Contents of this lecture

General about

- Small intro (E, T, P)
- Capacity, overfitting and underfitting
- Hyper parameters and model selection

Practical considerations

- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

Introduction to Machine Learning?

What is Machine Learning?

“Learning is any process by which a system improves performance from experience.”

“Machine Learning is concerned with computer programs that automatically improve their performance through experience. “

- Herbert Simon

A computer program is said to **learn**

from experience E with respect to some class of tasks T and performance measure P,

if its performance at tasks in T, as measured by P, improves with experience E.

- Tom Mitchell

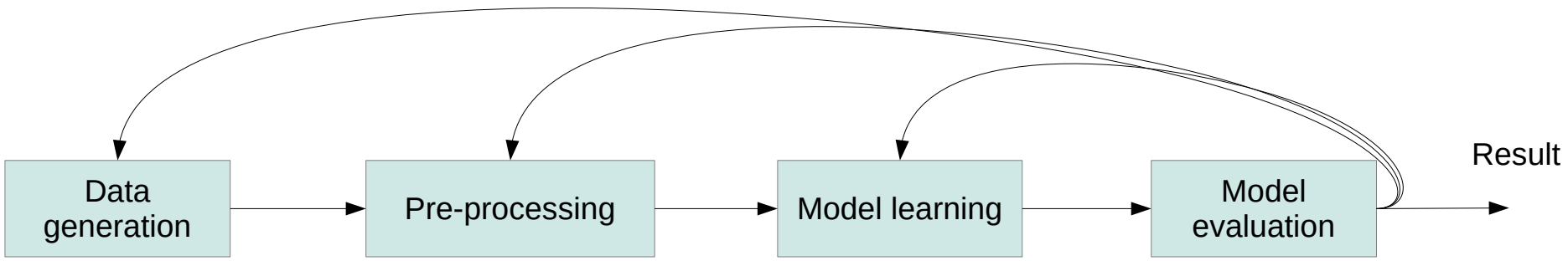
According to Tom Mitchell

Learning =

- Improve over task *T*
- With respect to performance *P*
- Based on experience *E*

Learning!

Typical learning process



Sensors
Experiments
Databases
...

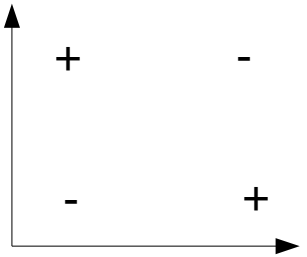
Normalization
Noise filtering
Feature extraction
Feature selection
Dimensionality reduction
...

Classification
Regression
Clustering
...

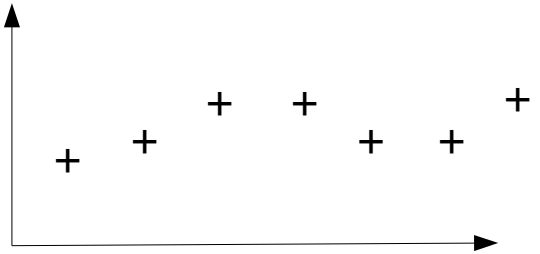
Cross validation
Bootstrap
...

Tasks (*T*)

Some common tasks:



Classification: The algorithm is asked to give one of *k* classes for which the input belongs to

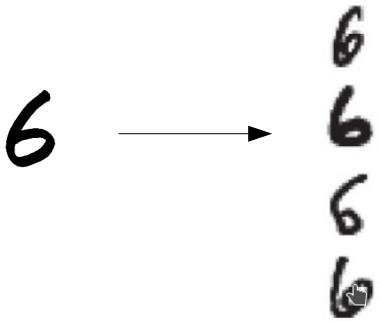


Regression: Predict a numerical output given an input

Translation: sequence in → sequence out

It's raining cats and dogs →
Det regnar katter och hundar

Synthesis and sampling: : Generate new data "similar" to the given data



Performance measures (P)

Accuracy: Used for classification problems. Easy to understand, but there plenty of others. Area under ROC curve is sometimes better.

Instead of performance one can also talk about errors. Mean squared error can be used for regression problems, but others exist!

Other tasks have other performance or error measures. The choice is very application dependent and may be difficult to formulate.

Note 1: It is important to that performances or error should be measured on independent data

Note 2: Error measures used to communicate results are typically not the same as error used during model training

Experience (*E*)

Based on the type of experience there are broadly two kinds of learning algorithms (excluding reinforcement learning) :

Supervised learning and Unsupervised learning

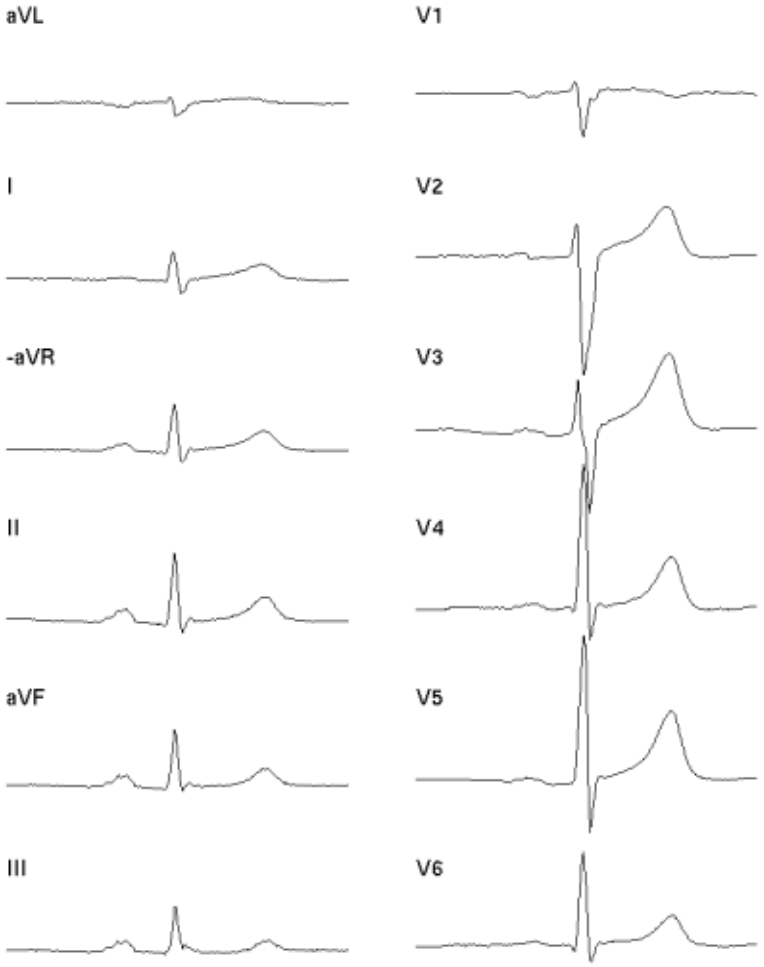
For all machine learning problems we a data set

$$\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \longrightarrow \text{Unsupervised learning}$$

If we add labels

$$\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} + \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_N\} \longrightarrow \text{Supervised learning}$$

An example



Classification of ECGs

Input data $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$

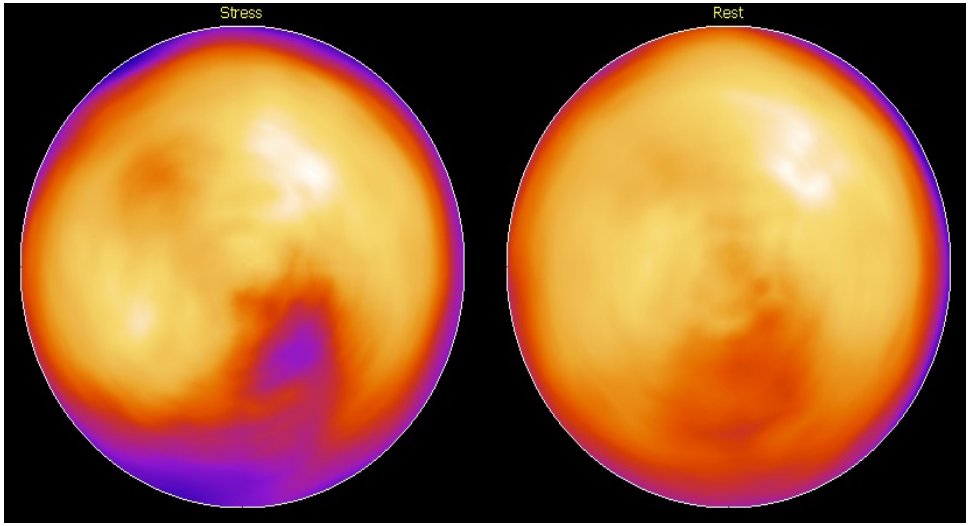
Extracted features
Raw signal

Labels $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_N\}$

AMI or not AMI
Ischemia or not ischemia

Another example

Classification of “heart” images



Input data $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$

Extracted features
Image (RGB pixel values)

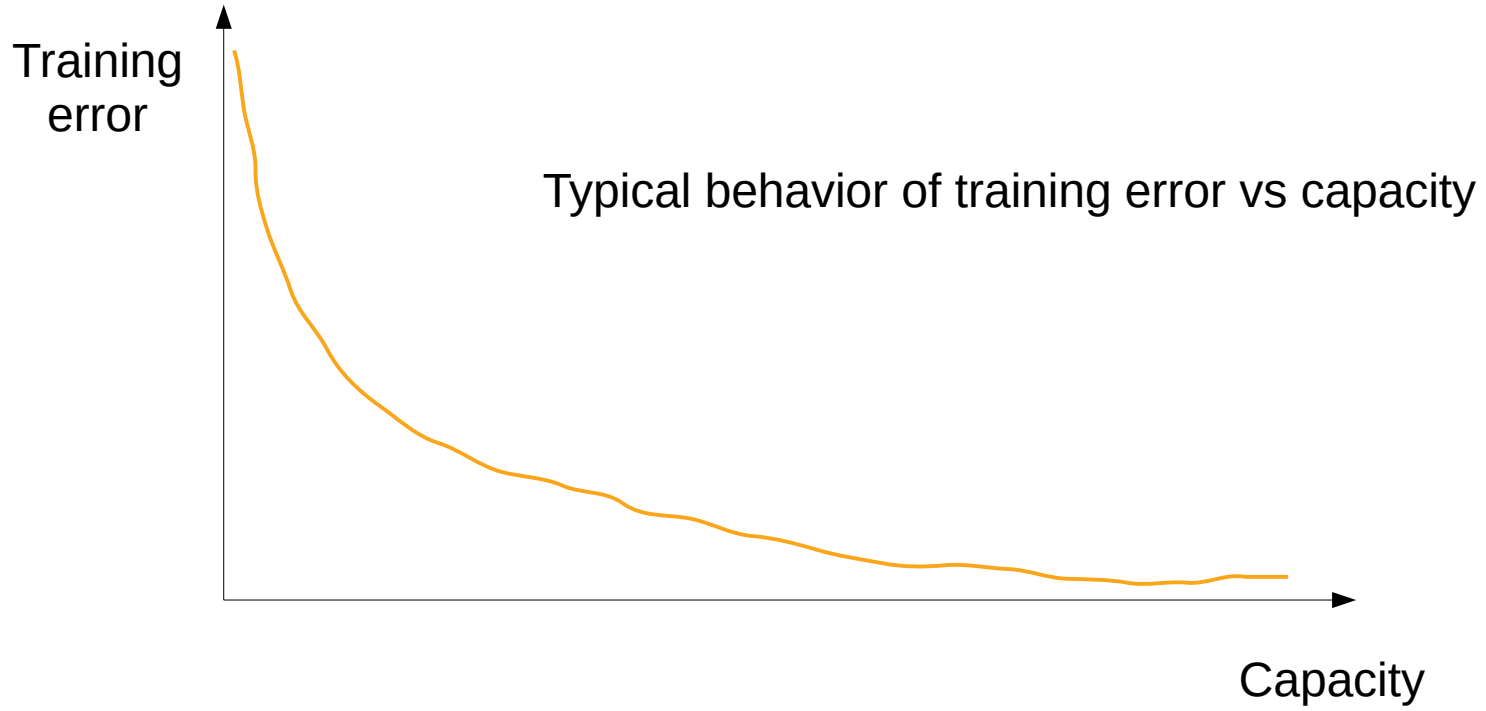
Labels $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_N\}$

AMI or not AMI
Ischemia or not ischemia

Capacity, overfitting and underfitting

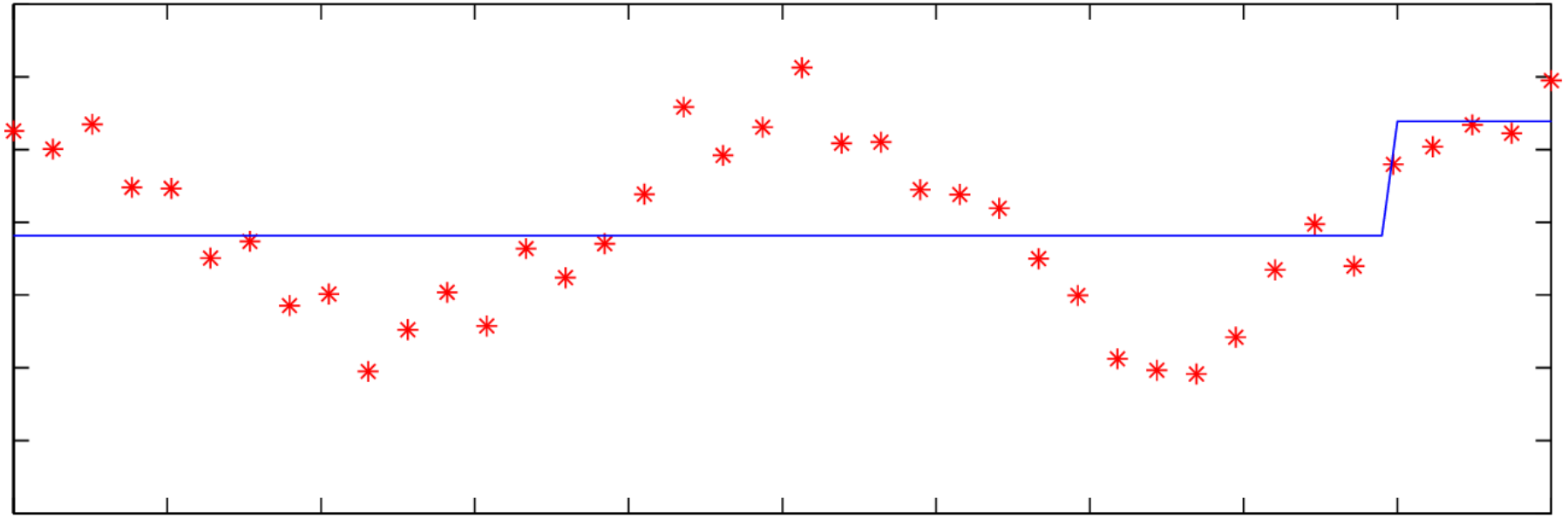
We have a data set *D* of cases (data points) used to train a model.

