

Neural Network Learning

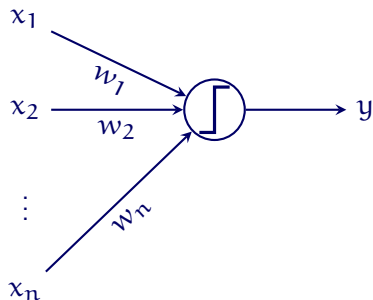
CS 114 Spring
February 7, 2020

Slides by Çağrı Çöltekin

Artificial neural networks (ANNs)

- ▶ Artificial neural networks are machine learning models *inspired* by the biological neural networks.
- ▶ Although some strong claims have been made about the link between the two, for our purposes, ANNs are just another statistical method in machine learning
- ▶ The founding blocks of ANNs are simple units (like biological neurons) that carry out simple calculations in parallel
- ▶ ANNs have many similarities to the linear models we discussed so far, but allow non-linearities that are often useful in practice
- ▶ The recent 'deep learning' methods are variations of ANN architectures.

The perceptron

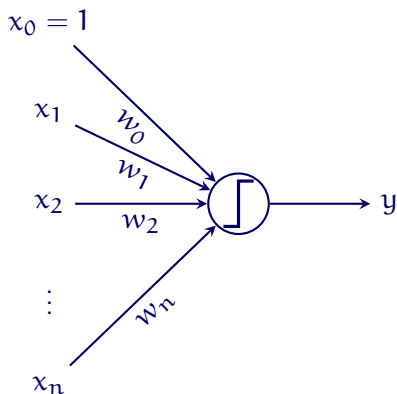


$$y = f \left(\sum_i^n w_i x_i \right)$$

where

$$f(x) = \begin{cases} +1 & \text{if } \sum_i^n w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

The perceptron



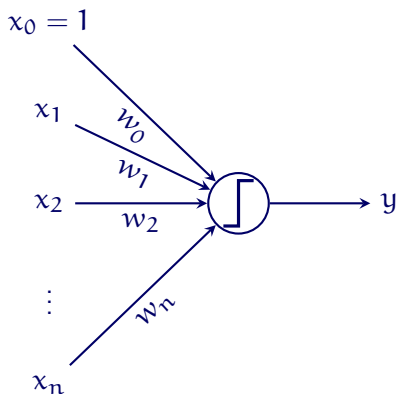
$$y = f \left(\sum_i^n w_i x_i \right)$$

where

$$f(x) = \begin{cases} +1 & \text{if } \sum_i^n w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

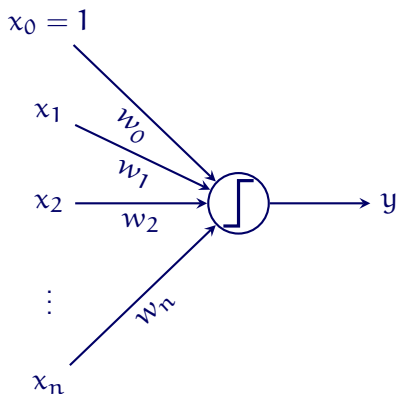
Similar to the *intercept* in linear models, an additional input x_0 which is always set to one is often used (called **bias** in ANN literature.)

The perceptron: in plain words



- ▶ Sum all input x_i weighted with corresponding weight w_i
- ▶ Pass it through a threshold function
- ▶ Classify the input a
 - positive if the perceptron fires (the sum is larger than 0),
 - negative otherwise

The perceptron: in plain words



- ▶ Sum all input x_i weighted with corresponding weight w_i
- ▶ Pass it through a threshold function
- ▶ Classify the input a
 - positive** if the perceptron fires (the sum is larger than 0),
 - negative** otherwise

The perceptron can solve *linearly separable* classification problems.

The perceptron algorithm

- ▶ For correctly classified examples, we do not update the parameters
- ▶ For misclassified example, we try to minimize

$$E(\mathbf{w}) = - \sum_i \mathbf{w} \mathbf{x}_i y_i$$

where i ranges over all misclassified examples.

- ▶ For each misclassified example, we update the weights

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \nabla E(\mathbf{w})$$

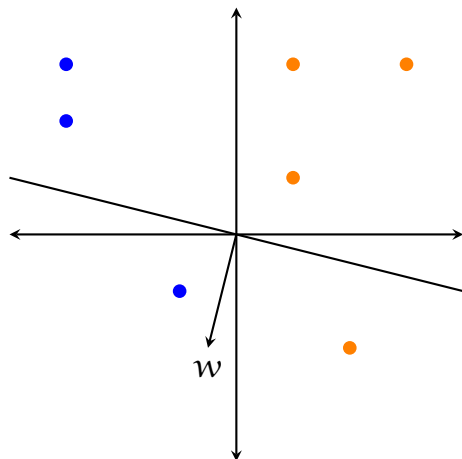
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}_i y_i$$

note that with every update the set of misclassified examples change.

