DATA SCIENCE FOR DIGITAL HUMANITIES 2

MACHINE LEARNING

PROF. DR. GORAN GLAVAŠ

Deep Learning – Detailed Overview

TO

What is „Deep Learning” and what is it used for?

AI vs. ML vs. DL

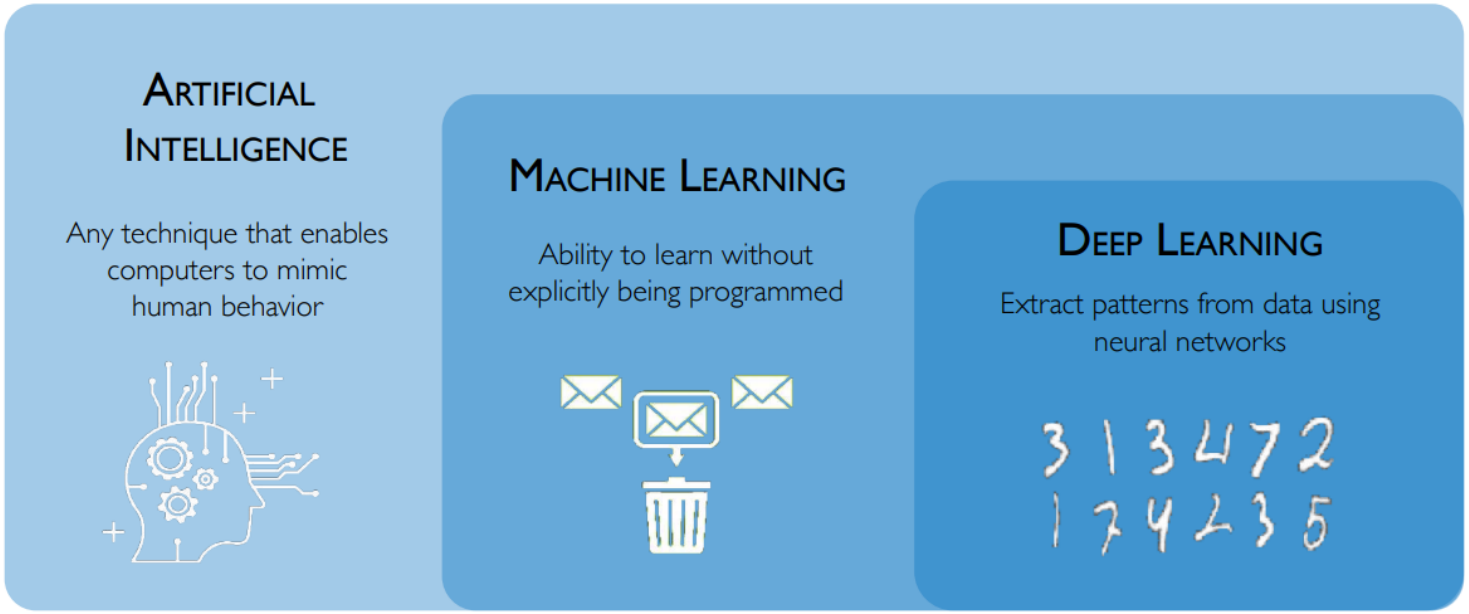


Image: © MIT 6.S191: Introduction to Deep Learning
introtodeeplearning.com

„Traditional ML" vs. „Deep Learning"

Traditional machine learning:

- Manual design of features:
    - We decide what is good input data / data representation
- **„Features"**: computed in a pre-processing step
- **Model**: characterize the data (classification or clustering) in terms of given fixed input representation / features
- **more bias in the models, less data required**

Deep/representation learning

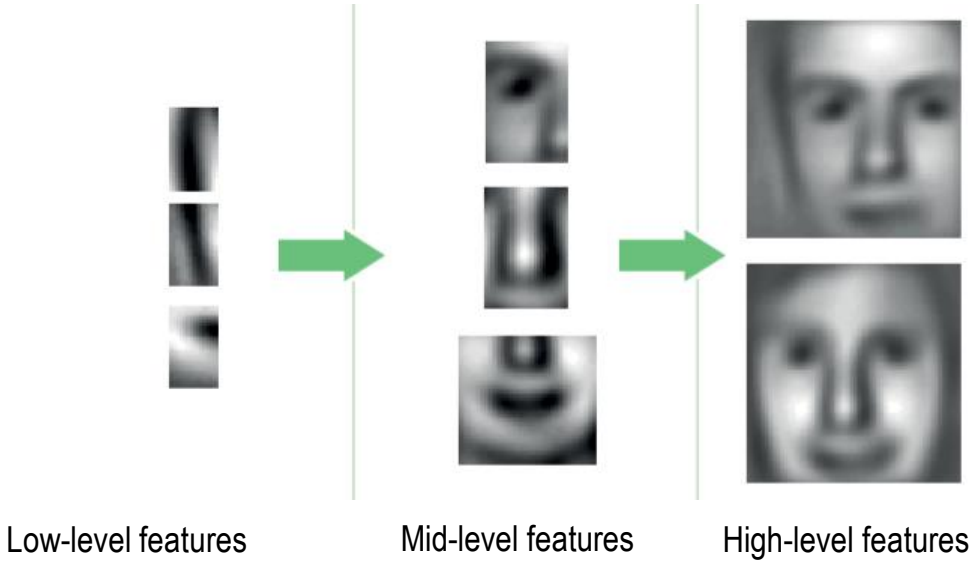
- Suitable representations of the input data are **also learned**
- **„Deep"** architectures: lower layers dedicated to learning data representations
- Less bias in the models, more data required

Why Deep Learning?