A model can be said to possess a certain capacity (= the ability to fit a range of different functions. With small capacity we can only fit a limited number of functions and increasing the capacity means fitting a larger set of functions).

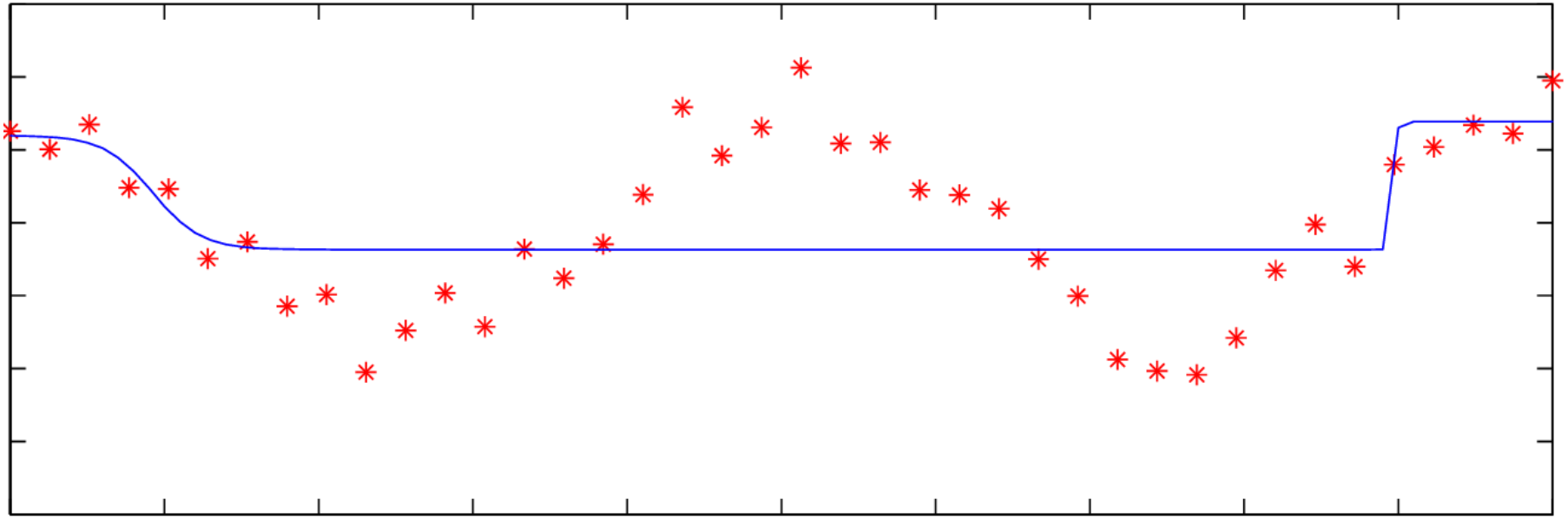


Typical behavior of training error vs capacity

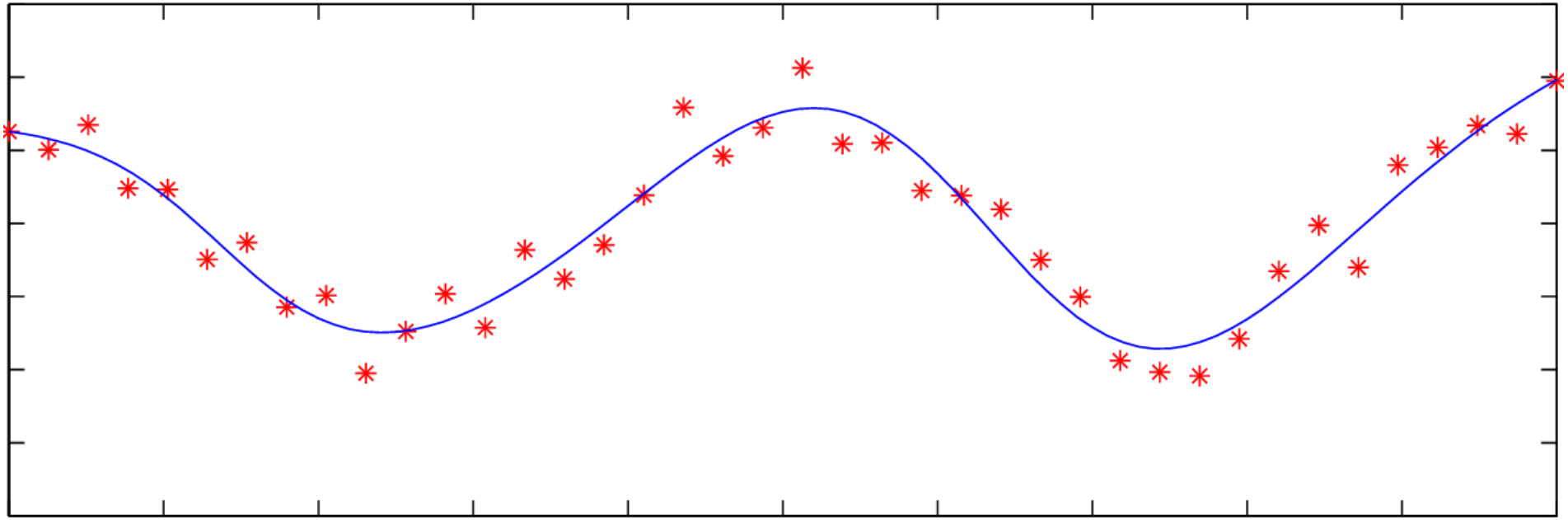
ML-model of degree “1”



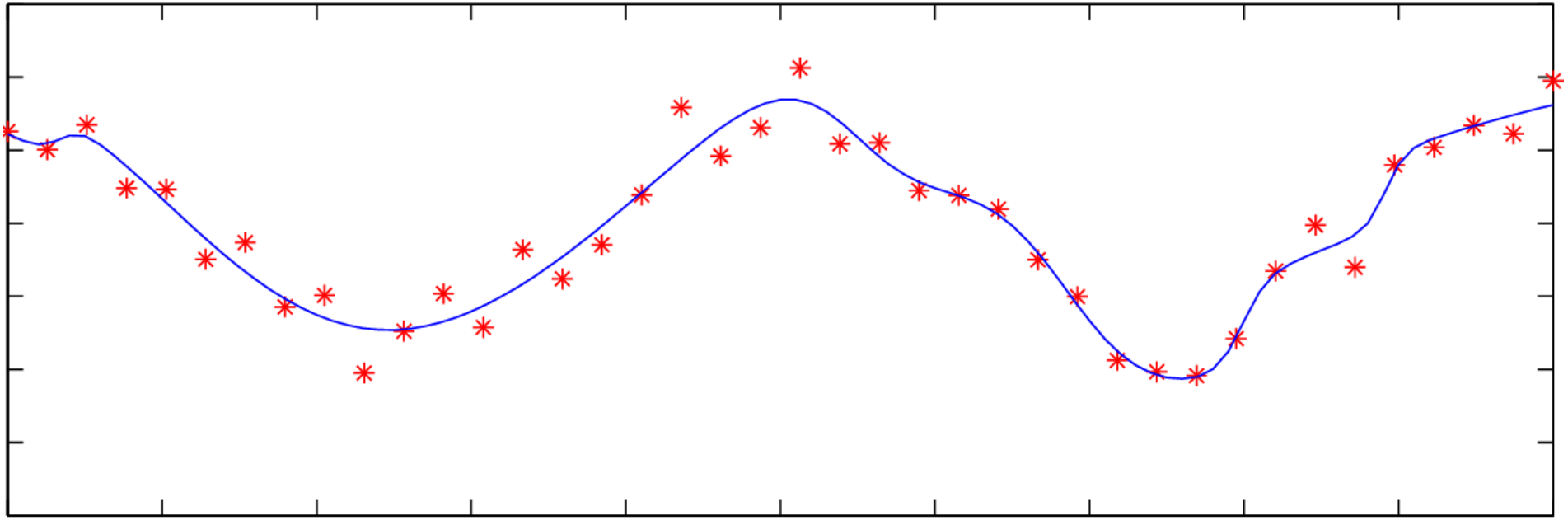
ML-model of degree “2”



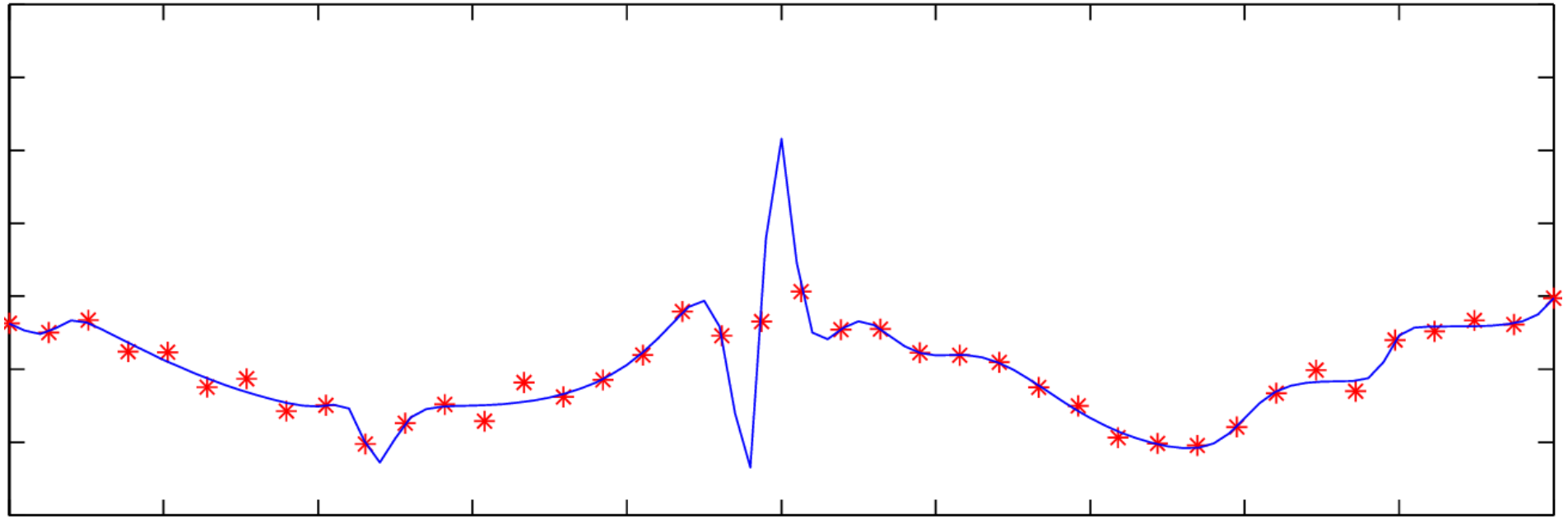
ML-model of degree “4”



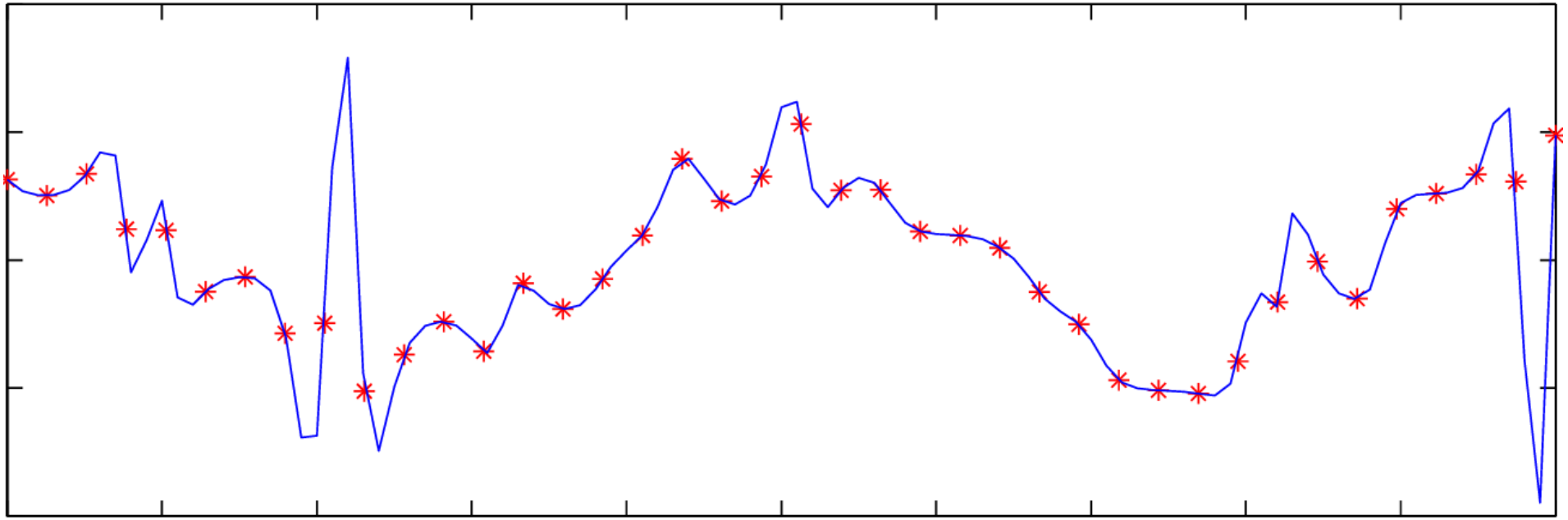
ML-model of degree “7”