The perceptron algorithm

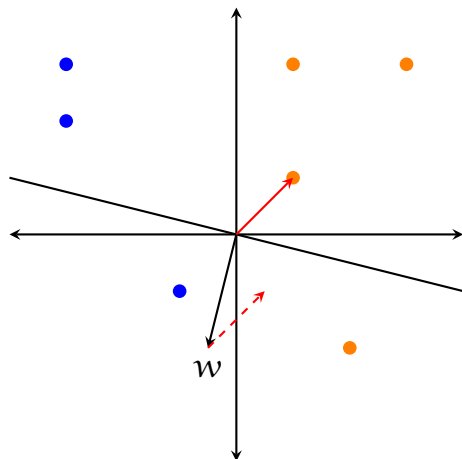
- ▶ The perceptron algorithm (eventually) converges to the global minimum if the classes are linearly separable.
- ▶ If the classes are not linearly separable, the perceptron algorithm will not stop
- ▶ We do not know whether the classes are linearly separable or not before the algorithm converges

Perceptron learning: demonstration



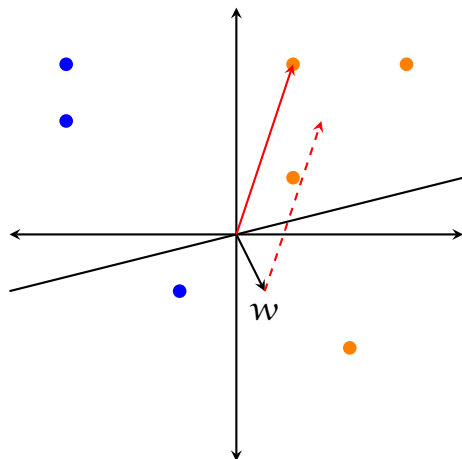
1. Randomly initialize \mathbf{w} the decision boundary is orthogonal to \mathbf{w}
2. Pick a misclassified example \mathbf{x}_i add it to \mathbf{w} .
3. Set $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$, repeat step 2 until convergence

Perceptron learning: demonstration



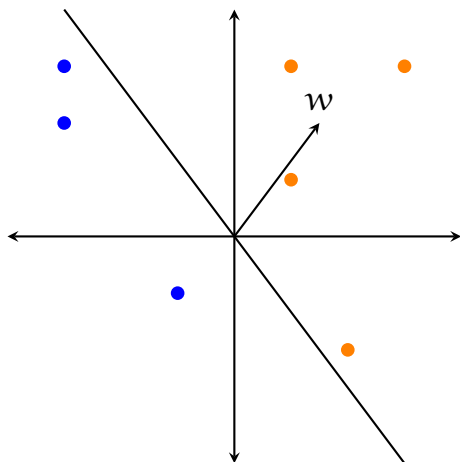
1. Randomly initialize w the decision boundary is orthogonal to w
2. Pick a misclassified example x_i add it to w .
3. Set $w \leftarrow w + y_i x_i$, repeat step 2 until convergence

Perceptron learning: demonstration



1. Randomly initialize \mathbf{w} the decision boundary is orthogonal to \mathbf{w}
2. Pick a misclassified example \mathbf{x}_i add it to \mathbf{w} .
3. Set $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$, repeat step 2 until convergence

Perceptron learning: demonstration



1. Randomly initialize w the decision boundary is orthogonal to w
2. Pick a misclassified example x_i add it to w .
3. Set $w \leftarrow w + y_i x_i$, repeat step 2 until convergence

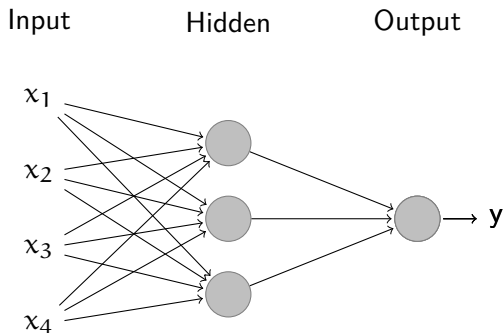
Perceptron: a bit of history

- ▶ The perceptron was developed in late 1950's and early 1960's (Rosenblatt 1958)
- ▶ It caused some excitement in many fields including computer science, artificial intelligence, cognitive science
- ▶ The excitement (and funding) died away in early 1970's (after the criticism by Minsky and Papert 1969)
- ▶ The main issue was the fact that perceptron cannot handle problems that are not linearly separable.

