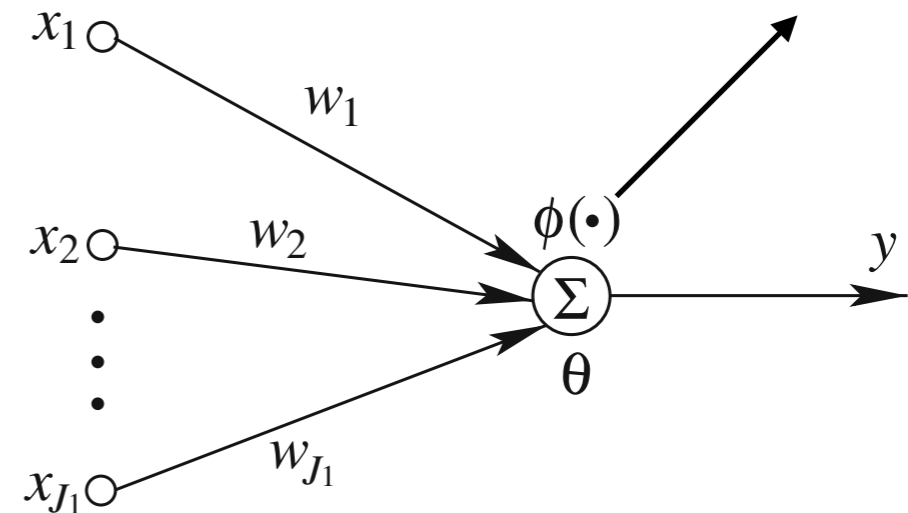
Neural Networks: One-Neuron Perceptron

Rosenblatt, 1958 Frank Rosenblatt published the first concept of the Perceptron learning rule

The activation function represents a linear or nonlinear mapping from the input to the output and is denoted by $\phi(\cdot)$

$$net = \sum_{i=1}^{J_1} w_i x_i - \theta = \mathbf{w}^T \mathbf{x} - \theta,$$

The output of the neuron



A perceptron receives multiple input signals, and if the sum of the input signals exceed a certain threshold it either returns a signal or remains “silent” otherwise

The Perceptron Learning Rule

- 1) Initialize the weights to 0 or small random numbers
- 2) For each training sample $\mathbf{x}(i)$
 1. Calculate the *output* value.
 2. Update the weights

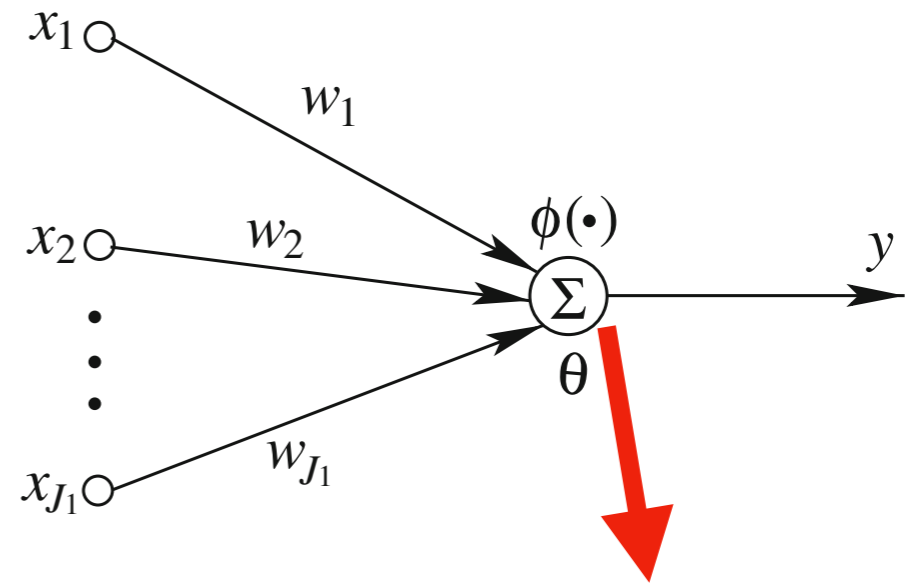
$$w_j := w_j + \Delta w_j$$

$$\Delta w_j = \eta (\text{target}(i) - \text{output}(i)) x(i)$$

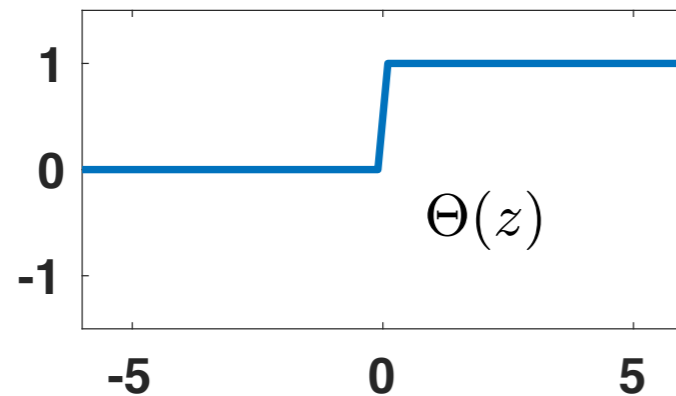
Neural Networks: One-Neuron Perceptron

Rosenblatt, 1958 Frank Rosenblatt published the first concept of the Perceptron learning rule

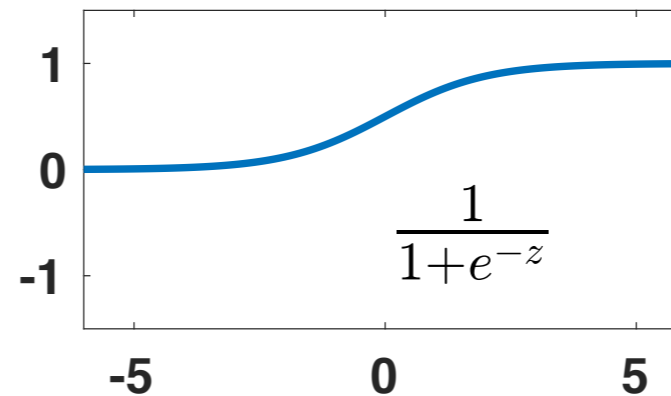
$$net = \sum_{i=1}^{J_1} w_i x_i - \theta = \mathbf{w}^T \mathbf{x} - \theta,$$



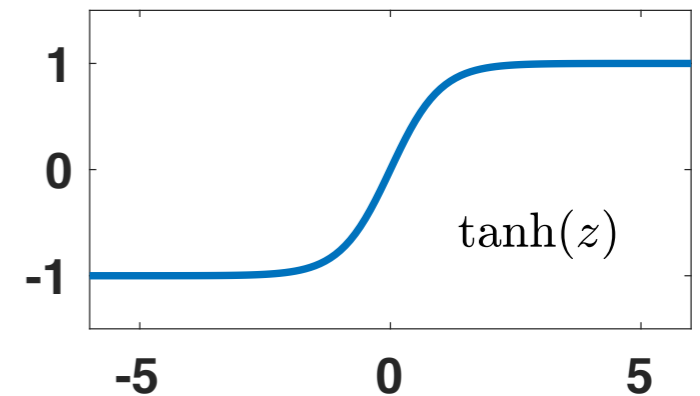
Perceptron



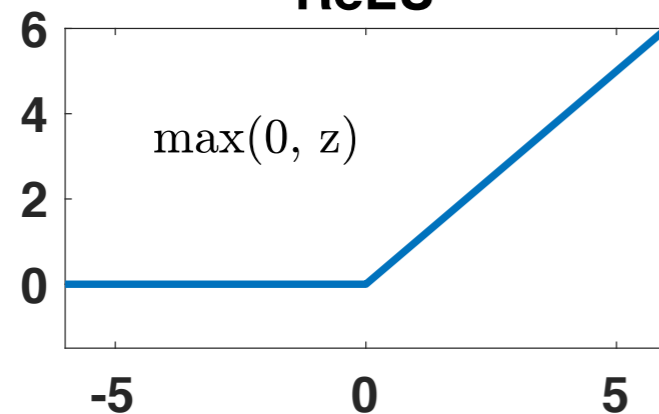
Sigmoid



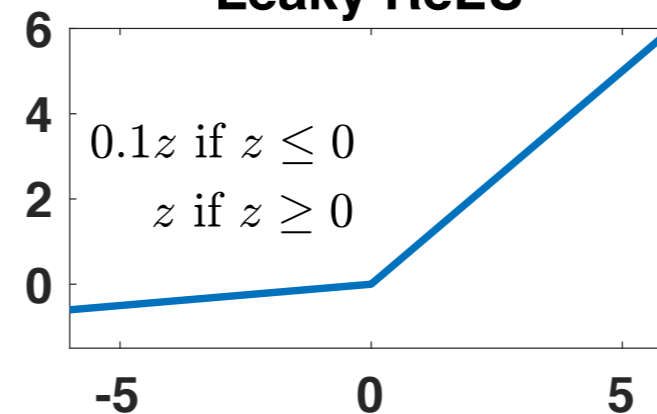
Tanh



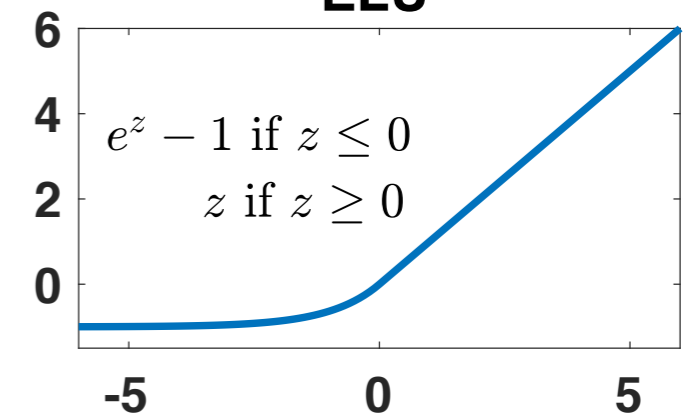
ReLU



Leaky ReLU



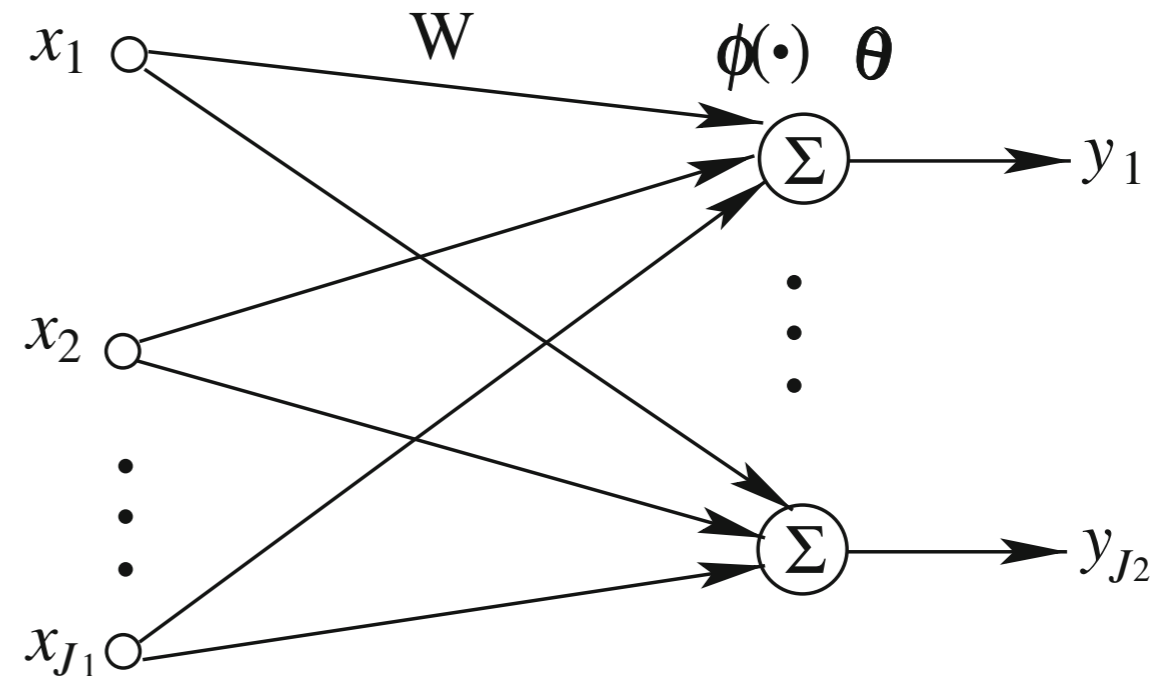
ELU



Neural Networks: Single-Layer Perceptron

When more neurons with the hard-limiter activation function are used, we have a **single-layer perceptron**.

$$\begin{aligned} \mathit{net} &= \mathbf{W}^T \mathbf{x} - \theta, \\ \hat{\mathbf{y}} &= \phi(\mathit{net}), \end{aligned}$$



The single-layer perceptron can be used to classify input vector data \mathbf{x} into more classes.

$$\mathit{net}_{t,j} = \sum_{i=1}^{J_1} x_{t,i} w_{ij}(t) - \theta_j = \mathbf{w}_j^T \mathbf{x}_t - \theta_j,$$

$$\hat{y}_{t,j} = \begin{cases} 1, & \mathit{net}_{t,j} > 0 \\ 0, & \text{otherwise} \end{cases},$$

$$e_{t,j} = y_{t,j} - \hat{y}_{t,j},$$

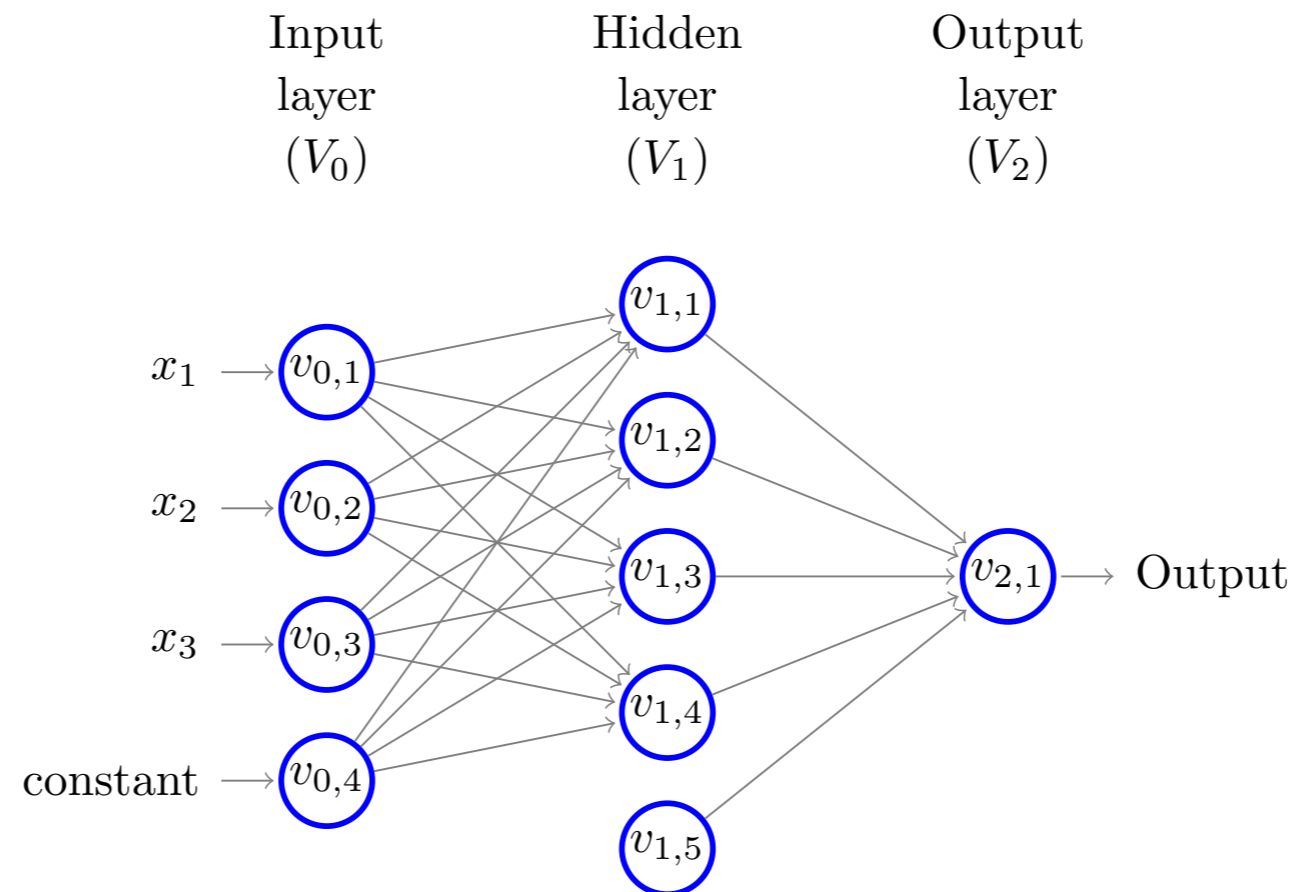
$$w_{ij}(t+1) = w_{ij}(t) + \eta x_{t,i} e_{t,j},$$



Neural Networks: The Multilayer Perceptron

In the case of the MLP, it includes an **input layer** (that does not do any processing), one **output layer** and at least one **hidden layer**.