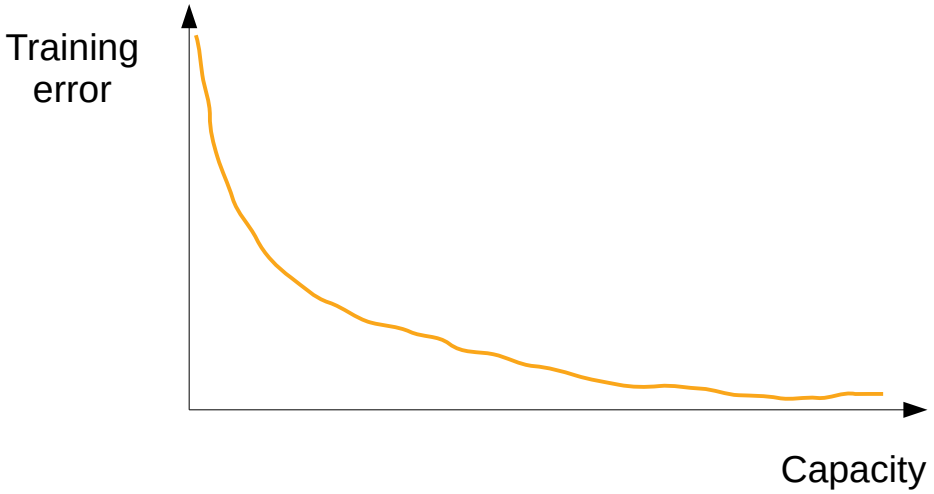


ML-model of degree “12”



ML-model of degree “25”





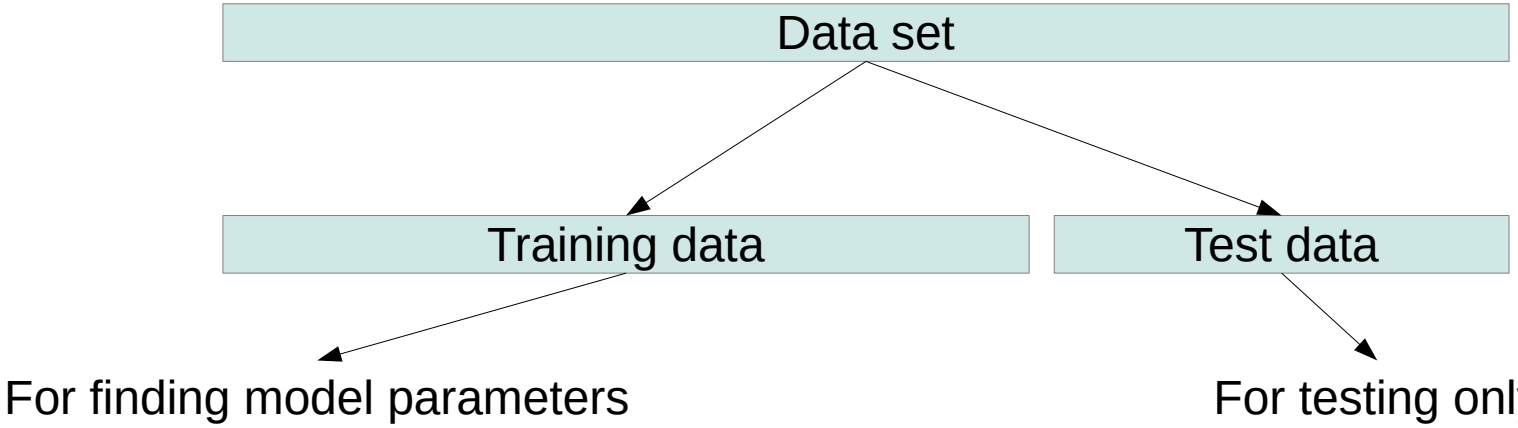
Problem: We cannot use training error or performance as an indicator how well my ML-model is doing!

The central challenge of machine learning is to perform well on **new previously unseen** data, **not** part of the training.

A model with such ability is said to **generalize**.

The **generalization performance/error** = expected performance/error on “new” inputs

Often,



Test error ≈ generalization error

We expect:

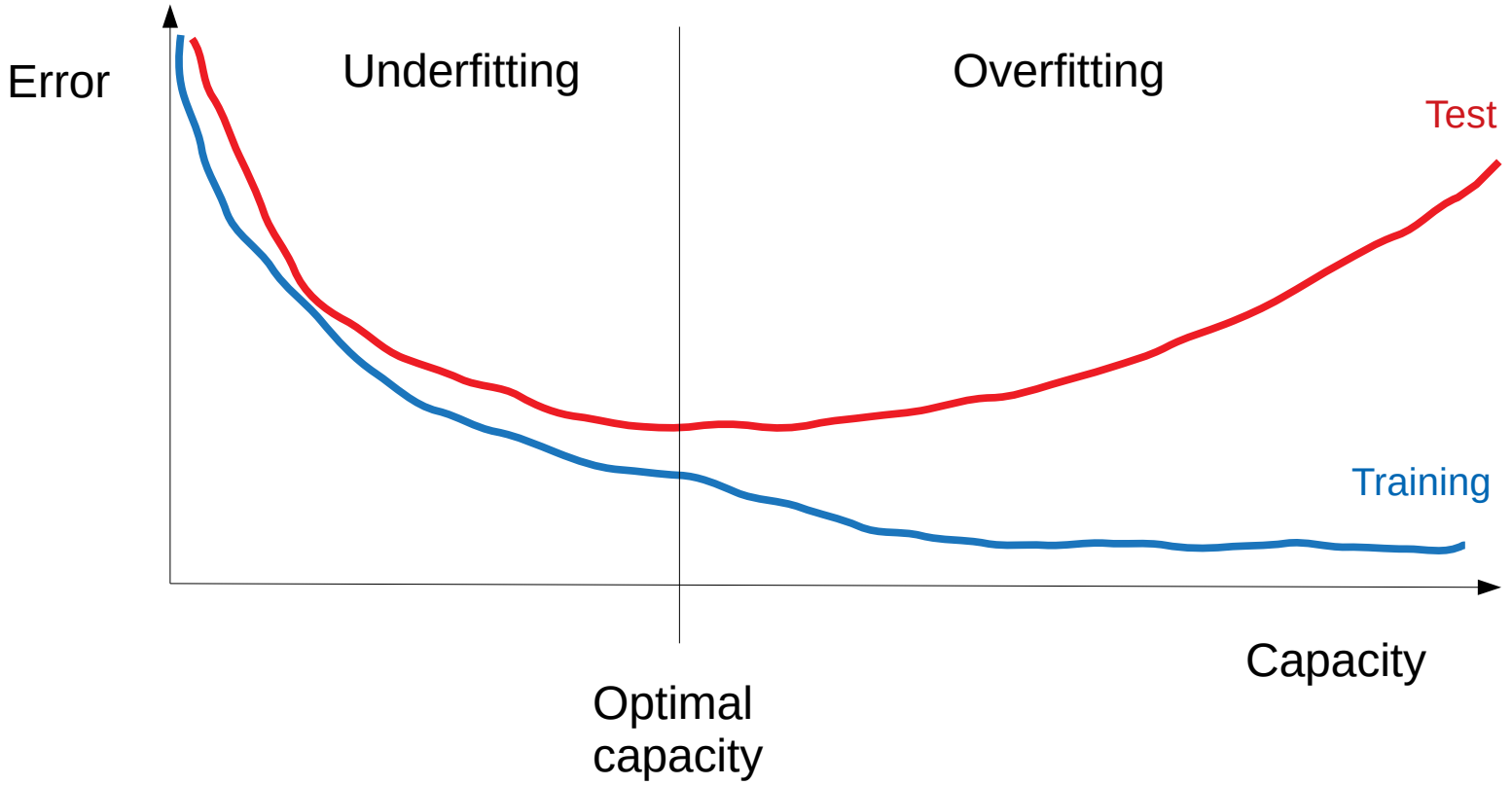
Test error ≥ Training error

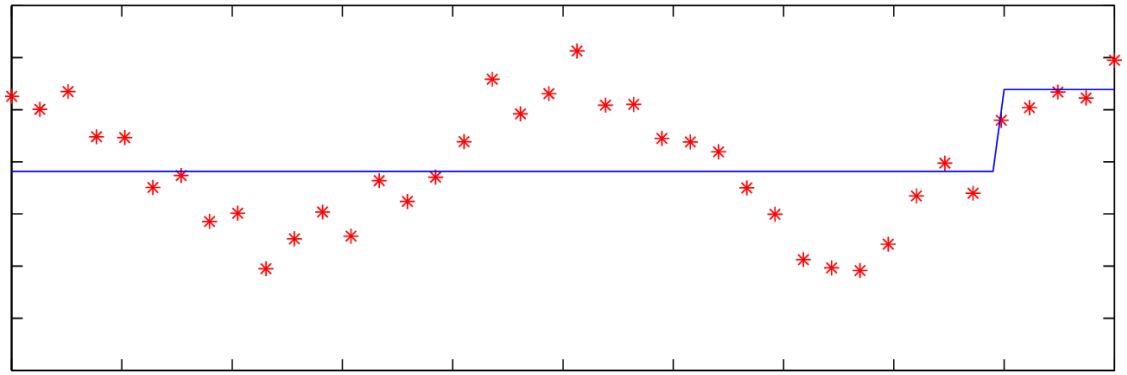
GOAL:

Minimize training error

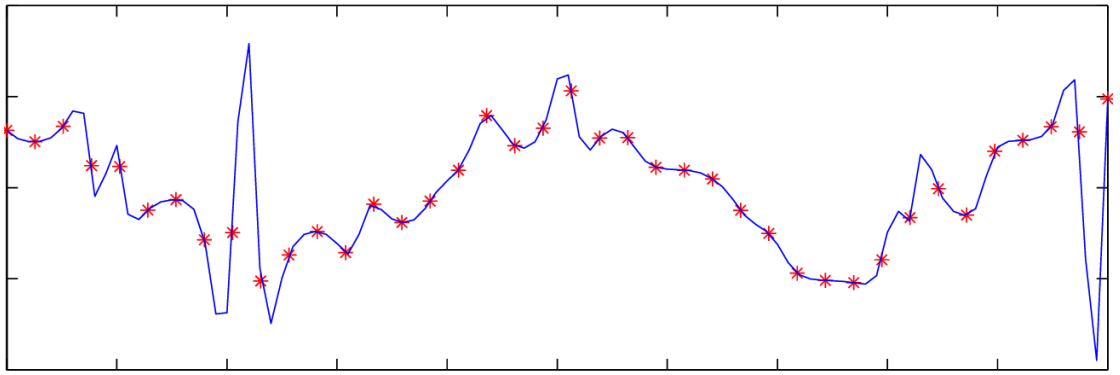
Make the gap between test error
and training error as small as possible.