Multi-layer perceptron

- ▶ Multi-layer perceptron (MLP) is a fully connected **feed-forward** network consisting of perceptron-like units
- ▶ The units in an MLP use a continuous activation function, unlike threshold function used in perceptron
- ▶ The MLP can be trained using gradient-based methods
- ▶ The MLP can represent many interesting machine learning problems. It can be used for both regression and classification

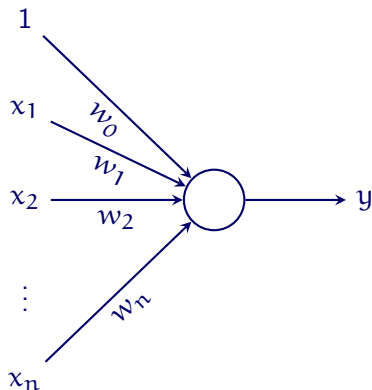
Multi-layer perceptron



Activation functions in MLP

- ▶ The activation functions in MLP are typically continuous (differentiable) functions
- ▶ For hidden units sigmoid functions (logistic sigmoid or tanh) are a common choice (more on this in a few weeks)
- ▶ The activation functions of the output units depends on the task
 - ▶ For regression, identity function
 - ▶ For binary classification, logistic sigmoid
 - ▶ For multi-class classification, softmax

Single neuron in an MLP



$$y = f \left(\sum_i^n w_i x_i \right)$$

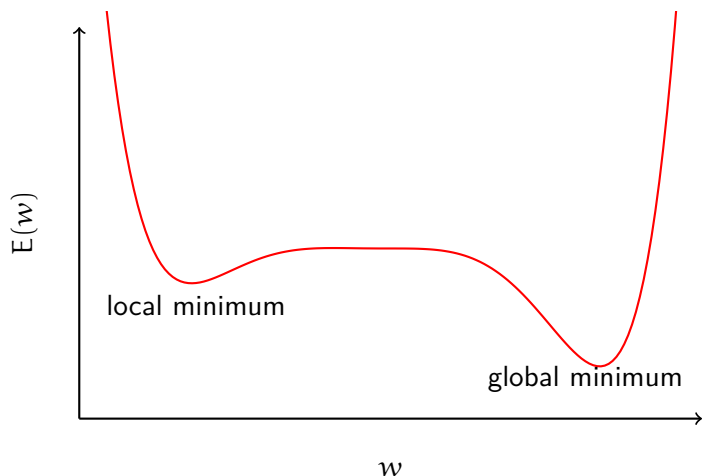
- ▶ With logistic sigmoid as the activation function

$$y = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + \dots + w_n x_n)}}$$

it is equivalent to the logistic regression

- ▶ We can now use gradient-descent to estimate the parameters

Finding minima of the error function



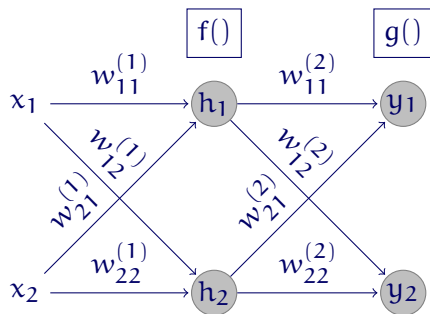
Gradient descent: a refresher

- ▶ The general idea is to approach a minimum of the error function in small steps

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$$

- ▶ ∇E is the gradient of the error function, it points to the direction of the maximum increase
- ▶ α is the learning rate
- ▶ The updates can be made in
 - batch** updates are performed for the complete training set
 - on-line** updates are performed for each training instance. This version is known as *stochastic gradient descent* (SGD)

MLP: a simple example

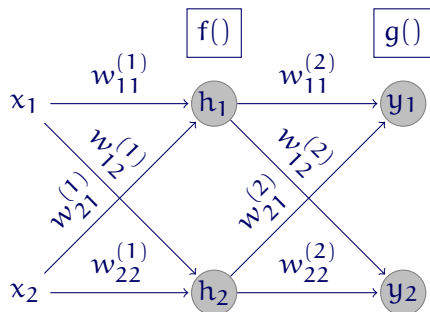


$$h_j = f \left(\sum_i w_{ij} x_i \right)$$

$$y_k = g \left(\sum_j w_{jk} h_j \right)$$

$$y_k = g \left(\sum_j w_{jk} f \left(\sum_i w_{ij} x_i \right) \right)$$

MLP: a simple example



- ▶ Alternatively, we can write the computations in matrix form

$$\mathbf{h} = f(W^{(1)}\mathbf{x})$$

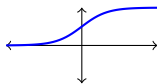
$$\begin{aligned} \mathbf{y} &= g(W^{(2)}\mathbf{h}) \\ &= g\left(W^{(2)}f(W^{(1)}\mathbf{x})\right) \end{aligned}$$

