Introduction to Machine Learning

Part 2

Tomer Galanti & Sagie Benaim

March 13, 2019

Today's Lecture

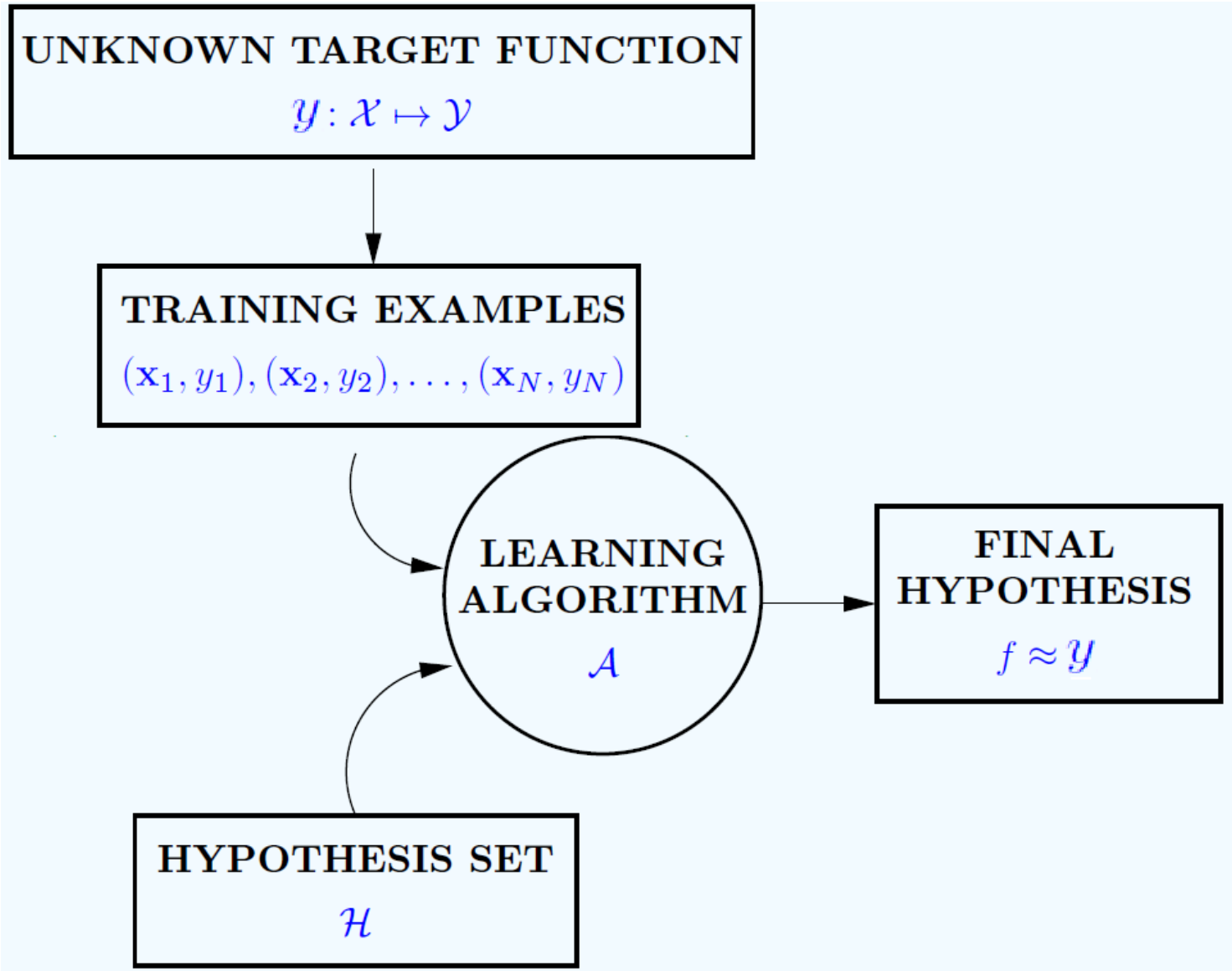
- A machine learning toolbox.
- Linear Regression.
- k-means.
- Support Vector Machine (SVM).
- Neural Networks (!).

Reminder: machine learning setting

The supervised learning protocol

- Unknown target function: y.
- Unknown distribution of instances: D.
- Training data: $S = \{(x_i, y(x_i))\}_{i=1}^m$ - dataset of i.i.d labelled instances.
- Hypothesis class: $\mathcal{H}$ - from which we select our candidates.
- Loss function: $\ell(f(x), y(x))$ - measures the per-sample error.
- Problem: We would like to return a function $f \in \mathcal{H}$ that has a low generalization error: $\mathcal{L}_D[f, y] := \mathbb{E}_{x \sim D} \ell(f(x), y(x))$.
- Solution: typically, we select $f \in \mathcal{H}$ that minimizes the empirical error $\mathcal{L}_S[f, y] := \frac{1}{m} \sum_{i=1}^m \ell(f(x_i), y(x_i))$ and hope that the generalization error is minimized as well.

Reminder: machine learning setting



Reminder: machine learning setting

Today's lecture:

- Unknown target function: y.
- Unknown distribution of instances: D.
- Training data: $S = \{(x_i, y(x_i))\}_{i=1}^m$ - dataset of i.i.d labelled instances.
- Hypothesis class: $\mathcal{H}$ - from which we select our candidates.
- Loss function: $\ell(f(x), y(x))$ - measures the per-sample error.
- Problem: We would like to return a function $f \in \mathcal{H}$ that has a low generalization error: $\mathcal{L}_D[f, y] := \mathbb{E}_{x \sim D} \ell(f(x), y(x))$.
- Solution: typically, we select $f \in \mathcal{H}$ that minimizes the empirical error $\mathcal{L}_S[f, y] := \frac{1}{m} \sum_{i=1}^m \ell(f(x_i), y(x_i))$ and hope that the generalization error is minimized as well.

Linear regression

Linear regression is.. a regression problem, i.e., continuous labels.

- Training dataset $S = \{(x_i, y_i)\}_{i=1}^{m}$.
- $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$.

Linear Regression

In the linear regression problem we are interested in minimizing a training error that corresponds to the L_2**-loss function**:

$$\ell(f(x), y(x)) = (f(x) - y(x))^2 \tag{1}$$

The training error is therefore,

$$\mathcal{L}_S[f, y] = \frac{1}{m} \sum_{i=1}^{m} (f(x_i) - y(x_i))^2 \tag{2}$$

Note: this kind of training error is also known as the **mean squared error**.

Linear regression

The output of linear regression is a linear function of the input.

$$\mathcal{H} = \{f_w(x) = \langle w, x \rangle \mid w \in \mathbb{R}^n\} \tag{3}$$

where, $w \in \mathbb{R}^n$ is a vector of parameters.

Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features.

Linear regression

To minimize $\mathcal{L}_S[f_w, y]$, we can simply solve for where its gradient is 0, i.e.,

$$\nabla_w \mathcal{L}_S[f_w, y] = \nabla_w \frac{1}{m} \sum_{i=1}^{m} (\langle w, x_i \rangle - y(x_i))^2 = 0 \tag{4}$$

The solution is:

$$w = (X^\top X)^{-1} X^\top v \tag{5}$$

where $X = (x_i)_{i=1}^m$ is the matrix where each row consists of a training instance and $v = (y(x_1), \ldots, y(x_m))$ is the vector of labels of the training instances.