- Feature engineering is:
  - Time consuming
  - Tedious
  - Not scalable (some feature computations can be complex)
- **Deep Learning** is a paradigm in which we learn the underlying features useful for the task directly from data
  - Deep = multiple layers, each capturing a different level of abstraction
  - Lower layers capture finer-grained features
    - E.g., in CV, lines, contures; in NLP word meaning and syntax
  - Higher layers capture coarser features
    - E.g., CV: surfaces, shapes or objects; NLP: semantics (word meaning interactions)

Why Deep Learning?

- Useful features automatically recognized as patterns in the raw data



Deep Learning

- **Manual feature engineering**
  - Encoding domain/expert knowledge into the features
  - Reduces the amount of domain knowledge that needs to be learned from data

- **Deep Learning**
  - No prior domain/expert knowledge: we learn everything from the data
  - **We need more data**!
    - To learn the mappings from raw input into features
  - We need to learn more parameters (more complex models)
    - learning the mappings from raw input into features and then from features to prediction
    - **We need more computational power**!

TO

Commonly used DL models

Deep Learning – Detailed Overview

01 What is „Deep Learning” ML and what is it used for?

02 Perceptron and feed-forward networks

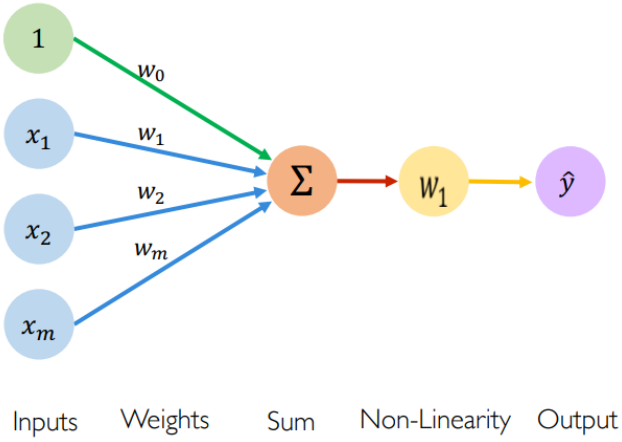
03 Autoencoders

04 Convolutional neural networks

05 Recurrent neural networks

Perceptron

- Takes inputs („features”) $\mathbf{x} = [1, x_1, x_2, \ldots, x_m]$ and computes a dot-product with weights (parameters) $w = [w_0, w_1, \ldots, w_m]$
- Applies a non-linear activation function g on $\mathbf{w}^T\mathbf{x}$



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

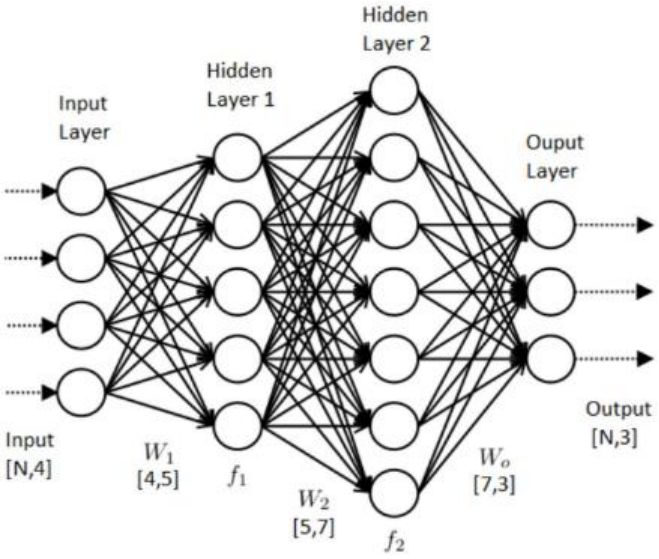
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Image from 6.S191 Introduction to Deep Learning
introtodeeplearning.com

Feed-forward network (FFDN)

- Also known as the **multilayer perceptron** (MLP)
- Layers of perceptrons
  - Outputs from previous = inputs for perceptrons of the next layer
  - Parameters of each layer (i.e., weights of perceptrons) can be written in the **matrix form**



In image:

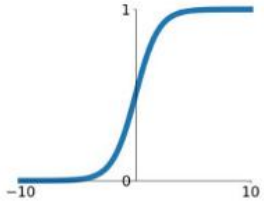
$$\mathbf{y} = \mathbf{W}_0\, f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x}))$$

Input: **x**, output: **y**, parameters: $\mathbf{W}_0$, $\mathbf{W}_1$, $\mathbf{W}_2$
Activation functions: f_1, f_2