The MLP generally learns by means of a **backpropagation algorithm**, which is basically a **gradient technique**.



The **backpropagation algorithm** looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem.

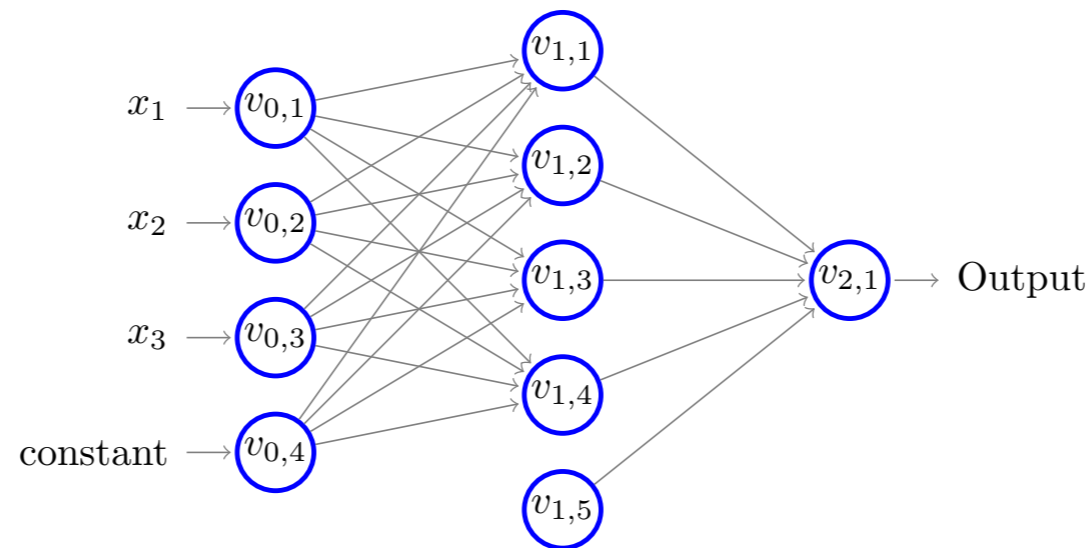
Neural Networks

A neural network consists of units (neurons), arranged in layers, which convert an input vector into some output.

Each unit takes an input, applies a (often nonlinear) function to it and then passes the output on to the next layer.

Generally the networks are defined to be **feed-forward**: a unit feeds its output to all the units on the next layer, but there is no feedback to the previous layer.

Weightings are applied to the signals passing from one unit to another, and it is these weightings which are tuned in the training phase to adapt a neural network to the particular problem at hand.



There are many network architectures available now like Feed-forward, Convolutional, Recurrent etc

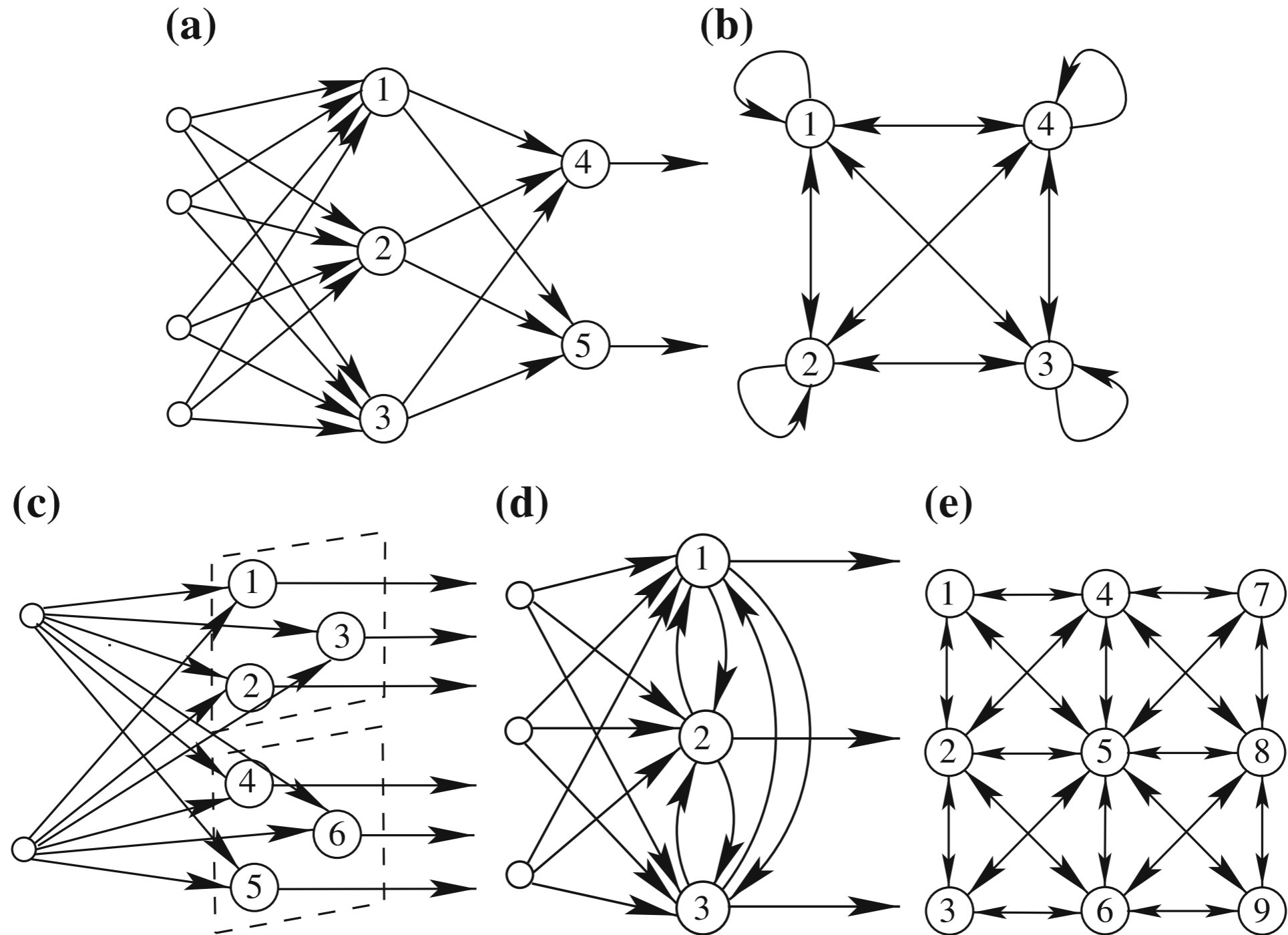


Fig. 1.5 Architecture of neural networks. **a** Layered feedforward network. **b** Recurrent network. **c** Two-dimensional lattice network. **d** Layered feedforward network with lateral connections. **e** Cellular network. The big numbered circles stand for neurons and the small ones for input nodes

Neural Networks: backpropagation algorithm

The basic procedure for training neural is the same as we used for training simpler supervised learning algorithms, such as logistic and linear regression: construct a cost/loss function and then use gradient descent to minimize the cost function.

Neural networks differ from these simpler supervised procedures in that generally they contain multiple hidden layers that make taking the gradient more computationally difficult.

The most successful algorithm for training neural networks is backpropagation, introduced to neural networks by Rumelhart et al. in 1985

Neural Networks: backpropagation algorithm

The backpropagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent.

The combination of weights which minimizes the error function is considered to be a solution of the learning problem.

Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function.

One of the more popular activation functions for backpropagation networks is the sigmoid, a real function defined by the expression.

Neural Networks: backpropagation algorithm

Rumelhart et al. in 1985

L layers in our network with $l = 1, \dots, L$

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) = \sigma(z_j^l), \quad z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l.$$

bias of this neuron

w_{jk} the weight for the connection from the k-th neuron in layer l-1 to the j-th neuron in layer l.

in a feed-forward neural network the activation a_j^l of the j-th neuron in the l-th layer can be related to the activities of the neurons in the layer l-1

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L}.$$

error Δ_j^L of the j-th neuron in the L-th layer as the change in cost function with respect to the weighted input z_j^L

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l} \sigma'(z_j^l),$$

the error of neuron j in layer l, Δ_j^l , as the change in the cost function

where in the last line we have used the fact that $\partial b_j^l / \partial z_j^l = 1$

Notice that the error function Δ_j^l can also be interpreted as the partial derivative of the cost function with respect to the bias b_j^l

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l} \frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l},$$

Neural Networks: backpropagation algorithm

Since the error depends on neurons in layer l only through the activation of neurons in the subsequent layer $l + 1$, we can use the chain rule to write

$$\begin{aligned}\Delta_j^l &= \frac{\partial E}{\partial z_j^l} = \sum_k \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \Delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l).\end{aligned}$$

The final equation can be derived by differentiating of the cost function with respect to the weight w_{jk}^l as

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}$$

Neural Networks: backpropagation algorithm

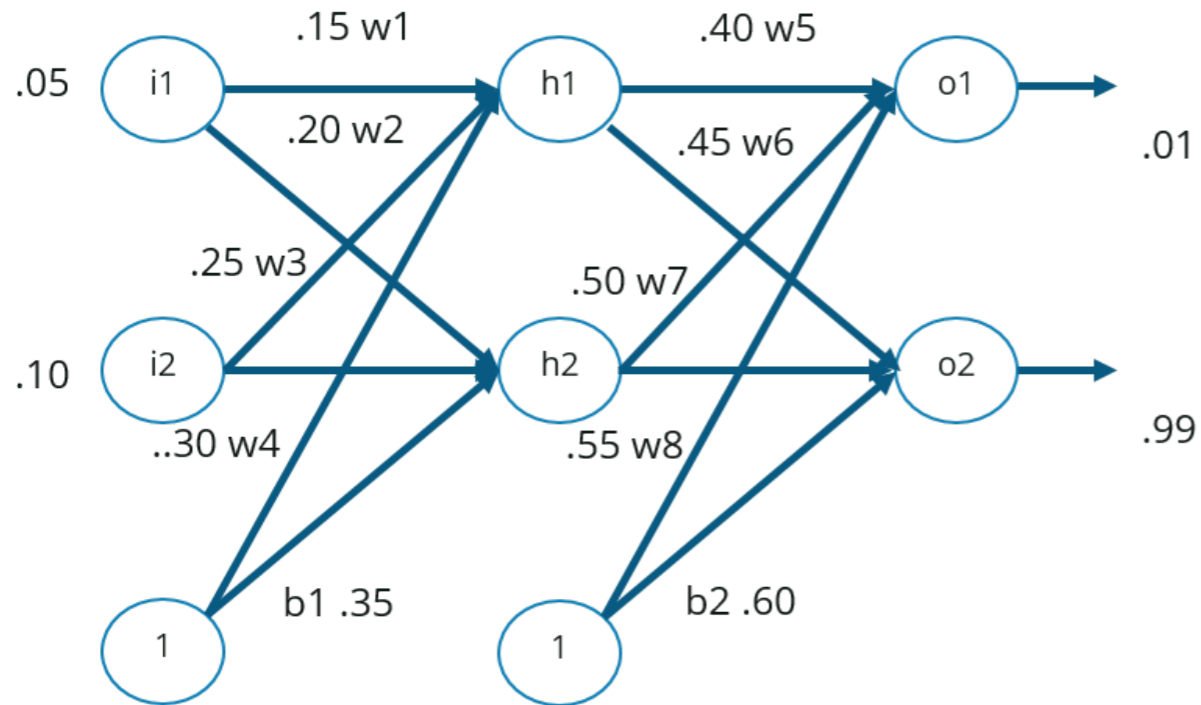
These equations can be combined into a simple, computationally efficient algorithm to calculate the gradient with respect to all parameters

Nielsen, Michael A (2015), Neural networks and deep learning

The Backpropagation Algorithm

1. **Activation at input layer:** calculate the activations a^1_j of all the neurons in the input layer.
2. **Feedforward:** starting with the first layer, exploit the feed-forward architecture to compute z^l and a^l for each subsequent layer.
3. **Error at top layer:** calculate the error of the top layer
4. **“Backpropagate” the error:** propagate the error backwards and calculate Δ^l_j for all layers.
5. **Calculate gradient:** calculate $\frac{\partial E}{\partial b^l_j}$ and $\frac{\partial E}{\partial w^l_{jk}}$.

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

Net Input For h1:

$$\text{net h1} = w1*i1 + w2*i2 + b1*1$$

$$\text{net h1} = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775$$

Output Of h1:

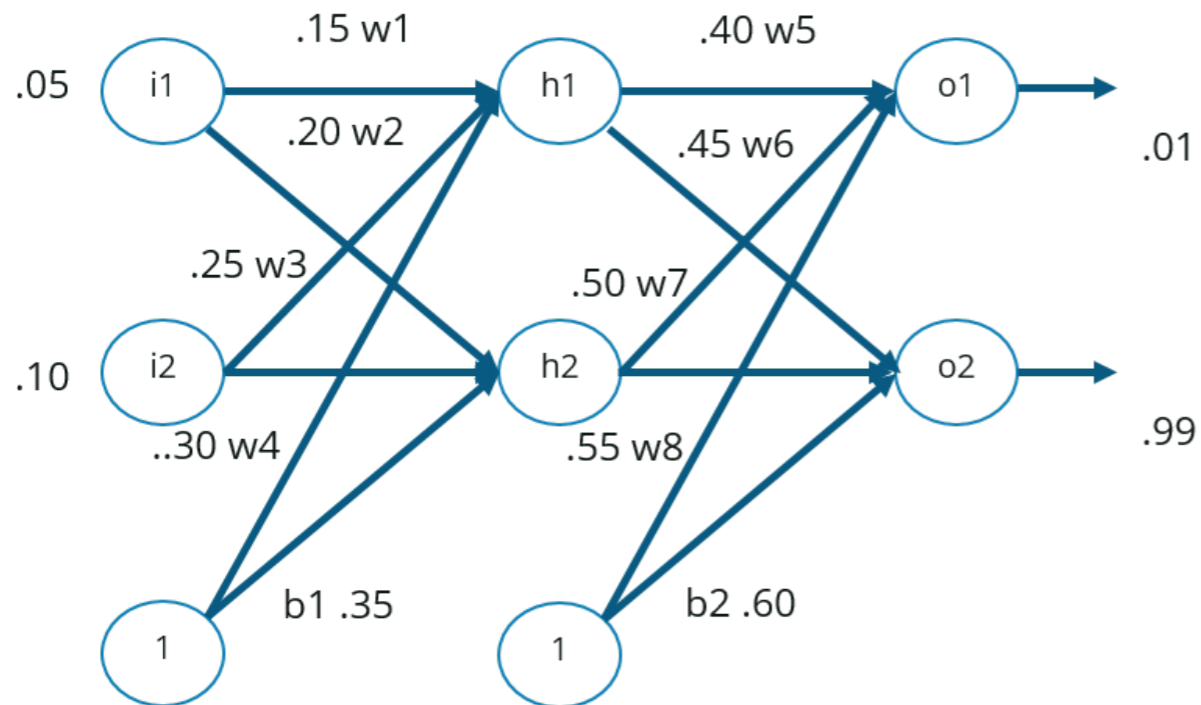
$$\text{out h1} = 1/1+e^{-\text{net h1}}$$

$$1/1+e^{.3775} = 0.593269992$$

Output Of h2:

$$\text{out h2} = 0.596884378$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