Linear regression

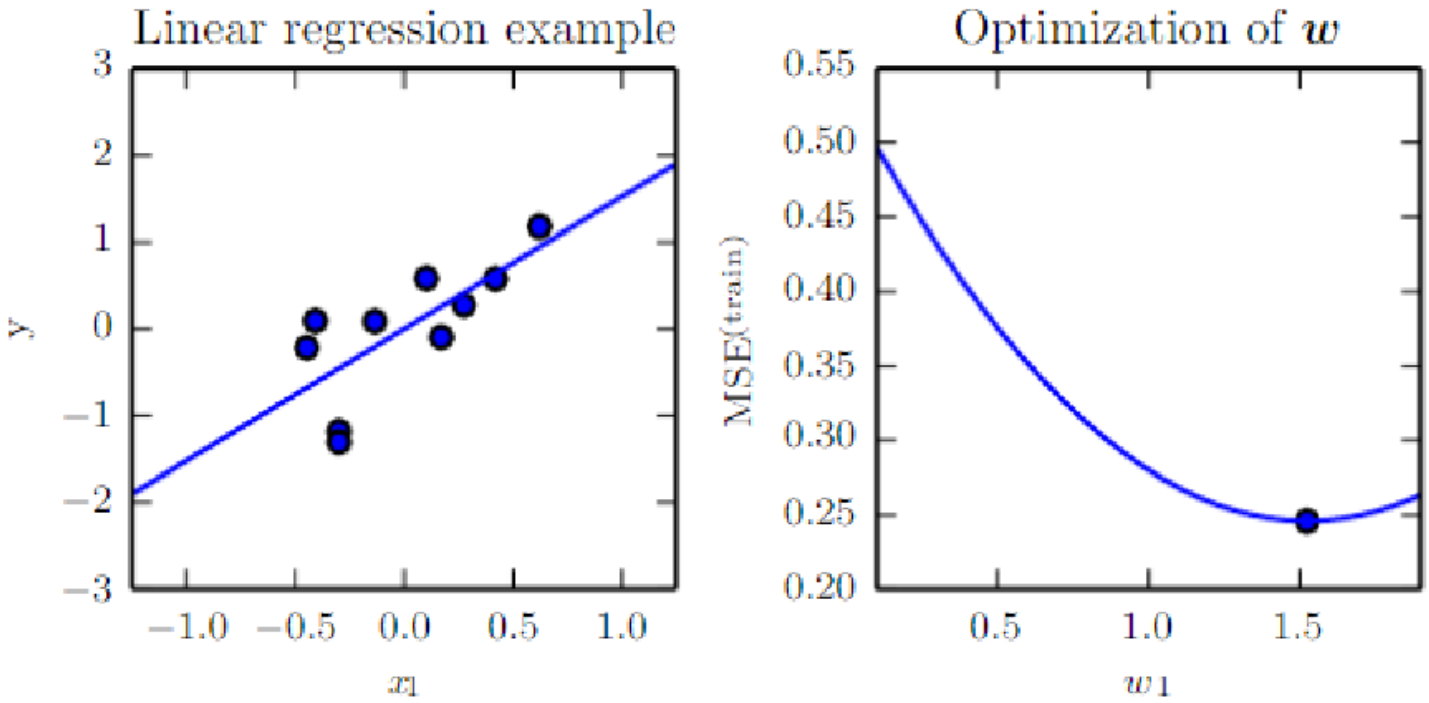


Figure: Linear regression, with a training set of ten data points, each containing one feature. Because there is only one feature, the weight vector w contains only a single parameter to learn, w_1. (Left) Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. (Right) The plotted point indicates the value of w_1 found by the closed form solution, which we can see minimizes the mean squared error on the training set.

k-means

Clustering algorithm, unsupervised learning.

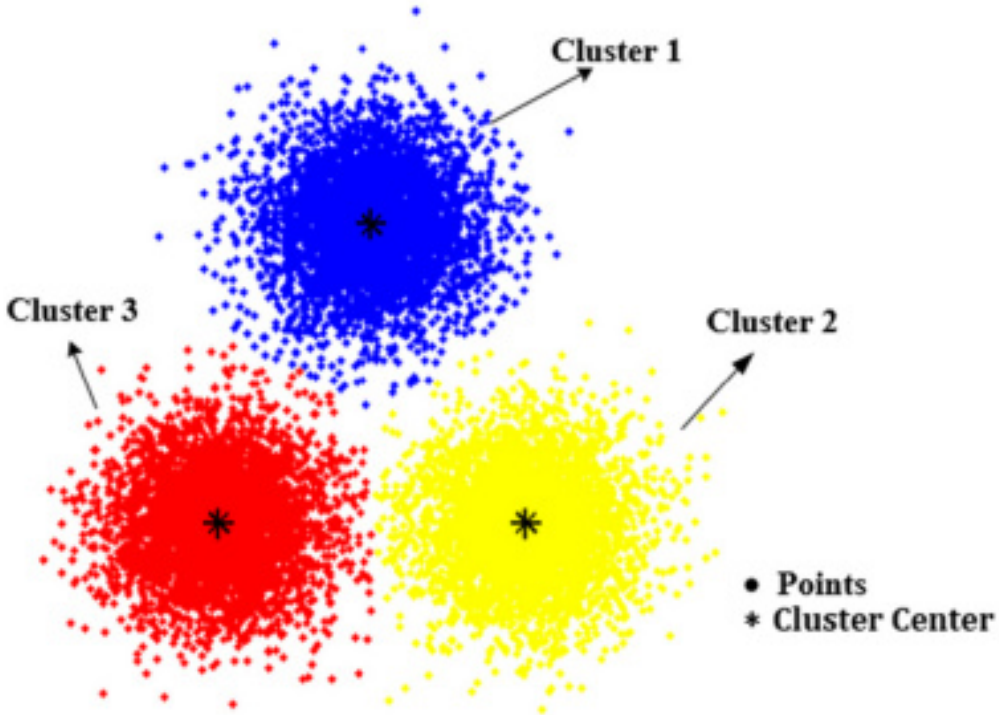


Figure: k-means algorithm with $k = 3$. The algorithm finds 3 means for the data points. Each new instance x is classified as i if it the i'th mean is closest to x.

k-means

Given a set of observations $(x_1, x_2, \ldots, x_m)$, where each observation is a n-dimensional real vector, k-means clustering aims to partition the n observations into k ($\leq m$) sets $P = \{P_1, P_2, \ldots, P_k\}$ so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\arg\min_{P} \sum_{i=1}^{k} \sum_{x \in P_i} \|x_i - \mu_i\|^2 \tag{6}$$

where μ_i is the mean of the cluster P_i.

k-means

- In general this problem is NP-hard in general Euclidean space of dimension n even for 2 clusters.
- NP-hard for a general number of clusters k even in the plane.

k-means

A heuristic algorithms.

Lloyd's algorithm:

1. Given an initial set of means $\mu_1, \ldots, \mu_k$, the algorithm proceeds by alternating between two steps:
   1. Assign each observation to the cluster whose mean has the least squared Euclidean distance (i.e., the "nearest" mean).
   2. Update step: calculate the new means to be the centroids of the observations in the new clusters.

k-means

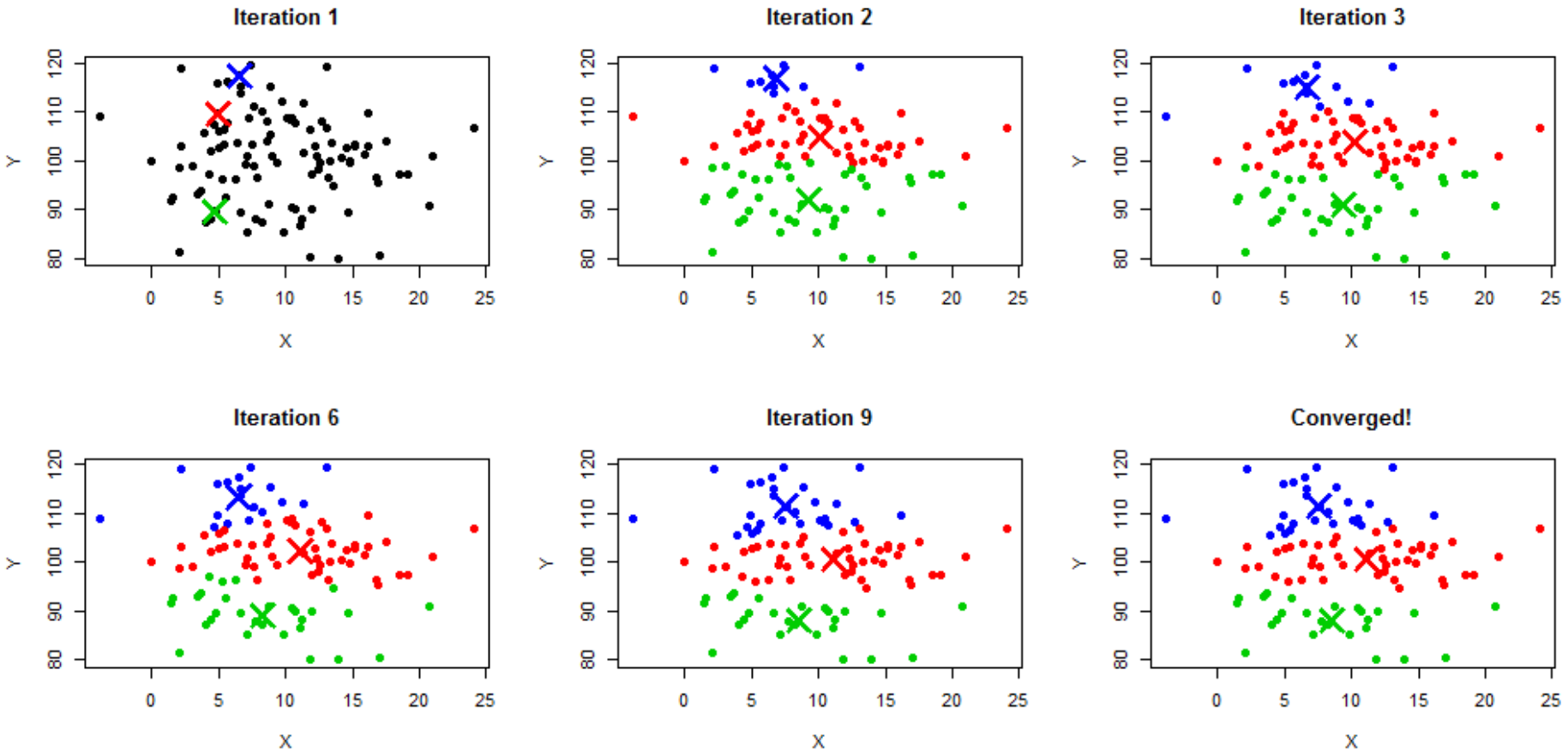


Figure: A run of Lloyd's algorithm. The algorithm initializes with three random clusters and updates them iteratively till convergence.

