Machine Learning for Computational Linguistics

Deep neural networks

Çağrı Çöltekin

University of Tübingen
Seminar für Sprachwissenschaft

June 21, 2016

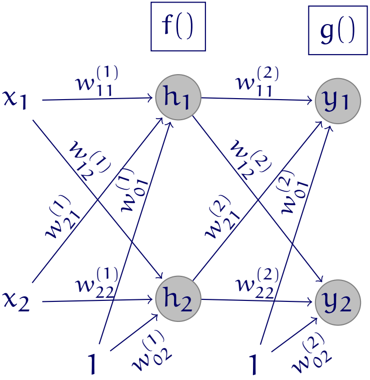
Deep learning

- 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

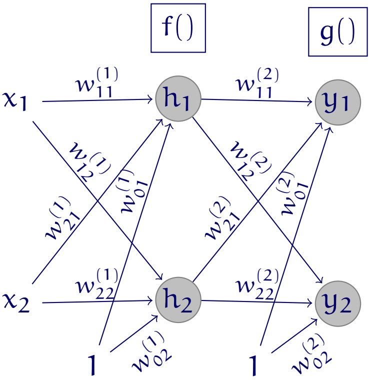


$$h_j = f\left(\sum_i w_{ij}^{(1)} x_i\right)$$

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

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

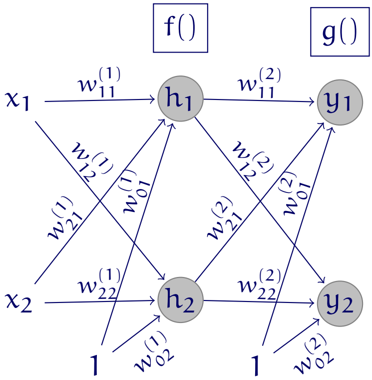
Feed-forward networks: with matrix/vector notation



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

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

Feed-forward networks: with matrix/vector notation



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

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

- 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.

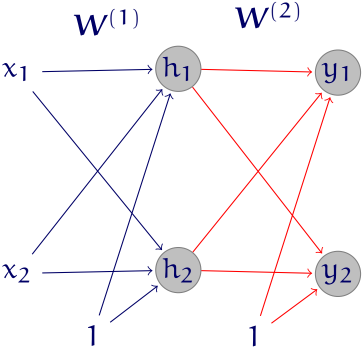
ANNs: activation functions

- 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
    $\mathtt{softmax}_j(z) = \frac{e^{z_j}}{\sum_k e^{z}_k}$
- 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

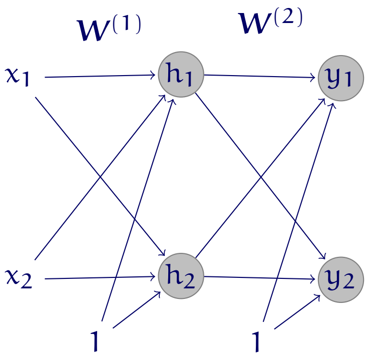
- ANNs implement complex functions: we need to use iterative optimization methods (e.g., gradient descent) to train them
- Typically error functions for ANNs are not convex, gradient descent will find a local minimum
- Optimization requires updating multiple layers of weights
- Assigning credit (or blame)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

Learning in ANNs: backpropagation



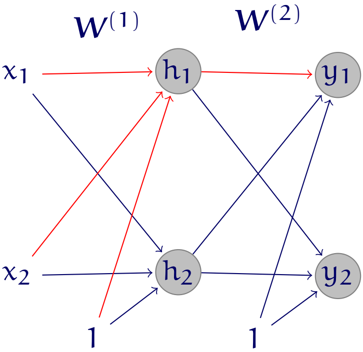
- Updating weights $W^{(2)}$ are easy: we can use gradient descent directly

Learning in ANNs: backpropagation



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

Learning in ANNs: backpropagation



- Updating weights $\boldsymbol{W}^{(2)}$ are easy: we can use gradient descent directly
- But the contribution of weights $\boldsymbol{W}^{(1)}$ 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, $y = w_0 + w_1 x_1 + \ldots + w_k x_k$

- 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., age)
  - Some effects are periodic (e.g., many measaures 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

$$y = w_0 + w_1 x_1 + w_2 x_1^2$$

  and

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2$$

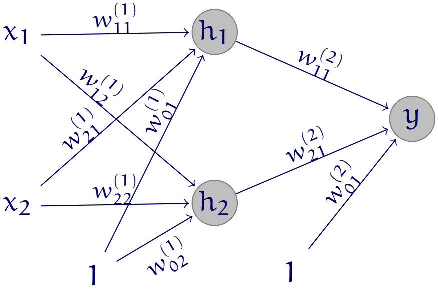
  are still linear in weights.