- ▶ This corresponds to a series of (non-linear) transformations of the input

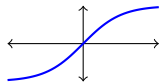
Activation functions

- ▶ Choice of activation functions depend on the application, and the type of unit
- ▶ For hidden units, common choices are sigmoid functions

Logistic $\sigma(z) = \frac{1}{1+e^z}$



Hyperbolic tangent $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



(We will later introduce a few others)

- ▶ For regression the common choice is a linear function, most typically the **identity** function $I(z) = z$
- ▶ For classification, either logistic function (for binary classification), or **softmax** (for multi-class)

$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Error functions in ANN training

- ▶ If we assume Gaussian noise, a natural choice is the minimizing the sum of squared error

$$E(\mathbf{w}) = \sum_i (y_i - \hat{y}_i)^2$$

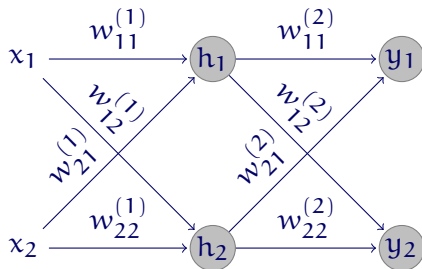
- ▶ For binary classification, we use *cross entropy*

$$E(\mathbf{w}) = - \sum_i y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

- ▶ Similarly, for multi-class classification, also cross entropy

$$E(\mathbf{w}) = - \sum_i \sum_k y_{i,k} \log \hat{y}_k$$

Learning in multi-layer networks: back propagation



- ▶ The final output of the network is computed by calculating the output each layer and passing it to the next (**forward propagation**)
- ▶ The weights are updated using a technique called **back propagation**
- ▶ Back-propagation algorithm makes use of chain rule of derivatives to efficiently propagate the error from output units to the input weights

Regularization in neural networks

- ▶ As in linear models we studied, we can use L1 and L2 regularization by adding a regularization term to the error function (known as weight decay). For example,

$$J(\mathbf{w}) = E(\mathbf{w}) + \|\mathbf{W}\|$$

- ▶ There are other ways to fight overfitting
 - ▶ With **early stopping**, one stops the training before it reaches to the smallest training error
 - ▶ With **dropout**, random units (with all of their connections) are dropped during training
 - ▶ Injecting noise at the output, as a way to (implicitly) model the noise in the target classes/values

How many layers, units

- ▶ A network with single hidden layer, is said to be a *universal approximator*: it can approximate any continuous function with arbitrary precision
- ▶ However, in practice multiple interconnected layers are useful and commonly used in modern ANN models
- ▶ The choice of layers, in general the architecture of the system, depends on the application

Another bit of history

- ▶ In 1980's ANNs became popular again
- ▶ One of the important developments that made this possible was the back propagation algorithm
- ▶ In 1990's the ANNs had again fallen 'out of fashion'. Mainly due to
 - ▶ From the engineering perspective: other, more successful algorithms (such as SVMs) performed generally better
 - ▶ From the cognitive science perspective: the fact that ANNs are complex systems that are difficult to interpret
- ▶ At present (after 2005 or so) they, once more, enjoy success stories and popularity with the name 'deep learning'
- ▶ We will study some aspects of the deep learning methods for the remainder of the course

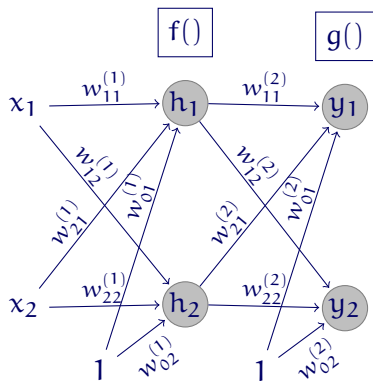
Deep learning

- During the last decade, a set of methods collectively known as *deep learning* became dominant
- They are (mostly) based on multi-layer neural networks
- They won many of the competitions against other ML methods (e.g., SVMs)
- The main premise is to learn useful features automatically: no need for feature engineering

