# Statistical Natural Language Processing

## Artificial Neural networks: an introduction

Çağrı Çöltekin

University of Tübingen
Seminar für Sprachwissenschaft

Summer Semester 2019

# Artificial neural networks

- Artificial neural networks (ANNs) are machine learning models inspired by biological neural networks
- ANNs are powerful non-linear models
- Power comes with a price: there are no guarantees of finding the global minimum of the error function
- ANNs have been used in ML, AI, Cognitive science since 1950's – with some ups and downs
- Currently they are the driving force behind the popular '*deep learning*' methods

# The biological neuron

(showing a picture of a real neuron is mandatory in every ANN lecture)

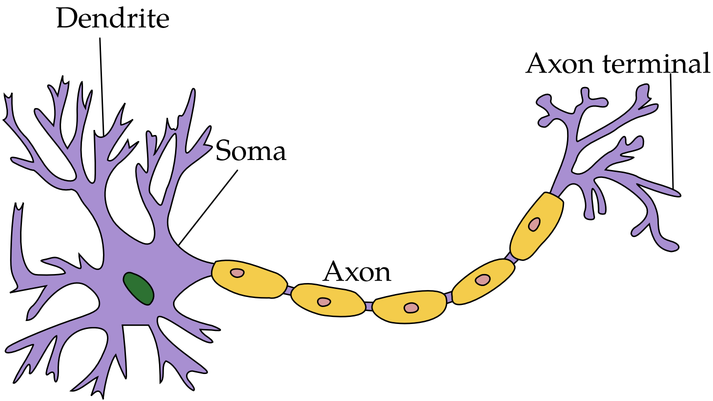



*Image source: Wikipedia

# Artificial and biological neural networks

- ANNs are *inspired* by biological neural networks
- Similar to biological networks, ANNs are made of many simple processing units
- Despite the similarities, there are many differences: ANNs do not mimic biological networks
- ANNs are practical statistical machine learning methods

# Recap: the perceptron

$$y = f\left(\sum_{j}^{m} w_j x_j\right)$$

where

$$f(x) = \begin{cases} +1 & \text{if} \quad \boldsymbol{wx} > 0 \\ -1 & \text{otherwise} \end{cases}$$

In ANN-speak $f(\cdot)$ is called an *activation function*.

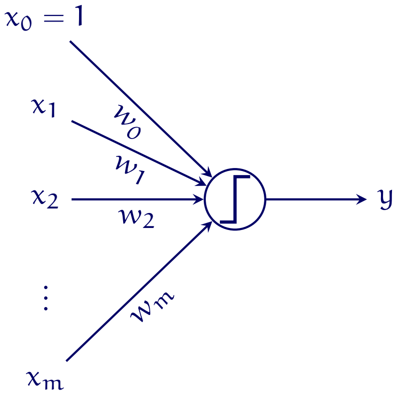
# Recap: the perceptron

$$y = f\left(\sum_{j}^{m} w_j x_j\right)$$

where

$$f(x) = \begin{cases} +1 & \text{if} \quad \boldsymbol{wx} > 0 \\ -1 & \text{otherwise} \end{cases}$$

In ANN-speak $f(\cdot)$ is called an *activation function*.



# Recap: logistic regression

$$P(y) = f\left(\sum_{j}^{m} w_j x_j\right)$$

where

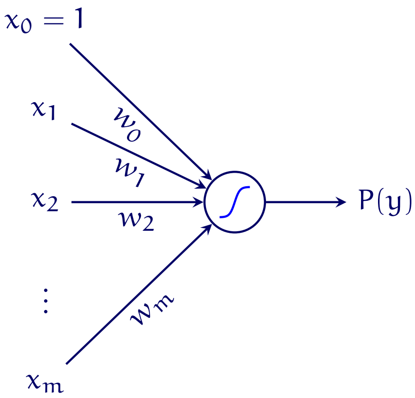
$$f(x) = \frac{1}{1 + e^{-wx}}$$

# Recap: logistic regression

$$P(y) = f\left(\sum_{j}^{m} w_j x_j\right)$$

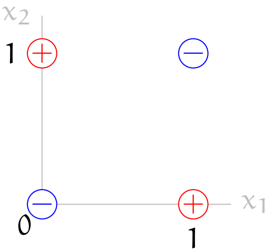
where

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



# Linear separability

- A classification problem is said to be *linearly separable* if one can find a linear discriminator
- A well-known counter example is the logical XOR problem




There is no line that can separate positive and negative classes.