Overfitting and underfitting





Underfitting



Overfitting

Bias-Variance tradeoff

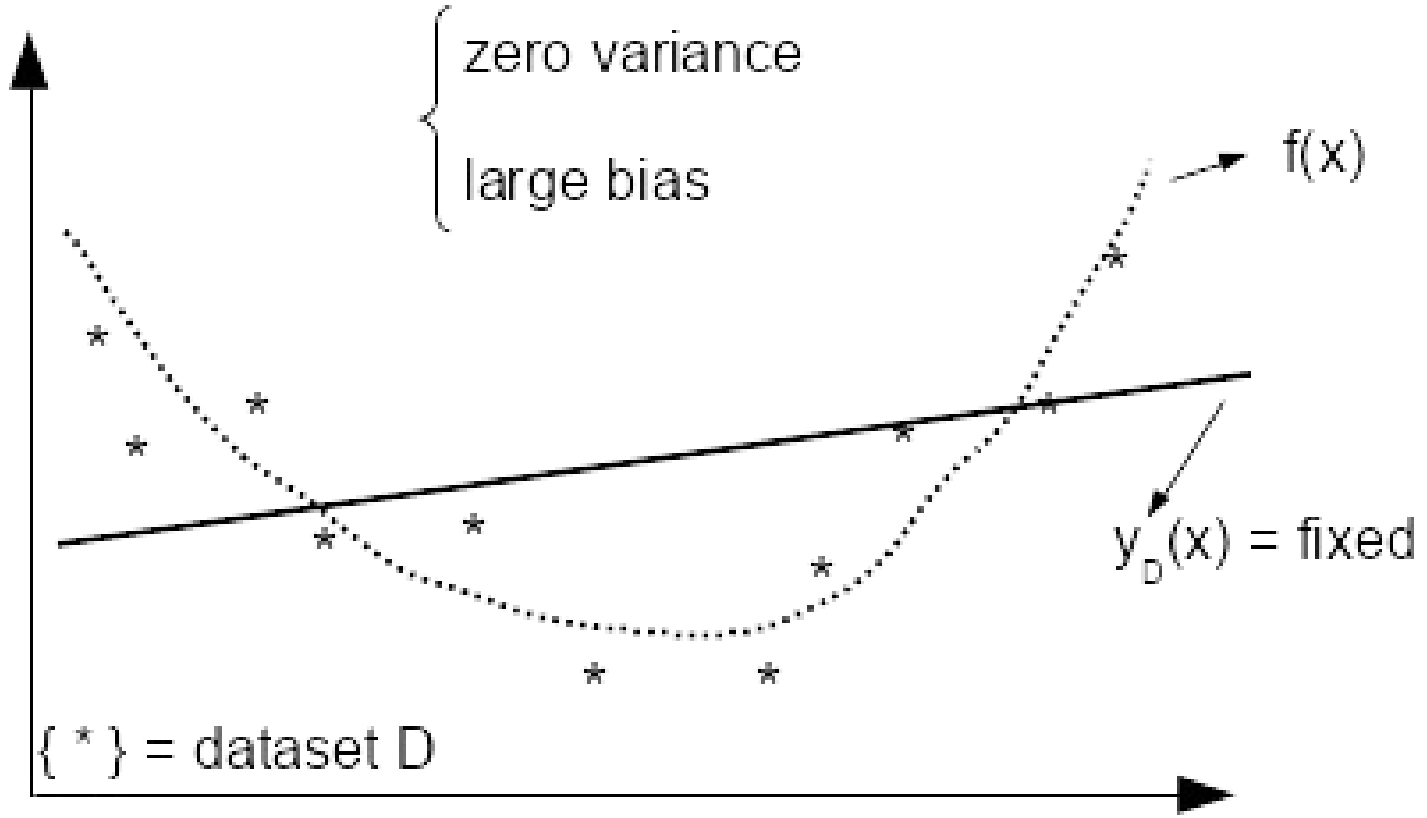
$D = \text{Data set of size } N$

$y_{\mathrm{D}}(\mathbf{x}) = \text{Model, trained on data set } D$

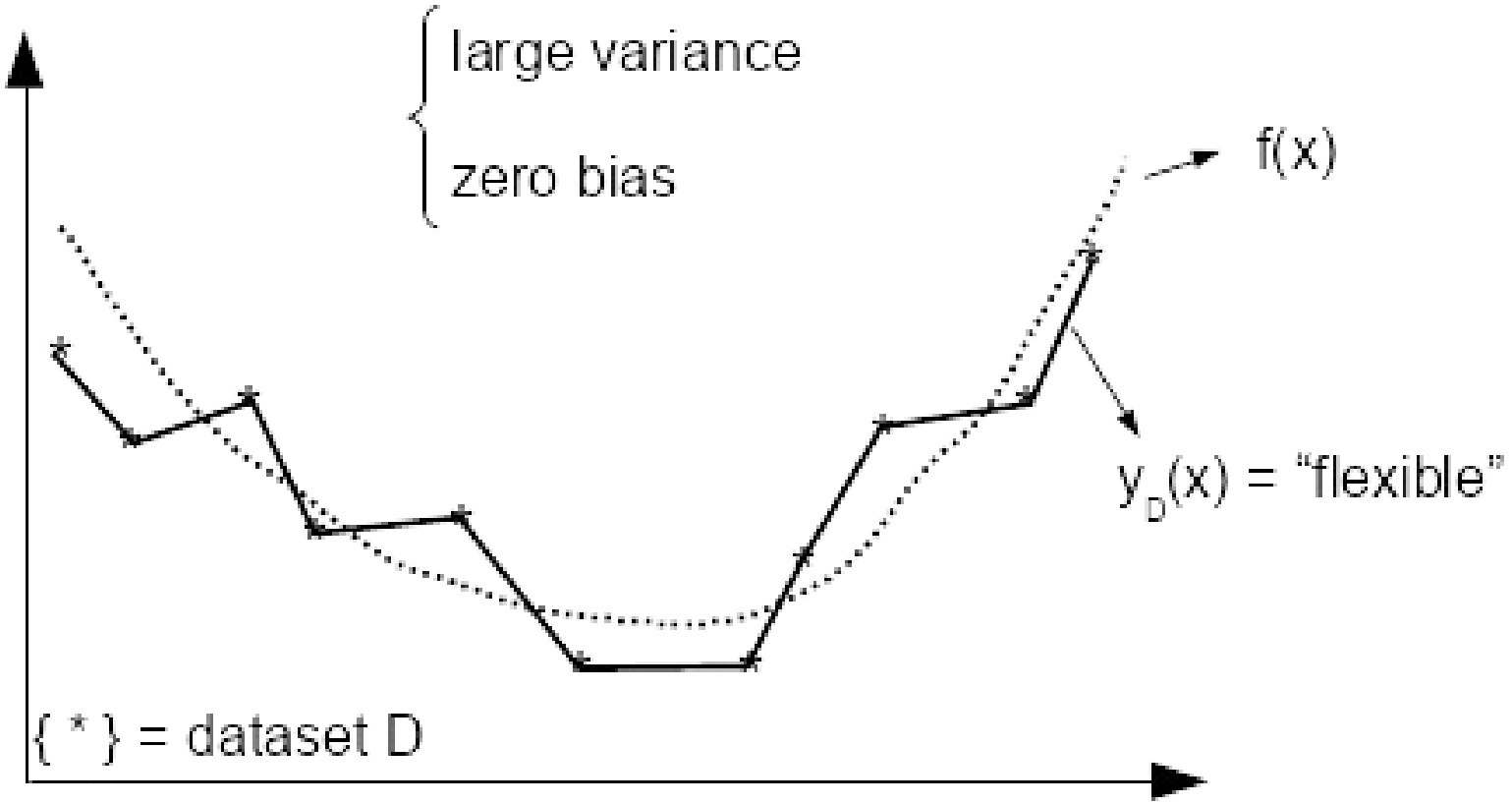
$\mathrm{E}[\cdot] = \text{Expectation over many } N\text{-sized data sets } D$

$$\mathrm{E}\left[\left(y_{\mathrm{D}}(\mathbf{x}) - f(\mathbf{x})\right)^2\right] =$$

$$= \underbrace{\left(\mathrm{E}\left[y_{\mathrm{D}}(\mathbf{x})\right] - f(\mathbf{x})\right)^2}_{\text{bias}^2} + \underbrace{\mathrm{E}\left[\left(y_{\mathrm{D}}(\mathbf{x}) - \mathrm{E}\left[y_{\mathrm{D}}(\mathbf{x})\right]\right)^2\right]}_{\text{variance}}$$



Underfitting situation



Overfitting situation

How do we find a balance between underfitting and overfitting?

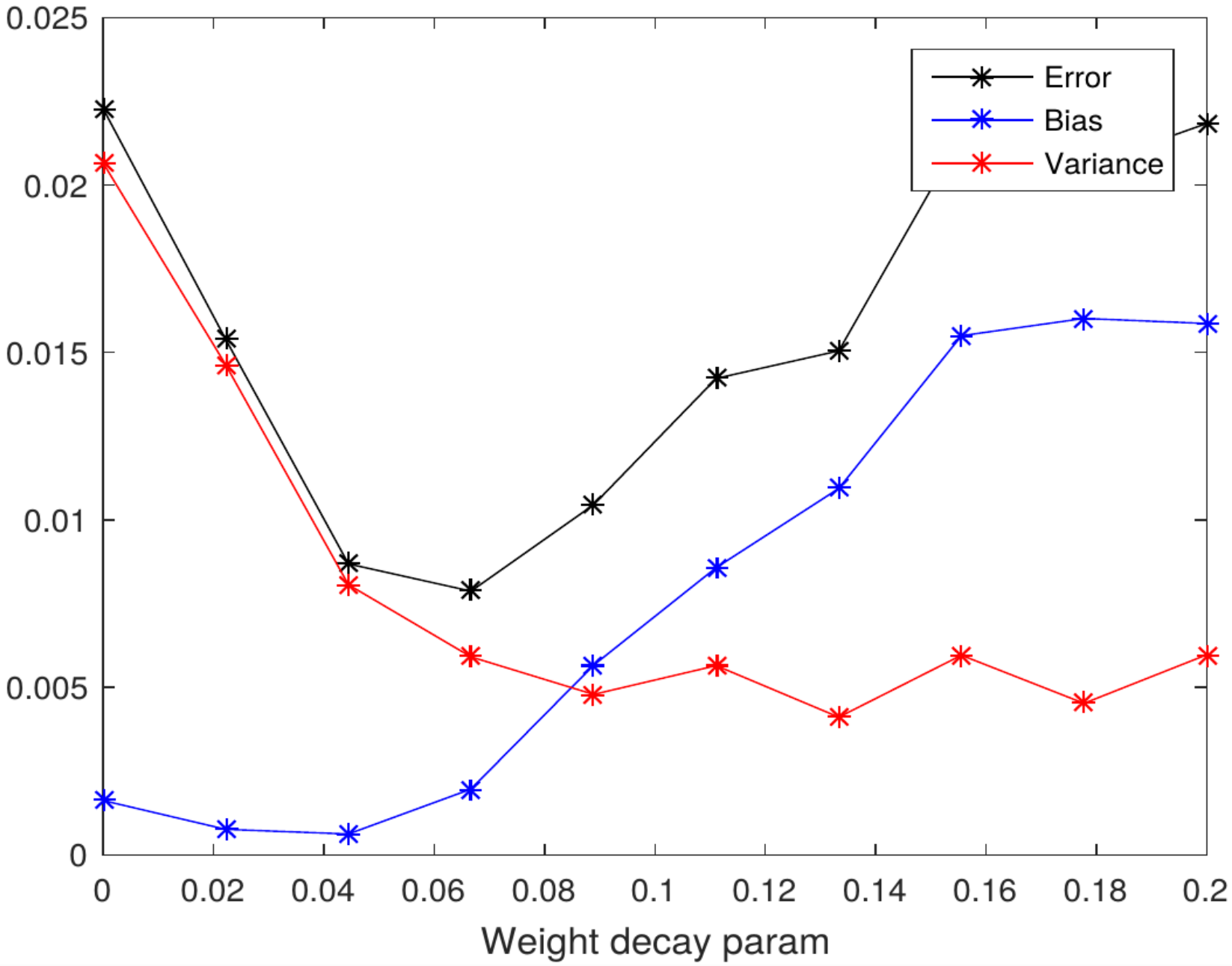
We want an efficient way of modifying the capacity of an ML model.

Regularization: Attempts to modify architectures, error functions or learning algorithms in order to reduce the generalization error

Example: L2 norm regularization

$$E(\mathbf{w}) = \mathrm{MSE} + \alpha||\mathbf{w}||^2$$

Simple regression problem



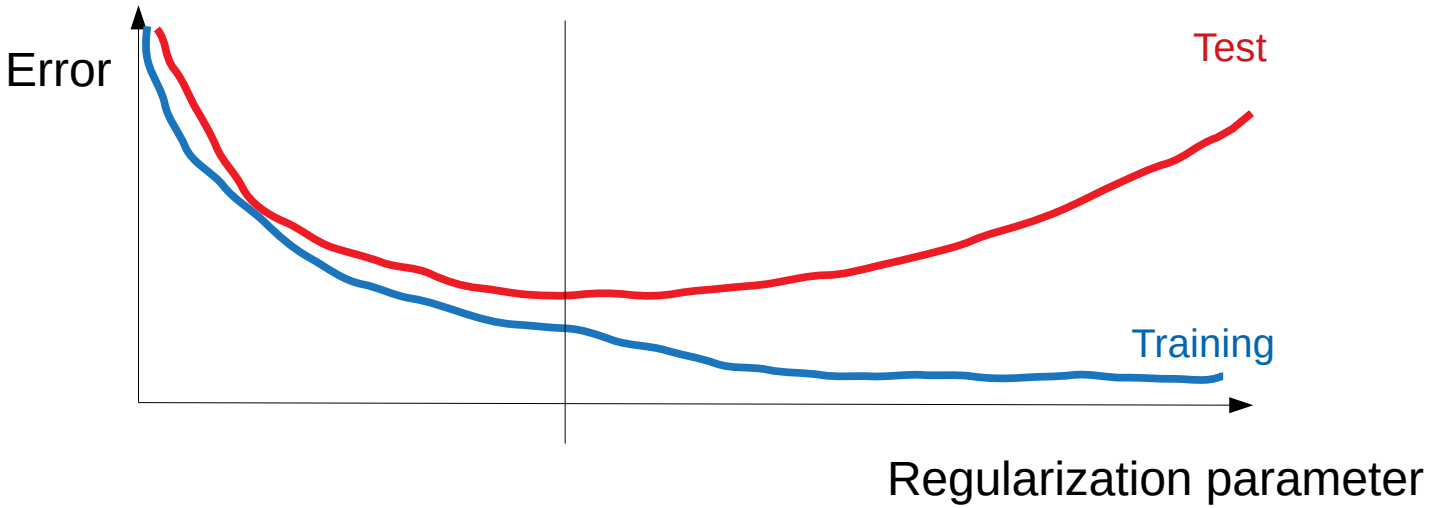
Hyper parameters and model selection

We have additional parameters that controls the ML model:

Size of the model: e.g. degree or number of hidden layers

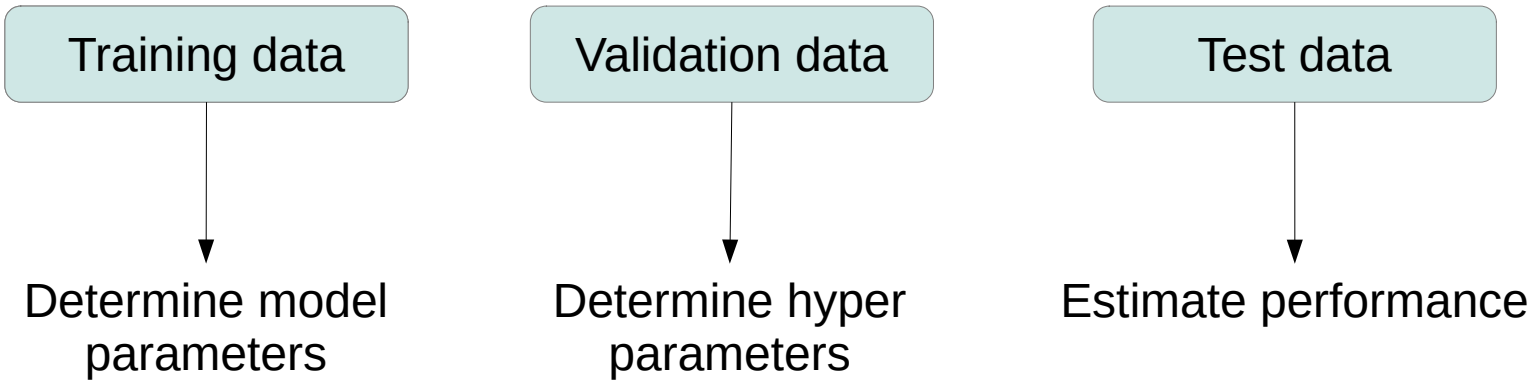
Regularization parameters: size of L2 norm or “dropout” parameter

Model selection = determination of hyper parameters



Model selection requires estimation
of generalization performance!!