Artificial neural networks

- ANNs are networks of units (neurons) each performing basic computation: calculate the weighted sum of the inputs, apply an *activation* function
- Combination of the units results in powerful learning machines
- The ANNs are inspired by biological neural networks, but they differ in quite a few ways
- The ANNs are also closely related to linear models

Feed-forward networks: the picture and the math

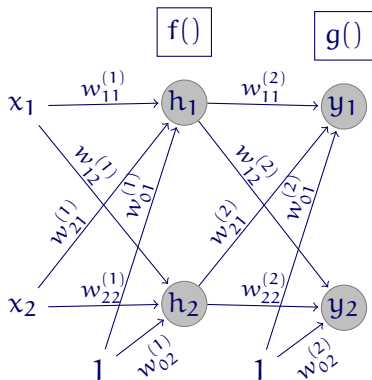


$$h_j = f \left(\sum_i w_{ij}^{(1)} x_i \right)$$

$$y_k = g \left(\sum_j w_{jk}^{(2)} h_j \right)$$

$$y_k = g \left(\sum_j w_{jk}^{(2)} f \left(\sum_i w_{ij}^{(1)} x_i \right) \right)$$

Feed-forward networks: with matrix/vector notation



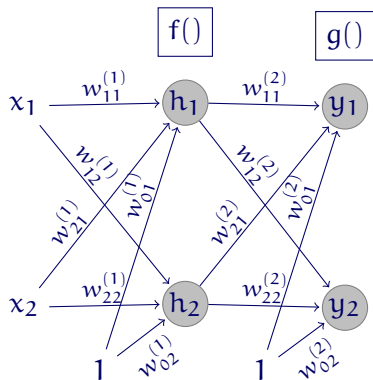
$$\mathbf{h} = f(\mathbf{W}^{(1)}\mathbf{x})$$

$$\mathbf{y} = g(\mathbf{W}^{(2)}\mathbf{h})$$

$$= g(\mathbf{W}^{(2)}f(\mathbf{W}^{(1)}\mathbf{x}))$$

- $f()$ and $g()$ are non-linear functions, such as *logistic sigmoid* or *tanh*

Feed-forward networks: with matrix/vector notation



$$\mathbf{h} = f(\mathbf{W}^{(1)}\mathbf{x})$$

$$\mathbf{y} = g(\mathbf{W}^{(2)}\mathbf{h})$$

$$= g(\mathbf{W}^{(2)}f(\mathbf{W}^{(1)}\mathbf{x}))$$

- $f()$ and $g()$ are non-linear functions, such as *logistic sigmoid* or *tanh*

Realizing that an ANN boils down to a series of matrix multiplications (linear transformations) and (non-linear) function applications is also essential for effectively using some of the libraries.

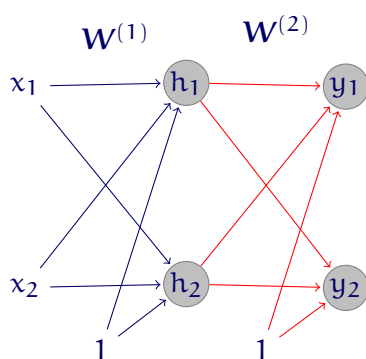
ANNs: activation functions

- Output layer activations depend on the problem
 - For *regression*, linear (e.g., identity function)
 $I(z) = z$
 - For *binary classification*, logistic sigmoid (called simply 'sigmoid' in the ANN literature)
 $\sigma(z) = \frac{1}{1+e^z}$
 - For *multi-class classification* softmax
 $\text{softmax}_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$
- For hidden layers, *logistic* or *tanh* used to be the norm in the earlier models (but more on this later in this lecture)

Learning in ANNs

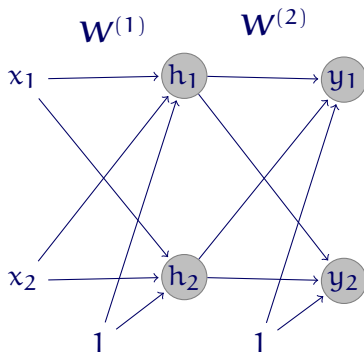
- ANNs implement complex functions: we need to use iterative optimization methods (e.g., gradient descent) to train them
- Typically error functions for ANNs are not convex, gradient descent will find a local minimum
- Optimization requires updating multiple layers of weights
- Assigning credit (or blame) to each weight during learning is not trivial
- An effective solution to the last problem is the **backpropagation** algorithm