Common Activation Functions

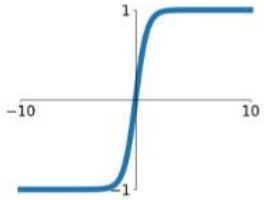
Sigmoid

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



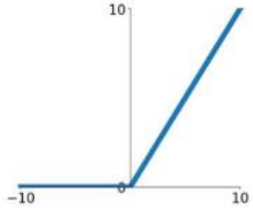
tanh

$$\tanh(x)$$



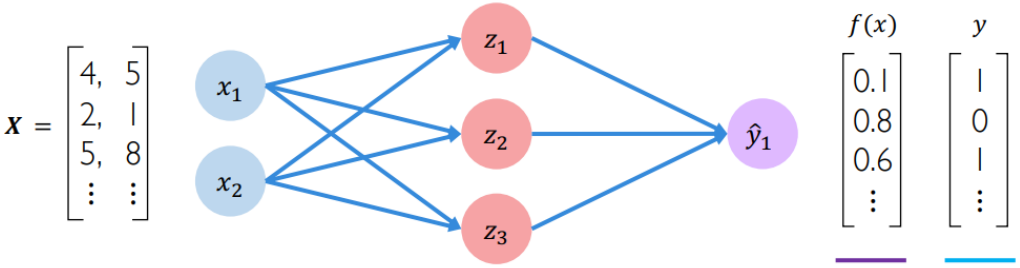
ReLU

$$\max(0, x)$$



Training Neural Networks

- We need to define some **loss function**
  - Remember the three components of each ML algorithm!
- For example, square loss/error or **cross-entropy error** (as for LR)
- Then, we need to **minimize the loss** on the **training data**!



$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log \underbrace{\left(f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}} + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \underbrace{\left(1 - f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}}$$

Image from 6.S191 Introduction to Deep Learning, introtodeeplearning.com

Training Neural Networks

- We need to **minimize the loss** on the **training data**!
  - Randomly initialize all network parameters **W**
- **Solve**:

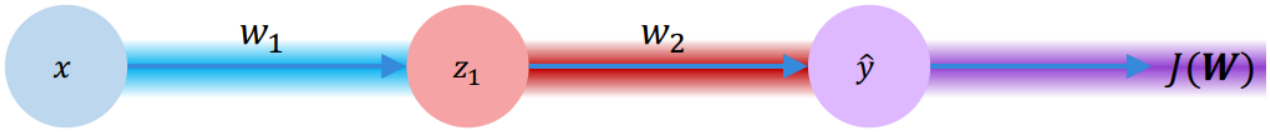
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

- The above equation has no closed-form solution => **iterative gradient-based optimization** (gradient descent and similar algorithms)
- **Gradient descent**: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
- **Backpropagation algorithm**: sequential computation of gradients in the reverse order (in the backward direction)

Training Neural Networks

- **Backpropagation algorithm**: sequential computation of gradients in the reverse order (in the backward direction)

- Based on the chain rule of differentiation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Image from 6.S191 Introduction to Deep Learning, introtodeeplearning.com

Deep Learning – Detailed Overview

01 What is „Deep Learning” ML and what is it used for?

02 Perceptron and feed-forward networks

03 Autoencoders

04 Convolutional neural networks

05 Recurrent neural networks

Autoencoders

- **Unsupervised** (sometimes called self-supervised) DL paradigm
  - We encode the raw input into a lower-dimensional space
  - Hoping to capture the patterns/regularities in raw data
  - Trying to learn **latent features**

- Commonly a pre-processing step for supervised learning

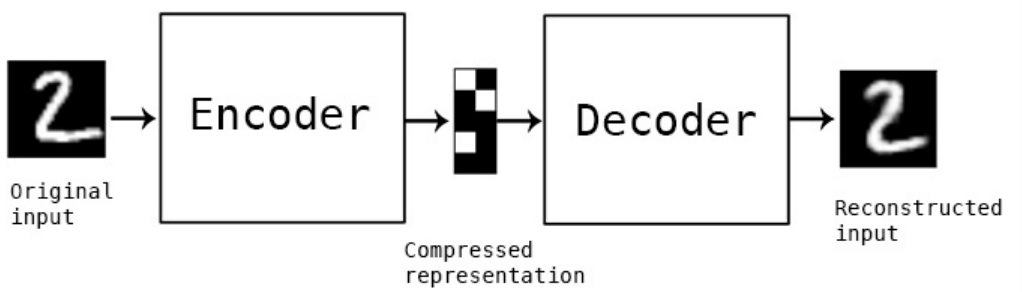


Image from https://blog.keras.io

Autoencoders

- **Autoencoding** is effectively a **data compression** algorithm
  - Compression: Encoder
  - Decompression: Decoder
  - Both neural networks' parameters are learned from the data