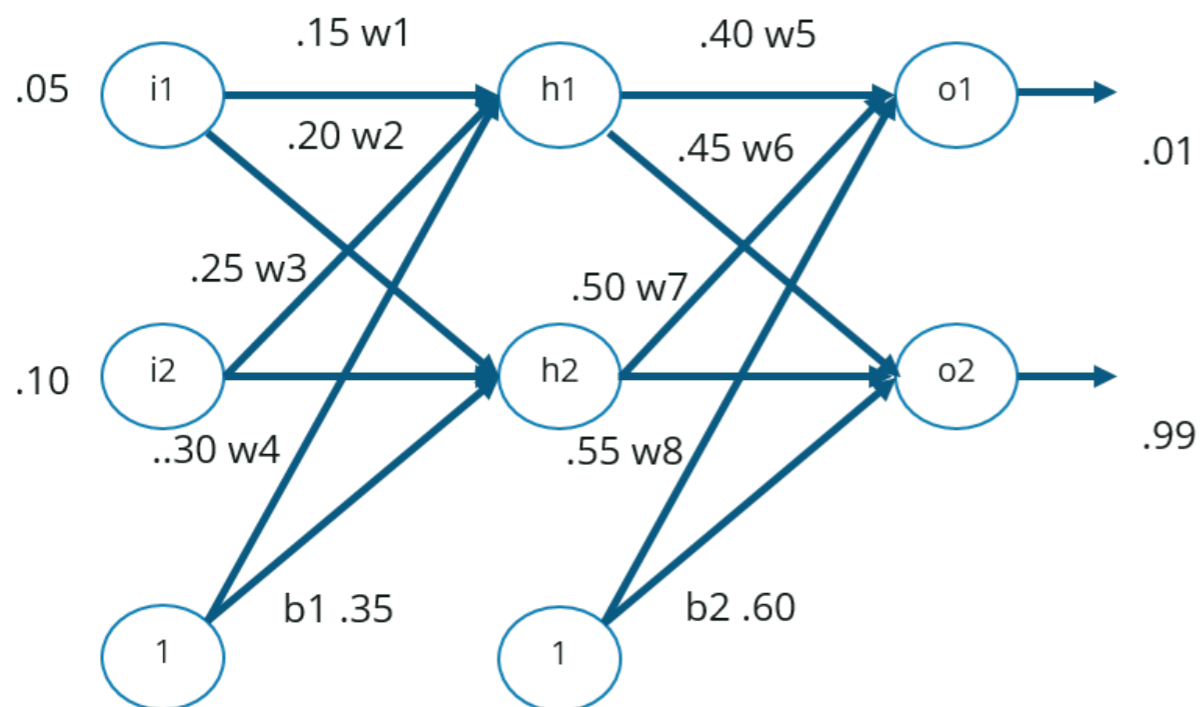
Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

We will repeat this process for the **output layer neurons**, using the output from the hidden layer neurons as inputs.

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

Output For o1:

$$\text{net } o1 = w5 * \text{out } h1 + w6 * \text{out } h2 + b2 * 1$$

$$0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

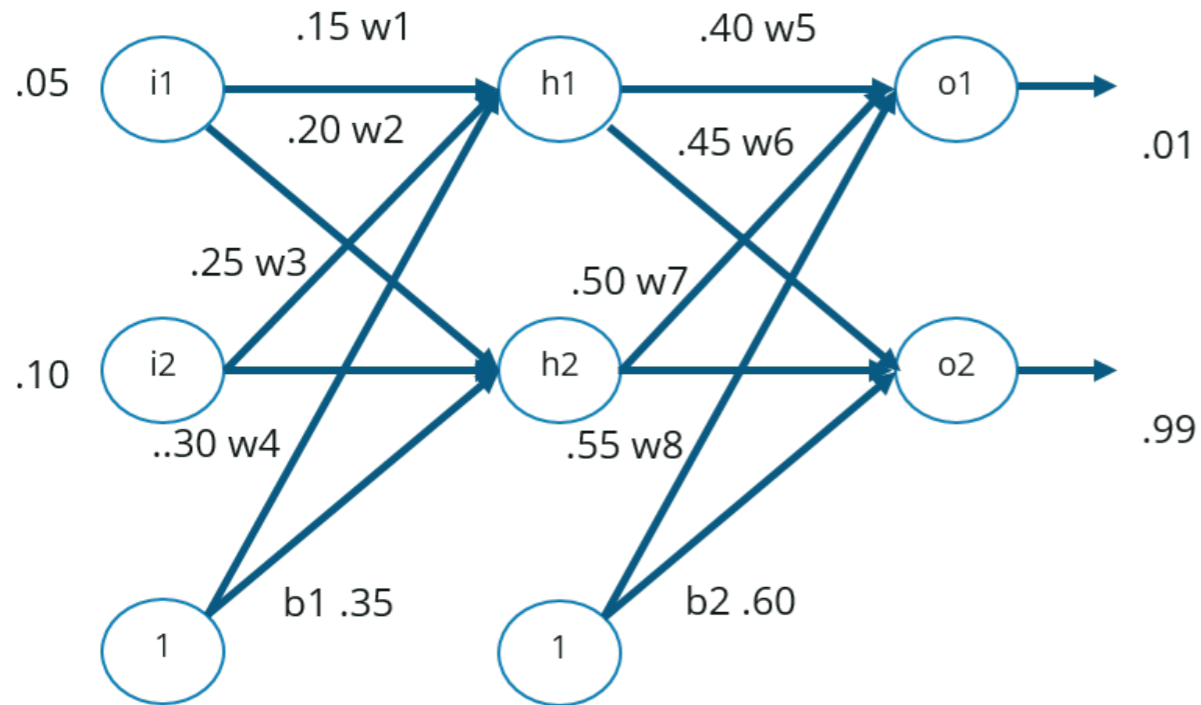
$$\text{Out } o1 = 1 / (1 + e^{-\text{net } o1})$$

$$1 / (1 + e^{-1.105905967}) = 0.75136507$$

Output For o2:

$$\text{Out } o2 = 0.772928465$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

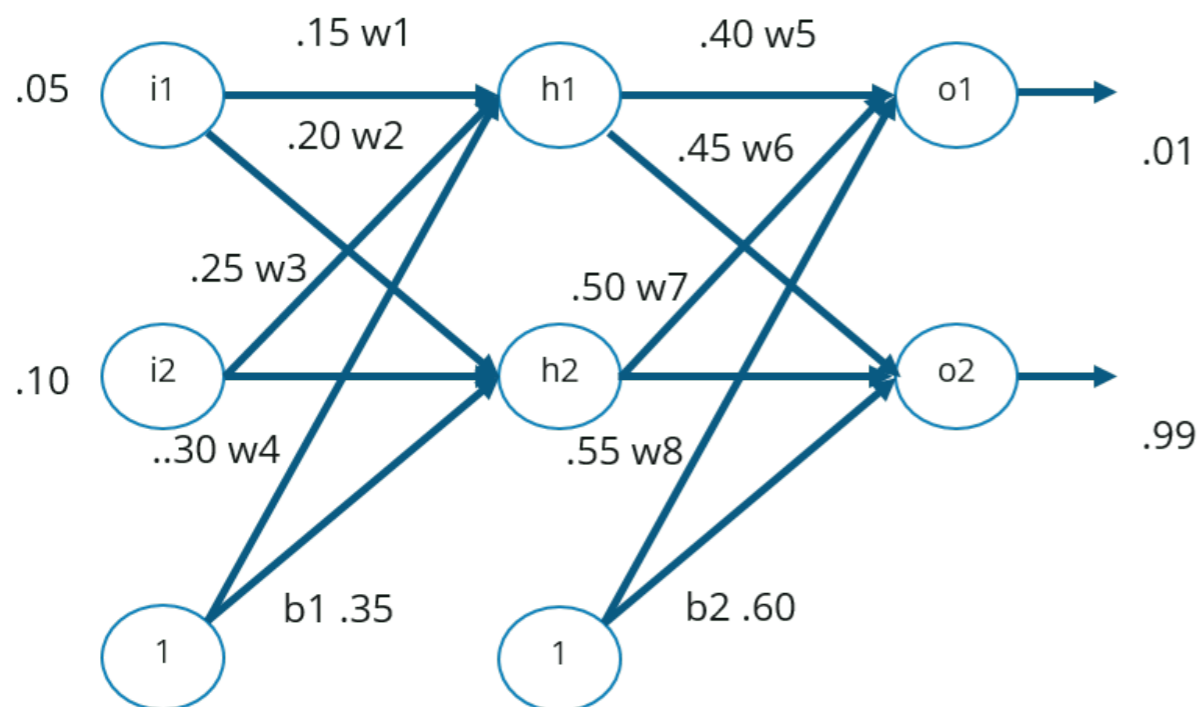
Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

Now, let's see what is the value of the error:

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

Error For o1:

$$E_{o1} = \sum 1/2(\text{target} - \text{output})^2$$

$$\frac{1}{2} (0.01 - 0.75136507)^2 = 0.274811083$$

Error For o2:

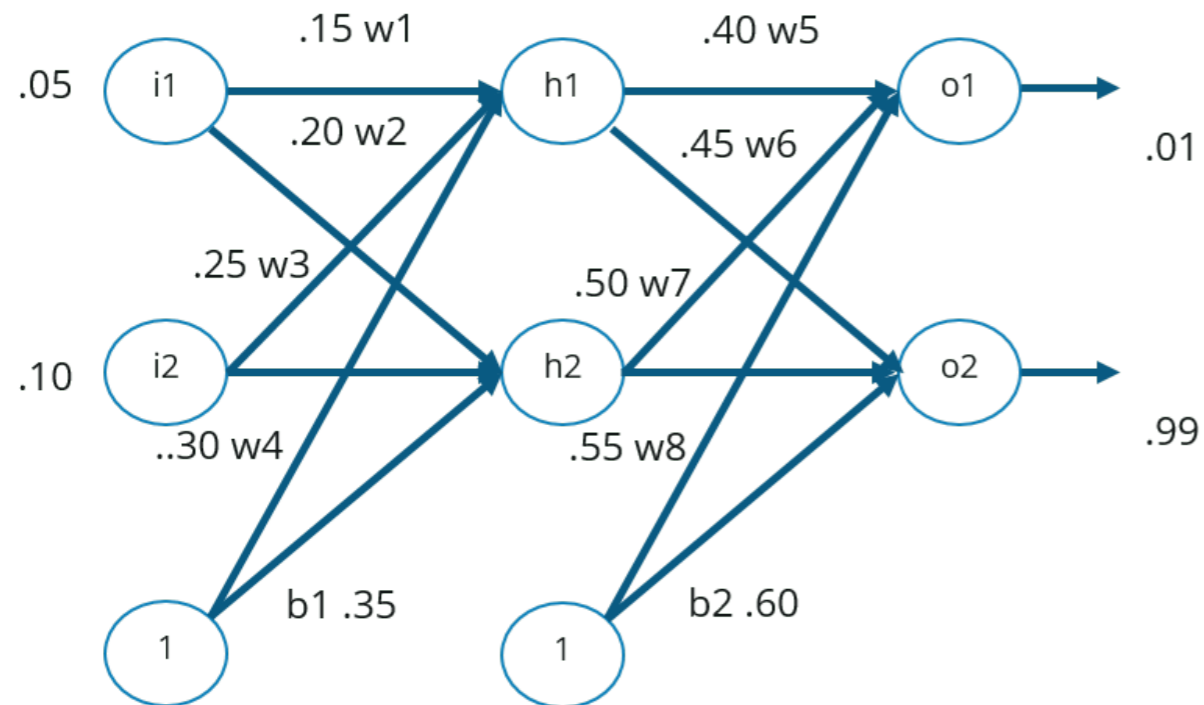
$$E_{o2} = 0.023560026$$

Total Error:

$$E_{\text{total}} = E_{o1} + E_{o2}$$

$$0.274811083 + 0.023560026 = 0.298371109$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

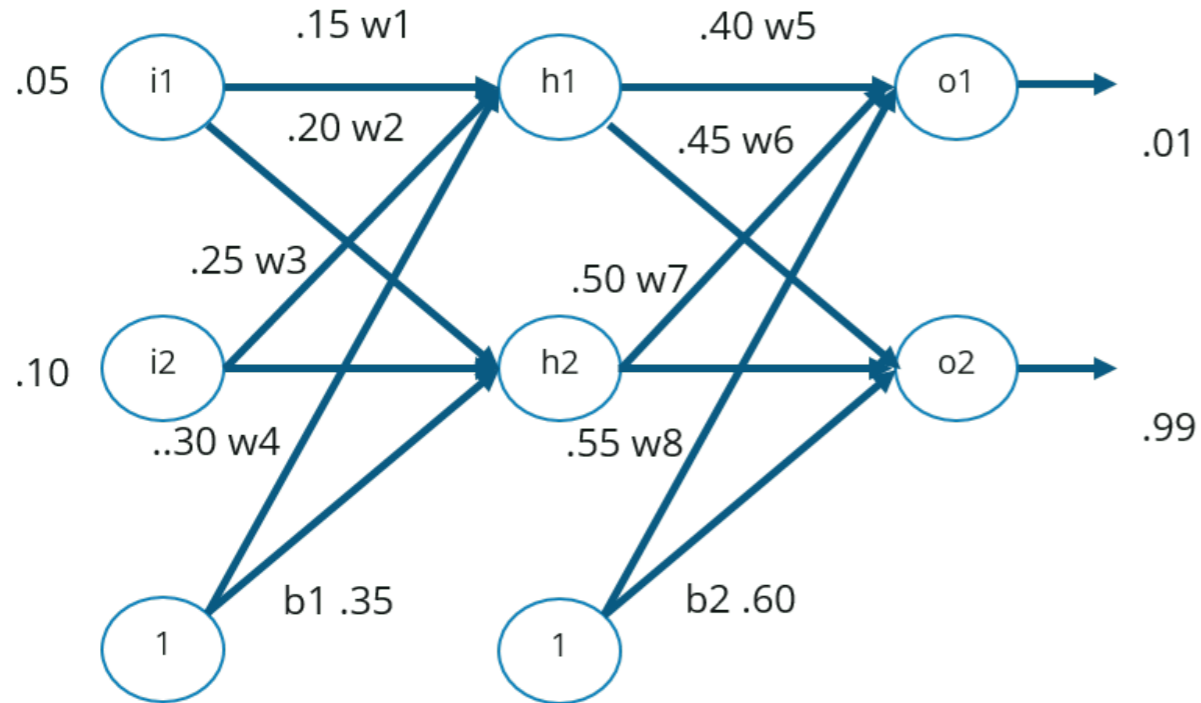
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W_5 , we will calculate the rate of change of error w.r.t change in weight W_5 .