Learning in ANNs: backpropagation



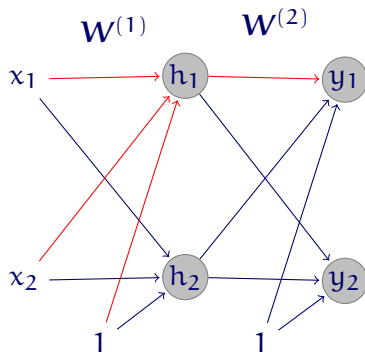
- Updating weights $W^{(2)}$ are easy: we can use gradient descent directly

Learning in ANNs: backpropagation



- Updating weights $W^{(2)}$ are easy: we can use gradient descent directly
- But the contribution of weights $W^{(1)}$ to the error is indirect

Learning in ANNs: backpropagation



- Updating weights $W^{(2)}$ are easy: we can use gradient descent directly
- But the contribution of weights $W^{(1)}$ to the error is indirect
- Backpropagation algorithm efficiently assigns credit to weights in the earlier layers

Where do non-linearities come from?

(a short divergence)

In a linear model, $y = w_0 + w_1x_1 + \dots + w_kx_k$

- The outcome is *linearly-related* to the predictors
- The effects of the inputs are *additive*

This is not always the case:

- Some predictors affect the outcome in a non-linear way
 - The effect may be strong or positive only in a certain range of the variable (e.g., age)
 - Some effects are periodic (e.g., many measures of time)
- Some predictors interact
 - ‘*not bad*’ is not ‘*not*’ + ‘*bad*’ (e.g., for sentiment analysis)

Dealing with non-linearities (1)

(a short divergence, contd.)

Non-linear transformations, kernels, feature engineering

- Note that both

$$y = w_0 + w_1x_1 + w_2x_1^2$$

and

$$y = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2$$

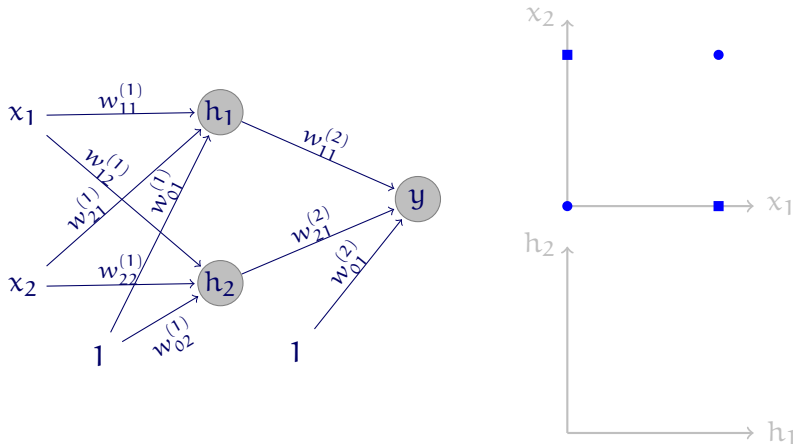
are still linear in weights.

- + The objective function is still convex, we have a global minimum
- Requires careful feature selection/engineering
- Often becomes slow to train

Dealing with non-linearities (2)

(a short divergence, contd.)

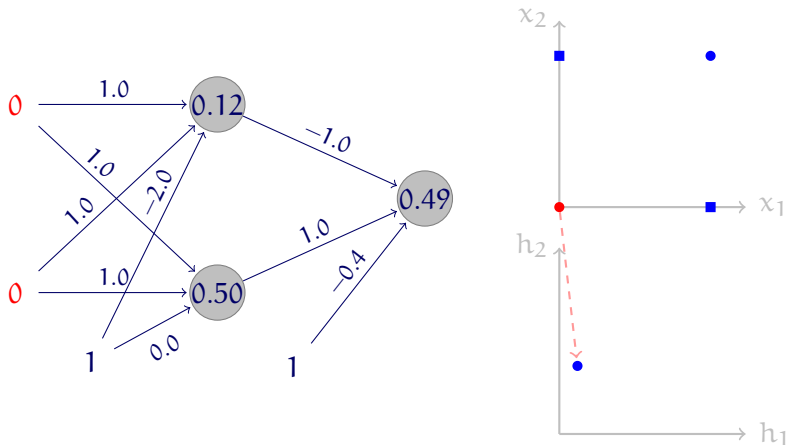
Multi-layer networks (a solution to the XOR problem)



Dealing with non-linearities (2)

(a short divergence, contd.)