Support Vector Machines

Before neural networks were a thing, we had **Support Vector Machines** (SVM for short).

- Training data: $S = \{(x_i, y(x_i))\}_{i=1}^m \subset \mathbb{R}^n \times \{\pm 1\}$.
- Problem: find a classifier $f \in \mathcal{H}$ with a small empirical error $\mathcal{L}_S[f, y] := \frac{1}{m} \sum_{i=1}^m \text{Ind}[f(x_i) \neq y(x_i)]$.
- Hypothesis class: the hypothesis class of standard SVMs is the class of **linear classifiers**:

$$\mathcal{H} = \left\{ x \mapsto \text{sign}(\langle w, x \rangle + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R} \right\} \tag{7}$$

Support Vector Machines

What is a linear classifier?

Any hyperplane can be represented as $\{x|\langle w, x\rangle + b = 0\}$ for some $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

This is the set of solutions x to a single linear equation.

Support Vector Machines

What is a linear classifier?

- Any hyperplane can be represented as $\langle w, x \rangle + b = 0$.
- The space is partitioned into: $\langle w, x \rangle + b \geq 0$ and $\langle w, x \rangle + b < 0$.

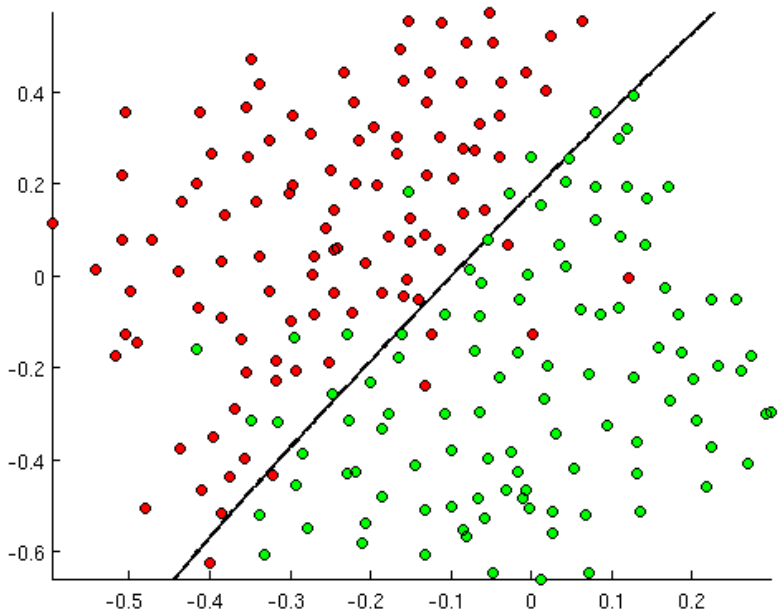


Figure: A linear classifier. Not necessarily a good classifier.

Support Vector Machines

Linearly separable data.

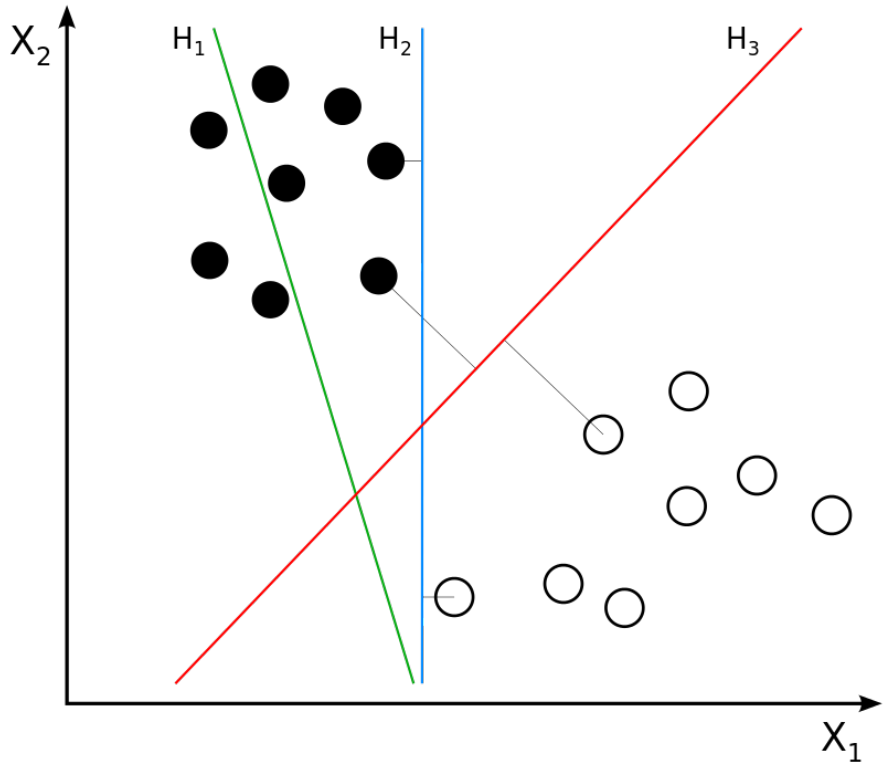


Figure: Linearly separable data. There are (infinitely) many alternative linear separators.

Support Vector Machines

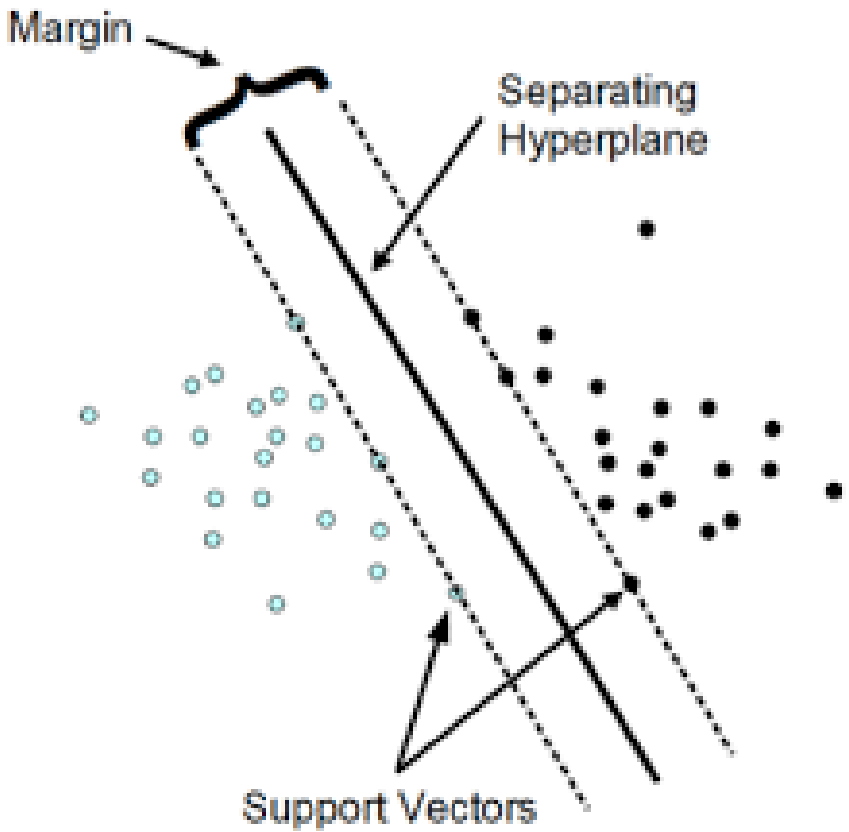
We focus only on linearly separable data.

Hard-margin linear SVM = an algorithm that given a linearly separable dataset, returns a max-margin linear classifier.

Extensions to the ”almost” separable case exist. It is called soft-margin linear SVM. We do not need it in the current discussion.

Support Vector Machines

In terms of generalization, we prefer to select a linear separator that maximizes the margin (Vapnik and Chervonenkis, 1965.).