Neural Networks: backpropagation algorithm



The above network contains the following:

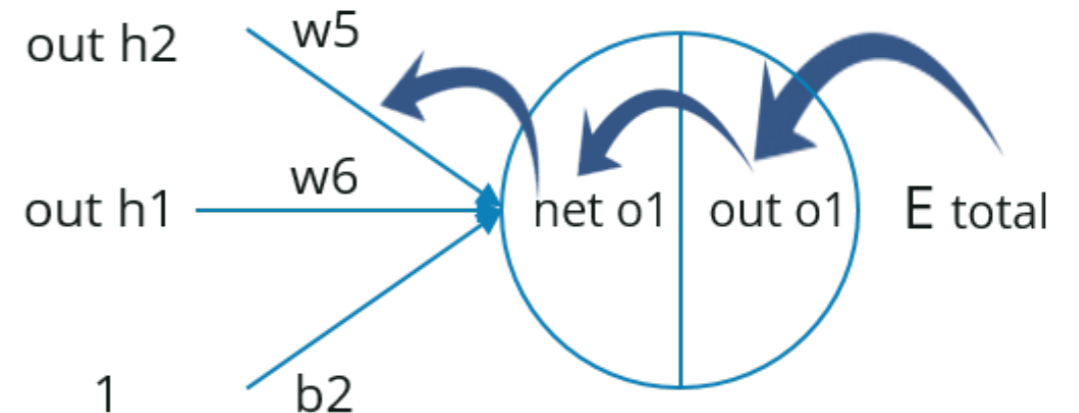
- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

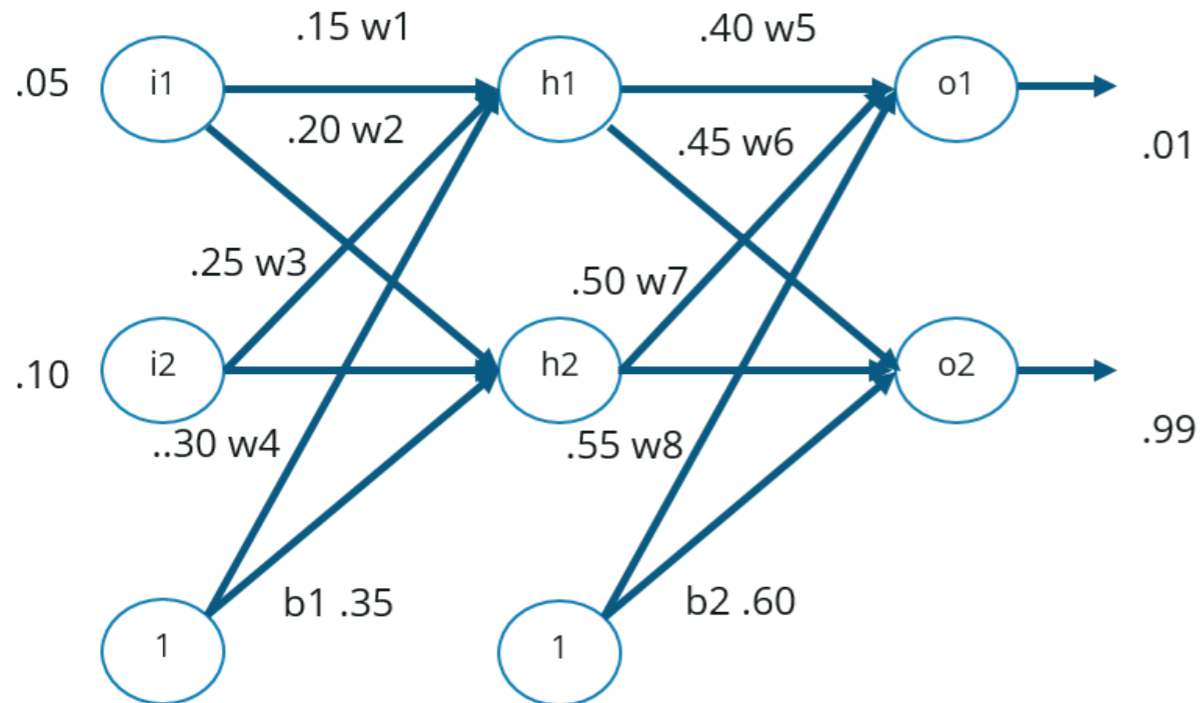
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

$$\frac{\delta E_{total}}{\delta w_5} = \frac{\delta E_{total}}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w_5}$$



Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

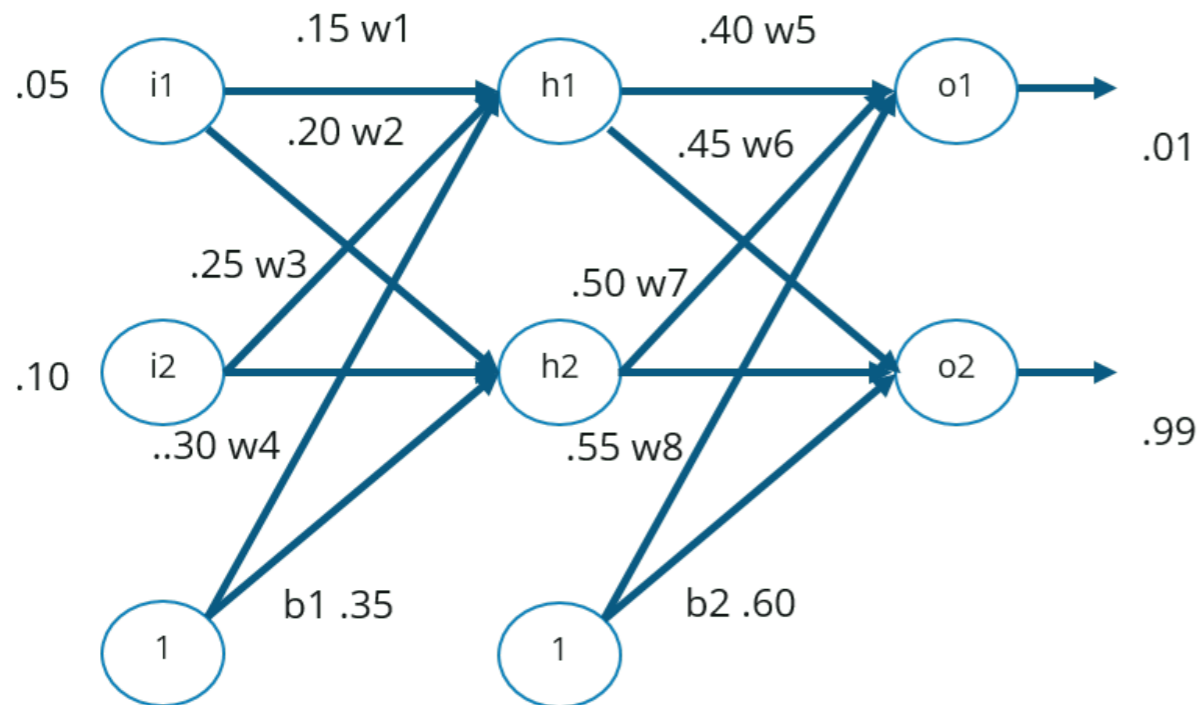
Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

calculate the change in total errors w.r.t the output O_1 and O_2 .

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

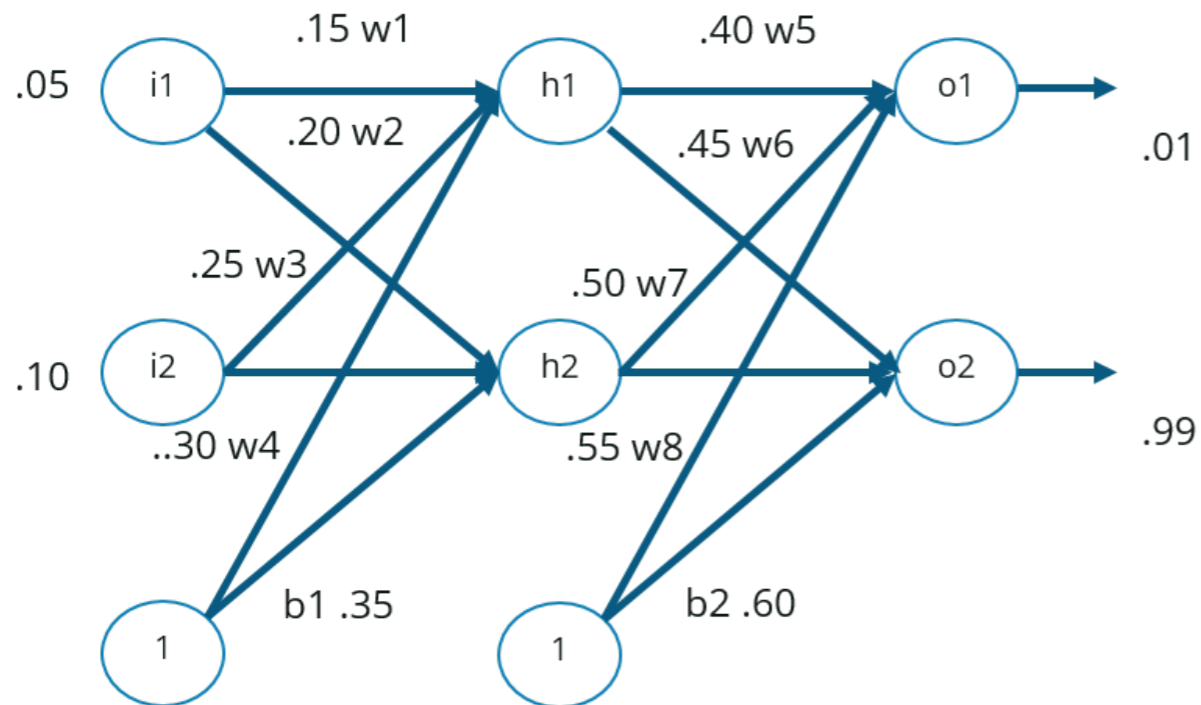
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

$$E_{\text{total}} = 1/2(\text{target } o1 - \text{out } o1)^2 + 1/2(\text{target } o2 - \text{out } o2)^2$$

$$\frac{\delta E_{\text{total}}}{\delta \text{out } o1} = -(\text{target } o1 - \text{out } o1) = -(0.01 - 0.75136507) = 0.74136507$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

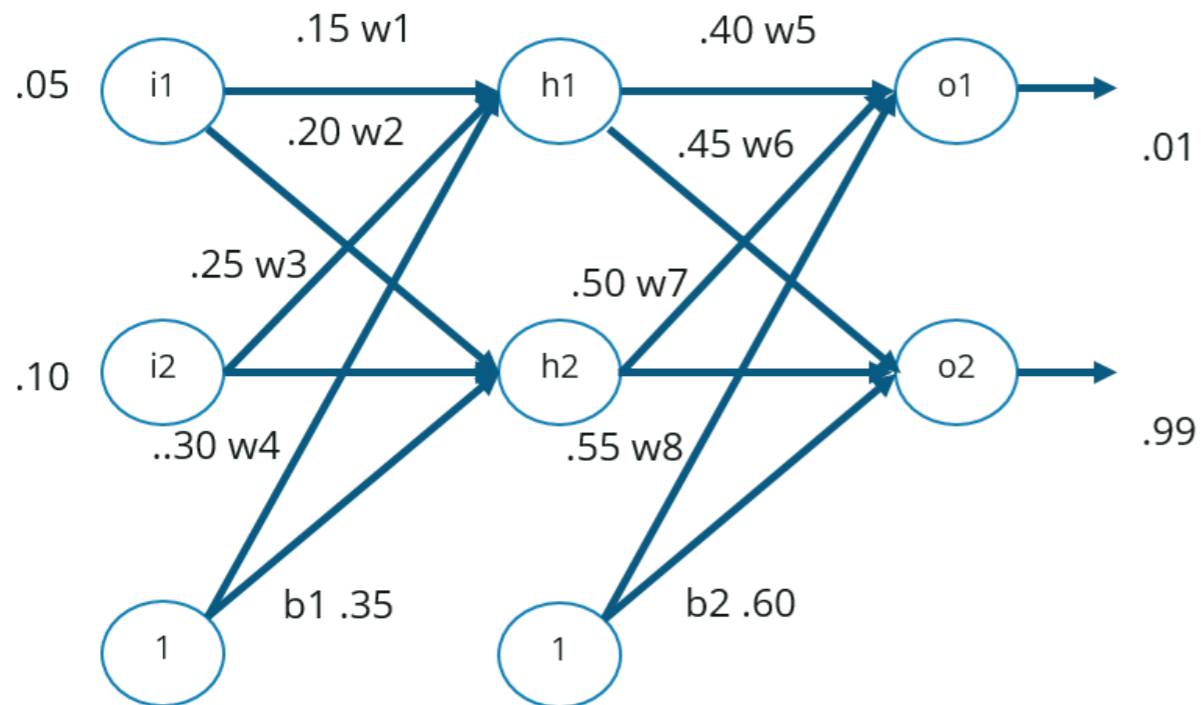
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

$$\text{out } o1 = 1 / (1 + e^{-neto1})$$

$$\frac{\delta_{out \ o1}}{\delta_{net \ o1}} = \text{out } o1 (1 - \text{out } o1) = 0.75136507 (1 - 0.75136507) = 0.186815602$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

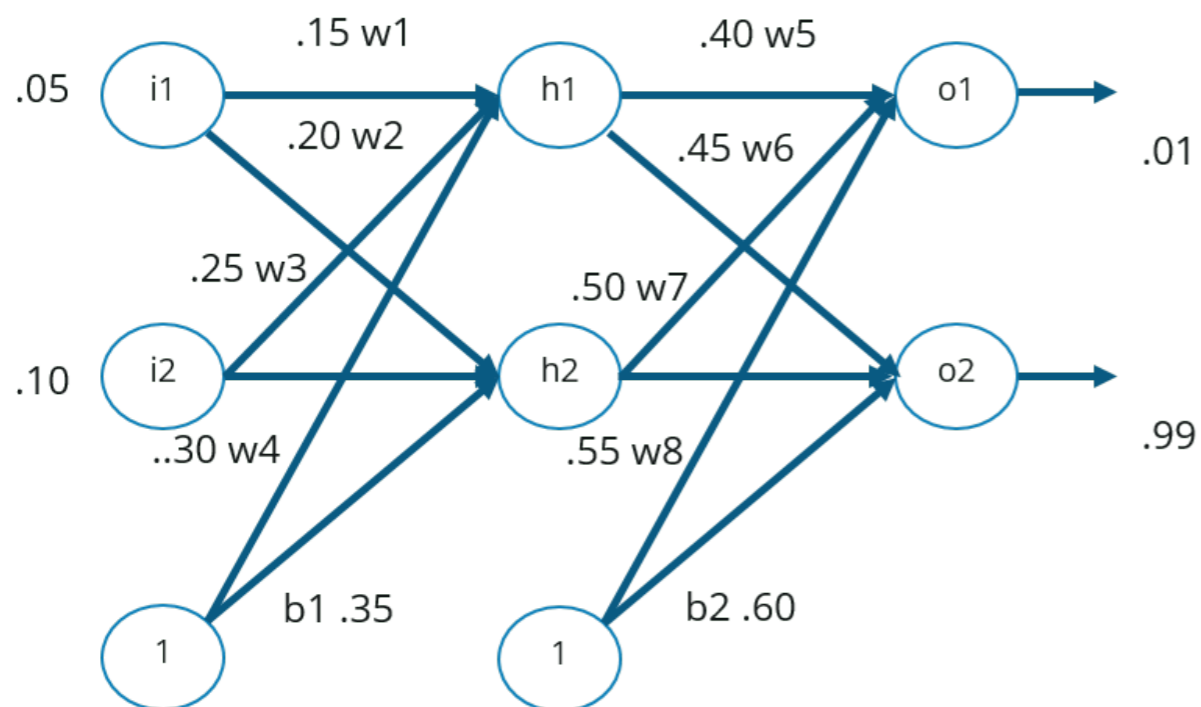
Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 2: Backward Propagation

$$\text{net } o1 = w5 * \text{out } h1 + w6 * \text{out } h2 + b2 * 1$$
$$\frac{\delta_{\text{net } o1}}{\delta w5} = 1 * \text{out } h1 * w5^{(1-1)} + 0 + 0 = 0.593269992$$

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

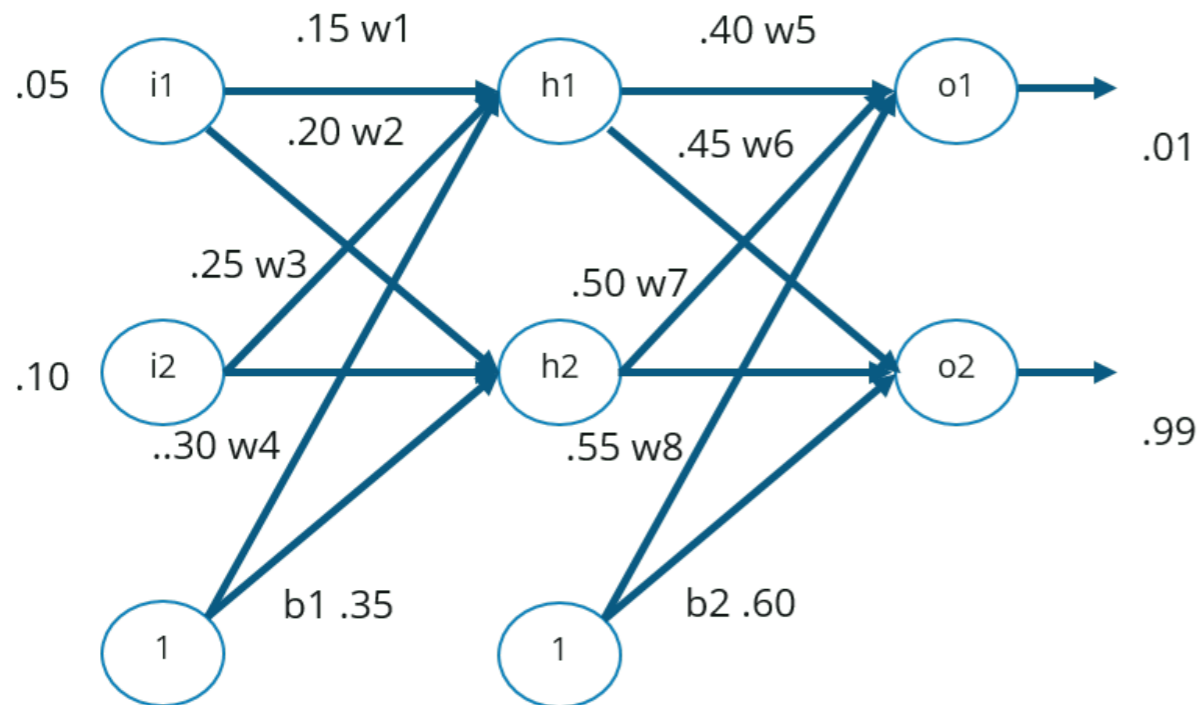
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 3: Putting all the values together and calculating the updated weight value

$$\frac{\delta E_{total}}{\delta w_5} = \frac{\delta E_{total}}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w_5}$$