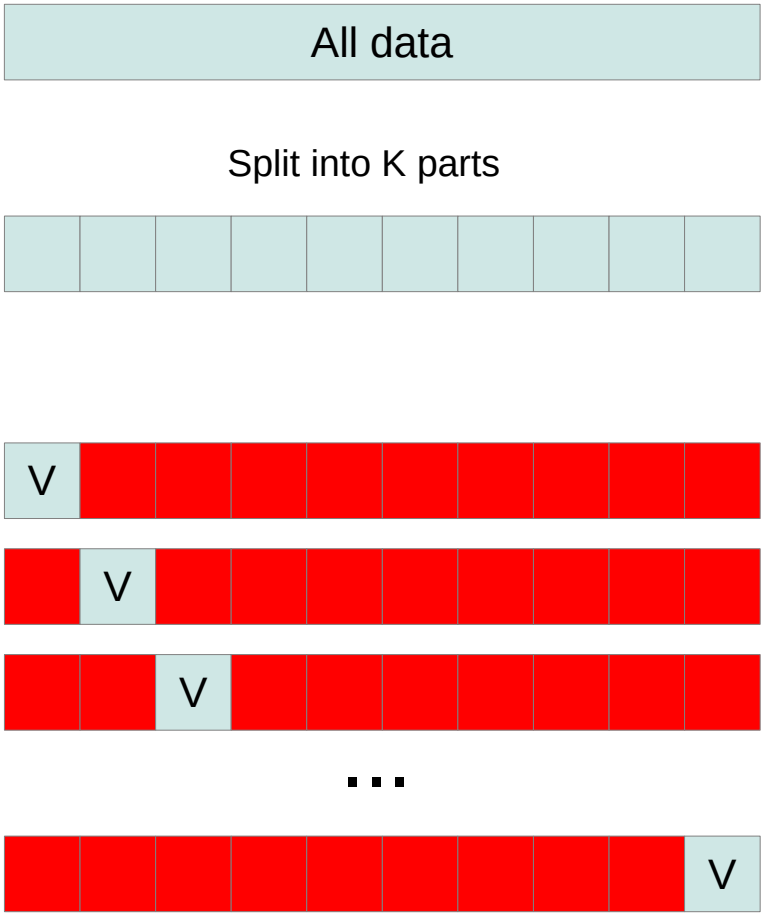
How can we do that?



Various approaches to estimate generalization performance

The holdout method

The K-fold cross validation method

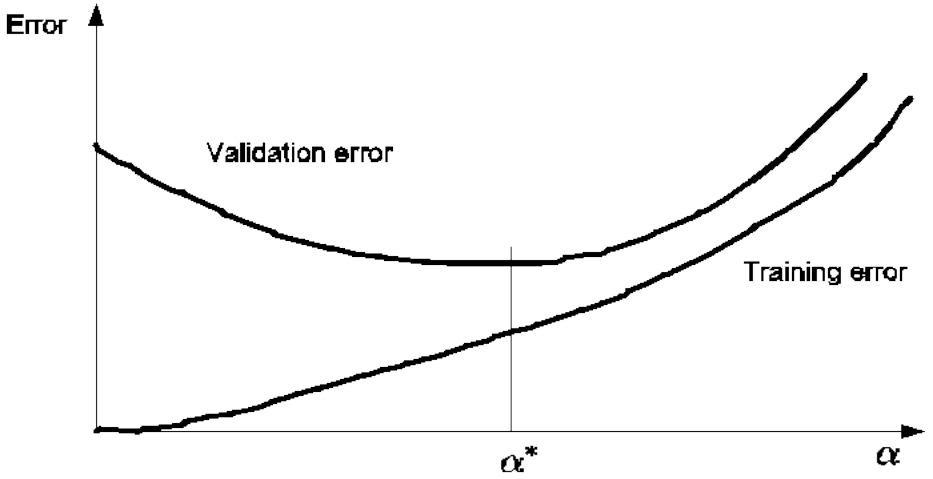


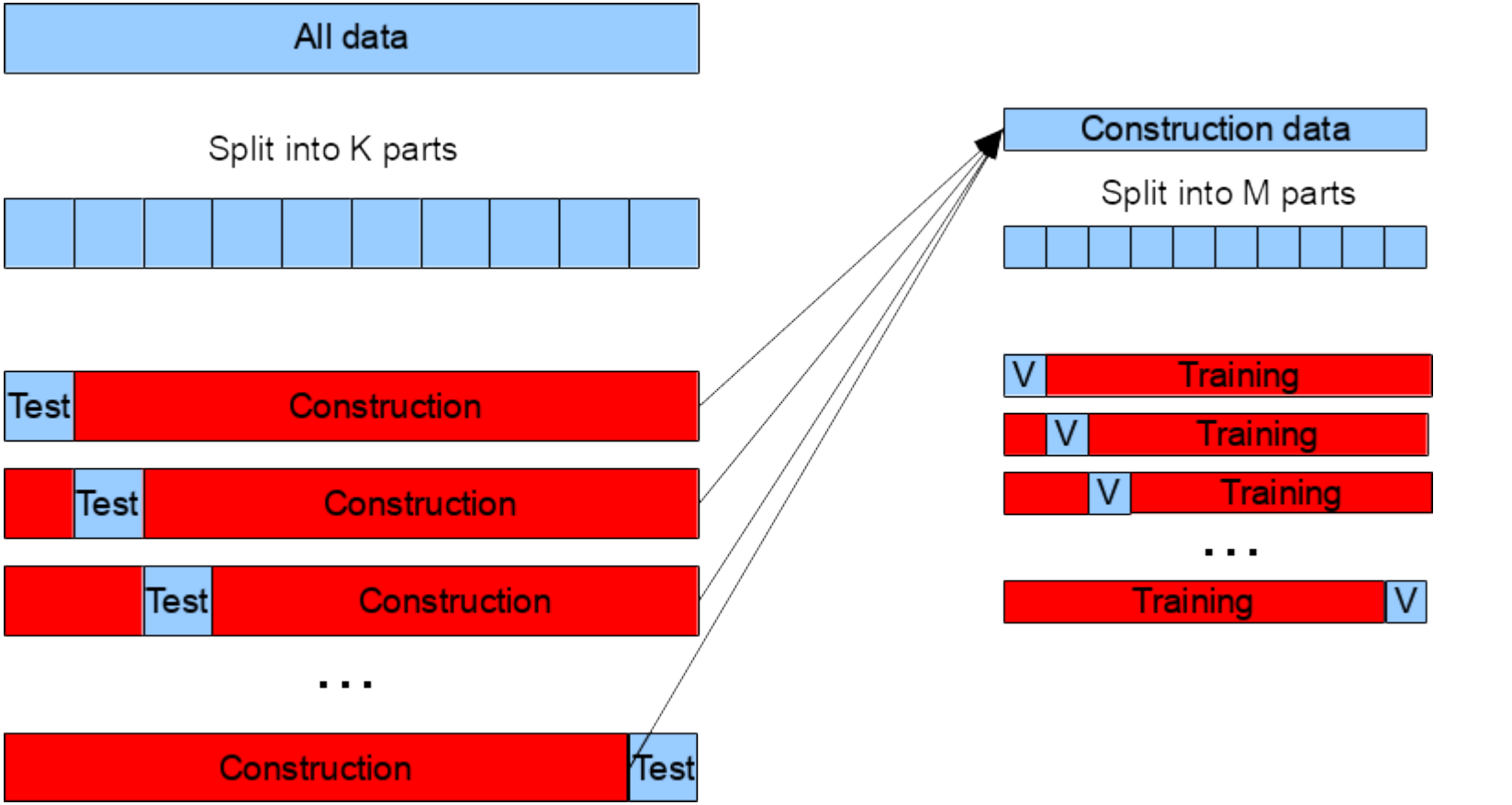
$$\hat{P} = \frac{1}{K} \sum_i P_i$$

Sometimes repeat *N* times

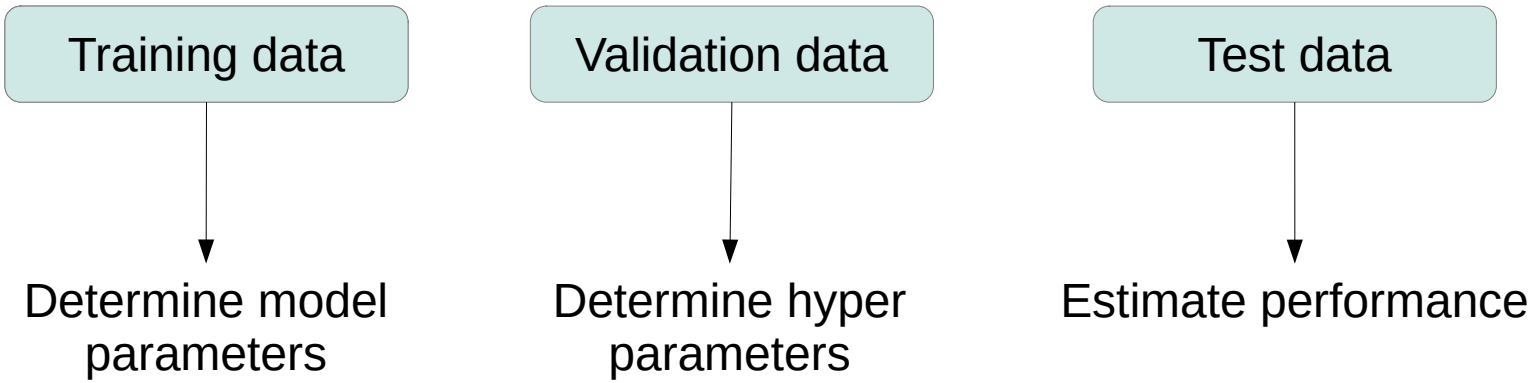
$$\hat{P} = \frac{1}{NK} \sum_n \sum_i P_{in}$$

Sometimes we need two loops!



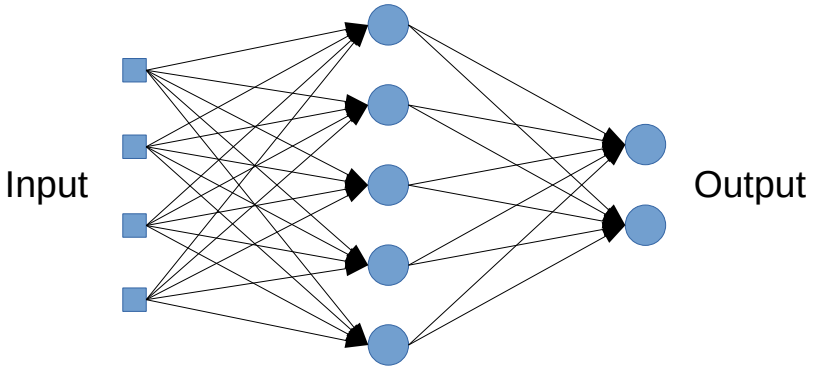


Important


Training data
Validation data
Test data
Determine model parameters
Determine hyper parameters
Estimate performance

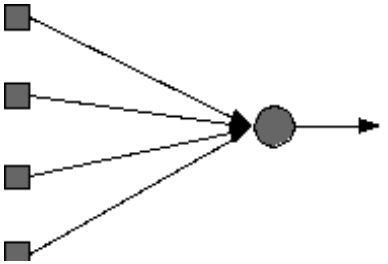
Practical considerations
(Test case the MLP)

- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

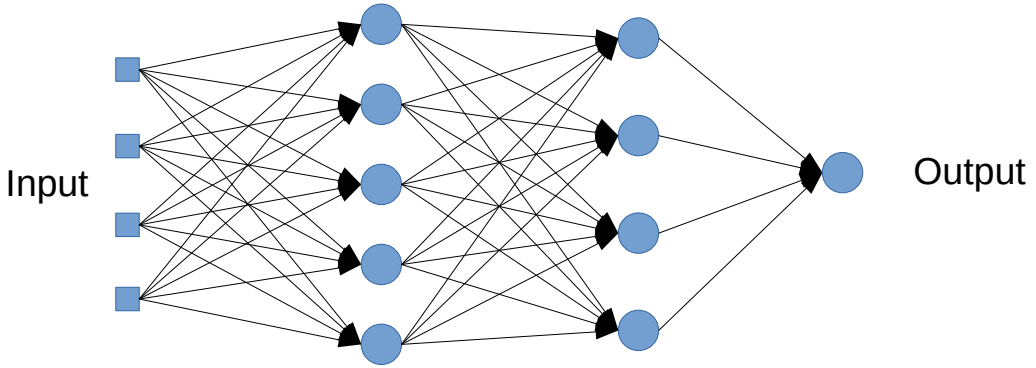


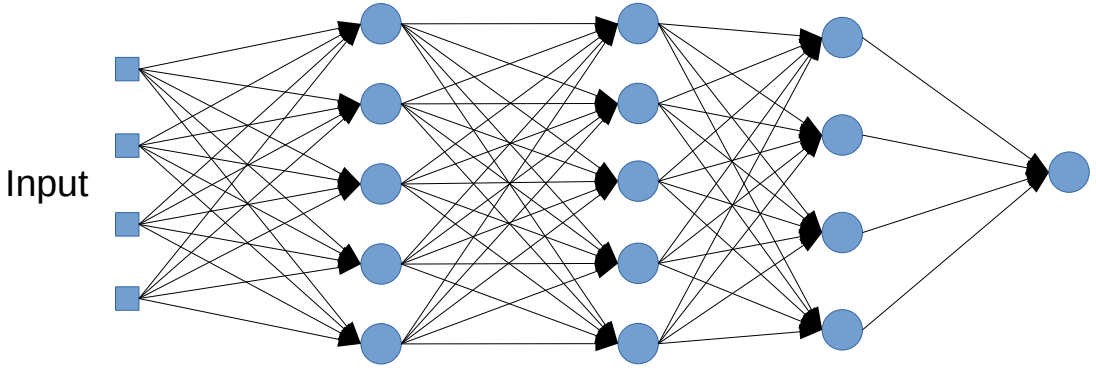
One hidden layer

NO hidden layers



Two hidden layers

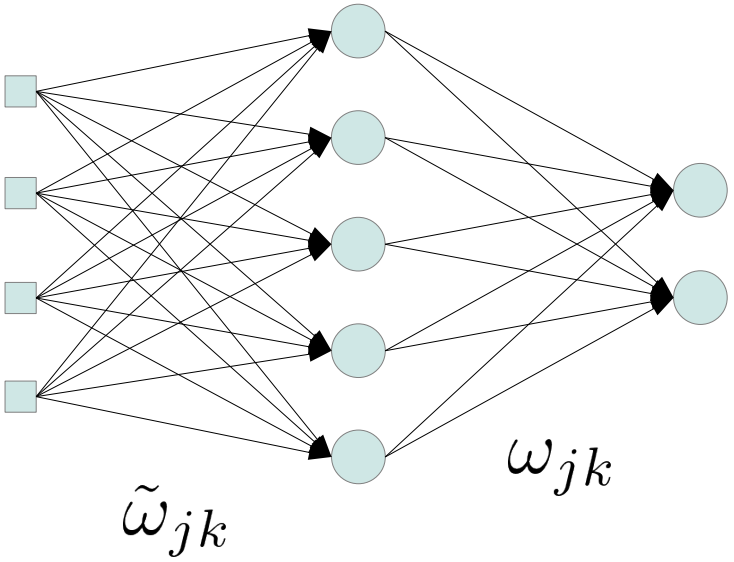




Three hidden layers (towards deep MLP)