Support Vector Machines

Before we continue, note that for all x:

$$\begin{aligned} & \text{Ind}[\text{sign}(\langle w, x \rangle + b) \neq y(x)] = 0 \\ \iff & y(x)(\langle w, x \rangle + b) \geq 0 \end{aligned} \tag{8}$$

Support Vector Machines

Distance between a hyperplane $H_{w,b} := \{x : \langle w, x \rangle + b = 0\}$ and a point $\bar{x}$ is:

$$d(H_{w,b}, \bar{x}) := \min_{x:\langle w,x\rangle+b=0} \|x - \bar{x}\|_2 = \frac{|\langle w, \bar{x} \rangle + b|}{\|w\|_2} \tag{9}$$

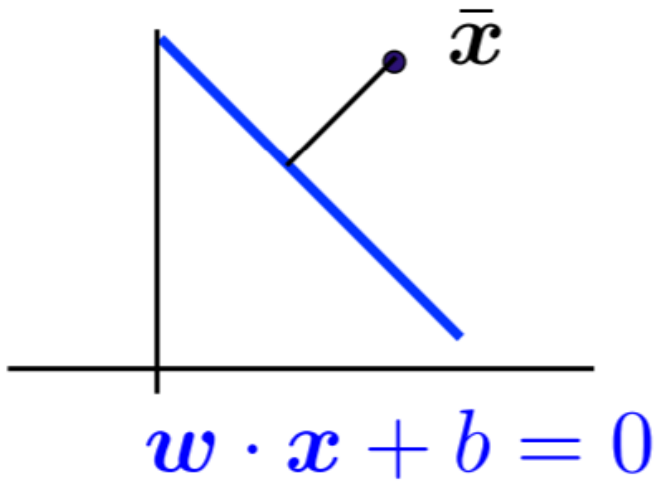


Figure: The distance between a hyperplane and a point.

Support Vector Machines

The maximal margin in a separable dataset $S = \{(x_i, y_i)\}_{i=1}^m$ is:

$$\begin{aligned}
&\max_{w,b:\text{ consistent with } S} \min_{i\in\{1,\ldots,m\}} d(H_{w,b}, x_i) \\
&= \max_{w,b:\forall i:\text{Ind}[\text{sign}(\langle w,x_i\rangle+b)\neq y(x_i)]=0} \min_{i\in\{1,\ldots,m\}} \frac{|\langle w, x_i\rangle + b|}{\|w\|_2} \\
&= \max_{w,b:\forall i: y(x_i)(\langle w,x_i\rangle+b)\geq 0} \min_{i\in\{1,\ldots,m\}} \frac{|\langle w, x_i\rangle + b|}{\|w\|_2} \\
&= \max_{w,b: y(x_i)(\langle w,x_i\rangle+b)\geq 1} \frac{1}{\|w\|_2}
\end{aligned} \tag{10}$$

Note: the derivation of the last equation is omitted.

Support Vector Machines

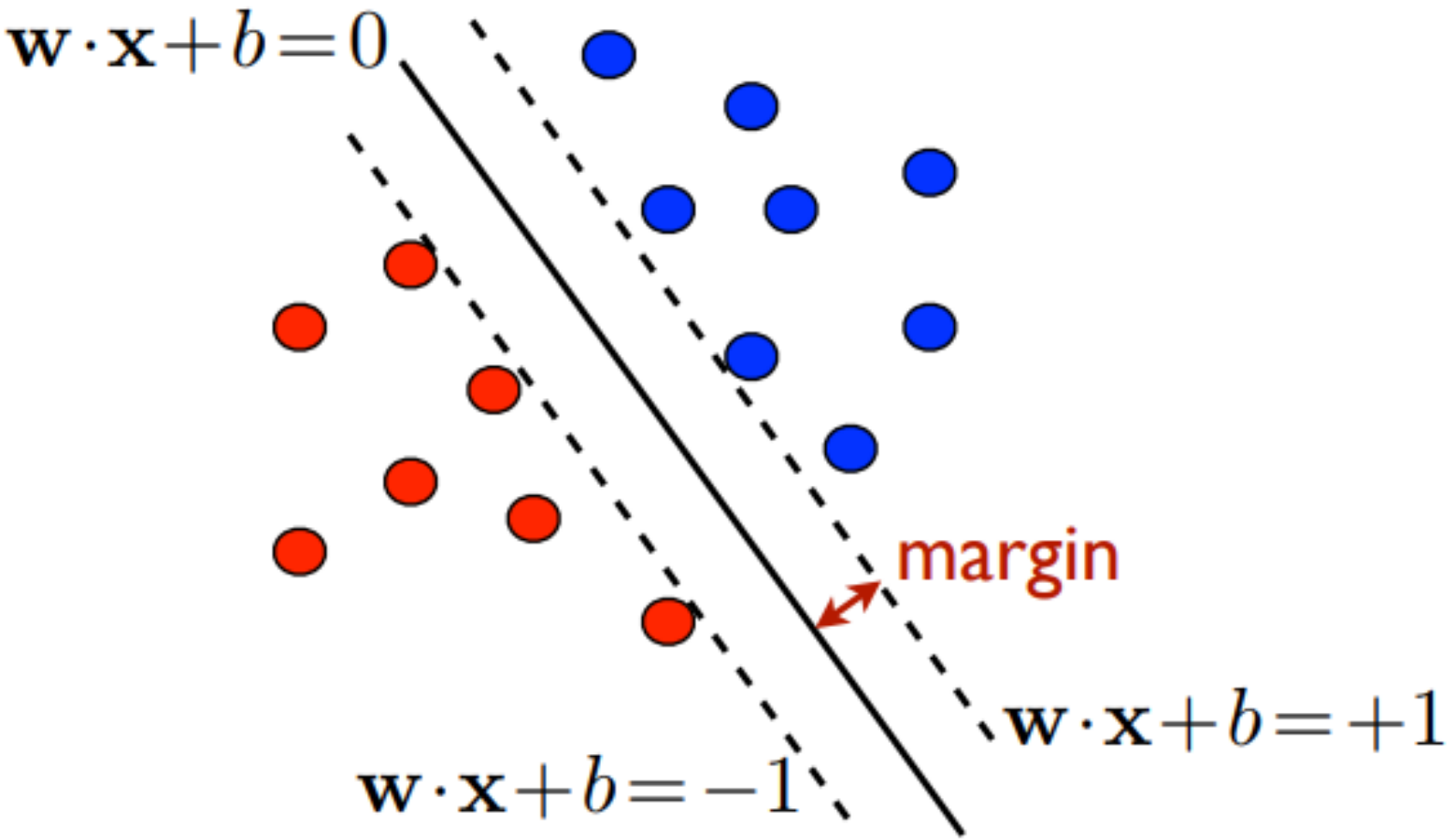


Figure: The maximal margin linear separator.

Support Vector Machines

This leads us to the following objective:

$$\underbrace{\max_{w,b} \frac{1}{\|w\|_2}}_{\text{maximal margin}} \quad \text{subject to:} \quad \underbrace{\forall i \in [m] : y(x_i)(\langle w, x_i \rangle + b) \geq 1}_{\text{consistent with the data}} \tag{11}$$

Or alternatively, we can represent it as the **Hard-margin SVM** objective for selecting w and b:

$$\underbrace{\min_{w,b} \frac{1}{2}\|w\|_2^2}_{\text{maximal margin}} \quad \text{subject to:} \quad \underbrace{\forall i \in [m] : y(x_i)(\langle w, x_i \rangle + b) \geq 1}_{\text{consistent with the data}} \tag{12}$$

Support Vector Machines

What happens if the data is not linearly separable, but it is linearly separable after transforming the data using a non-linear transformation $\phi(x)$?

We call the function ϕ a **feature map**.

Support Vector Machines

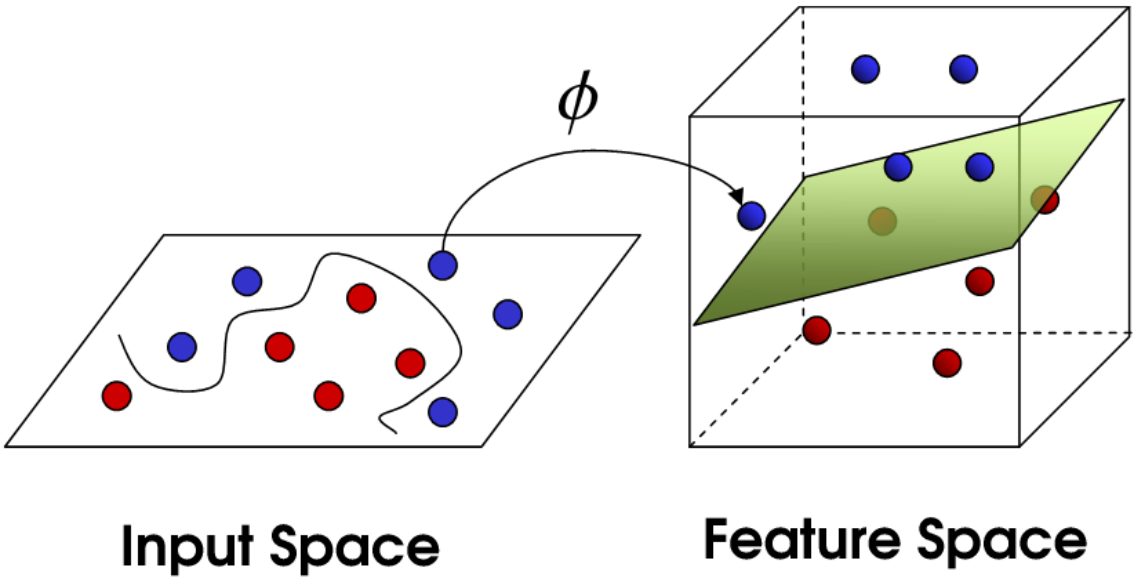


Figure: Feature map that transforms a non linearly separable data into a linearly separable data in a higher dimensional space.