0.082167041

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 3: Putting all the values together and calculating the updated weight value

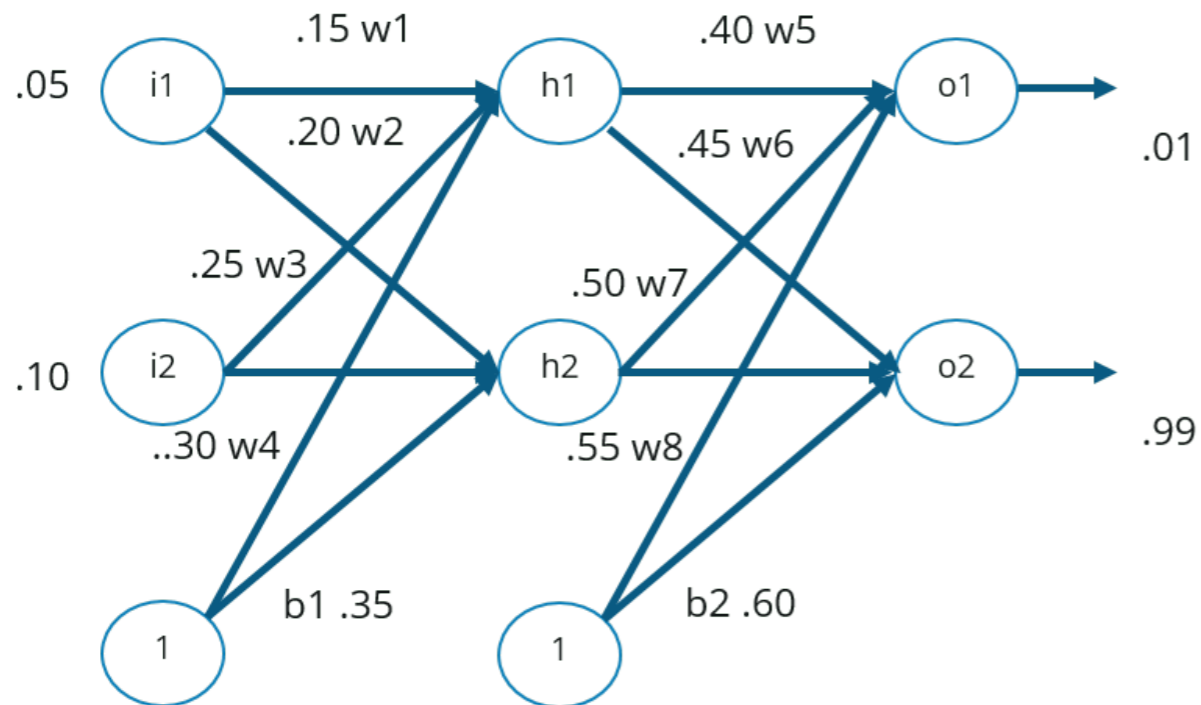
$$w5^+ = w5 - \eta \frac{\delta E_{total}}{\delta w5}$$

$$w5^+ = 0.4 - 0.5 * 0.082167041$$

Updated w5

0.35891648

Neural Networks: backpropagation algorithm



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

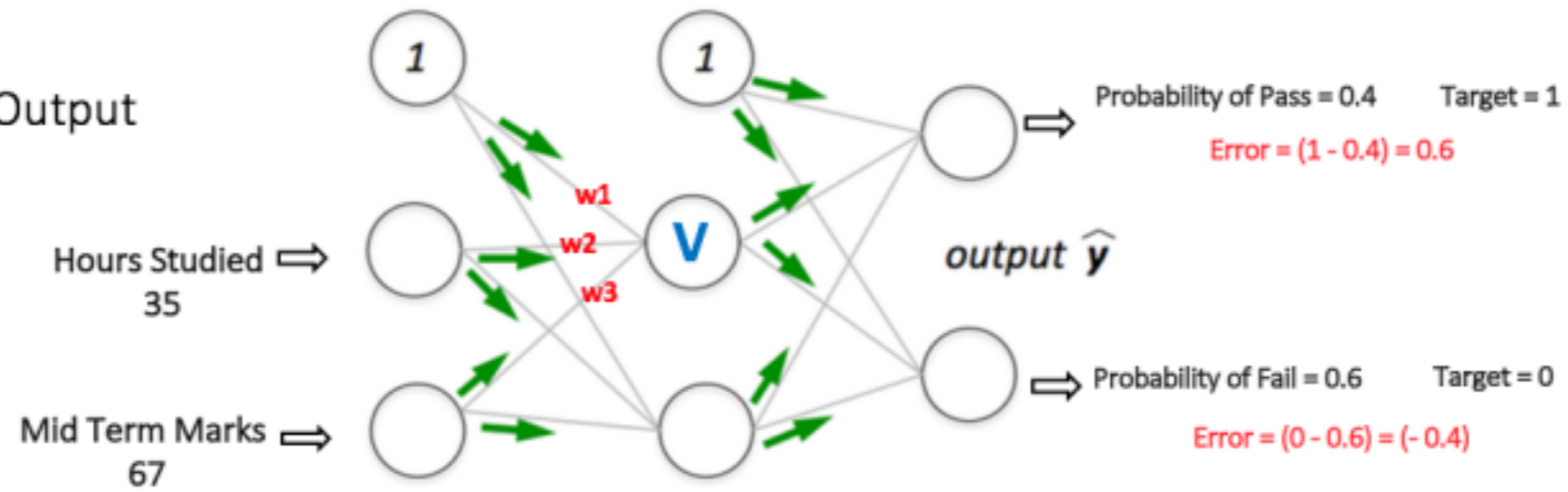
Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

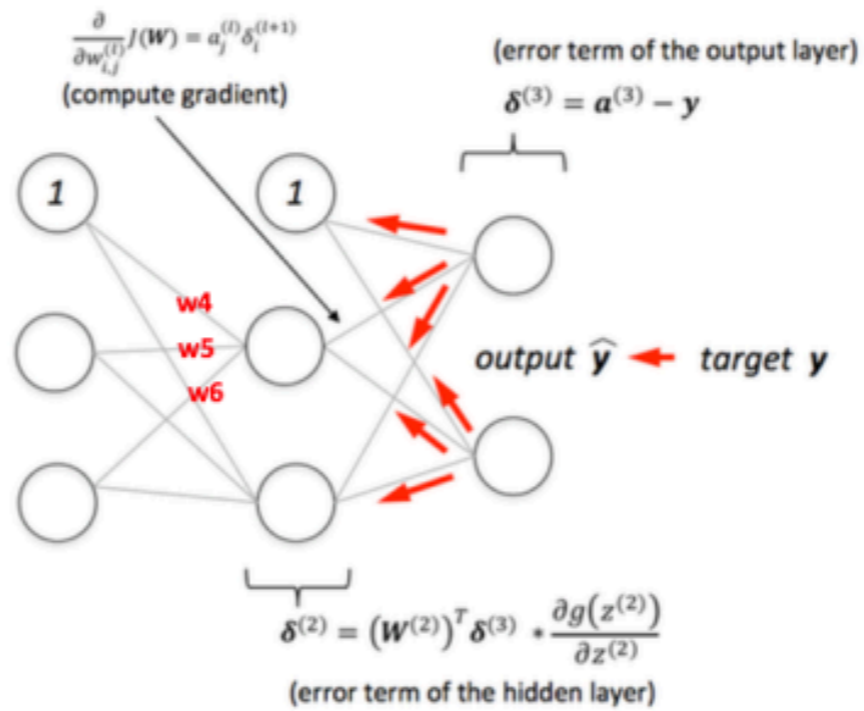
Step – 3: Putting all the values together and calculating the updated weight value

- Similarly, we can calculate the other weight values as well.
- After that we will again propagate forward and calculate the output. Again, we will calculate the error.
- If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.
- This process will keep on repeating until error becomes minimum.

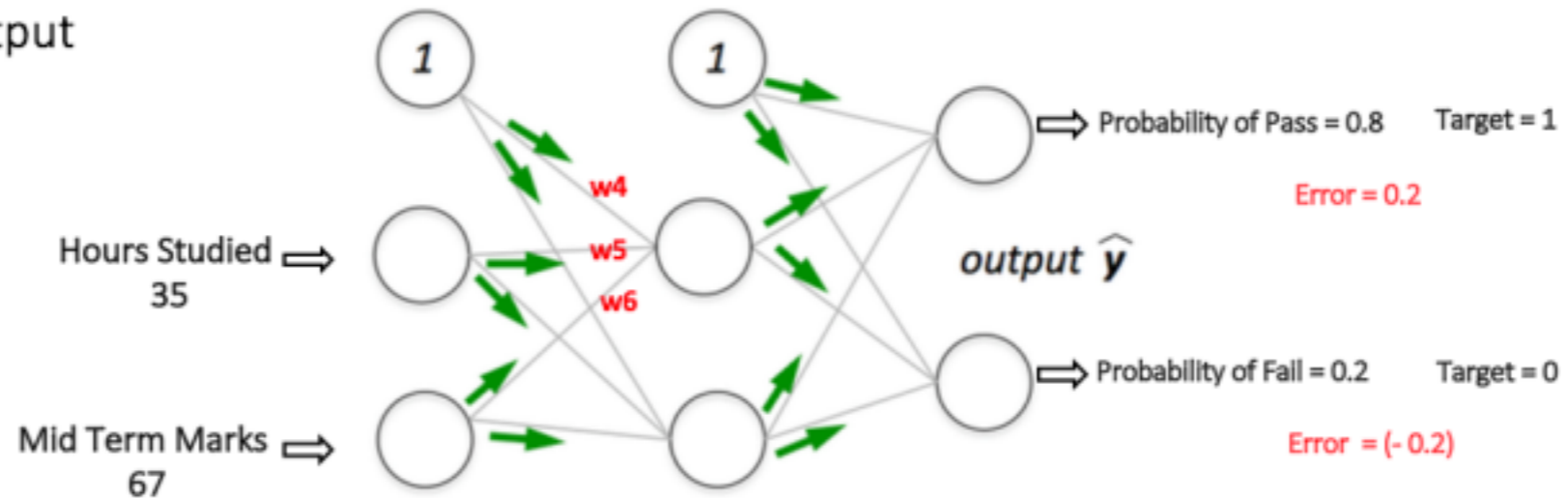
Incorrect Output



Backpropagation + Weights Adjusted



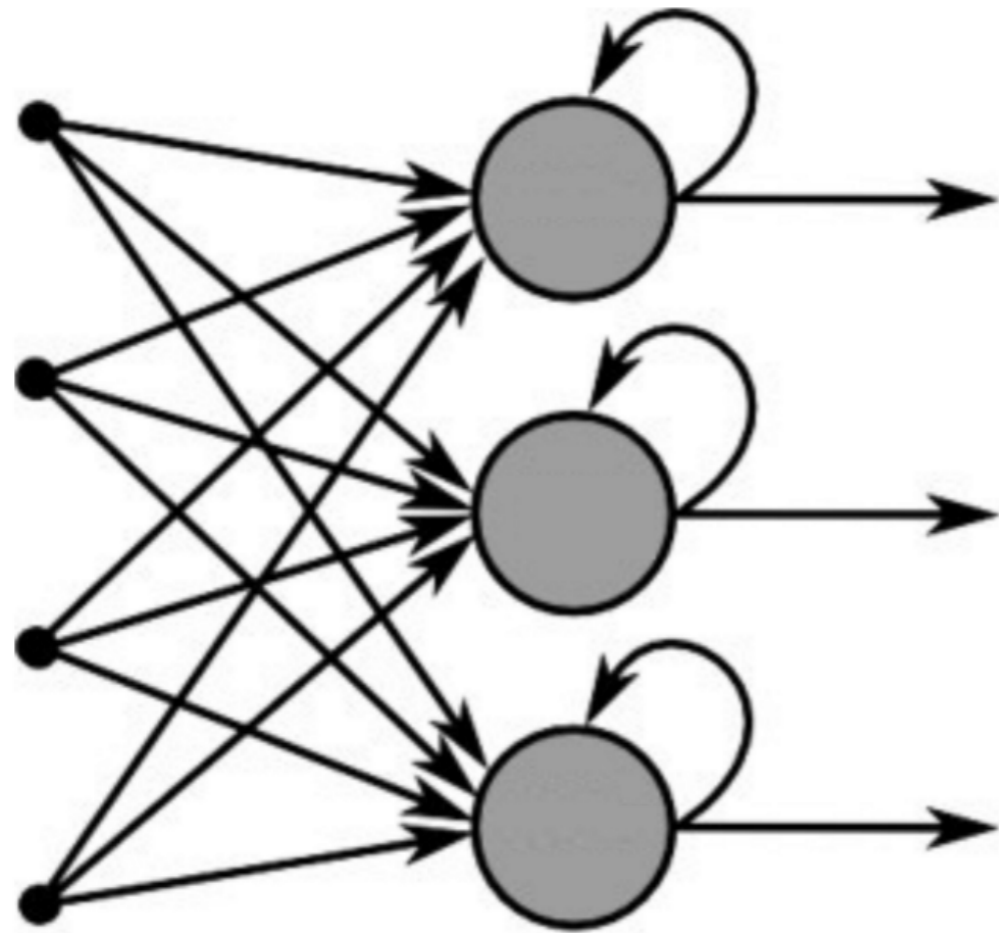
Correct Output



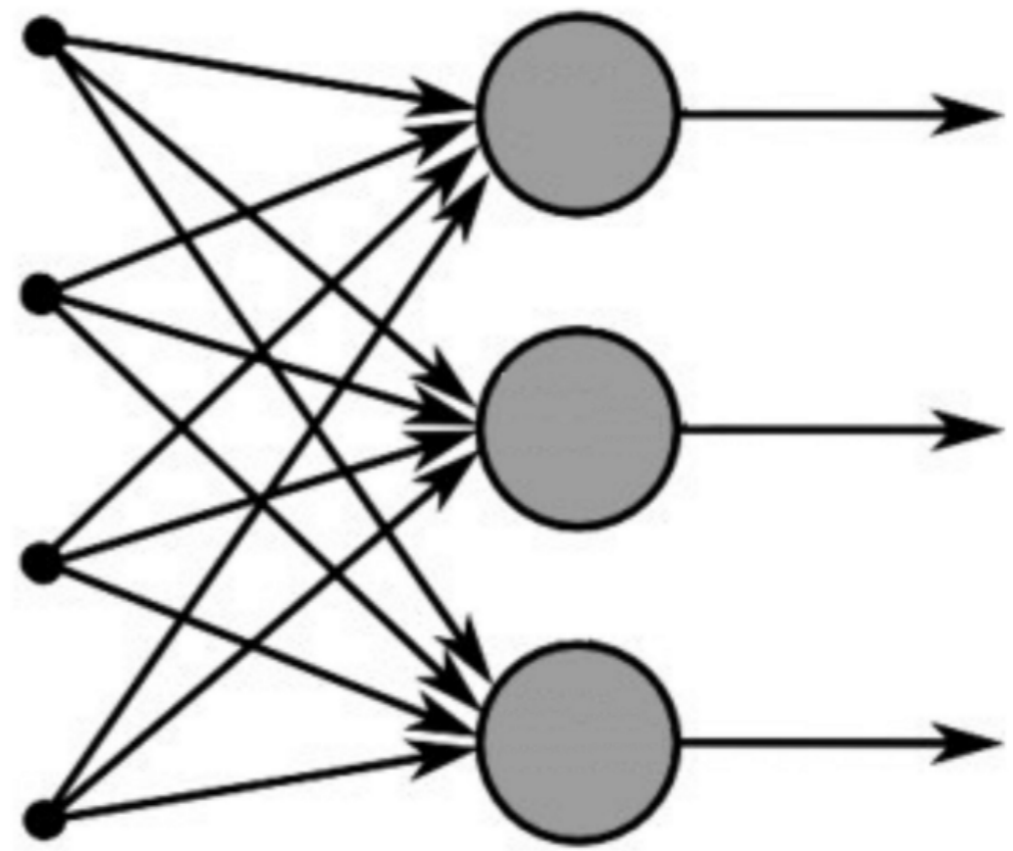
Recurrent Neural Network (RNN)

A usual RNN has a short-term memory

Recurrent Neural Networks add the immediate past to the present.