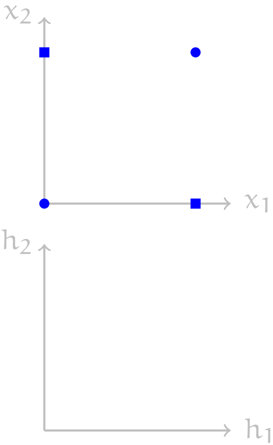
+ The objective function is still convex, we have a global minimum
– Requires careful feature selection/engineering
– Often becomes slow to train

Dealing with non-linearities (2)

(a short divergence, contd.)

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

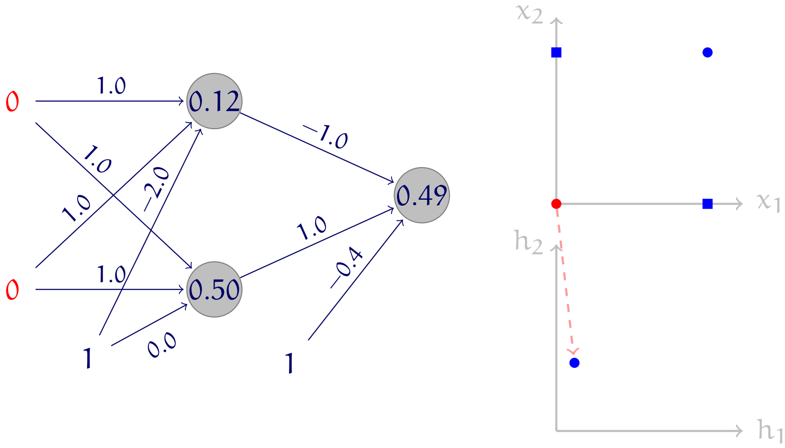




Dealing with non-linearities (2)

(a short divergence, contd.)

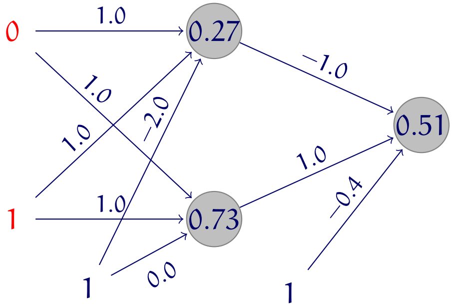
Multi-layer networks (a solution to the XOR problem)

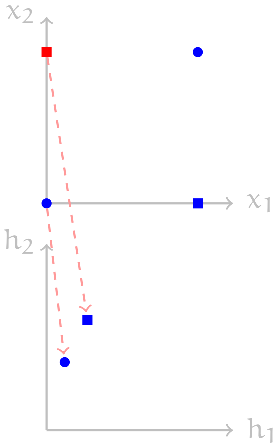


Dealing with non-linearities (2)

(a short divergence, contd.)

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

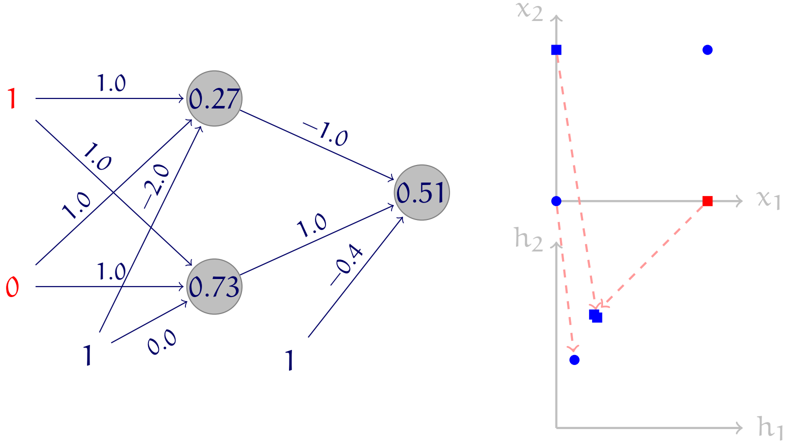




Dealing with non-linearities (2)

(a short divergence, contd.)

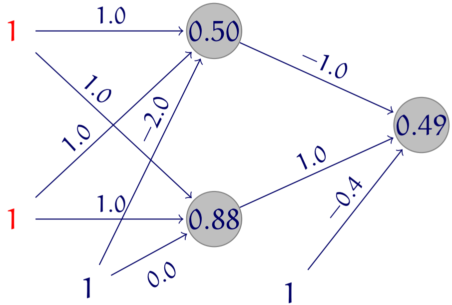
Multi-layer networks (a solution to the XOR problem)

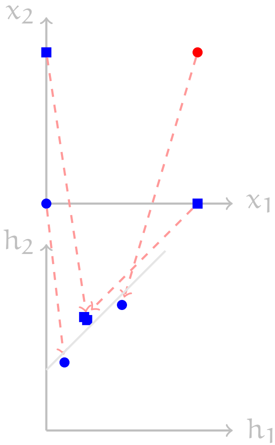


Dealing with non-linearities (2)

(a short divergence, contd.)