Support Vector Machines

- Consider a case where the positive points are in the unit circle $\{(x, y) \mid x^2 + y^2 \leq 1\}$ and the negative points outside the unit circle.
- The transformation $\phi(x, y) = (x^2 - 0.5, y^2 - 0.5)$ takes the data and transforms it into a linearly separable data.
- The linear separator: $w = (-1, -1)$ and $b = 0$.

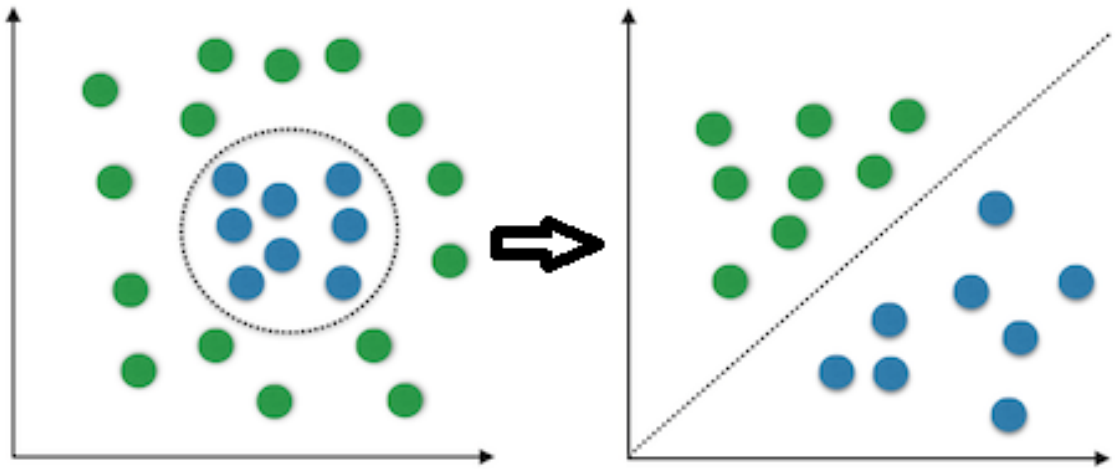


Figure: Feature map that transforms a non linearly separable data into a linearly separable data in a higher dimensional space.

Support Vector Machines

In this case, we have a **pre-defined feature mapping** $\phi : \mathbb{R}^n \to \mathbb{R}^m$. The hypothesis class becomes:

$$\mathcal{H} = \left\{ \text{sign}(\langle w, \phi(x) \rangle + b) \mid w \in \mathbb{R}^m, b \in \mathbb{R} \right\} \tag{13}$$

Instead of:

$$\mathcal{H} = \left\{ \text{sign}(\langle w, x \rangle + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R} \right\} \tag{14}$$

Do you see the difference?

Support Vector Machines

This is a very nice framework, since, now we are able to separate data that is not necessarily linearly separable.

The main problem with this framework is that a-priori we do not know how to select ϕ. There are many different types of feature maps, each has its own properties.

Support Vector Machines

There are two solutions to this problem.

Option 1: use a very generic ϕ, such as the infinite-dimensional ϕ. If $\phi(x)$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor. Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

Support Vector Machines

There are two solutions to this problem.

Option 2: manually engineer ϕ. For a long time this was the dominant approach. It requires decades of human eort for each separate task, with practitioners specializing in dierent domains, such as speech recognition or computer vision, and with little transfer between domains.

Support Vector Machines

Even if we are very talented in engineering feature maps:

- Not automatic: we do not have a principled way to select the feature map automatically.
- Not generic: for each dataset we have a tailor-made feature map.
- Non-adaptive: in more complicated scenarios, what happens if the data changes over time?
- Non transferable: we cannot transfer the feature map from one domain to another.

Support Vector Machines

Deep learning is not a cure all solution.

But, it is a very good solution to the proposed problem. Using deep learning we can easily learn a feature map in a principled way.

The process of learning a feature map is also called **representation learning** and neural networks are very good in representation learning (we even have a conference for representation learning).

The brain

Neurons in the brain look like this:

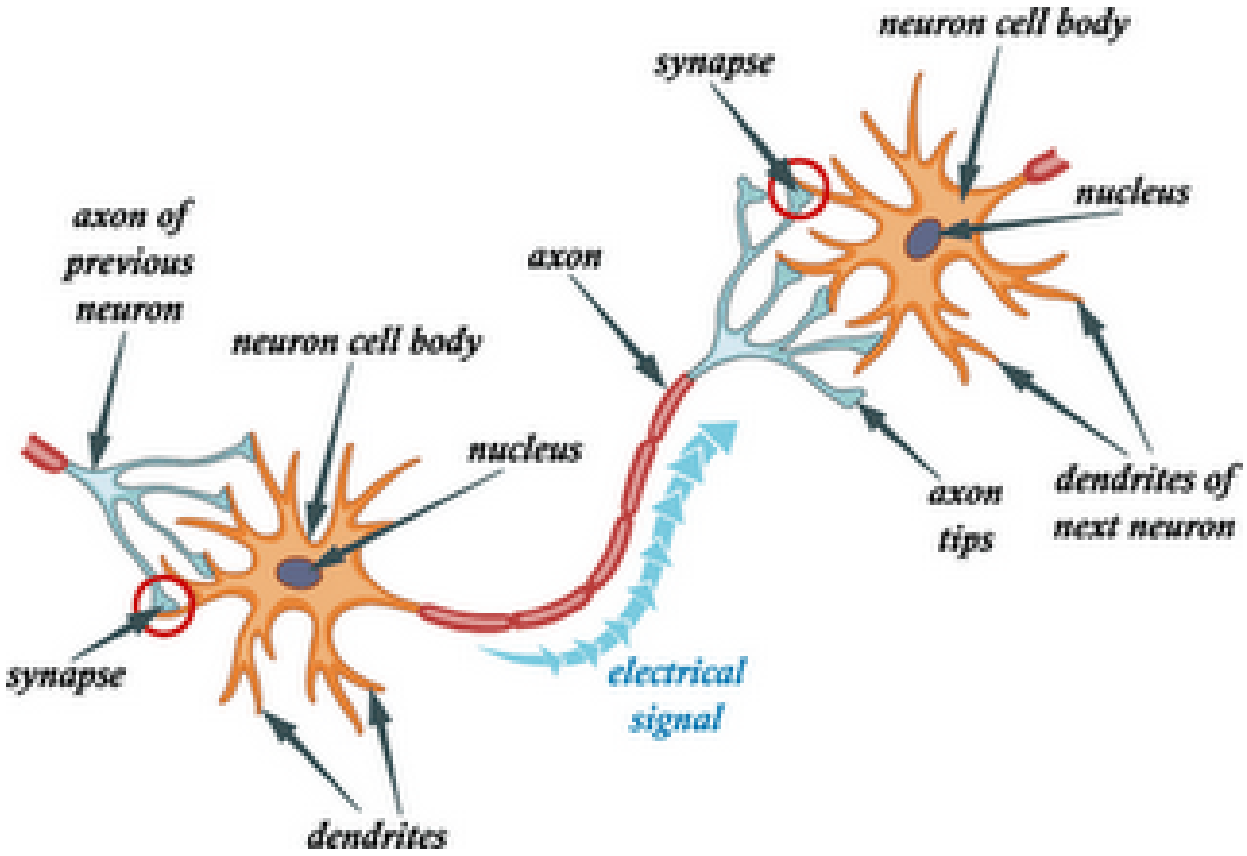


Figure: An illustration of two neurons connected to each other. Each neuron takes electric signals from previous neurons and returns an electric electric to the next neurons.

An artificial neuron (perpectron)

An **artificial neuron** (**perceptron** or **neuron** for short) is a function $x \mapsto \sigma(\langle w, x \rangle + b)$, where,

- $x \in \mathbb{R}^n$ is the **input** of the neuron.
- $w \in \mathbb{R}^n$ is the **weights** of the neuron.
- $b \in \mathbb{R}$ is the **bias** of the neuron.
- $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear **activation function** of the neuron.