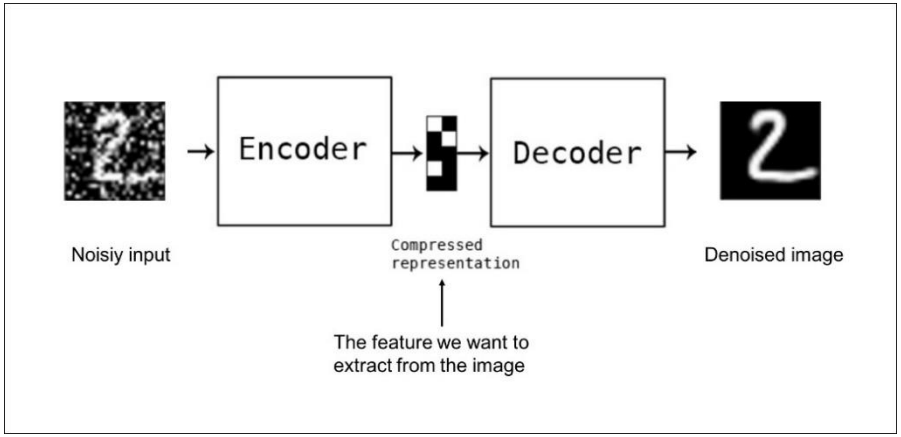
- Autoencoders are **data-specific**
  - Learn from data: they can only compress data similar (same distribution!) as that they were trained on
- But, autoencoders are **not** used for data compression
  - Lossless reconstruction cannot be guaranteed

- They are used for **self-supervised dimensionality reduction** as a pre-processing step for supervised learning with limited training data
- A type of **„continuous clustering"**

Denoising Autoencoders

- Learning to reconstruct the input is often not „stimulating enough” for the encoder network to learn useful latent features

- Some manipulation of the input (**introducing noise**)
- Reconstructing the **clean** input (i.e., without the noise)
- Encoder forced to **learn to remove the noise**: that way learns more useful latent representations



Deep Learning – Detailed Overview

01 What is „Deep Learning” ML and what is it used for?

02 Perceptron and feed-forward networks

03 Autoencoders

04 Convolutional neural networks

05 Recurrent neural networks

Convolutional Neural Networks

- Originate from **Computer Vision**
    - Biological inspiration: human eye and its receptive field mechanisms
- Gradually reducing the resolution of the image with **convolutional layers** and **pooling layers**

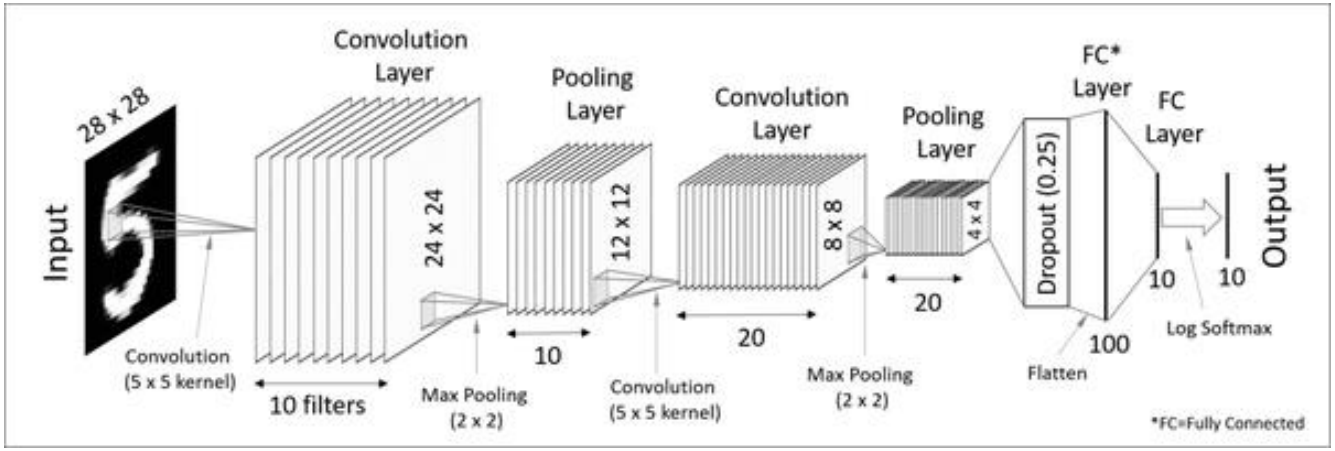


Image from:
https://codetolight.files.wordpress.com

Convolutional Neural Networks

- **Convolution**:
  - Operation between a slice/area of the input and the **„filter matrix"** (also known as **„kernel"**)

- Filter matrix is typically much smaller in dimensions than the input
  - E.g., an image is 28x28, but a filter can be, e.g., 3x3 or 5x5

- There are typically multiple filters of the same size

$$\text{Conv(A, B)} = \sum_{i=1}^{M} \sum_{j=1}^{N} A_{ij} * Bij$$

- **Pooling:**
  - Reducing an area of some size (e.g., 3X3) to a single value
  - Different types of pooling: max-pooling, average-pooling

Convolutional Neural Network

- Filter matrices (aka kernels) are the parameters of the CNN
- CNN by itself is not a classifier, it just encodes the input representation into a smaller-dimensional feature vector
- CNN is commonly coupled with a feed-forward classification net