# Can a linear classifier learn the XOR problem?

# Can a linear classifier learn the XOR problem?

- We can use non-linear basis functions

$$w_0 + w_1 x_1 + w_2 x_2 + w_3 \phi(x_1, x_2)$$

  is still linear in $\boldsymbol{w}$ for any choice of $\phi(\cdot)$

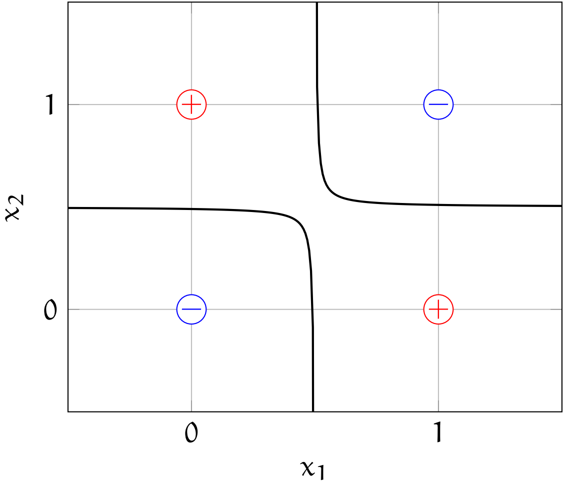
- For example, adding the product $x_1 x_2$ as an additional feature would allow a solution like: $x_1 + x_2 - 2x_1 x_2$

| $x_1$ | $x_2$ | $x_1 + x_2 - 2x_1 x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

- Choosing proper basis functions like $x_1 x_2$ is called *feature engineering*

# Non-linear basis functions

## solution in the original input space



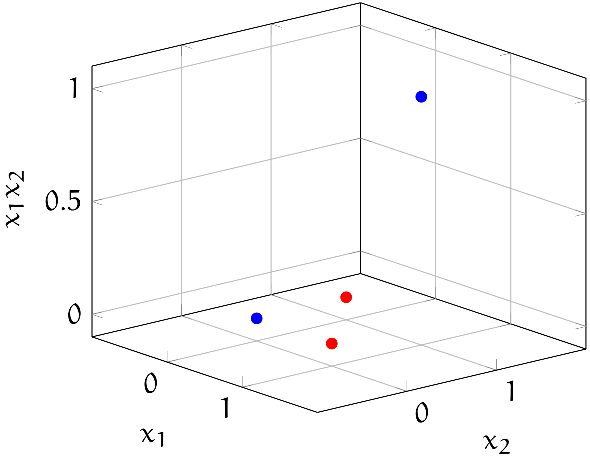

The solution to

$$x_1 + x_2 - 2x_1x_2 - 0.5 = 0$$

is a (non-linear) discriminant that solves the problem

# Non-linear basis functions

## solution in the 3D input space




- The additional basis function maps the problem into 3D
- In the new, mapped space, the points are linearly separable

# Where do non-linearities come from?

## non-linearities are abundant in nature, it is not only the XOR problem

In a linear model, $y = w_0 + w_1 x_1 + \ldots + w_k x_k$

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

This is not always the case:

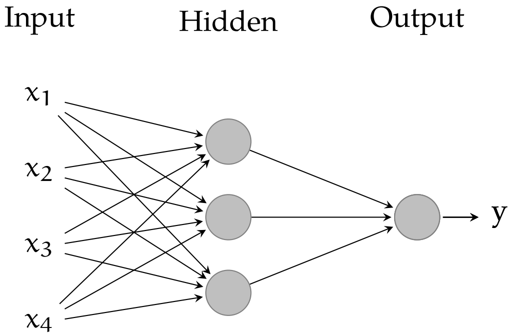
- Some predictors affect the outcome in a non-linear way
  - The effect may be strong or positive only in a certain range of the variable (e.g., reaction time change by age)
  - Some effects are periodic (e.g., many measures of time)
- Some predictors interact
  '*not bad*' is not '*not*' + '*bad*' (e.g.,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