Exasmple one hidden layer

Input x_k



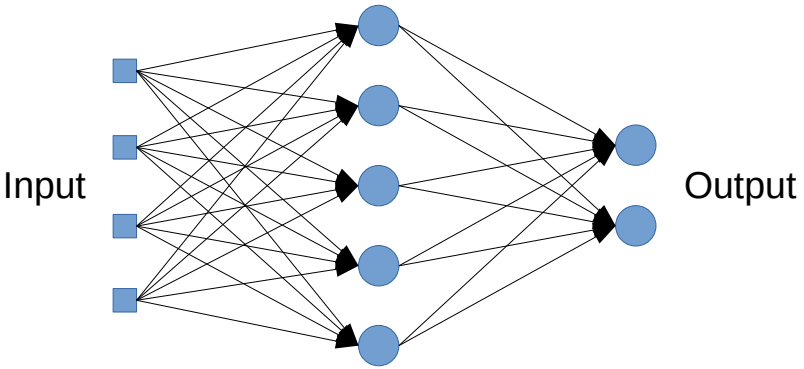
y_i Output

Remember: A mapping from “input” to “output”

$$y_i(\mathbf{x}) = \varphi_o\left(\sum_j \omega_{ij}\varphi_h\left(\sum_k \tilde{\omega}_{jk}x_k\right)\right)$$

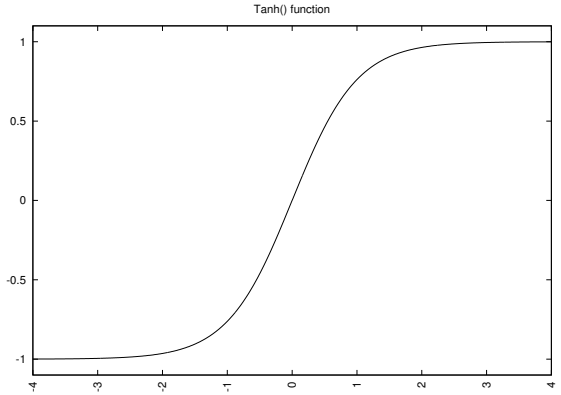
Tunable weights

It is important that we have **non-linear** activation functions in the hidden layer

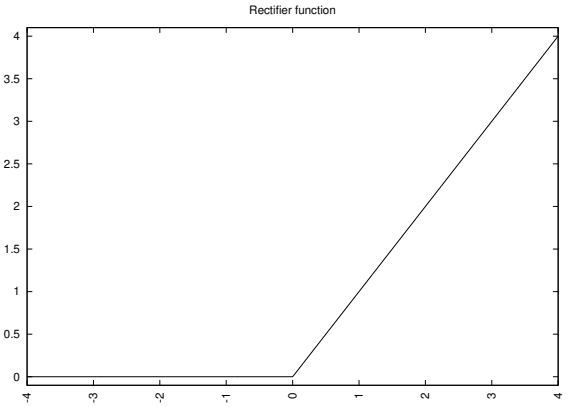


Common activation functions for the **hidden** layers

"Old and shallow"

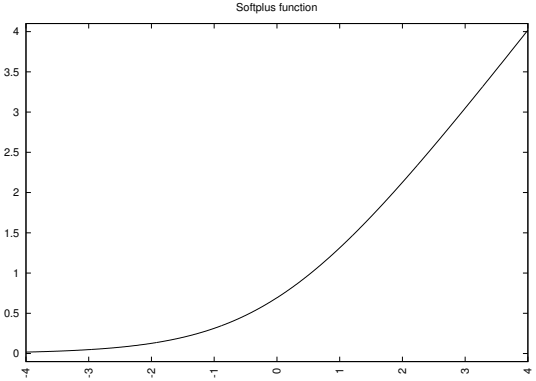


Tanh: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$



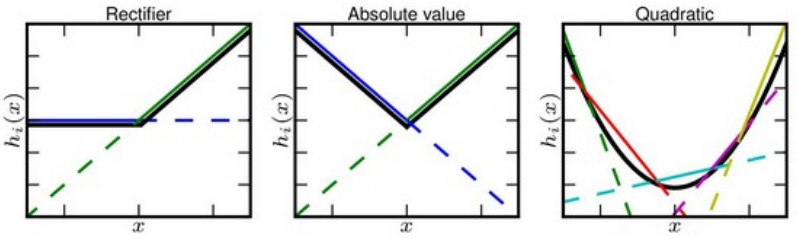
ReLU: $\max(0, x)$

"New and deep"



"Soft" ReLU

Also possible



Maxout units

Activation functions for the **output** layer

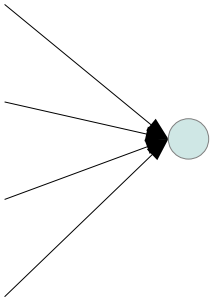
The MLP is mostly used for two kinds of tasks:

Classification: The algorithm is asked to predict one of *k* classes for which the input belongs to

Regression: Predict numerical outputs given an input

For classification problems we typically
make a distinction between binary
and multiple classification problems

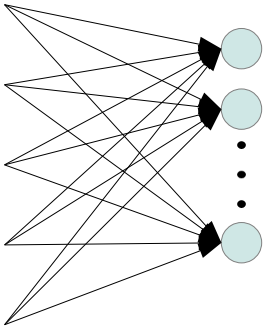
Binary



A single output node:

Class 0: target value = 0
Class 1: target value = 1

M classes

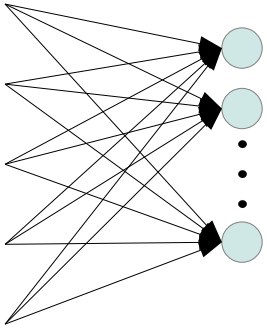


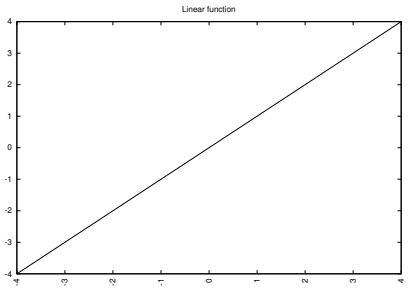
M output nodes:
One-hot-encoding

Class 1: [1 0 0 … 0]
Class 2: [0 1 0 … 0]
…
Class M: [0 0 0 … 1]

Output activation functions for the MLP

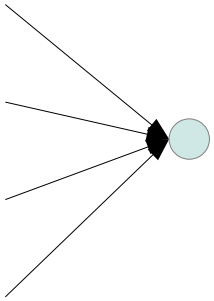
Regression

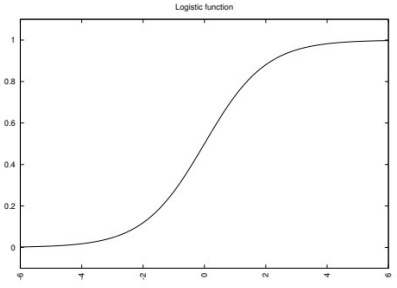




Linear

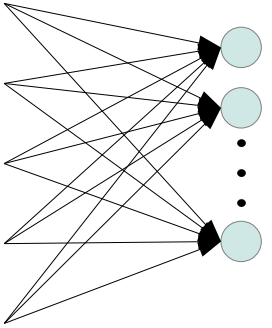
Binary classification





Logistic

M classes



Softmax

$$y_{ni} = \frac{e^{a_{ni}}}{\sum_{i'} e^{a_{ni'}}}$$

$a_{ni} = \text{net input to output node } i \text{ for pattern } n$

Practical considerations
(Test case the MLP)

- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

Some common loss functions

Regression problem

$$\begin{aligned} &\text{targets} && \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_N\} \\ &\text{inputs} && \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \end{aligned}$$

$$E(\theta) = \frac{1}{N} \sum_{n}^{N} \left(d_n - y(\mathbf{x}_n; \theta)\right)^2$$

Mean squared error

Binary classification problem

$$\begin{aligned} &\text{targets} && \{d_1, d_2, ..., d_N\},\ d_n \in \{0, 1\} \\ &\text{inputs} && \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \end{aligned}$$

$$E(\theta) = -\frac{1}{N} \sum_{n}^{N} \left[d_n \log y(\mathbf{x}_n; \theta) + (1 - d_n) \log \left(1 - y(\mathbf{x}_n; \theta)\right) \right]$$

Cross entropy error

M-class classification problem

$$\begin{aligned} &\text{targets} && \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_N\},\ \mathbf{d}_n \in \text{one-hot-encoding} \\ &\text{inputs} && \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \end{aligned}$$

$$E() = -\frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{M} d_{ni} \log y_{ni}$$

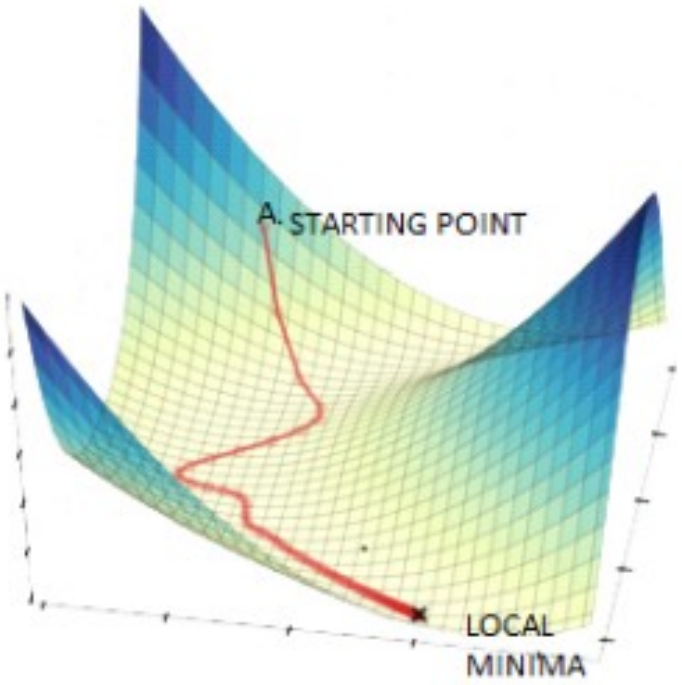
Cross entropy error

Practical considerations
(Test case the MLP)

- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

Stochastic gradient descent

Training an ML model, especially neural network models,
involves minimizing the loss function with respect to model parameters



Very often one have to rely on numerical minimization procedures. The most common approach is ***gradient descent*** based methods

$$\Delta\theta_i = -\eta \frac{\partial E(\theta)}{\partial \theta_i}$$

Now very often $$E(\theta) = \frac{1}{N}\sum_{n}^{N} E_n(\theta)$$

We can write $$\Delta\theta_i = \frac{1}{N}\sum_{n} \Delta\theta_{ni}$$

per pattern update

Stochastic gradient descent (SGD)

We can approximate the gradient with a smaller set of patterns

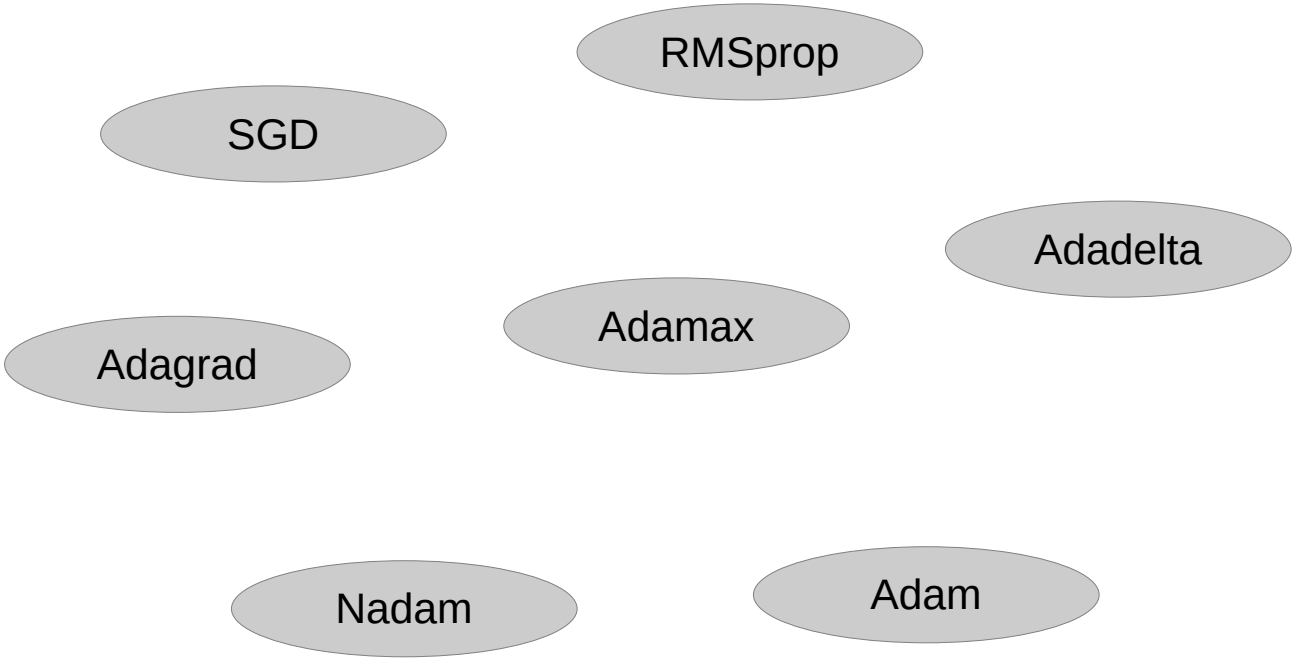
$$\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\} \qquad \Delta\theta_i \approx \frac{1}{m}\sum_k^m \Delta\theta_{ki}$$

$$m \ll N$$

Each set $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$ is called a **minibatch** of patterns.