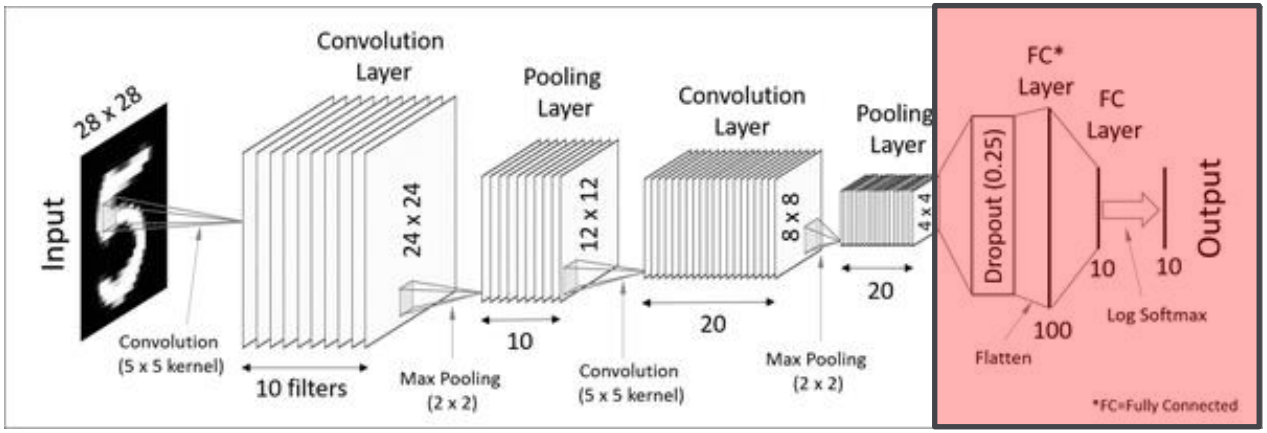
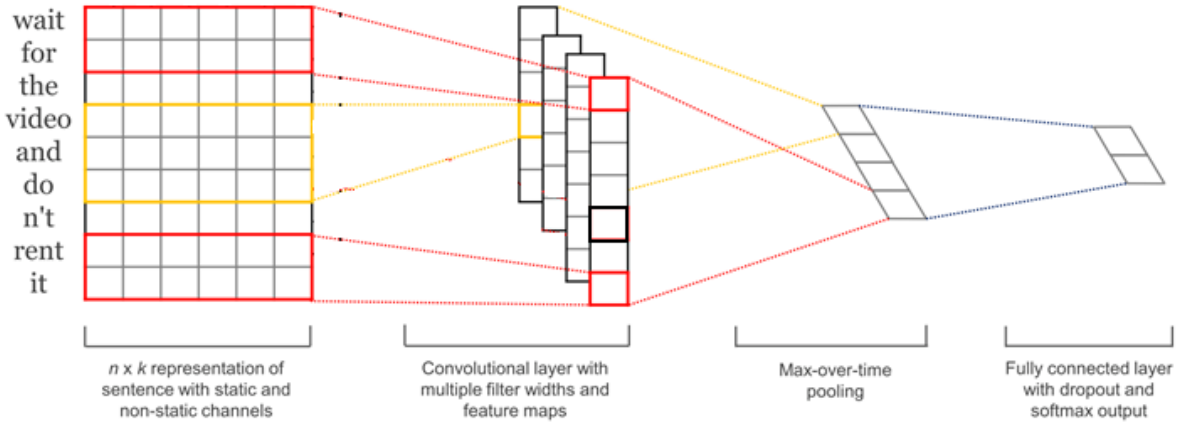


Image from: https://codetolight.files.wordpress.com

Convolutional Neural Networks for Text

- Although they come from the Computer Vision world, CNNs have been useful in many text classification tasks as well
- For text classification, we use 1-D CNNs (we stride over only 1 dim.)



Convolutional Neural Networks for Text

- Let d be the length of word embedding vectors
- **Filters** are CNN parameter matrices of size K x d, where K is small number, typically between 3 and 5
    - K is called the size of the filter

- One CNN typically has many filters, often of different sizes
    - E.g., 32 filters of size 3, 64 filters of size 4, and 32 filters of size 5

- **Convolution layer**:
    - Each filter strides down the input sequence and produces a convolution score with each input subsequence of size K
- Let $\mathbf{F}_K$ be one filter (matrix) od size K (i.e., dimensions K x d)
- Let $\mathbf{X}_{[a:b]}$ be the submatrix of the input matrix **X** consisting of rows a to b

- We then compute the vector of following convolutions

$$\mathbf{C}(\mathbf{F}_K) = [\text{Conv}(\mathbf{X}_{[1:K]}, \mathbf{F}_K); \text{Conv}(\mathbf{X}_{[2:K+1]}, \mathbf{F}_K); \text{Conv}(\mathbf{X}_{[3:K+2]}, \mathbf{F}_K); \ldots; \text{Conv}(\mathbf{X}_{[N-K+1:N]}, \mathbf{F}_K)]$$

Convolutional Neural Networks for Text

- **Pooling layer**:
  - Each filter F_K will produce a vector of convolution scores $C(F_K)$ over the input sequence
  - We want to keep only the „most salient" local sequences
- That's why we tipically select only k largest values from the convolution vector of each filter – this is called **k-max pooling**
- **Most often 1-max pooling is used (only the largest value is kept)**