An artificial neuron (perceptron)

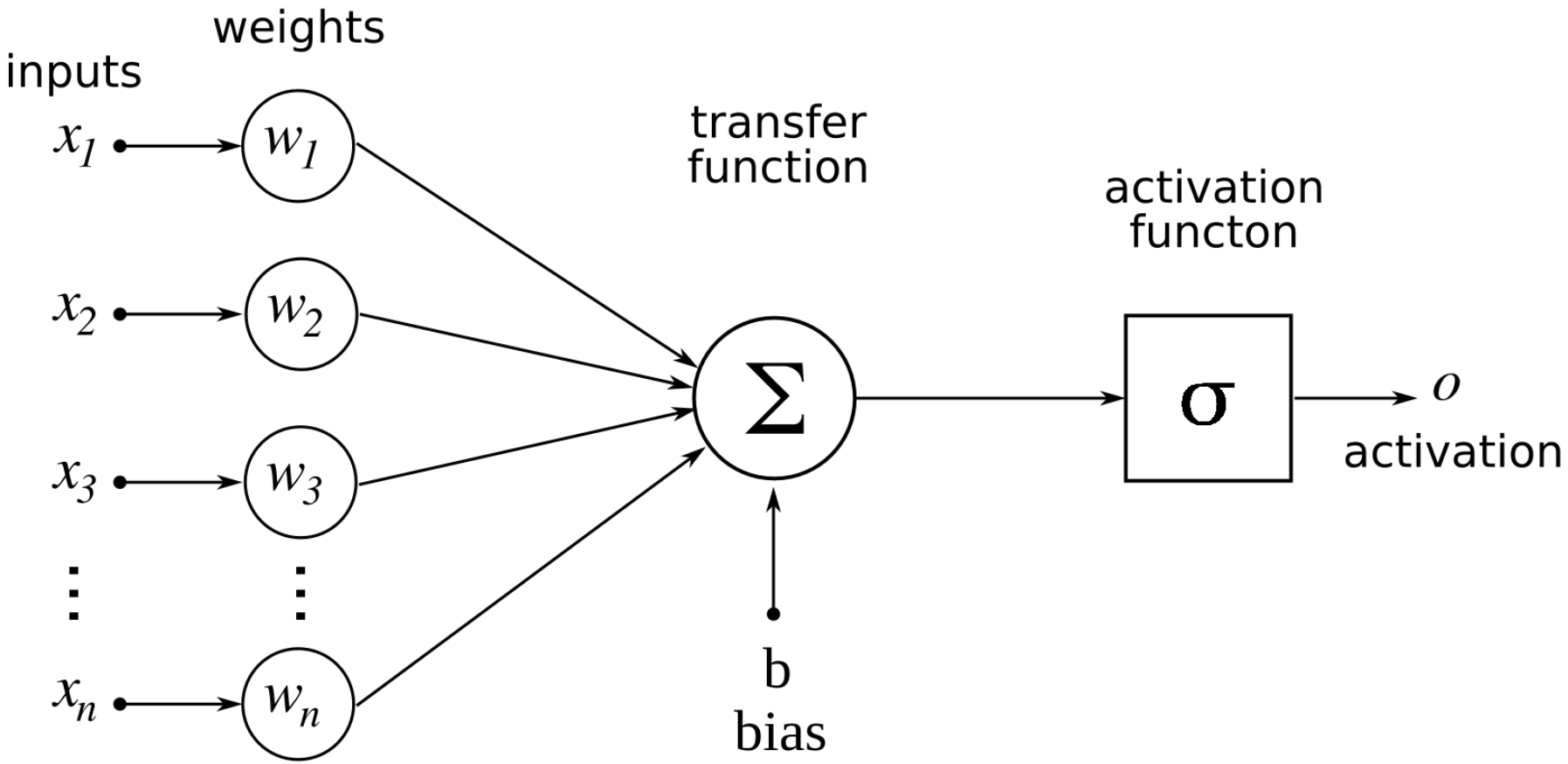


Figure: A neuron with n inputs, weights $w_1, \ldots, w_n$, bias b and a non-linear activation function σ.

Activation functions

Sign function: $\text{sign}(x) = \pm 1$ depending on if x is positive or negative.

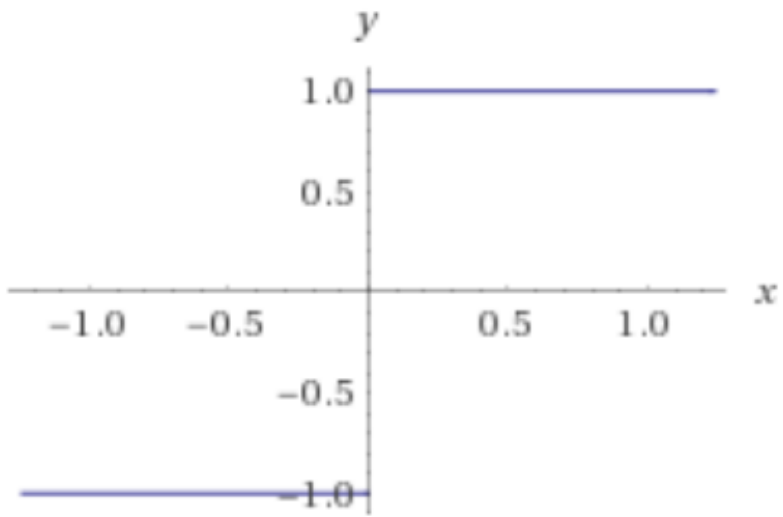


Figure: The sign activation function.

Activation functions

Rectified linear unit: $(x) = \max(0, x)$.

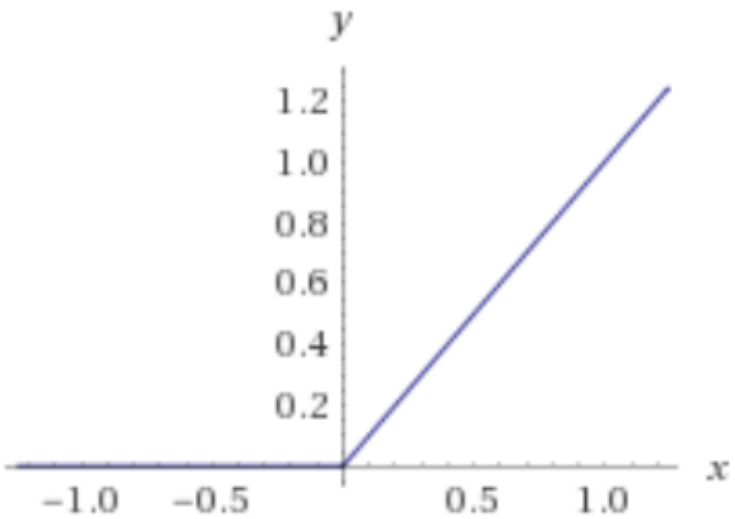


Figure: The ReLU activation function.

Activation functions

Hyperbolic tangent: $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$.

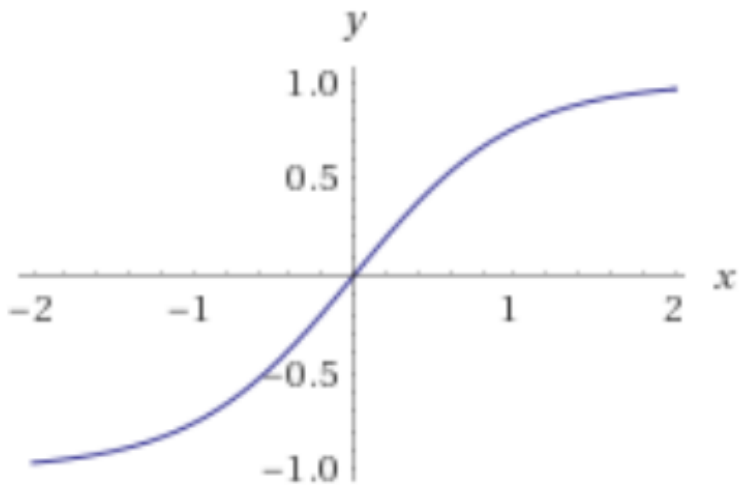


Figure: The tanh activation function.

Activation functions

Sigmoid/Logistic function: $s(x) = \frac{1}{1+\exp(-x)}$.

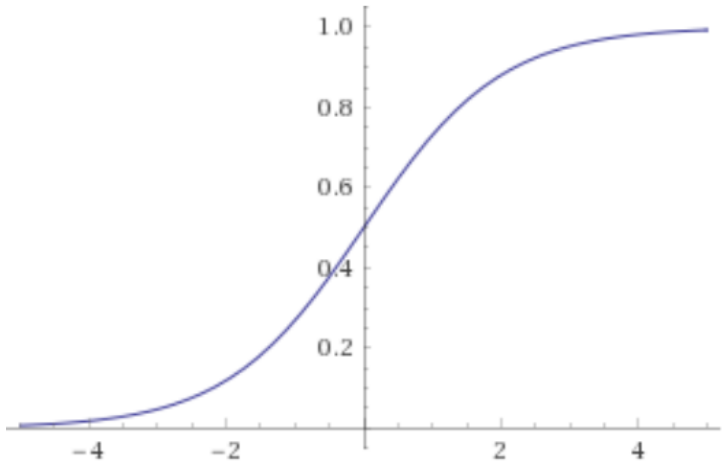


Figure: The sigmoid activation function.

Neural networks

There are many kinds of neural networks. The simplest variant is the **Multi-Layered Perceptron** (MLP for short).

$$h_{w,b}(x) := \sigma_d(W_d(\dots\sigma_1(W_1 x + b_1)\dots) + b_d) \tag{15}$$

Where:

- $w = (W_1, \dots, W_d)$ and $b = (b_1, \dots, b_d)$ are together the collection of all of the parameters in the neural network, $W_i \in \mathbb{R}^{h_i \times h_{i+1}}$ and $b_i \in \mathbb{R}^{h_{i+1}}$.
- Each σ_i is an element-wise activation function, i.e., we apply σ_i on each coordinate separately.