There are many methods centered around the SGD idea!!

Many different methods



Practical considerations

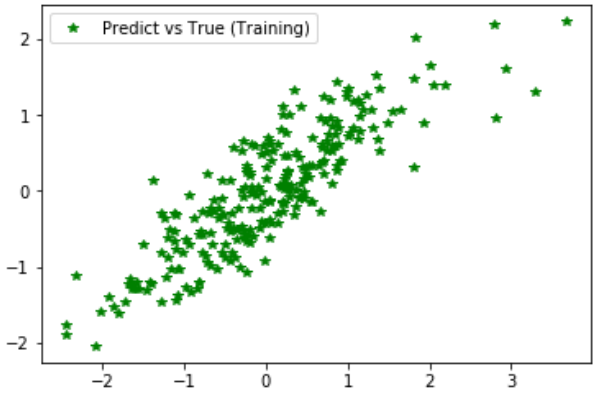
- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

How to measure performance – regression problems

RMSE $$E = \sqrt{\frac{1}{N}\sum_{n}^{N}\left\|\mathbf{d}_n - \mathbf{y}(\mathbf{x}_n, \boldsymbol{\omega}^*)\right\|^2}$$

Normalized MSE $$E = \frac{\sum_n \left\|\mathbf{d}_n - \mathbf{y}(\mathbf{x}_n, \boldsymbol{\omega}^*)\right\|^2}{\sum_n \left\|\mathbf{d}_n - < \mathbf{d} >\right\|^2}$$

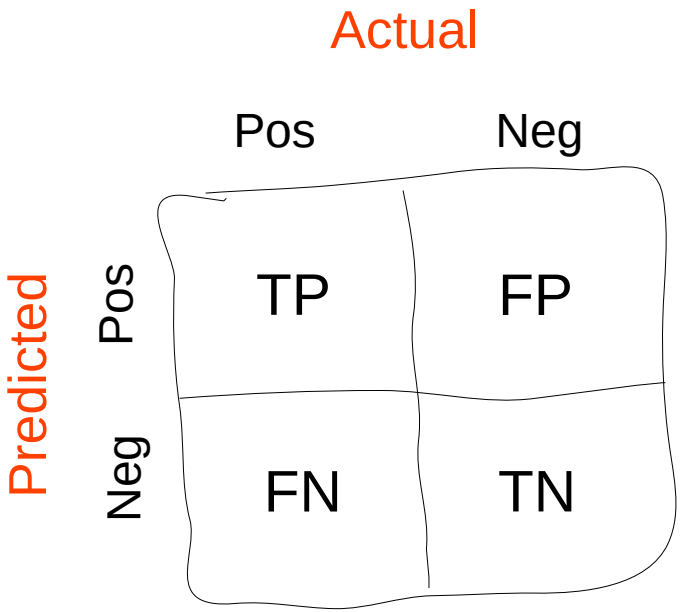
Scatterplot
True vs. predicted



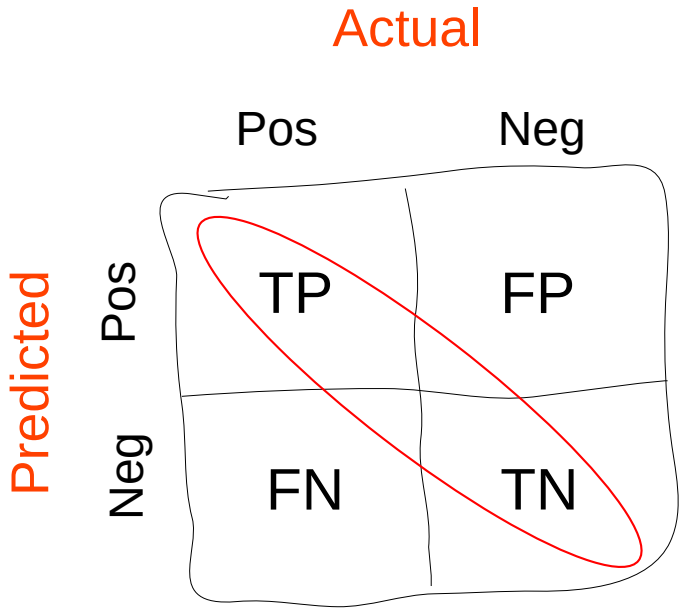
Compute correlation!

How to measure performance – binary classification problems

Confusion matrix

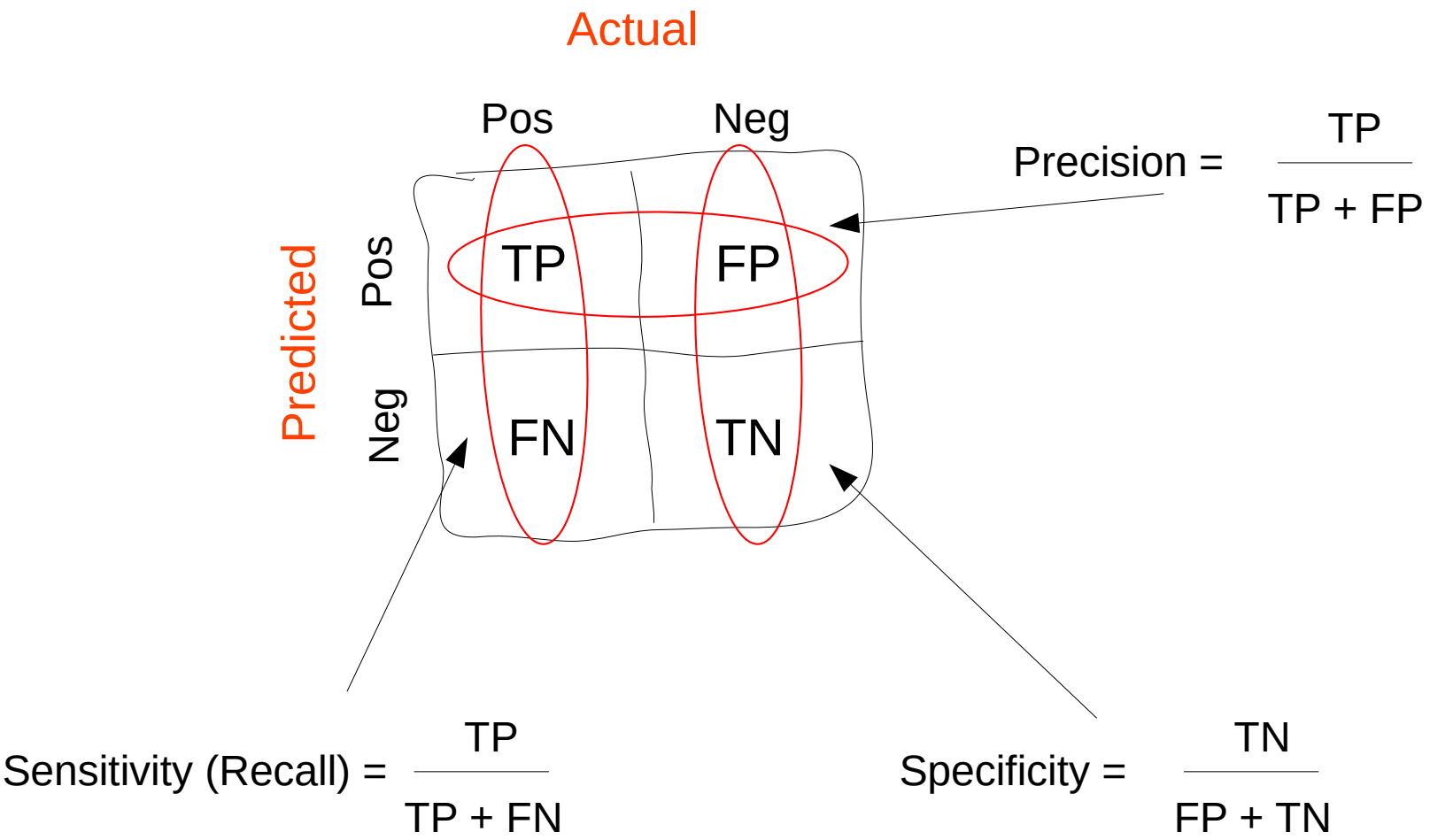


TP = True Positives
TN = True Negatives
FP = False Positives
FN = False Negatives

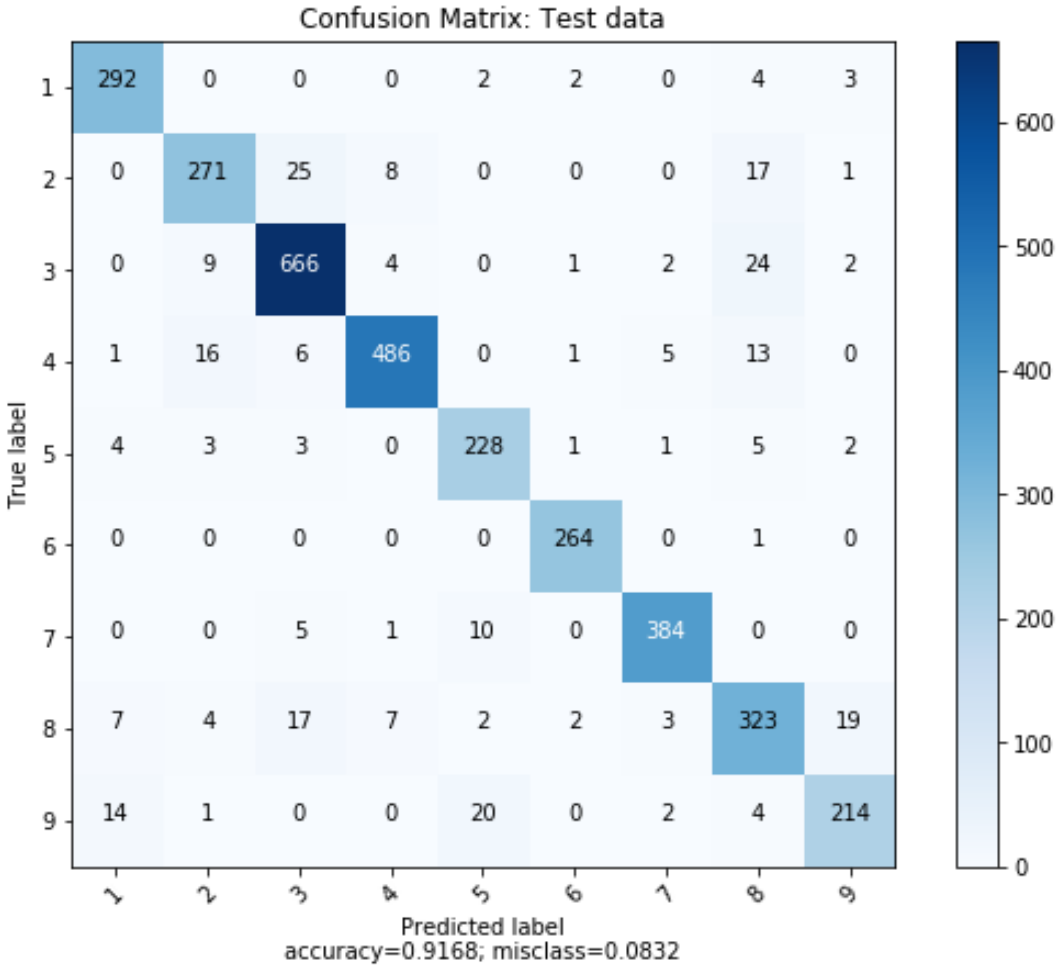


$$\text{Accuracy} = \frac{TP + TN}{TP+FP+FN+TN}$$

Why can accuracy be misleading?