- **Latent text representation**:
  - We concatenate the results of pooling for each of the filters into a single vector which is the latent representation of the text

Deep Learning – Detailed Overview

01 What is „Deep Learning” ML and what is it used for?

02 Perceptron and feed-forward networks

03 Autoencoders

04 Convolutional neural networks

05 Recurrent neural networks

Recurrent neural networks

- **Recurrent neural networks** are neural models that explicitly model sequences / time series
- RNNs represent „arbitrarily" long sequences into a fixed-size vector
- But unlike CNNs, which capture only local sequences of small length (3-5), RNNs aim encode dependencies over entire sequences
- **General RNN model**:
  - **Input**: sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$
  - RNN is a function that converts an arbitrary size sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$ into a fixed size output vector $\mathbf{y}_n$
  - Analogously, the subsequence $\mathbf{x}_1, \ldots, \mathbf{x}_i$ will produce the output $\mathbf{y}_i$
  - The output vector $\mathbf{y}_{i-1}$ of the previous step (i-1) is combined with the current input $\mathbf{x}_i$ to produce the output $\mathbf{y}_i$
- The RNN network is, at time step i, represented with its current state $\mathbf{s}_i$

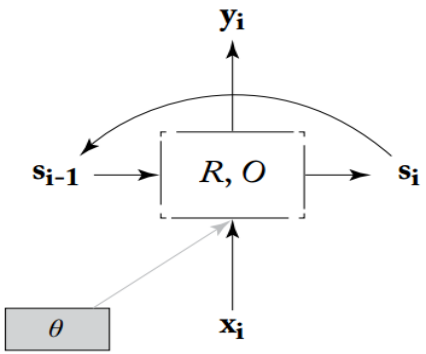
Recurrent neural networks

General RNN model:

- Defined by two functions:
- Function R defines how the next state $\mathbf{s}_i$ is computed from the previous state $\mathbf{s}_{i-1}$ and current input $\mathbf{x}_i$
- Function O defines how the current output $\mathbf{y}_i$ is computed from the current state $\mathbf{s}_i$
- Obviously, RNN is defined **recursively**

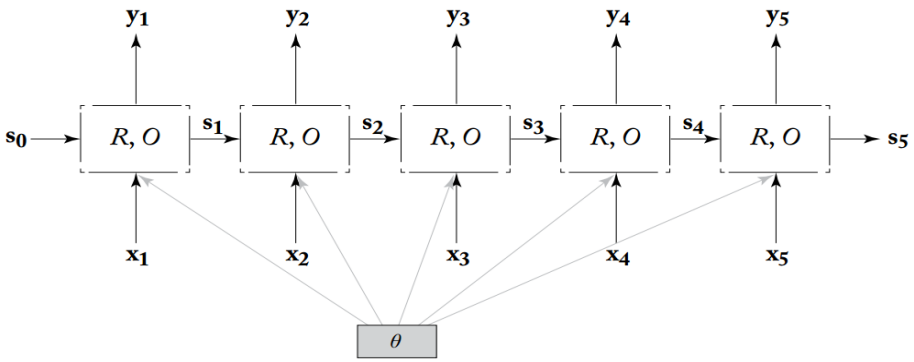
$$\mathbf{y}_n = O(\mathbf{s}_n)$$
$$\mathbf{s}_n = R(\mathbf{x}_n, \mathbf{s}_{n-1})$$
$$= R(\mathbf{x}_n, R(\mathbf{x}_{n-1}, \mathbf{s}_{n-2})$$
$$\ldots$$
$$= R(\mathbf{x}_n, R(\ldots(R(\mathbf{x}_1, \mathbf{s}_0))$$

- $\boldsymbol{\theta}$ are RNN parameters (in R)



Recurrent neural networks

- For some input sequence of finite length, we can „unroll” the RNN recursion



- $\mathbf{s}_n$ (and $\mathbf{y}_n$) can be thought of as the encoding of the whole sequence
- This general RNN model is instantiated into concrete models by defining functions *R* and *O*

Simple (Elman) RNN

- The simplest RNN formulation that still captures the ordering of the elements in the sequence
- **Model:**
  - R = Non-linear transformation g (usually hyperbolic tangent or sigmoid) applied to a linear combination of the input and previous state

$$\mathbf{s}_i = R(\mathbf{x}_i, \mathbf{s}_{i-1})$$
$$= g(\mathbf{x}_i \mathbf{W}^x + \mathbf{s}_{i-1} \mathbf{W}^s + \mathbf{b});$$

- O = identity function

$$\mathbf{y}_i = O(\mathbf{s}_i)$$
$$= \mathbf{s}_i$$

- **Parameters**
  - $\boldsymbol{\theta} = (\mathbf{W}^x, \mathbf{W}^s, \mathbf{b})$ if inputs are fixed