# Multi-layer perceptron

- The simplest modern ANN architecture is called multi-layer perceptron (MLP)
- The MLP is a *fully connected*, *feed-forward* network consisting of perceptron-like units
- Unlike perceptron, the units in an MLP use a continuous activation function
- 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

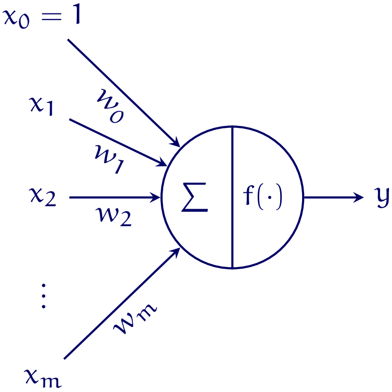
# Multi-layer perceptron

## the picture




Each unit takes a weighted sum of their input, and applies a (non-linear) *activation function*.

# Artificial neurons




- The unit calculates a weighted sum of the inputs

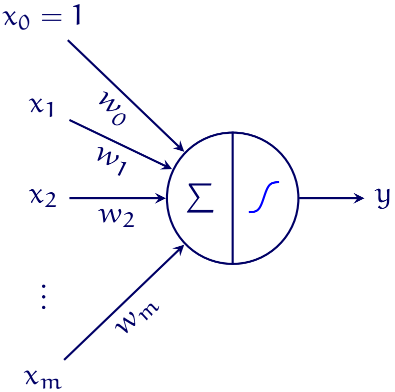
$$\sum_{j}^{m} w_j x_j = \boldsymbol{w}\boldsymbol{x}$$

- Result is a linear transformation
- Then the unit applies a non-linear activation function $f(\cdot)$
- Output of the unit is

$$y = f(\boldsymbol{w}\boldsymbol{x})$$

# Artificial neurons

## an example




- A common activation function is *logistic sigmoid* function

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

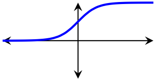
- The output of the network becomes

$$y = \frac{1}{1 + e^{-wx}}$$

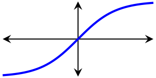
# Activation functions in ANNs

## hidden units

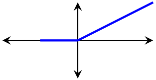
- The activation functions in MLP are typically continuous (differentiable) functions
- For hidden units common choices are



Sigmoid (logistic) $\frac{1}{1+e^{x}}$



Hyperbolic tangent (tanh) $\frac{e^{2x}-1}{e^{2x}+1}$



Rectified linear unit (relu) $\max(0, x)$

# Activation functions in ANNs

## output units

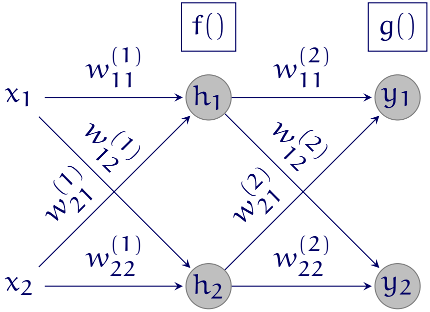
- The activation functions of the output units depends on the task. Common choices are
  - For regression, identity function
  - For binary classification, logistic sigmoid