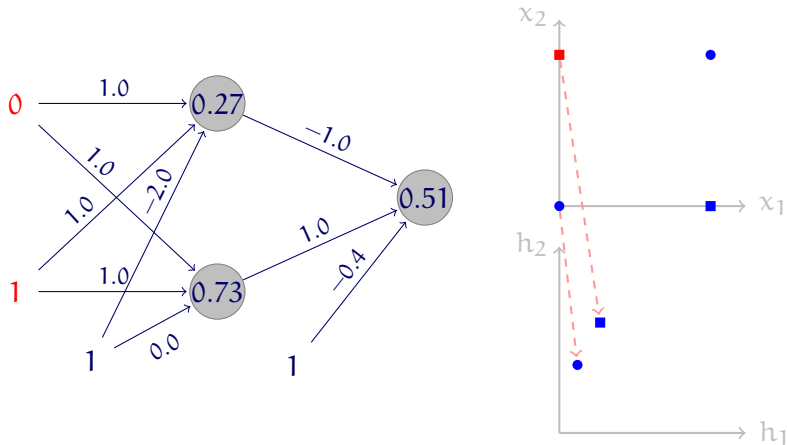
Multi-layer networks (a solution to the XOR problem)



Dealing with non-linearities (2)

(a short divergence, contd.)

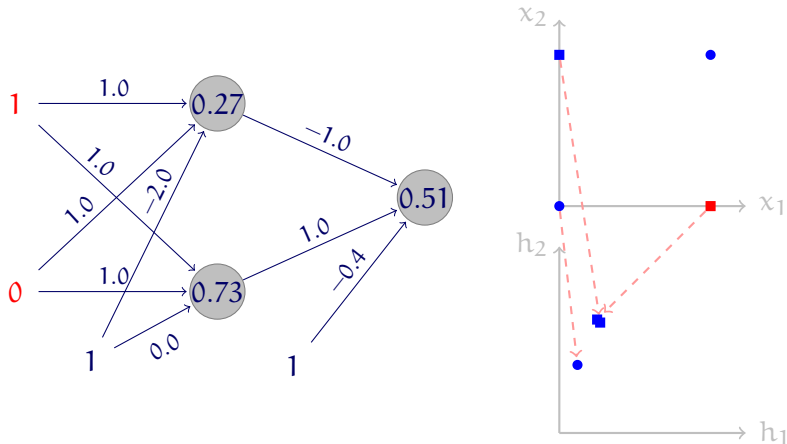
Multi-layer networks (a solution to the XOR problem)



Dealing with non-linearities (2)

(a short divergence, contd.)

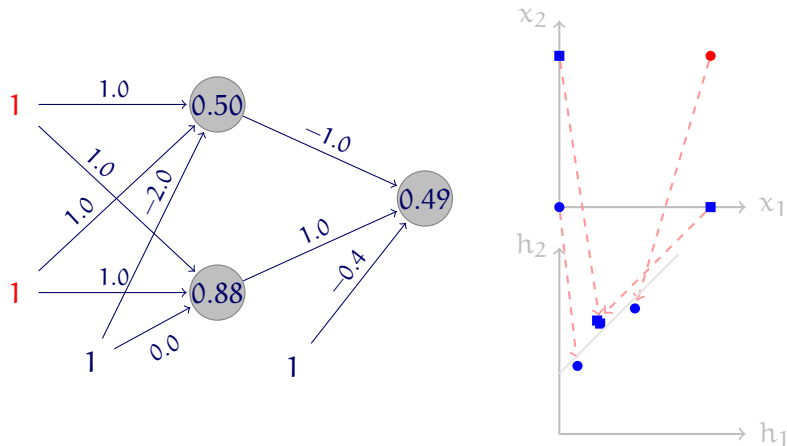
Multi-layer networks (a solution to the XOR problem)



Dealing with non-linearities (2)

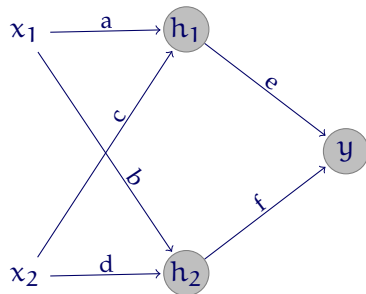
(a short divergence, contd.)