Recurrent Neural Network

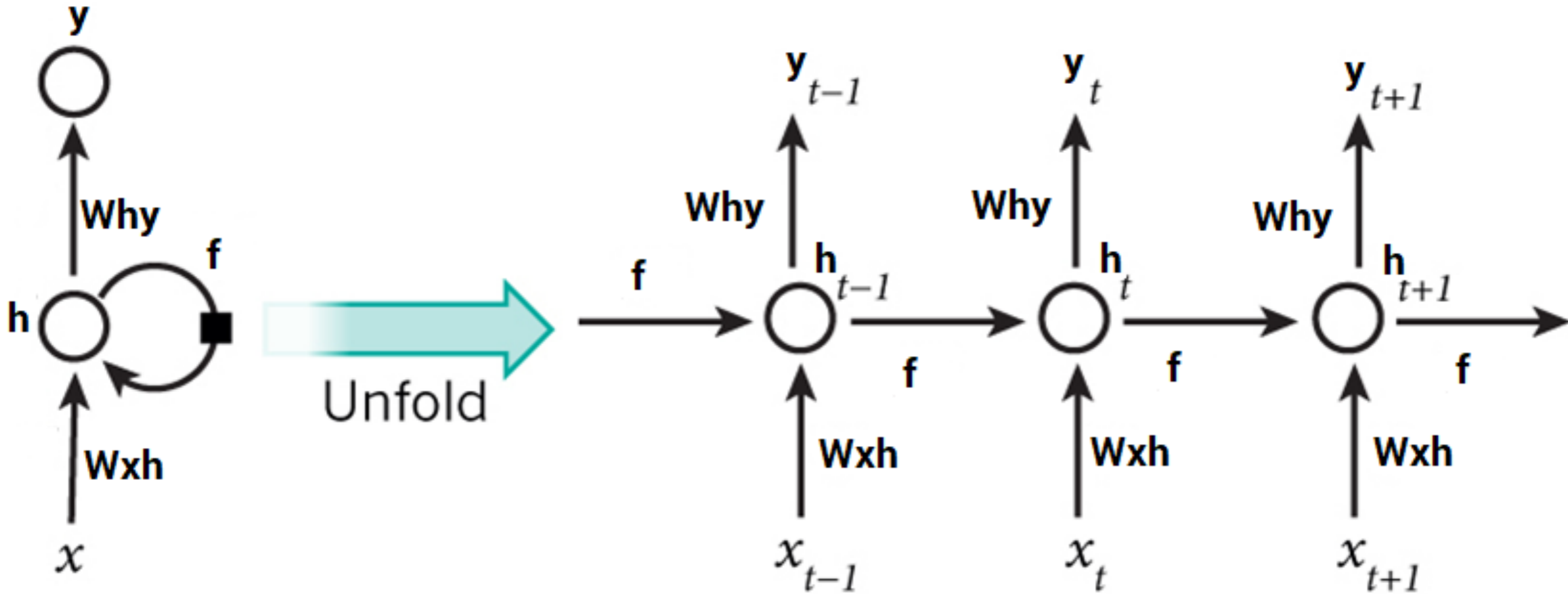


Feed-Forward Neural Network

Recurrent Neural Network (RNN)

To understand and visualize the back propagation, let's unroll the network at all the time steps.

In case of a backward propagation in this case, we are figuratively going back in time to change the weights, hence we call it the Back propagation through time(BPTT)

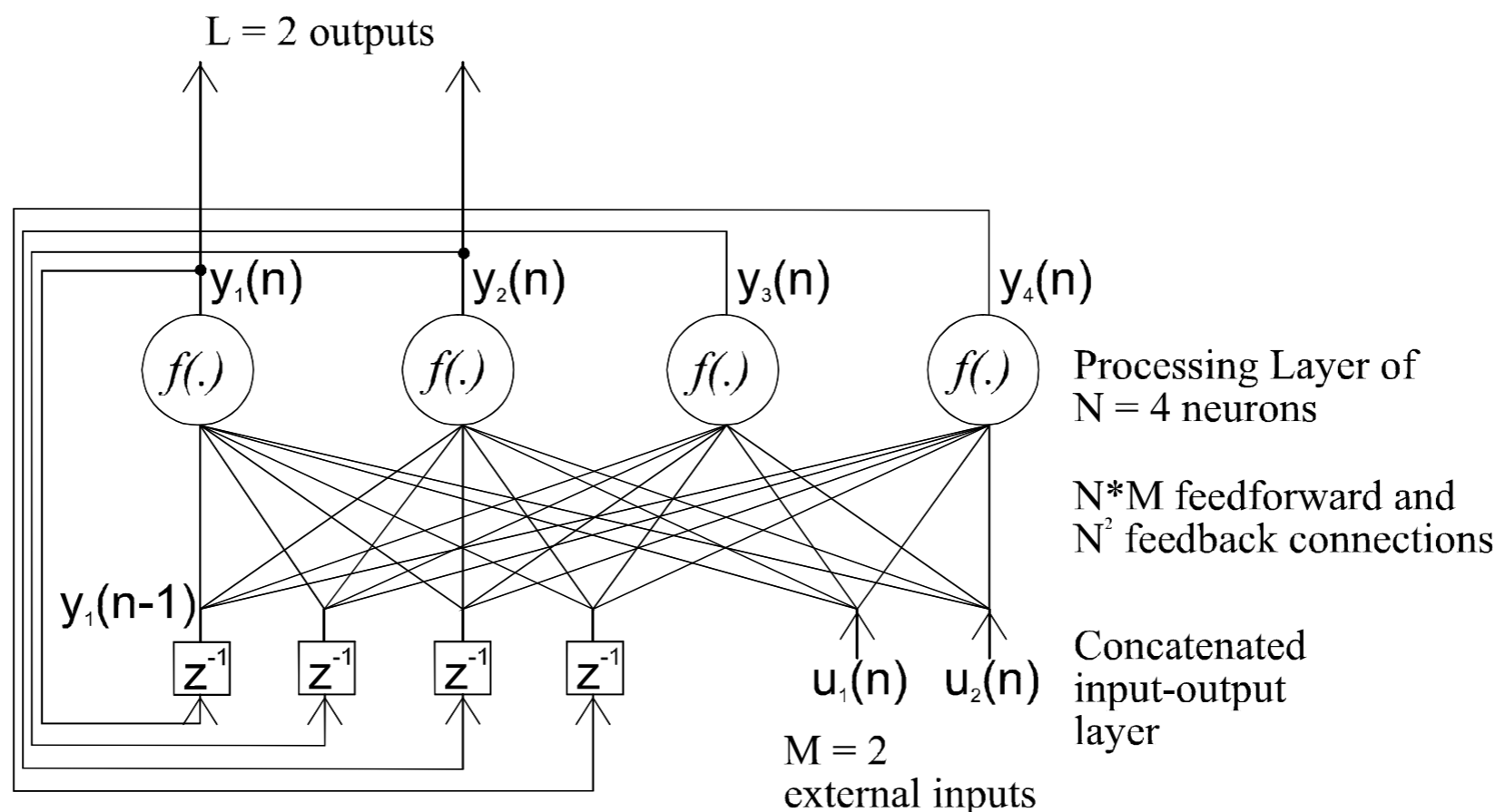


Recurrent Neural Network (RNN)

The Backpropagation Through Time (BPTT) algorithm is an algorithm that performs an exact computation of the gradient of the error measure for use in the weight adaptation.

$$E(n_0, n) = \sum_{m=n_0}^n \sum_{i=1}^L e_i^2(m)$$

The error measure may be minimized by gradient descent: $\Delta w_{ij} = -\eta \frac{\partial E(n_0, n)}{\partial w_{ij}}$



Synchronous and asynchronous networks

A relevant issue for the correct design of recurrent neural networks is the adequate **synchronization** of the computing elements.

In the case of **McCulloch-Pitts** networks we solved this difficulty by assuming that the activation of each computing element consumes a unit of time.

The synchronization of the output was achieved by requiring that all computing elements evaluate their inputs and compute their output **simultaneously**.

Asynchronous networks

In an **asynchronous** network each unit computes its excitation at random times and changes its state to 1 or -1 independently of the others and according to the sign of its total excitation.

There will not be any delay between computation of the excitation and state update

Asynchronous networks are of course more realistic models of biological networks, although the assumption of zero delay in the computation and transmission of signals lacks any biological basis.

The Hopfield Model

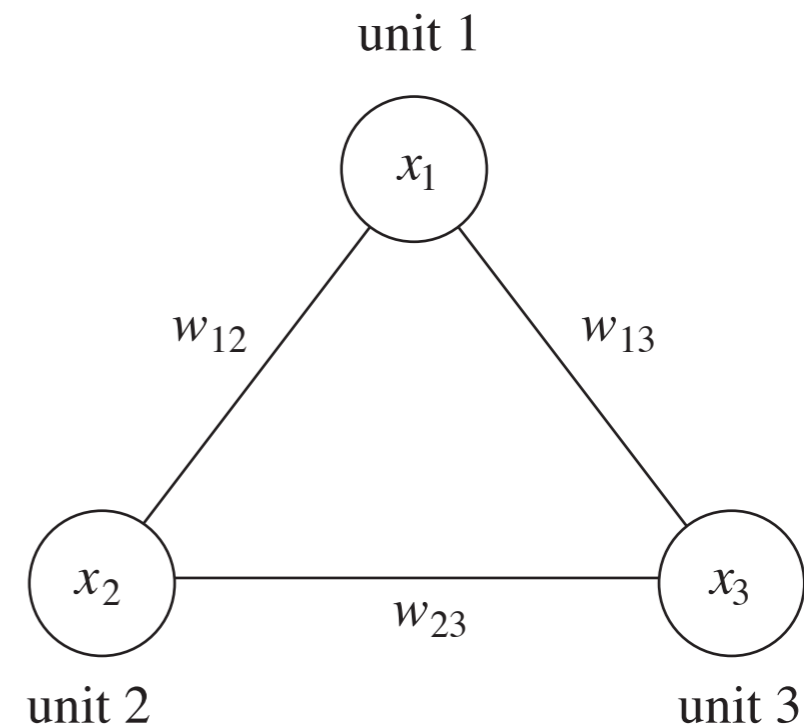
In 1982 the American physicist John Hopfield proposed an asynchronous neural network model which made an immediate impact in the AI community

No synchronization is required, each unit behaving as a kind of elementary system in complex interaction with the rest of the ensemble

In the Hopfield model it is assumed that the individual units preserve their individual states until they are selected for a new update. The selection is made randomly.

A Hopfield network consists of n totally coupled units, that is, each unit is connected to all other units except itself.

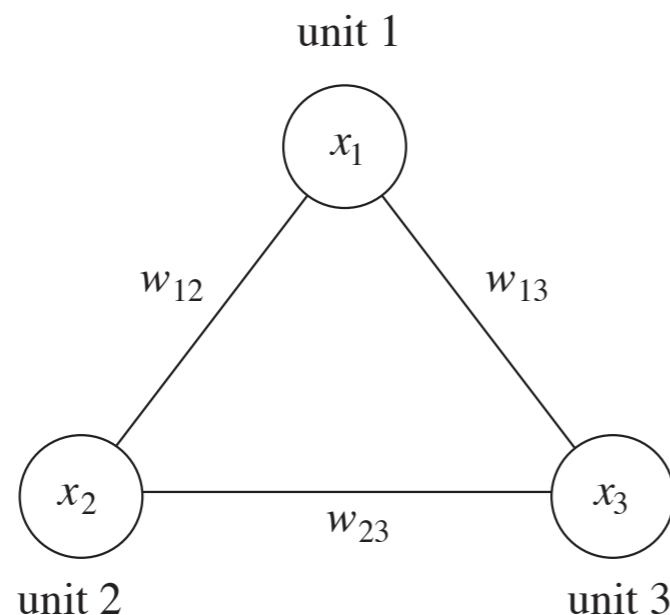
The network is symmetric



The Hopfield Model

Each one of them can assume the state 1 or -1 .

The connections in a Hopfield network with n units can be represented using an $n \times n$ weight matrix $\mathbf{W} = \{w_{ij}\}$ with a zero diagonal.



$$\mathbf{W} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

The units of a Hopfield network can be assigned a threshold θ different from zero. In this case each unit selected for a state update adopts the state 1 if its total excitation is greater than θ , otherwise the state -1 . This is the activation rule for perceptrons, so that we can think of Hopfield networks as asynchronous recurrent networks of perceptrons.

$$E(\mathbf{x}) = -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n w_{ij} x_i x_j + \sum_{i=1}^n \theta_i x_i.$$

The Hopfield Model

In order to characterize the performance of the network, the concept of energy is introduced and the following energy function defined

$$E(\mathbf{x}) = -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n w_{ij} x_i x_j + \sum_{i=1}^n \theta_i x_i.$$

The Hopfield Model

Hopfield network consists of a set of interconnected neurons which update their activation values asynchronously. The activation values are binary, usually $\{-1,1\}$. The update of a unit depends on the other units of the network and on itself. A unit i will be influenced by another unit j with a certain weight w_{ij} , and have a threshold value

$$x_i(t + 1) = \text{sign}\left(\sum_{j=1}^n x_j(t)w_{ij} - \theta_i\right)$$

$$X = \text{sign}(XW - T)$$

– X is the activation value of the n units/neurons : $X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$

– W is the weight matrix : $W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & & \ddots & \vdots \\ w_{n1} & \dots & \dots & w_{nn} \end{pmatrix}$ where w_{ij} can be interpreted

as the influence of neuron i over neuron j (and reciprocally)

– T is the threshold of each unit : $T = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{pmatrix}$

– the sign function is defined as :

$$\begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$\bar{J}'_{ij} = \bar{J}_{ij} + \alpha \frac{dE}{d\bar{J}_{ij}}$$

Hebb Learning

Learning in biologically relevant neural-network models usually relies on **Hebb learning rules**. The typical implementations of these rules change the synaptic strength on the basis of the co-occurrence of the neural events taking place at a certain time in the pre- and post-synaptic neurons.

The **Hebbian rule** was the first learning rule. In 1949 *Donald Hebb* developed it as learning algorithm of the **unsupervised neural network**. We can use it to identify how to improve the weights of nodes of a network.

The **Hebb learning rule** assumes that – If two neighbor neurons activated and deactivated at the same time. Then the weight connecting these neurons should increase.

For neurons operating in the opposite phase, the weight between them should decrease. If there is no signal correlation, the weight should not change.

The Hebbian learning rule describes the formula as follows:

$$W_{ij} = x_i * x_j$$

Delta Learning Rule

Developed by *Widrow* and *Hoff*, the delta rule, is one of the most common learning rules. **It depends on supervised learning.**

This rule states that the modification in synaptic weight of a node is equal to the multiplication of error and the input.

In Mathematical form the delta rule is as follows:

$$\Delta w = \eta (t - y) x_i$$

We can use the delta learning rule with both single output unit and several output units.

While applying the delta rule assume that the error can be directly measured.

The aim of applying the delta rule is to reduce the difference between the actual and expected output that is the error.

Correlation Learning Rule

The **correlation learning rule** based on a similar principle as the Hebbian learning rule. It assumes that weights between responding neurons should be more positive, and weights between neurons with opposite reaction should be more negative.