$$P(y = 1 \mid x) = \frac{1}{1 + e^{-wx}} = \frac{e^{wx}}{1 + e^{-wx}}$$

  - For multi-class classification, softmax

$$P(y = k \mid x) = \frac{e^{w_k x}}{\sum_j e^{w_j x}}$$

# MLP: a simple example

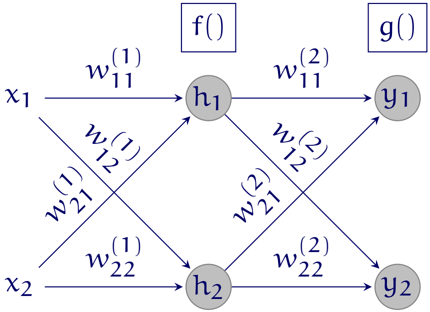



$$h_j = f\left(\sum_i w_{ij}^{(1)} x_i\right)$$

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

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

# MLP: a simple example




- Alternatively, we can write the computations in matrix form

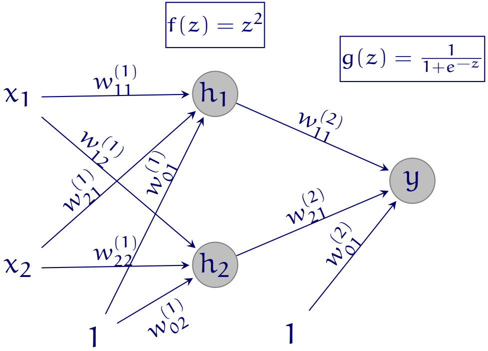
$$\mathbf{h} = f(W^{(1)}\mathbf{x})$$

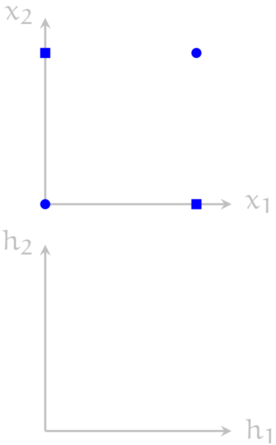
$$\begin{aligned}\mathbf{y} &= g(W^{(2)}\mathbf{h}) \\ &= g\left(W^{(2)}f(W^{(1)}\mathbf{x})\right)\end{aligned}$$

- This corresponds to a series of transformations followed by elementwise (non-linear) function applications