Multi-layer networks (a solution to the XOR problem)



Non-linear activation functions are necessary

Without non-linear activation functions, the ANN is equivalent to a linear model.



$$h_1 = ax_1 + cx_2$$

$$h_2 = bx_1 + dx_2$$

$$y = eh_1 + fh_2$$

$$= (ea + fb)x_1 + (ec + fd)x_2$$

y is still a linear function of x_i

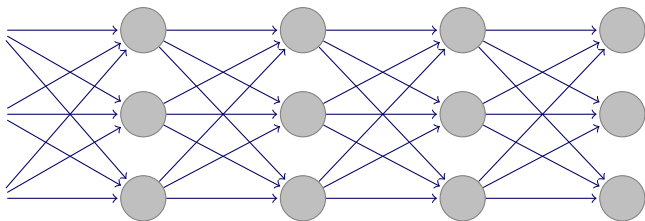
Deep learning

- ‘Deep learning’ refers to a set of ML methods, mainly based on ANNs
- The major difference from what we have seen so far is the ‘deep’ architectures with many hidden layers
- The deep ANNs have been successful many areas of ML, since about 2006, winning competitions on most reference data sets
- The main premise is learning useful features automatically: no need for feature engineering!

Deep networks: why do we need them?

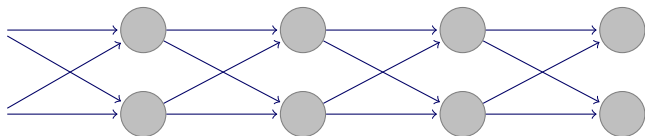
- We noted earlier that (large) MLPs with a single hidden layer are universal approximators: they can approximate any continuous function with arbitrary precision. However,
 - theoretical results do not limit the number of units: it may require number of units exponential in the size of the data
 - being able to ‘represent’ does not mean being able to ‘learn’
 - some families of functions can efficiently be approximated by deep networks, but require much larger networks if depth is smaller

Deep networks: many hidden layers



- Deep networks are simply feed-forward networks with many layers
- The additional hidden layers bring some advantages, as well as some difficulties
- The number of (effective) layers may exceed hundreds in practice

Training deep networks: problems



- A major problem is unstable gradients, they are backpropagated to earlier layers. The gradients may
 - **vanish**, learning becomes too slow
 - **explode**, convergence becomes difficult (or impossible)
- In general, training deep networks is (still) difficult. Models are sensitive to initialization, many parameters, numeric stability issues, ...

Dealing with unstable gradients

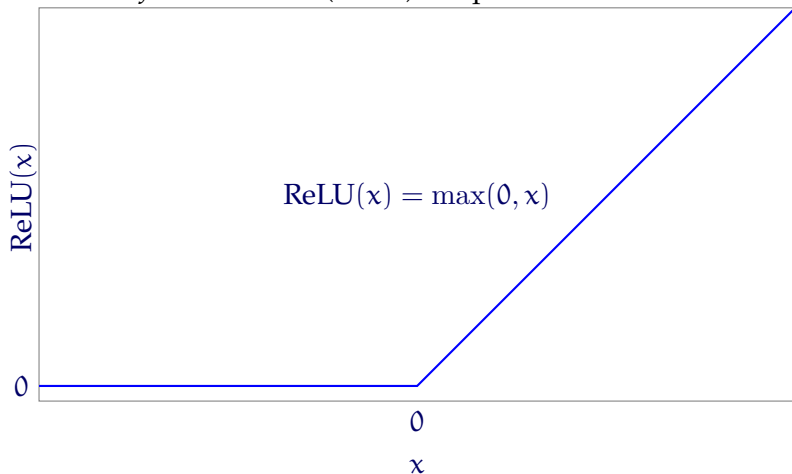
- Standard regularization methods (L1 or L2) help avoiding exploding gradients
- Another popular approach against exploding gradients is gradient *clipping* or *scaling*
- Some activation functions are more prone to unstable gradients. For example, derivative of the logistic sigmoid is close to zero for most part of its input space
- For vanishing gradients, special regularization schemes that encourage information flow, or special architectures are suggested

Activation functions in deep networks

- The choice of activation function for the output units are similar to MLP
 - Linear functions for regression
 - Logistic sigmoid for binary classification
 - Softmax for multi-class classification
- Instead of continuous sigmoid functions, piecewise linear functions are more popular
 - + Fast computation
 - + Smaller chance for vanishing/exploding gradients
 - + 'Universal approximation' is still possible
 - Not suitable with certain architectures

ReLU

Rectified linear unit (ReLU) is a piecewise function



ReLU is (currently) the most common activation function in feed-forward deep networks.

Common/popular models in deep learning

- Convolutional networks: for detecting local patterns
- Recurrent neural networks: for sequence learning
- Autoencoders/decoders: unsupervised methods using (deep) neural networks
- Combining different types of networks are often possible
- We will cover these three network types next