Neural networks

There are many kinds of neural networks. The simplest variant is the **Multi-Layered Perceptron** (MLP for short).

$$h_{w,b}(x) := \sigma_d(W_d(\dots\sigma_1(W_1 x + b_1)\dots) + b_d) \tag{16}$$

- $h_1 = n$ is the input dimension.
- $\sigma_i(W_d(\dots\sigma_1(W_1 x + b_1)\dots) + b_i)$ is the $(i+1)$**'th layer** and h_{i+1} is it's dimension, also known as **width** of the layer.
- d is the **depth** of the network.
- The intermediate layers (excluding input and output layers) are called **hidden layers**.
- The **width** of the network is the maximal width of the hidden layers.

Neural networks

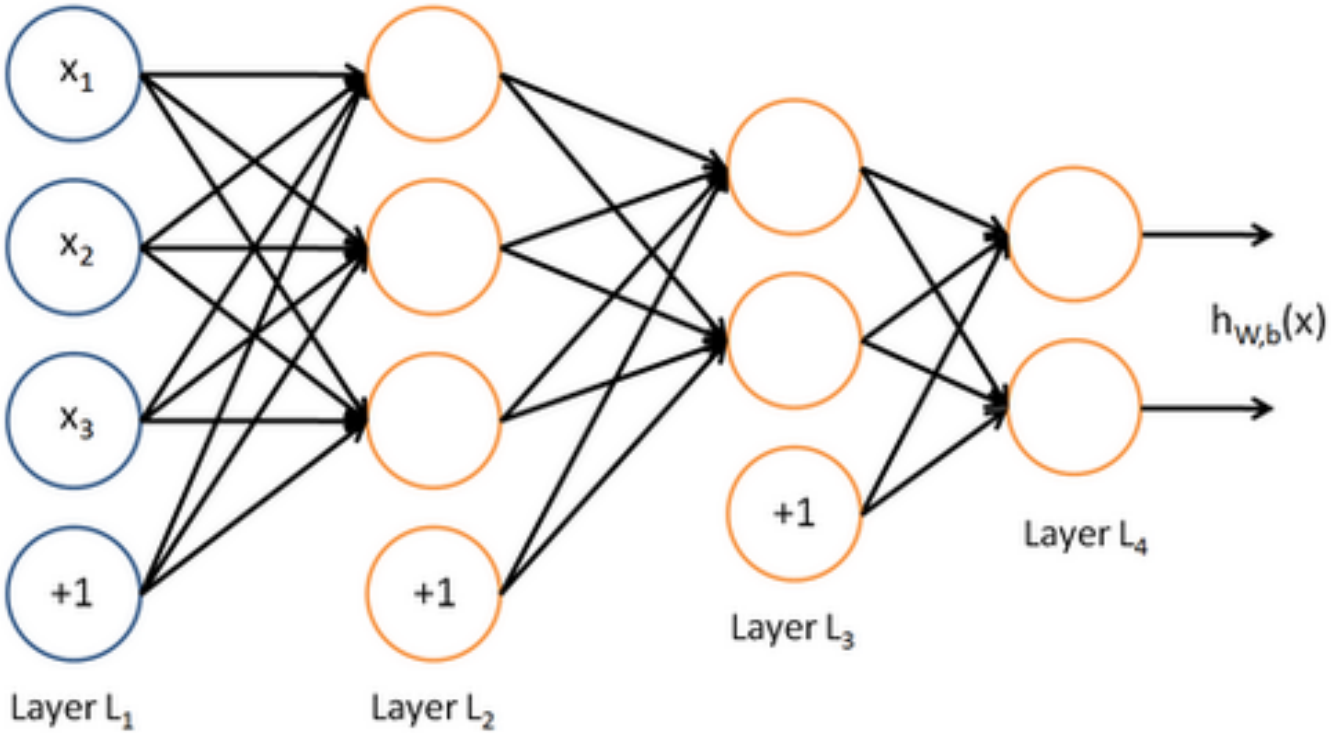


Figure: A neural network with 3 input nodes, two hidden layers of dimensions 3 and 2 and two output nodes.

Neural networks

$$h_{w,b}(x) := \sigma_d(W_d(\dots\sigma_1(W_1 x + b_1)\dots) + b_d) \tag{17}$$

A neural network with only one hidden layer is considered **shallow** and a neural network with more than one hidden layer is considered **deep**.

Neural networks

We can think of a neural network as a multi-layered function:

$$h_{w,b}(x) = f_d \circ f_{d-1} \circ \cdots \circ f_1(x) \tag{18}$$

where

$$f_i(z) = \sigma_i(W_i z + b_i) \tag{19}$$

Each layer is responsible for computing a representation of the input.

We can think of it as moving from ”low level” perception to ”high level” perception.

Neural networks

A slightly different interpretation of neural networks is:

$$h_{w,b}(x) = \sigma_d(W_d\phi(x) + b_d) \tag{20}$$

where $\phi(x)$ is the first $d-1$ layers.

With this form we can think of $\phi(x)$ is a feature map and the final layer as a classifier!

Exactly the same formulation as SVMs on top of a feature map:

$$x \mapsto \text{sign}(\langle w, \phi(x)\rangle + b) \tag{21}$$

Neural networks

We can think of the first $d - 1$ of a neural network as an implementations of a feature map. The final layer is a classifier on top of the feature map.

If we can train neural networks efficiently, we have an automatic method for selecting feature maps.

Neural networks

A real life neural network. Impressive huh?

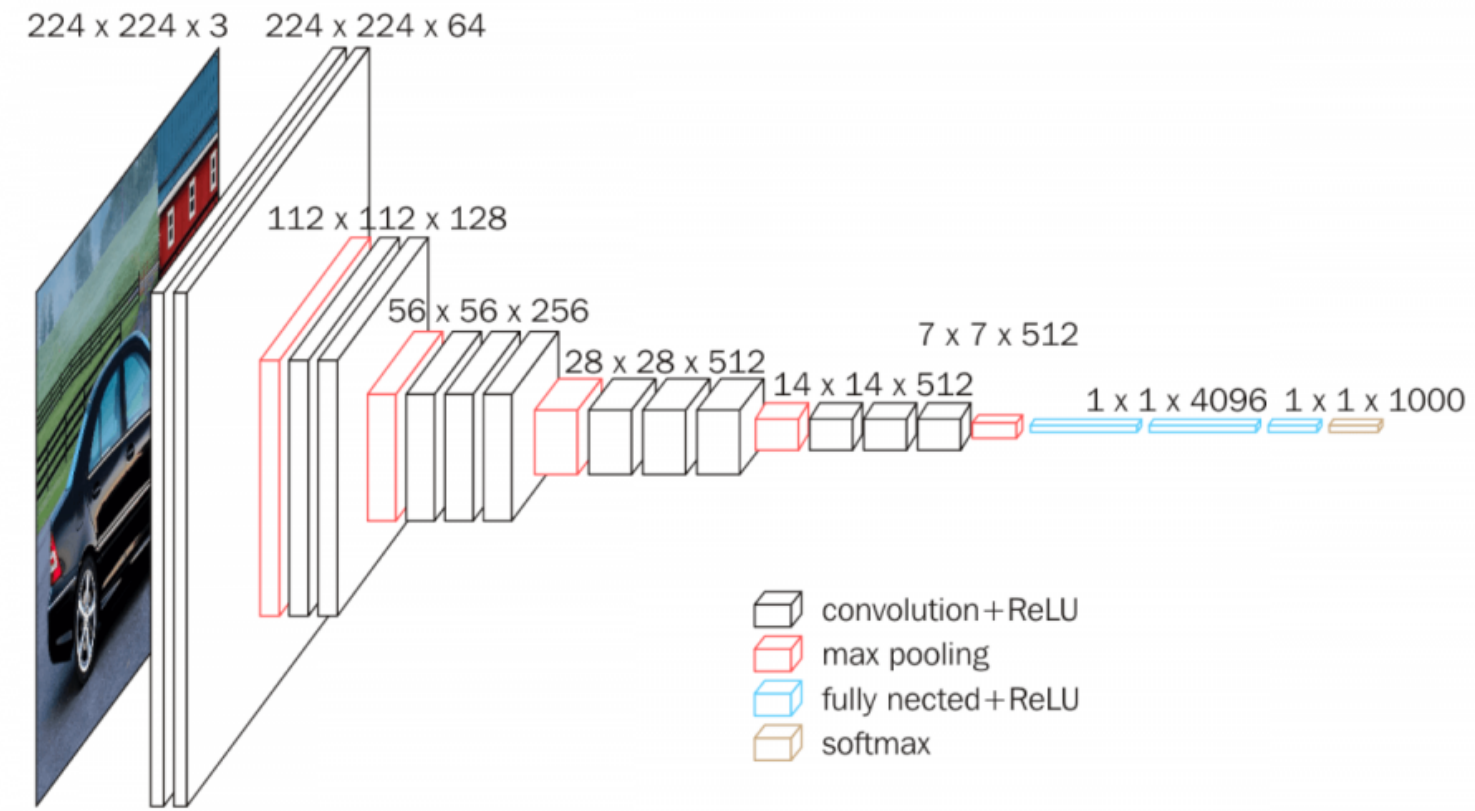


Figure: The VGG neural network. The width decreases as the network deepens.