# Solving non-linear problems with ANNs

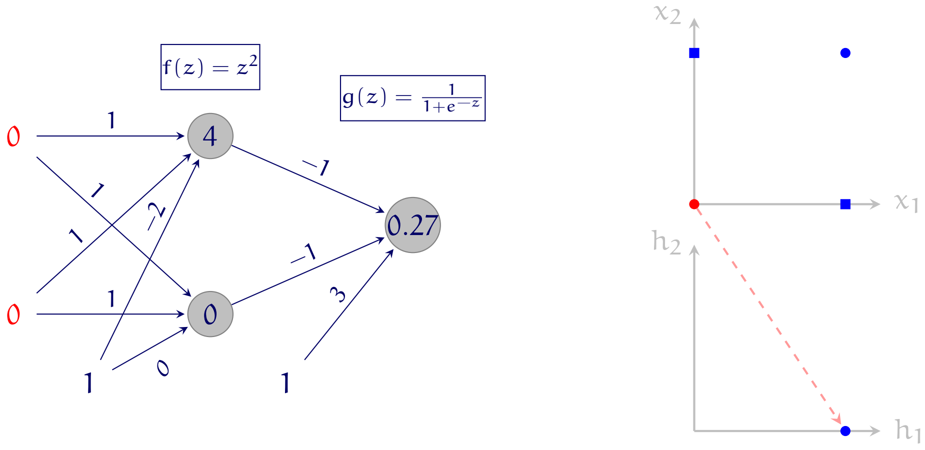
## a solution to XOR problem



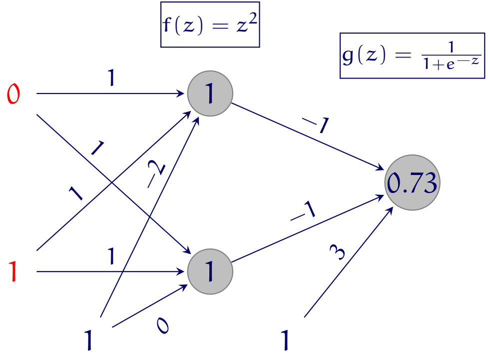
# Solving non-linear problems with ANNs

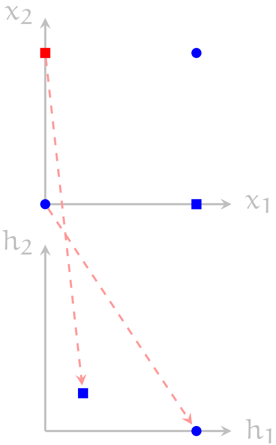
## a solution to XOR problem



# Solving non-linear problems with ANNs

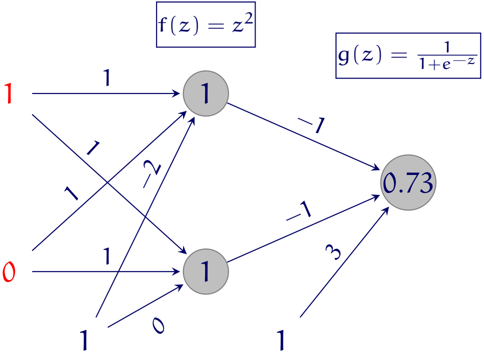
## a solution to XOR problem

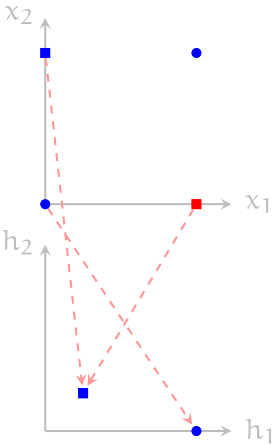


# Solving non-linear problems with ANNs

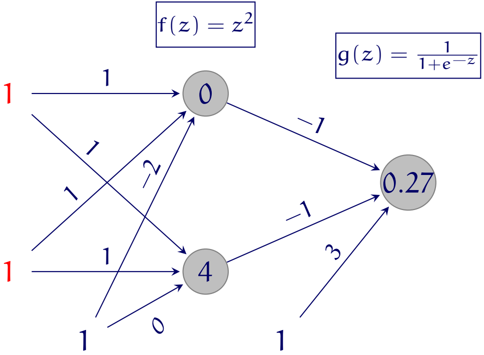
## a solution to XOR problem

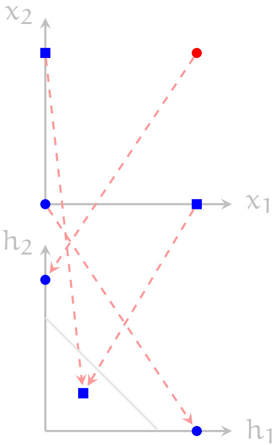


# Solving non-linear problems with ANNs

## a solution to XOR problem

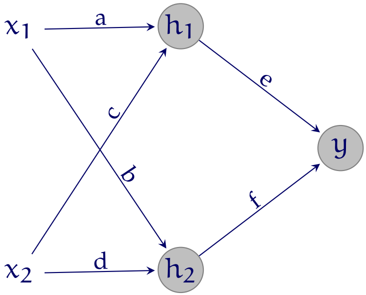



Is this different from non-linear basis functions?

# Non-linear activation functions are necessary

Without non-linear activation functions, an ANN with any number of layers is equivalent to a linear model.




$$
\begin{aligned}
h_1 &= ax_1 + cx_2 \\
h_2 &= bx_1 + dx_2 \\
y &= eh_1 + fh_2 \\
&= (ea + fb)x_1 + (ec + fd)x_2
\end{aligned}
$$

$y$ is still a linear function of $x_i$

# Gradient descent: a refresher

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

$$\boldsymbol{w} \leftarrow w - \eta \nabla J(\boldsymbol{w})$$

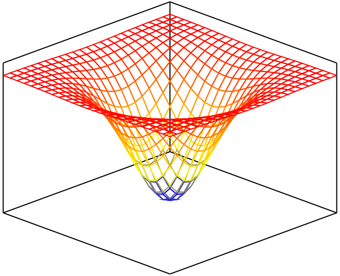
  - $\nabla J$ is the gradient of the loss function, it points to the direction of the maximum increase
  - $\eta$ is the learning rate
- The updates can be performed