Gated cell architectures in RNNs

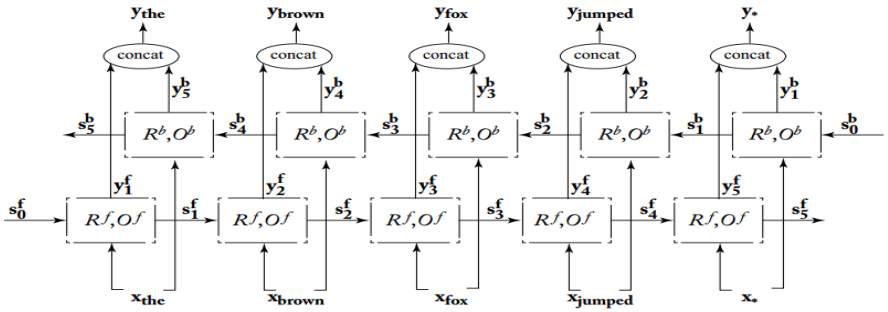
- Simple architecture suffers from a problem known as vanishing gradients
  - Error signals / gradients from later steps in the sequence diminish quickly in the backpropagation algorithm
  - The updates for early inputs that come from errors in later steps are very small
    - Esentially, Simple RNN has difficulties capturing long-distance dependencies
  - At each step, the whole RNN state is rewritten

- **Gated architectures idea**
  - Do not update the whole state at every step
  - Introduce parameters that decide which parts of the state to update
  - Introducing gate vectors:
    - They define which parts of the new state are taken from previous state and which from the current input
  - Models: **Long short-term memory (LSTM)**, Gated Recurrent Unit

Bi-directional RNNs

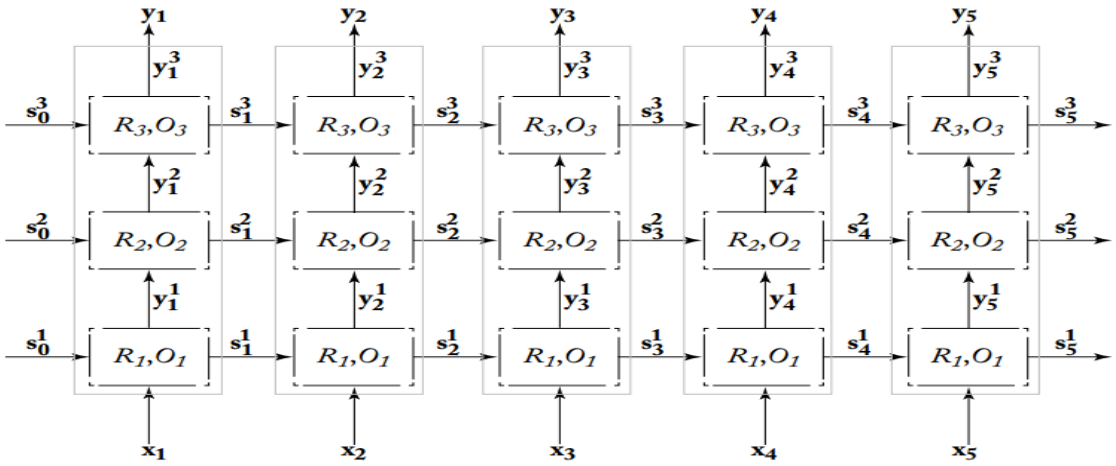
- Standard RNN at each step only encodes the sequence from one side of the current token
- For many time-series and sequence labeling tasks we want to incorporate knowledge about the **context from both sides**

- **Bidirectional RNN** is a model that combines two uni-directional RNNs encoding in opposite directions
- Output at step *i* is the concatenation of output vectors of both RNNs



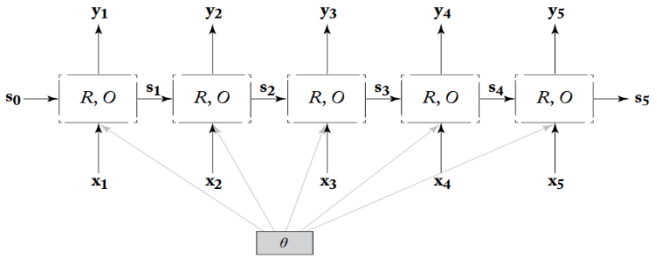
Multi-Layer (Stacked) RNNs

- RNNs can be stacked in layers, forming a grid
- The input for the first RNN are the actual input $\mathbf{x}_1, \ldots, \mathbf{x}_n$
- The input for all other layers are the outputs of previous layer RNN
- This architecture is called Deep RNN



RNN training

- As „unrolled”, an RNN is just a (very deep) feed-forward network
- Same parameters are shared across layers
- Additional input added at each layer

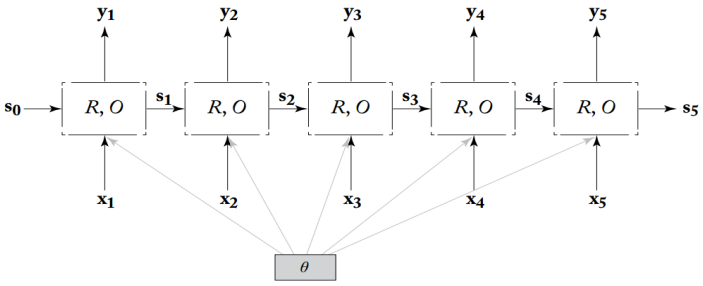


- To train an RNN network, we need to
  1. Create an unrolled computation graph for a given input sequence
  2. Add a loss function node to the unrolled computation graph
  3. Compute the loss (i.e., error) for a given input sequence
  4. Update the RNN parameters (**W** matrices and **b** bias vectors) to minimize the loss

RNN usage patterns

- **Q:** What is the loss function for an RNN?