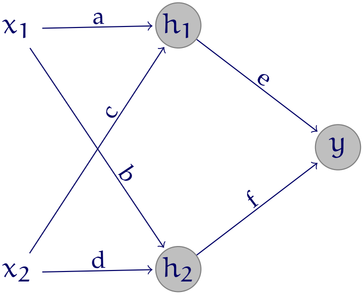
Multi-layer networks (a solution to the XOR problem)





Non-linear activation functions are necessary

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



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

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: why do they make sense?

Another important (and practical) reason is that some problems are easier to represent using multiple layers

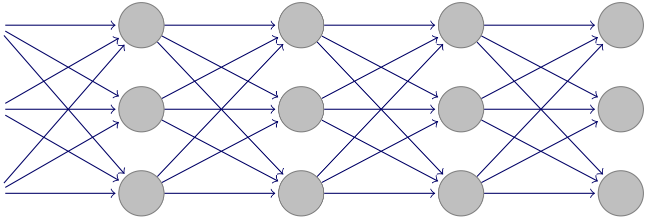
- Image processing
  pixels → edges → shapes → objects → …
- Language processing
  speech signal → phonemes → syllables → words → …

If problem can be simplified by learning a hierarchy of features, deep networks will probably be more useful.

Deep networks: why now?

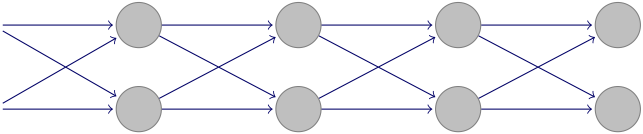
- Most models/methods used in deep networks today are proposed as early as 1980's
- The reason for 'renewed' popularity/success has to do with
  - Computer hardware became faster, especially GPUs are useful for training/using ANNs
  - ANNs are more efficient (e.g., in comparison to main rival SVMs), with big amount of data
  - Some developments allowed training models that were difficult to train before

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

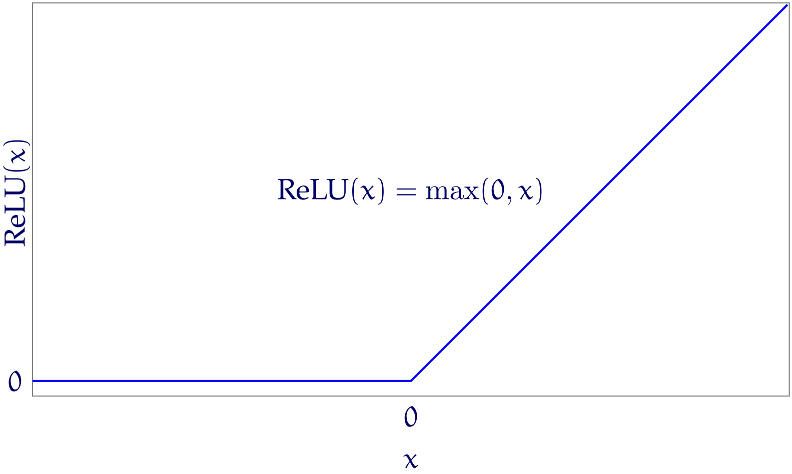
- Standard regularization methods (L1 or L2) help avoiding exploding gradients
- Another popular approach against exploding gradients is gradient *clipping* or or *scaling*
- Some activation functions are more prone to unstable gradients. For example, derivative of the logistic sigmoid is close to zero 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

Deep networks: interim summary

- Deep neural networks are simply ANNs with multiple hidden layers
- They have recently been successful in many ML tasks, including in NLP
- Like ANNs, they are powerful learners (but often opaque to interpretation)
- But beware of the hype: try simpler models first, often we do not need the power of a complex network
- The field is still active, many methods and tools are still experimental

Deep networks: interim summary

- Deep neural networks are simply ANNs with multiple hidden layers
- They have recently been successful in many ML tasks, including in NLP
- Like ANNs, they are powerful learners (but often opaque to interpretation)
- But beware of the hype: try simpler models first, often we do not need the power of a complex network
- The field is still active, many methods and tools are still experimental

Next: CNNs and RNNs

In-class exercise

Task: train an MLP for learning the XOR problem using keras `http://keras.io/`. Here is a quick reference:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
# create a new model
m = Sequential()
# the hidden layer -- try others: 'relu', 'sigmoid', ...
m.add(Dense(input_dim=2, output_dim=2, activation='tanh'))
# the output layer
m.add(Dense(output_dim=1, activation='sigmoid'))
# input and the output
x = np.matrix('0, 0; 0, 1; 1, 0; 1, 1')
y = np.array([1, 0, 0, 1])
# fit the model and predict
m.compile(loss='binary_crossentropy', optimizer='sgd')
m.fit(x, y, nb_epoch=10000)
m.predict(x)
```