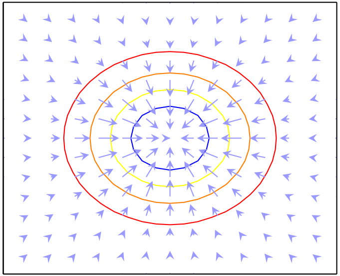
batch: for the complete training set

on-line: after every training instance
– this is known as *stochastic gradient descent* (SGD)

mini-batch: after small fixed-sized batches

# Gradient descent: the picture

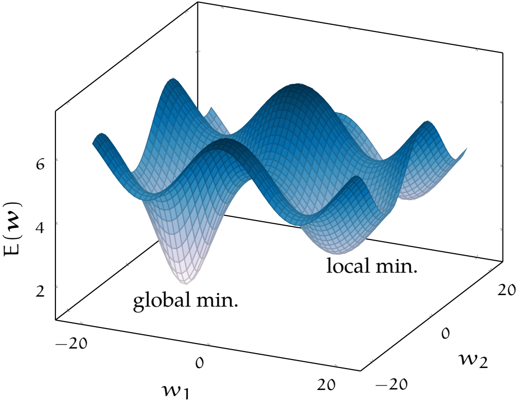




$$\nabla f(x_1, \ldots, x_n) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

A function is *convex* if there is only one (global) minimum.

# Global and local minima



# Error functions in ANN training

## depend on the task

- For regression, a natural choice is the minimizing the sum of squared error

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

- For binary classification, we use *cross entropy*

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

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

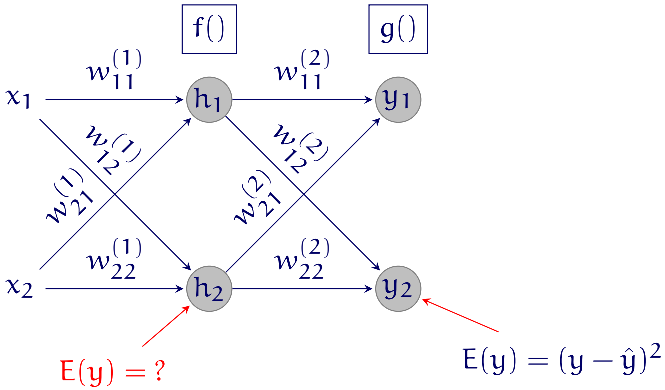
$$E(w) = -\sum_i \sum_k y_{i,k} \log \hat{y}_k$$

In practice, the ANN loss functions will not be convex.

# Learning in ANNs

- ANNs implement complex functions: we need to use 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

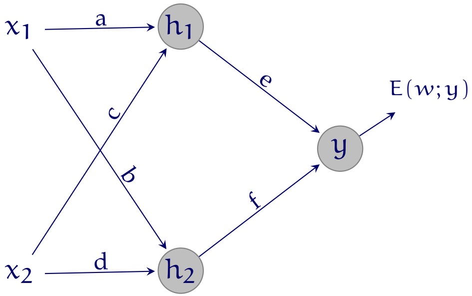
# Learning in multi-layer networks: the problem




We want a way to update non-final weights based on final error.

# Calculating gradient on a neural network

(with some simplification)



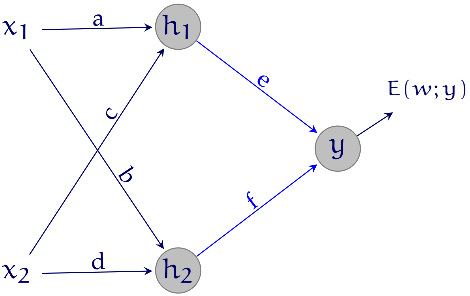

- We need to calculate the gradient:

$$\nabla E = \left( \frac{\partial E}{\partial a}, \frac{\partial E}{\partial b}, \frac{\partial E}{\partial c}, \frac{\partial E}{\partial d}, \frac{\partial E}{\partial e}, \frac{\partial E}{\partial f} \right)$$

we can use gradient descent directly

# Calculating gradient on a neural network

(with some simplification)




- We need to calculate the gradient:

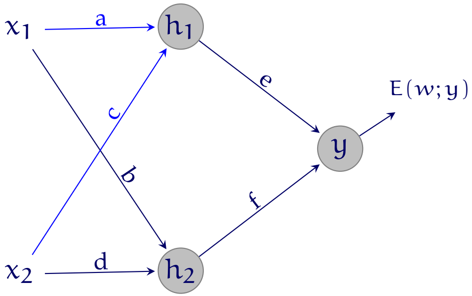
$$\nabla E = \left( \frac{\partial E}{\partial a}, \frac{\partial E}{\partial b}, \frac{\partial E}{\partial c}, \frac{\partial E}{\partial d}, \frac{\partial E}{\partial e}, \frac{\partial E}{\partial f} \right)$$

  we can use gradient descent directly

- $\frac{\partial E}{\partial e}$ and $\frac{\partial E}{\partial f}$ is easy, they do not depend on other variables