In Mathematical form the correlation learning rule is as follows:

$$\Delta W_{ij} = \eta x_i d_j$$

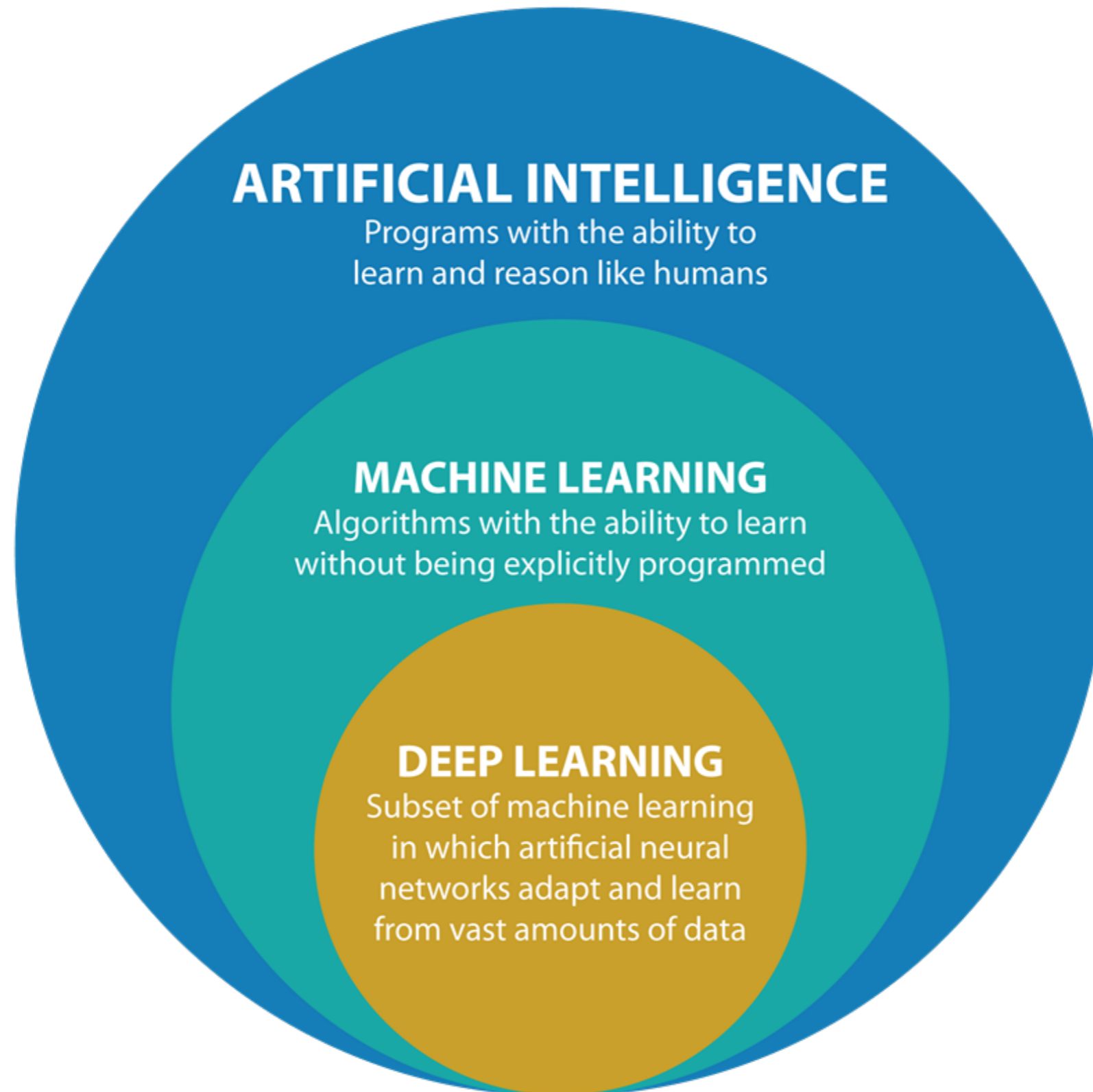
where d_j is the desired value of output signal. This training algorithm usually starts with the initialization of weights to zero.

Since assigning the desired weight by users, the correlation learning rule is an example of supervised learning

Learning Rule

In conclusion to the learning rules in Neural Network, we can say that most promising feature of the Artificial Neural Network is its ability to learn. The learning process of brain alters its neural structure. The increasing or decreasing the strength of its synaptic connections depending on their activity.

Deep Learning?

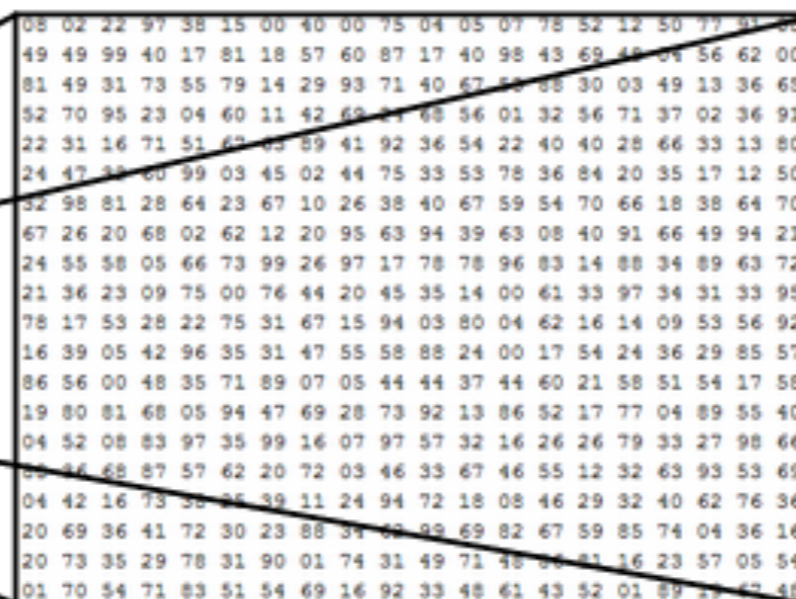


Deep Learning?

Deep Learning: Falling hardware prices and the development of GPUs for personal use in the last few years have contributed to the development of the concept of **Deep learning which consists of multiple hidden layers in an artificial neural network.**

This approach tries to model the way the human brain processes light and sound into vision and hearing. Some successful applications of deep learning are **computer vision** and **speech recognition.**

Convolutional Neural Network (CNN)



What the computer sees

image classification →
82% cat
15% dog
2% hat
1% mug

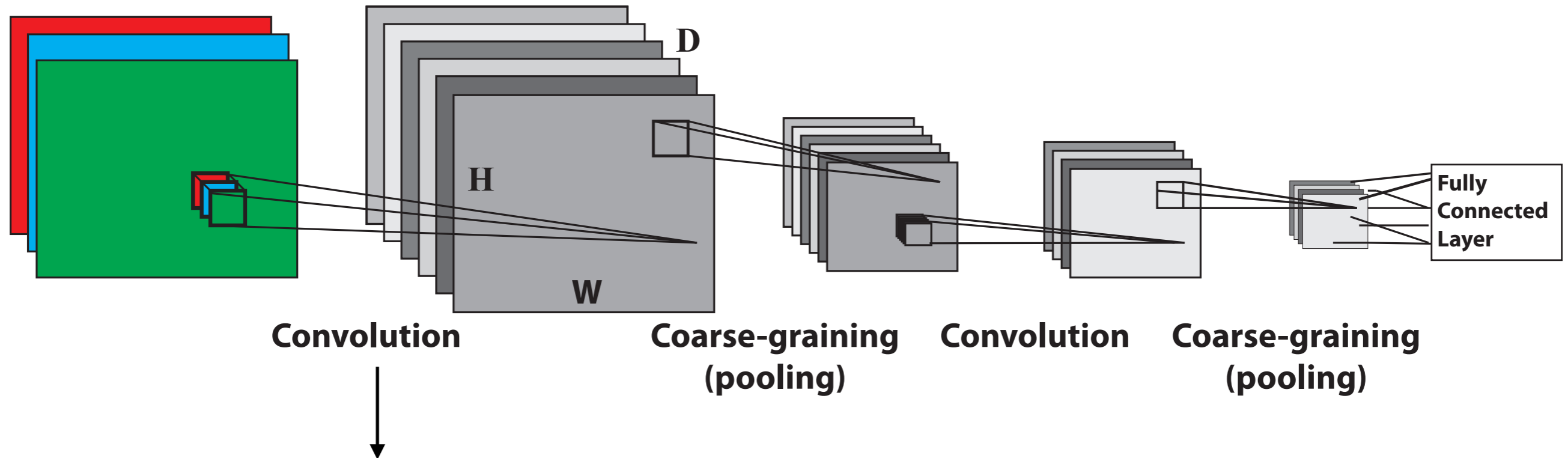
CNNs are the backbone of many modern **deep learning** applications and here we just give a high-level overview of CNNs that should allow the reader to delve directly into the specialized literature.

Convolutional Neural Networks (ConvNets or CNNs) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification.

Later, in 1998, Convolutional Neural Networks were introduced in a paper by Bengio, Le Cun, Bottou and Haffner. Their first Convolutional Neural Network was called **LeNet-5** and was able to classify digits from hand-written numbers

Convolutional Neural Network (CNN)

We perform a series **convolution + pooling** operations, followed by a number of **fully connected layers**. If we are performing multiclass classification the output is softmax. We will now dive into each component.



Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a *convolution filter* to produce a *feature map*

Convolutional Neural Network (CNN)

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

On the left side is the input to the convolution layer, for example the input image. On the right is the convolution *filter*, also called the *kernel*, we will use these terms interchangeably. This is called a *3x3 convolution* due to the shape of the filter. We perform the convolution operation by sliding this filter over the input

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4		

Feature Map

1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

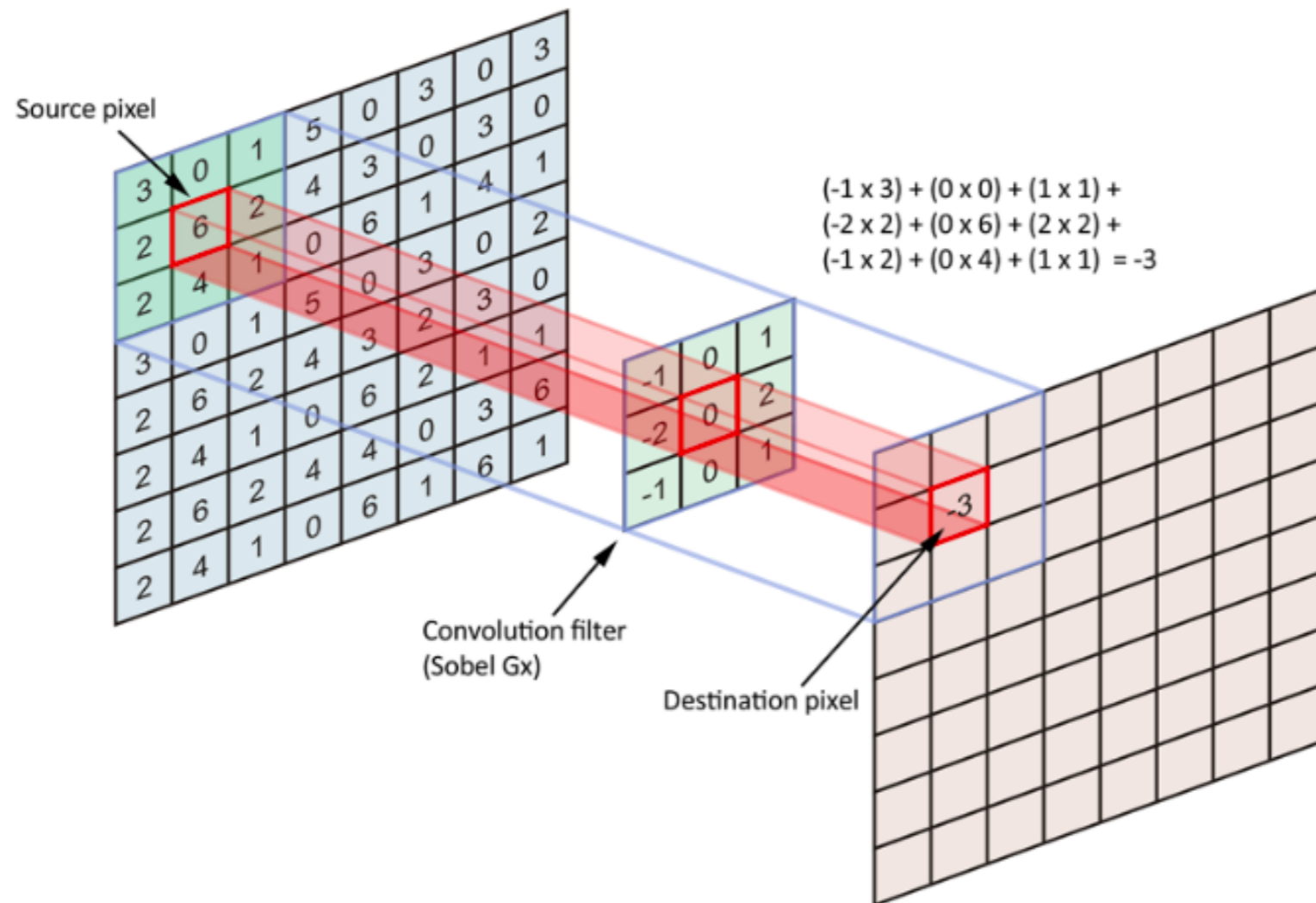
Input x Filter

4	3	

Feature Map

This was an example convolution operation shown in 2D using a 3x3 filter

Convolutional Neural Network (CNN)



We perform numerous convolutions on our input, where each operation uses a different filter. This results in different feature maps. In the end, we take all of these feature maps and put them together as the final output of the convolution layer.

Just like any other Neural Network, we use an **activation function** to make our output non-linear.

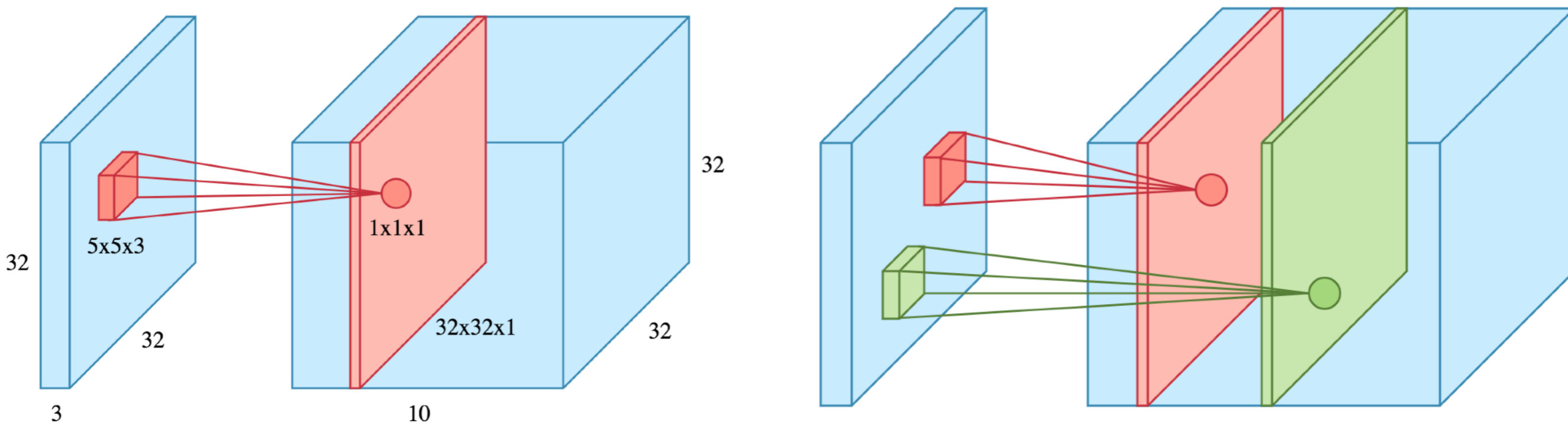
After a convolution layer, it is common to add a **pooling layer** in between CNN layers. The function of pooling is to continuously reduce the dimensionality to reduce the number of parameters and computation in the network

The most frequent type of pooling is **max pooling**, which takes the maximum value in each window

After the convolution and pooling layers, our classification part consists of a few fully connected layers