- **A**: That depends on the type of task for which we are performing



- **Type of tasks** addressed using RNNs (i.e., RNN usage patterns):
  1. **Encoder** – a prediction for the whole sequence
     - Loss based only on the last state (encoding the whole sequence)
     - Classification of the whole sequence
  2. **Transducer** – one prediction made at every position in the sequence
     - This is the **sequence labelling** usage of RNNs

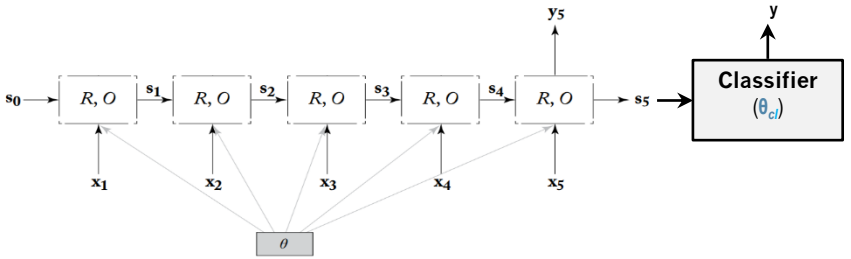
RNN usage patterns

RNN as encoder

- Assumptions:
- the last state vector (s_n)

  encodes the whole sequence
- Thus, if we need to make a prediction

  for the whole sequence, we can use the last state as its representation



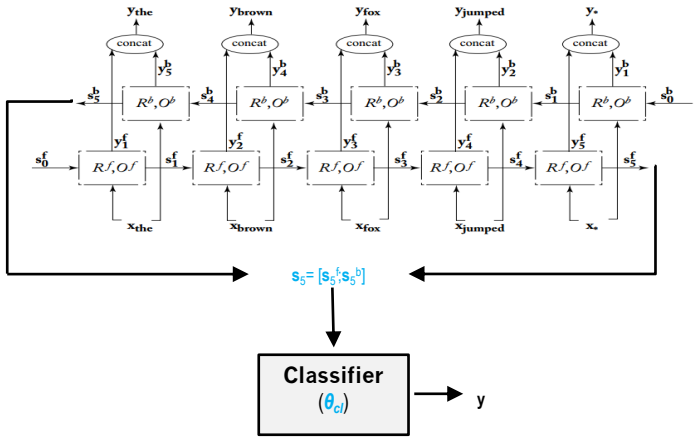
- The representation of the whole sequence, the last state, is fed into a classifier
  - E.g., a ternary classifier determining whether the sentence has positive, negative, or neutral sentiment
  - The classifier is usually a feed-forward network with its own set of parameters ($\boldsymbol{\theta}_{cl}$)

$$\mathbf{y} = \text{softmax}(\mathbf{s}_n\mathbf{W}_{cl} + \mathbf{b})$$

RNN usage patterns

RNNs as encoders

- Bidirectional RNN consists of two unidirectional RNN
- We have two states (s_n^f, s_n^b) that encode the whole sequence
- s_n^f encodes left-to-right
- s_n^b encodes right-to-left

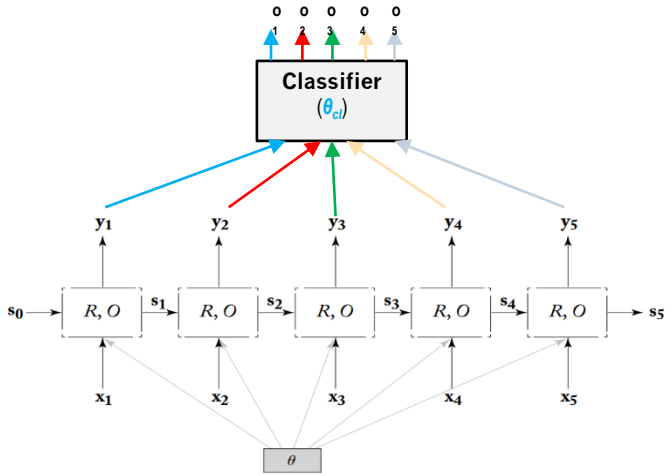


- **Q:** How do we create a sequence representation for classification?
- **A:** We concatenate the two final states: $s_n = [s_n^f; s_n^b]$

RNN usage patterns

RNN as transducer

- When RNN is used for sequence labelling
- We need to predict the class at every time step of the RNN



- Again, we couple the RNN with a feed-forward classifier
- But now we predict the class at every position in the sequence, instead of only from the final sequence state as in the encoder usage pattern
- The prediction loss for the whole sequence is simply the sum of prediction losses of all token-level predictions

Key Takeaways

Main application areas: text/language and vision (images / videos)

Deep Learning: raw data as input, lower layers induce representations for classification

Numeric parameter optimization, via backpropagation

Traditional ML: precomputed features = modeling bias

Prominent DL architectures: autoencoders, convolutional, and recurrent networks