# Calculating gradient on a neural network

(with some simplification)



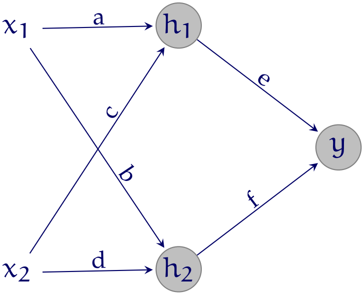

- We need to calculate the gradient:

$$\nabla E = \left( \frac{\partial E}{\partial a}, \frac{\partial E}{\partial b}, \frac{\partial E}{\partial c}, \frac{\partial E}{\partial d}, \frac{\partial E}{\partial e}, \frac{\partial E}{\partial f} \right)$$

  we can use gradient descent directly
- $\frac{\partial E}{\partial e}$ and $\frac{\partial E}{\partial f}$ is easy, they do not depend on other variables
- We factor others using chain rule

$$\frac{\partial E}{\partial a} = \frac{\partial h1}{\partial a} \frac{\partial E}{\partial h1} \quad \text{and} \quad \frac{\partial E}{\partial c} = \frac{\partial h1}{\partial c} \frac{\partial E}{\partial h1}$$

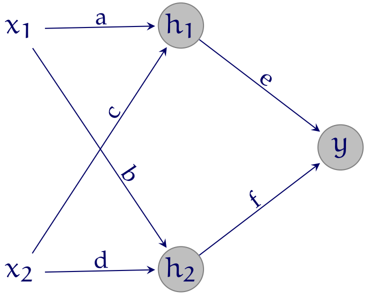
# Backpropagation




- So far, it is just math

$$\frac{\partial E}{\partial a} = \frac{\partial h1}{\partial a}\frac{\partial E}{\partial h1} \quad \text{and} \quad \frac{\partial E}{\partial c} = \frac{\partial h1}{\partial c}\frac{\partial E}{\partial h1}$$

# Backpropagation

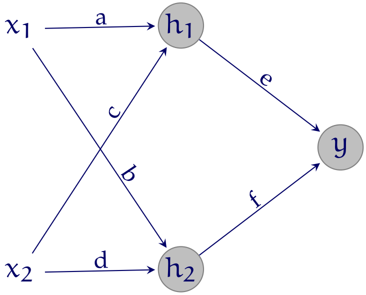



- So far, it is just math

$$\frac{\partial E}{\partial a} = \frac{\partial h1}{\partial a}\frac{\partial E}{\partial h1} \quad \text{and} \quad \frac{\partial E}{\partial c} = \frac{\partial h1}{\partial c}\frac{\partial E}{\partial h1}$$

- But a naive implementation does many repeated calculations

# Backpropagation

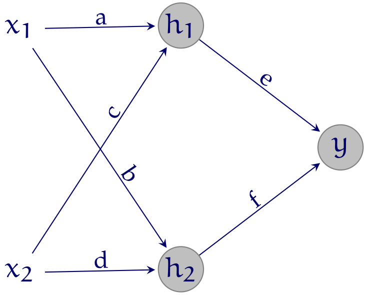



- So far, it is just math

$$\frac{\partial E}{\partial a} = \frac{\partial h1}{\partial a}\frac{\partial E}{\partial h1} \quad \text{and} \quad \frac{\partial E}{\partial c} = \frac{\partial h1}{\partial c}\frac{\partial E}{\partial h1}$$

- But a naive implementation does many repeated calculations
- Backpropagation is an efficient (dynamic programming) algorithm that avoids repeated calculations