Learning the XOR function

To make the idea of a neural network more concrete, we begin with an example of a MLP network on a very simple task: learning the XOR function.

Learning the XOR function

The XOR function (exclusive or):

$$\text{XOR} : \{0,1\}^2 \to \{0,1\}, \quad \text{XOR}(x_1, x_2) = 0 \iff x_1 = x_2 \tag{22}$$

The XOR function provides the target function y that we want to learn.

Learning the XOR function

Our model provides a hypothesis class

$$\mathcal{H} = \left\{ h_\theta \mid \theta \in \Theta \right\} \tag{23}$$

Our learning algorithm will adapt the parameters θ to make h_θ as similar as possible to y.

Learning the XOR function

In this simple example, we will not be concerned with statistical generalization.

We want our network to perform correctly on the four points $X = \{(0,0),(0,1),(1,0),(1,1)\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

Learning the XOR function

We can treat this problem as a regression problem and use a L_2-loss function $\ell(h_\theta(x), y(x)) = (h_\theta(x) - y(x))^2$.

Note: we have chosen this loss function to simplify the math for this example as much as possible. In practical applications, the L_2-squared loss is usually not an appropriate cost function for modeling binary data.

Learning the XOR function

Evaluated on our whole training set, the empirical error function is

$$\mathcal{L}_S[h_\theta, y] = \sum_{i=1}^{4} (h_\theta(x_i) - y(x_i))^2 \tag{24}$$

Note: if D is uniform on X, then, we also have: $\mathcal{L}_S[h_\theta, y] = \mathcal{L}_D[h_\theta, y]$.

Learning the XOR function

Modeling: we choose the form of our model, $h_\theta(x)$.

Suppose that we choose a linear model, with θ consisting of a vector of weights $w \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$. Our hypothesis class is:

$$\mathcal{H} = \left\{ h_{w,b}(x) = \langle w, x \rangle + b \mid w \in \mathbb{R}^n, b \in \mathbb{R} \right\} \tag{25}$$

Learning the XOR function

We can minimize $\mathcal{L}_S[h_{w,b}, y]$ in a closed form with respect to w and b using the normal equations.

After solving this task, the normal equations provide us $w = 0$ and $b = 0.5$. The linear model simply outputs 0.5 everywhere.

Why does this happen?

Learning the XOR function

Why does this happen?

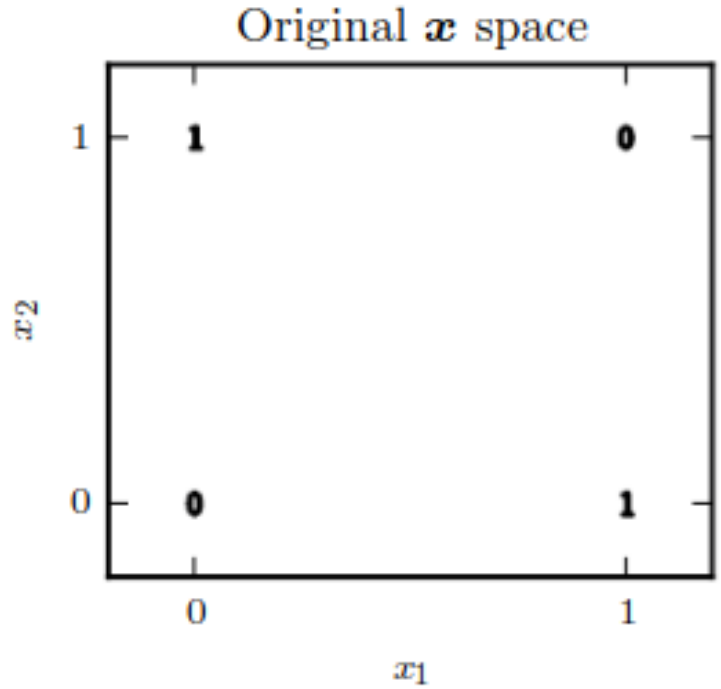


Figure: The bold numbers = the output of y at each point. A linear model cannot implement the XOR function. When $x_1 = 0$, the models output must increase as x_2 increases. When $x_1 = 1$, the models output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2. The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem.

Learning the XOR function

One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Specifically, we will introduce a simple MLP with one hidden layer containing two hidden units.

Learning the XOR function

The MLP we employ:

$$h_{w,b}(x) = W_2 \text{ReLU}(W_1 x + b_1) + b_2 \tag{26}$$

The hidden layer $z(x) := \text{ReLU}(W_1 x + b_1)$ is of dimension 2 and the output layer $W_2 \text{ReLU}(W_1 x + b_1) + b_2$ is of dimension 1. Here, $w = (W_1, W_2)$ and $b = (b_1, b_2)$ together collect the parameters of the neural network.

Learning the XOR function

$$h_{w,b}(x) = W_2 \text{ReLU}(W_1 x + b_1) + b_2 \tag{27}$$

The output layer is still just a linear regression model, but now it is applied to the hidden layer $z(x)$ rather than to x.

The network now contains two functions chained together,

$$f_1(x) := \text{ReLU}(W_1 x + b_1) \text{ and } f_2(z) := W_2 z + b_2 \tag{28}$$

with the complete model being $h_{w,b}(x) = f_2(f_1(x))$.

Learning the XOR function

We can now specify a solution to the XOR learning problem:

$$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad W_2 = \begin{bmatrix} 1 & -2 \end{bmatrix}, \quad b_2 = 0 \tag{29}$$

Learning the XOR function

We can now walk through the process of the neural network:

$$X^\top = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{30}$$

Learning the XOR function

After applying the first linear transformation,

$$W_1 X^\top = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix} \tag{31}$$

Next, we add the first bias vector b_1,

$$\begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix} \tag{32}$$

Learning the XOR function

We apply the activation function on each coordinate separately:

$$\begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{33}$$

Finally, we apply the second linear transformation and obtain:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \tag{34}$$

The neural network obtained the the correct answer for every example in the dataset.

Learning the XOR function

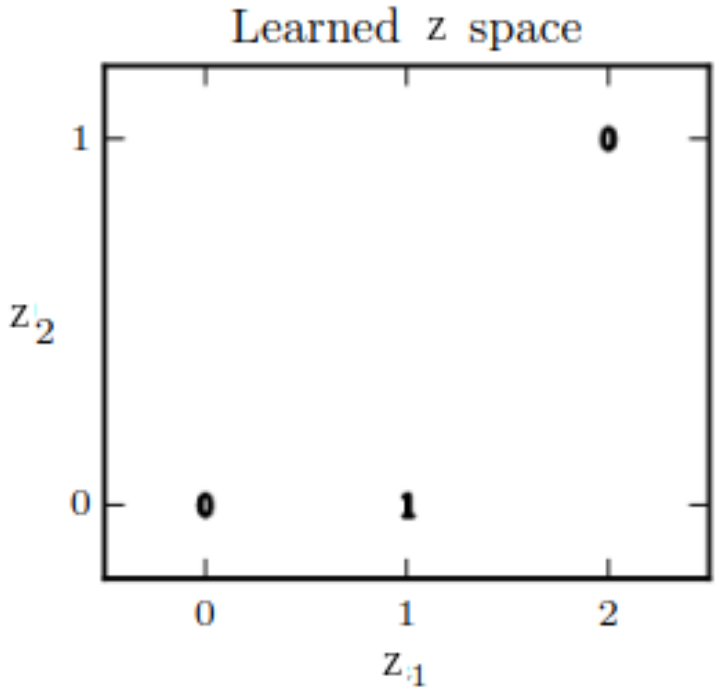


Figure: In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $x = (1, 0)$ and $x = (0, 1)$ to a single point in feature space, $z = (1, 0)$. In more realistic applications, learned representations can also help the model to generalize.

Learning the XOR function

In this example: we simply specified the solution, then showed that it obtained zero error.

In reality: billions of model parameters and thousands (or even billions) of training examples. So one cannot simply guess the solution as we did here.

Learning the XOR function

Instead, a gradient-based optimization algorithm can find parameters that produce very little error.

The solution we described to the XOR problem is at a global minimum of the error function, so gradient-based optimization could converge to this point.

Learning the XOR function

There are other equivalent solutions to the XOR problem that gradient based optimization methods could also find. The convergence point of these methods depend on the initial values of the parameters.

In practice, gradient based methods would usually not find clean, easily understood, integer-valued solutions like the one we presented here.