Convolutional Neural Network (CNN)

Let's say we have a $32 \times 32 \times 3$ image and we use a filter of size $5 \times 5 \times 3$

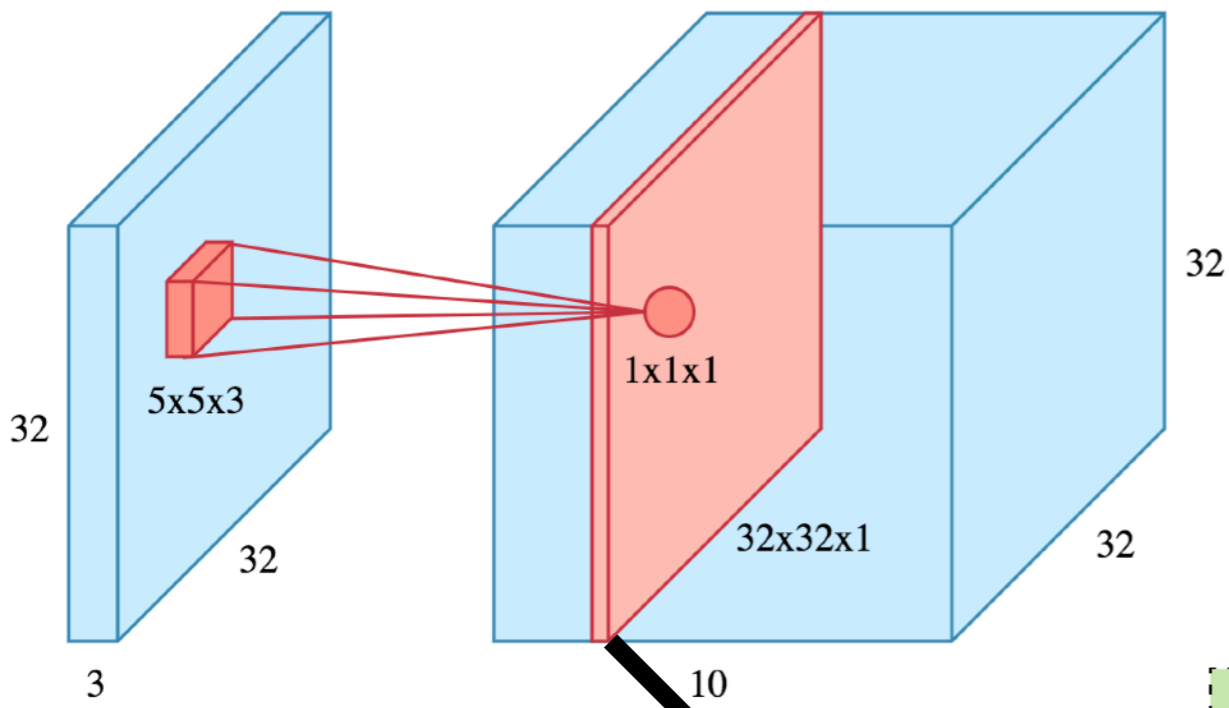


The convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

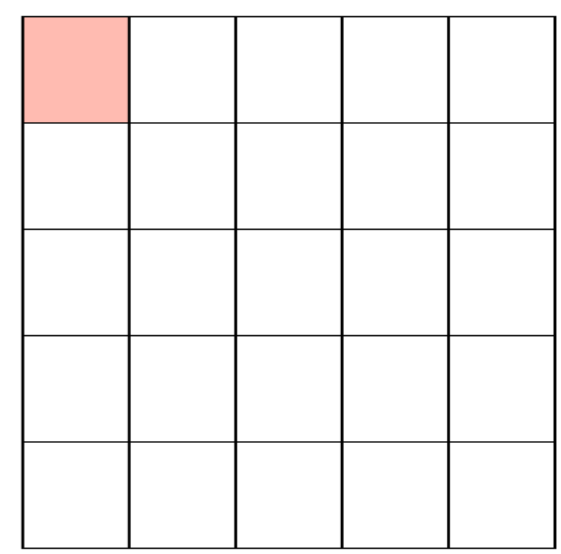
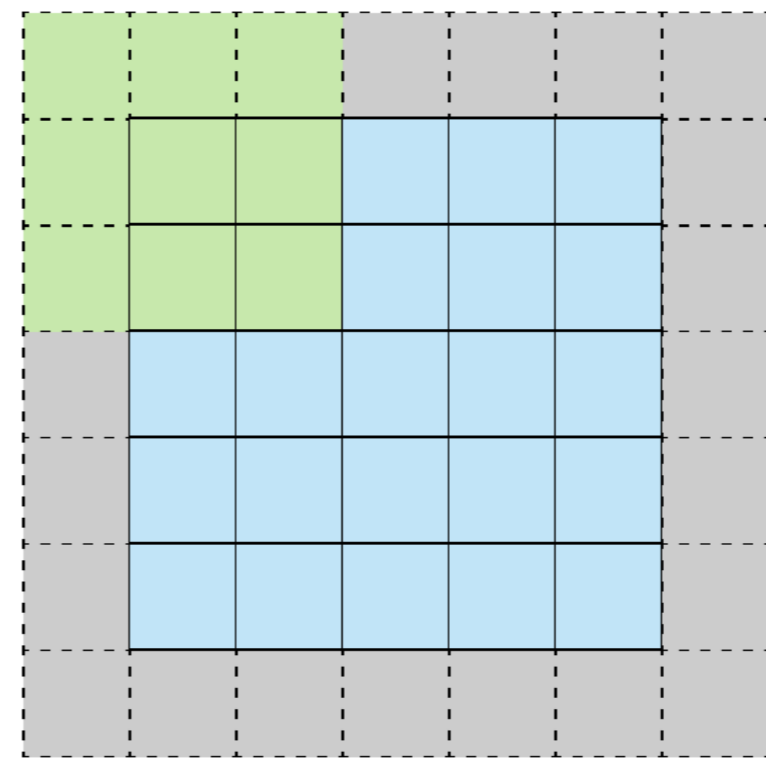
For any kind of neural network to be powerful, it needs to contain non-linearity

The 3D convolution figures we saw above used padding, that's why the height and width of the feature map was the same as the input (both 32×32), and only the depth changed.

Convolutional Neural Network (CNN)



The gray area around the input is the padding. We either pad with zeros or the values on the edge. Now the dimensionality of the feature map matches the input. Padding is commonly used in CNN to preserve the size of the feature maps, otherwise they would shrink at each layer, which is not desirable



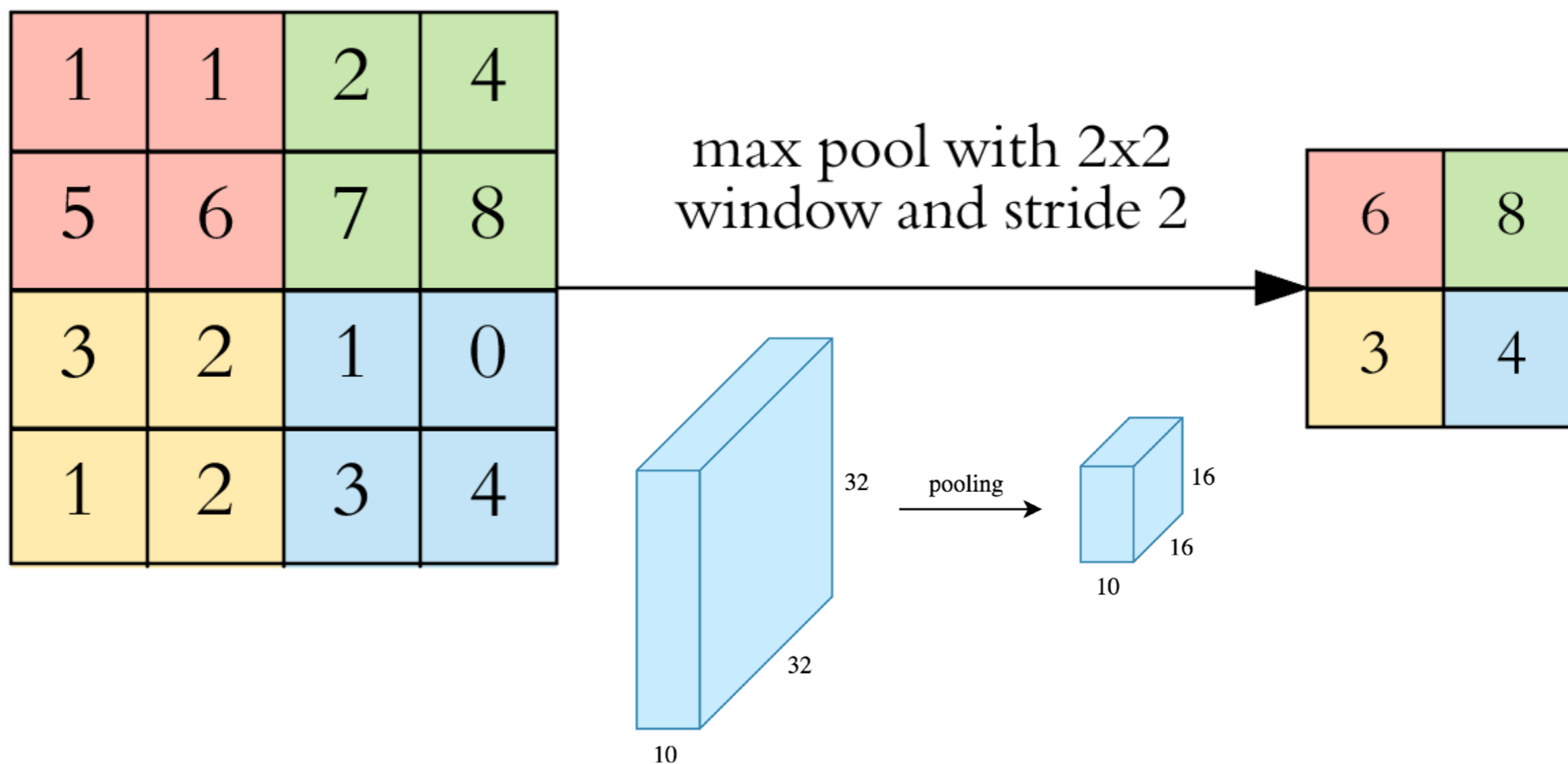
Stride 1 with Padding

Feature Map

Convolutional Neural Network (CNN)

After a convolution operation we usually perform *pooling* to reduce the dimensionality

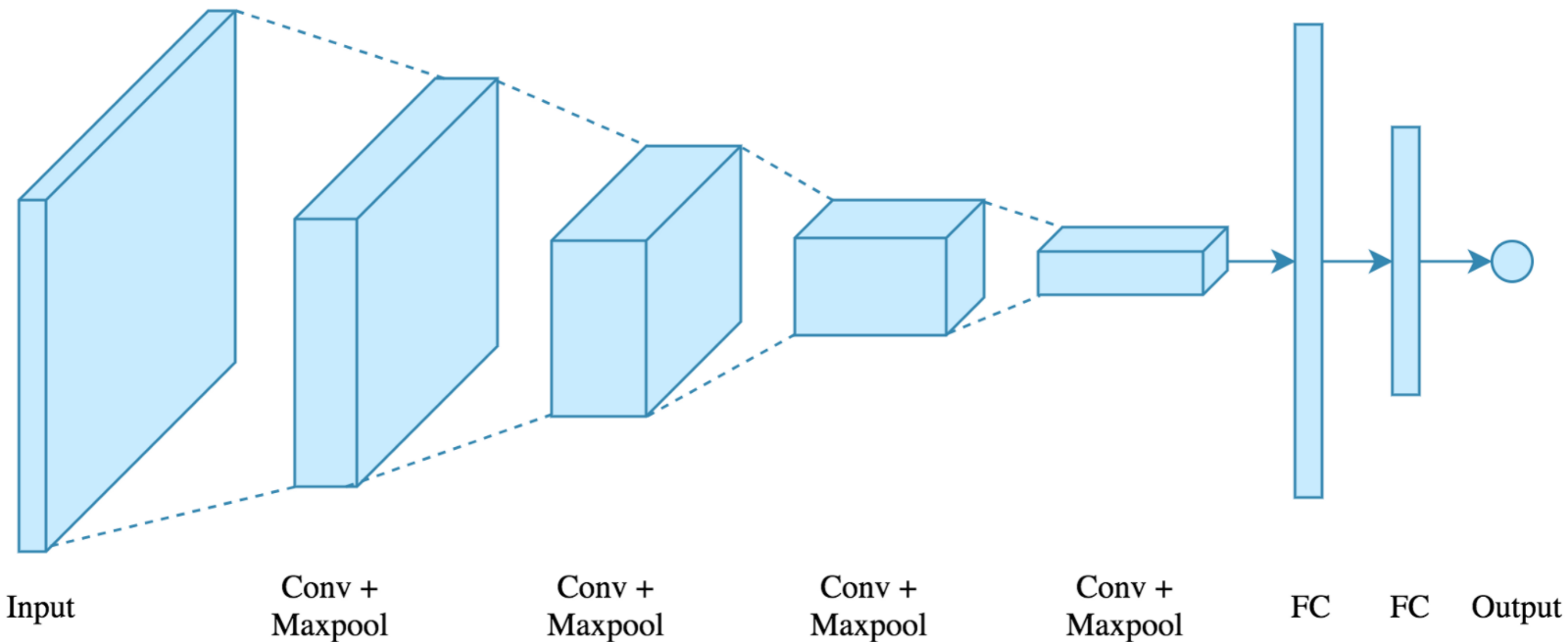
The most common type of pooling is *max pooling* which just takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters



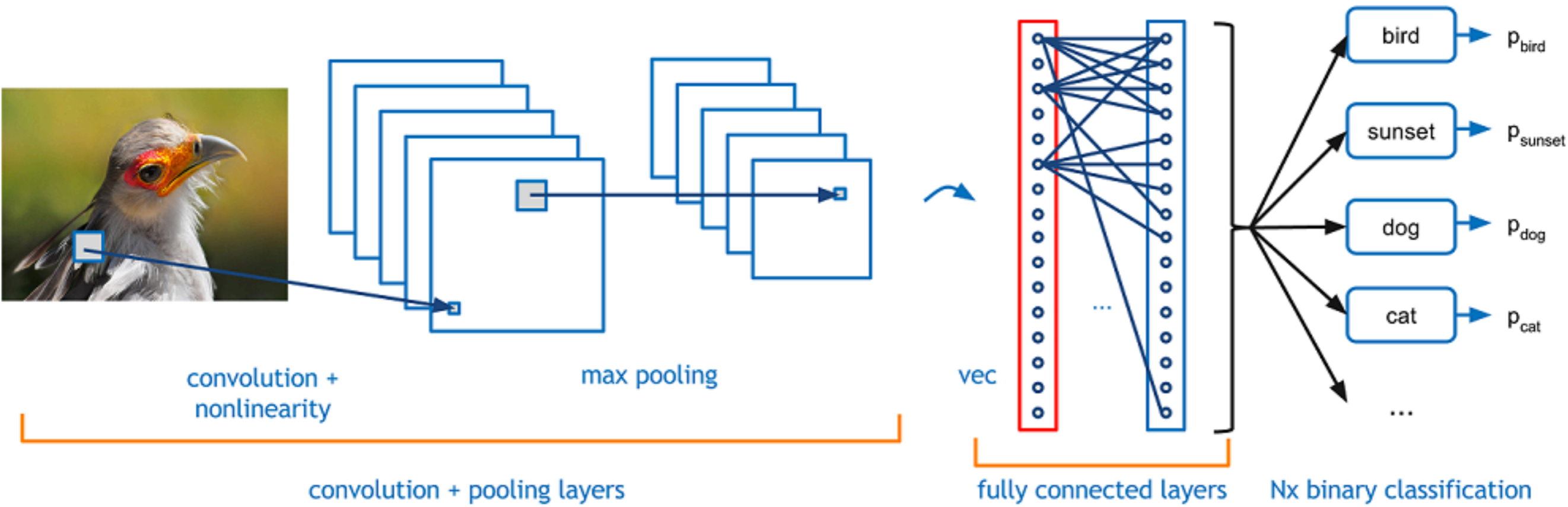
Convolutional Neural Network (CNN)

Fully Connected

After the convolution + pooling layers we add a couple of fully connected layers



Convolutional Neural Network (CNN)



the end