# Backpropagation



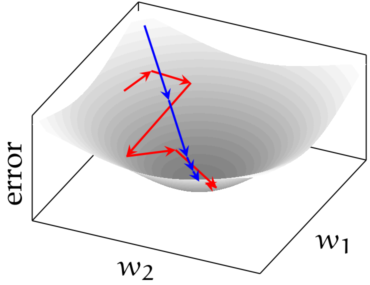

- So far, it is just math

$$\frac{\partial E}{\partial a} = \frac{\partial h1}{\partial a}\frac{\partial E}{\partial h1} \quad \text{and} \quad \frac{\partial E}{\partial c} = \frac{\partial h1}{\partial c}\frac{\partial E}{\partial h1}$$

- But a naive implementation does many repeated calculations
- Backpropagation is an efficient (dynamic programming) algorithm that avoids repeated calculations
- Backpropagation works for any *computation graph* without cycles

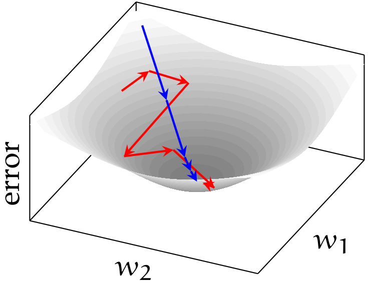
# Stochastic gradient descent

- Standard (batch) gradient descent is computationally expensive: it updates weight at every *epoch*
- Stochastic gradient descent (SGD) updates weights for every training instance
- SGD may take more steps, but converges to the same solution



# Stochastic gradient descent

- Standard (batch) gradient descent is computationally expensive: it updates weight at every *epoch*
- Stochastic gradient descent (SGD) updates weights for every training instance
- SGD may take more steps, but converges to the same solution




- In practice a *mini-batch* is more common
- Correct *batch size* is not only about efficiency, it also affects accuracy

# Preventing overfitting in neural networks

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

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

- 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

# Adapting learning rate

- The choice of learning rate $\eta$ is important

too small: slow convergence

too big: overshooting - may fluctuate around the minimum, or even jump away

- The idea is to adapt the learning rate during learning
- A common trick is adding a momentum:
  if we move in the same direction a long time accelerate

$$\Delta w_{ij}(t) = \eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)$$

- There are many adaptive optimization algorithms:
  Adagrad, Adadelta, RMSprop, Adam, ...

# 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

# A bit of history

1950-60 ANNs (perceptron) became popular:
lots of excitement in AI, cognitive science

1970s Not much interest

- criticism on perceptron: linear separability

1980s ANNs became popular again

- backpropagation algorithm
- multi-layer networks

1990s ANNs had again fallen ‘out of fashion’

- Engineering: other algorithms (such as SVMs) performed generally better
- From the cognitive science perspective: ANNs are difficult to interpret

present ANNs (again) enjoy a renewed popularity with the name ‘deep learning’

# Summary

- ANNs are powerful non-linear learners
  - based on some inspiration from biological NNs
  - using many simple processing units
  - built on linear models (logistic regression)
- For non-linear problems we need non-linear activation functions, and at least one hidden layer
- ANNs can be used for both regression and classification
- ANN loss functions are not convex, what we find is a local minimum
- They (typically) are trained with *backpropagation* algorithm

# Summary

- ANNs are powerful non-linear learners
  - based on some inspiration from biological NNs
  - using many simple processing units
  - built on linear models (logistic regression)
- For non-linear problems we need non-linear activation functions, and at least one hidden layer
- ANNs can be used for both regression and classification
- ANN loss functions are not convex, what we find is a local minimum
- They (typically) are trained with *backpropagation* algorithm

Next:

Mon/Fri Unsupervised learning

# Additional reading, references, credits

- Third edition (draft) of Jurafsky and Martin, has a new chapter on neural networks
- Hastie, Tibshirani, and Friedman (2009, ch.11) also includes an accessible introduction
- For a reivew of use of ANNs in NLP, including more advanced topics, see Goldberg 2016

# Additional reading, references, credits (cont.)


Goldberg, Yoav (2016). "A primer on neural network models for natural language processing". In: *Journal of Artificial Intelligence Research* 57, pp. 345–420.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer series in statistics. Springer-Verlag New York. ISBN: 9780387848587. URL: http://web.stanford.edu/~hastie/ElemStatLearn/.

Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.