Actual
Pos
Neg
Predicted
Pos
Neg
TP
FP
FN
TN
Precision = TP / (TP + FP)
Sensitivity (Recall) = TP / (TP + FN)
Specificity = TN / (FP + TN)

Confusion matrix, also for many classes



For binary classification problems it is common to use the Receiver Operating Characteristics (ROC) curve and the area under it (AUC)

To make a decision for a class, we need a cut value (C)

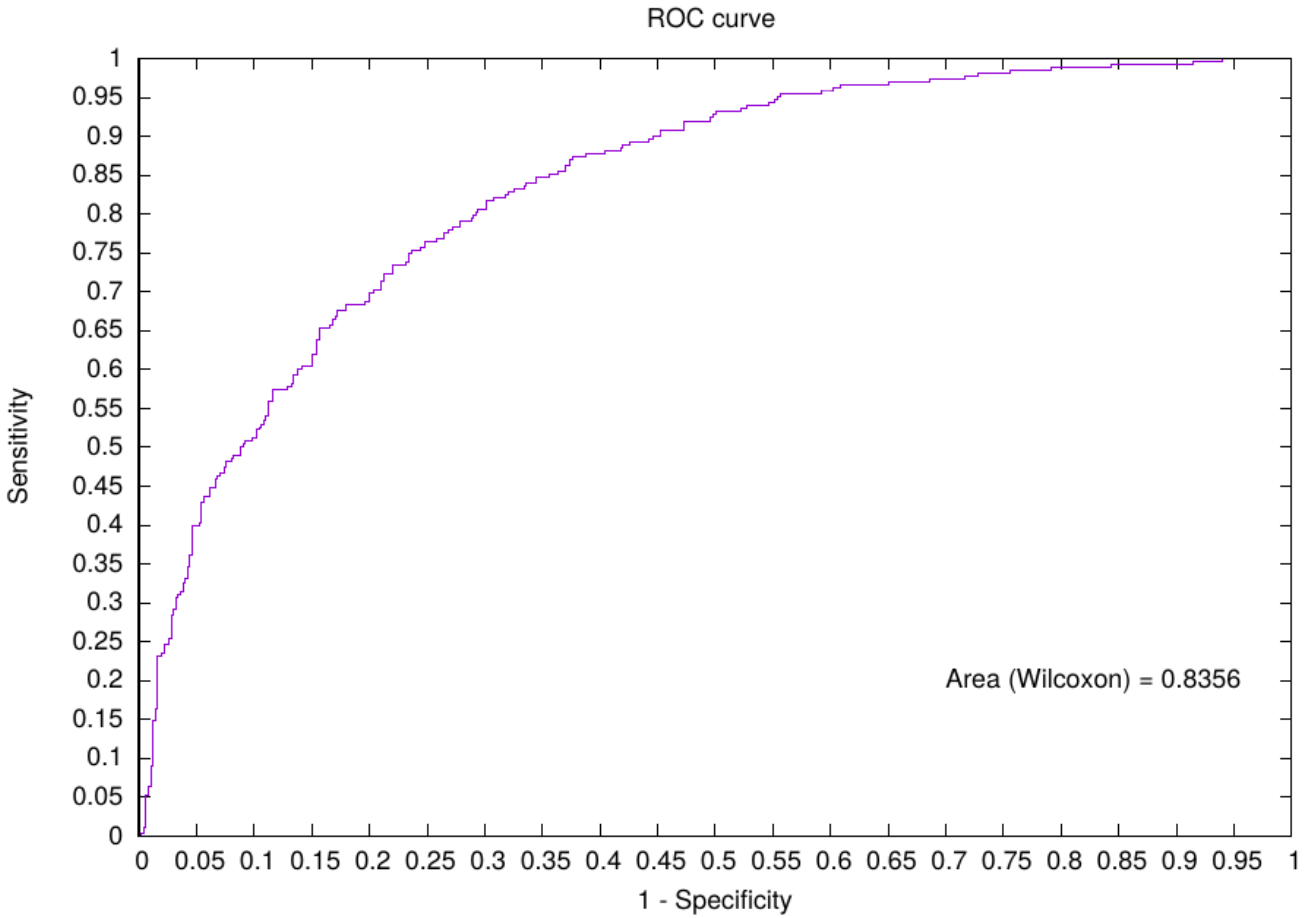
$$\left\{ \begin{array}{lll} \text{class 1 (pos)} & \text{if} & y(\mathbf{x}) > \mathrm{C} \\ \text{class 0 (neg)} & \text{if} & y(\mathbf{x}) \leq \mathrm{C} \end{array} \right\}$$

For each C we get a Sensitivity / Specificity pair

Vary C between [0,1] and plot all Sens vs (1-Spec)

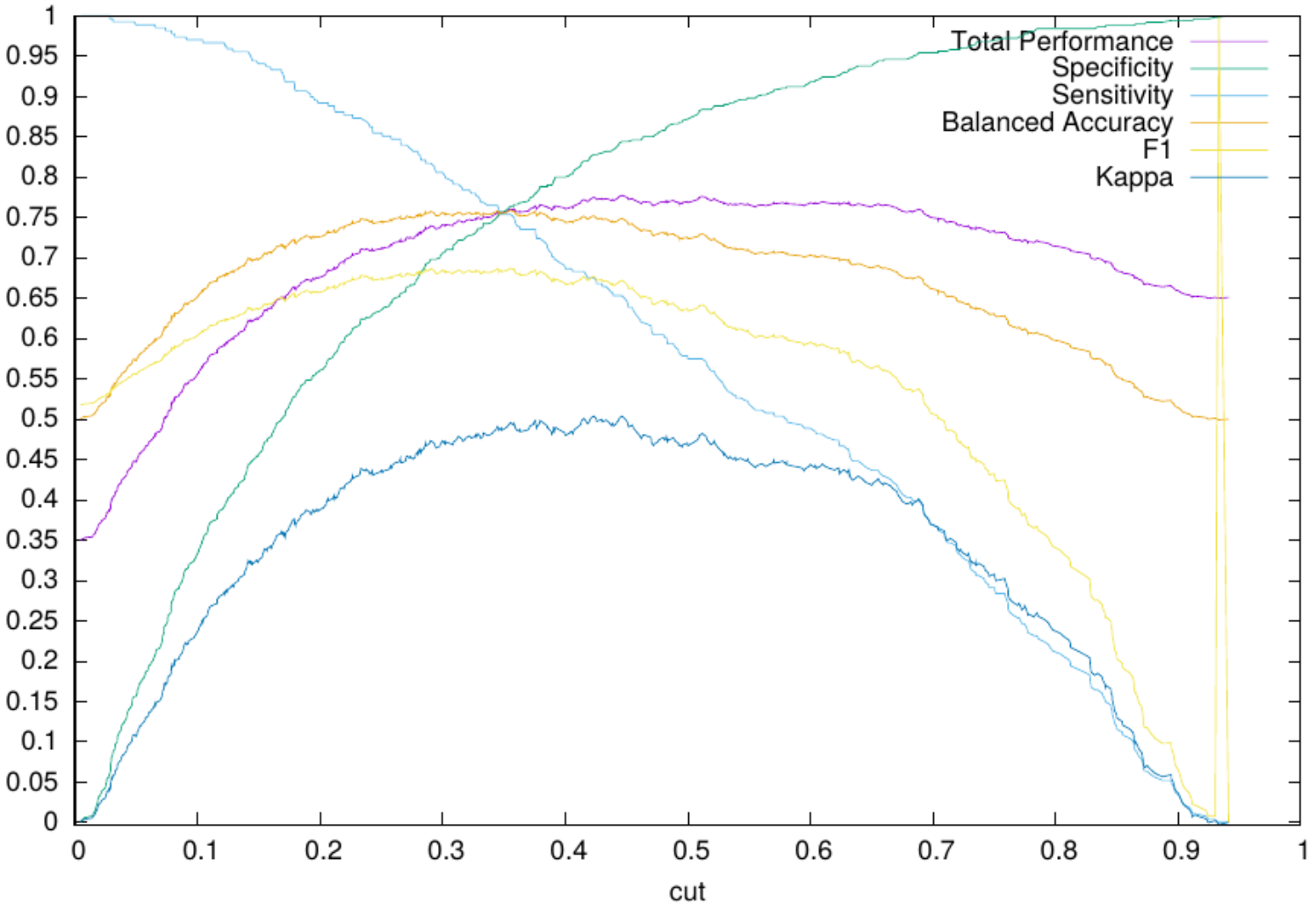
This is the ROC curve!

An example



What does the area mean?

We can plot more things!

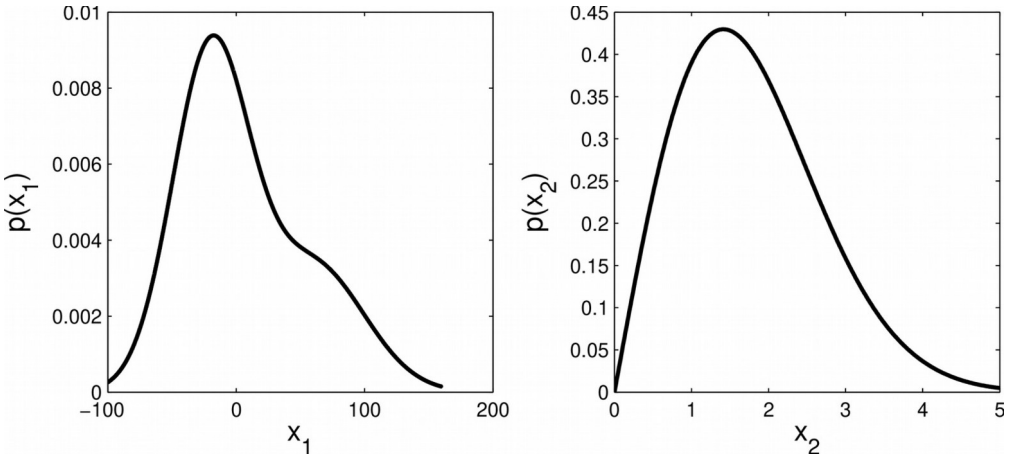


Practical considerations

- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

Pre-processing of input data

We need to compensate for different input sizes



Compute

Mean $\mu_k = \frac{1}{N}\sum_{n=1}^{N} x_{nk}$ *Std* $\sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_{nk}-\mu_k\right)^2}$

Transform

$$x_{nk} \rightarrow \frac{x_{nk}-\mu_k}{\sigma_k} \quad \forall\, n,k$$

More pre-processing of input data

- Missing data imputation
- Encoding
- Dimensionality reduction
- Feature selection
- ...

Encoding

Feature:	**Input**:
Numerical value	Numerical value
Binary category	Often 0/1 encoding
Many categories < 5 years [5, 10] years > 10 years	Often one-hot-encoding [1 0 0] [0 1 0] [0 0 1]
Text	“Many possibilities”, e.g. word2vec
….	….

More pre-processing of input data

- Missing data imputation
- Encoding
- Dimensionality reduction
- Feature selection
- ...

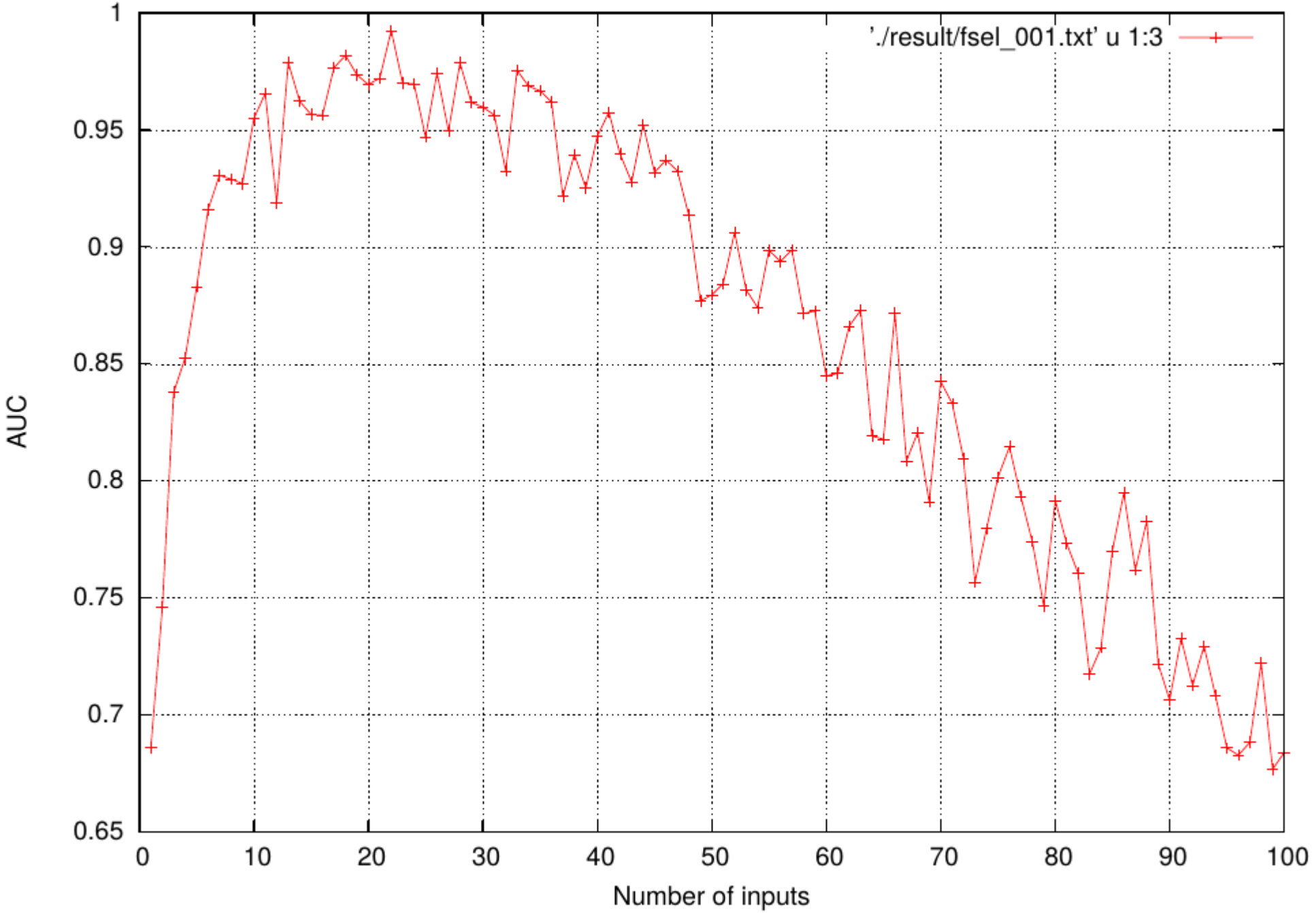
How can we reduce the input dimensionality?

- PCA (Pros/Cons?)
-
- Single variable hypothesis testing (Pros/Cons?)

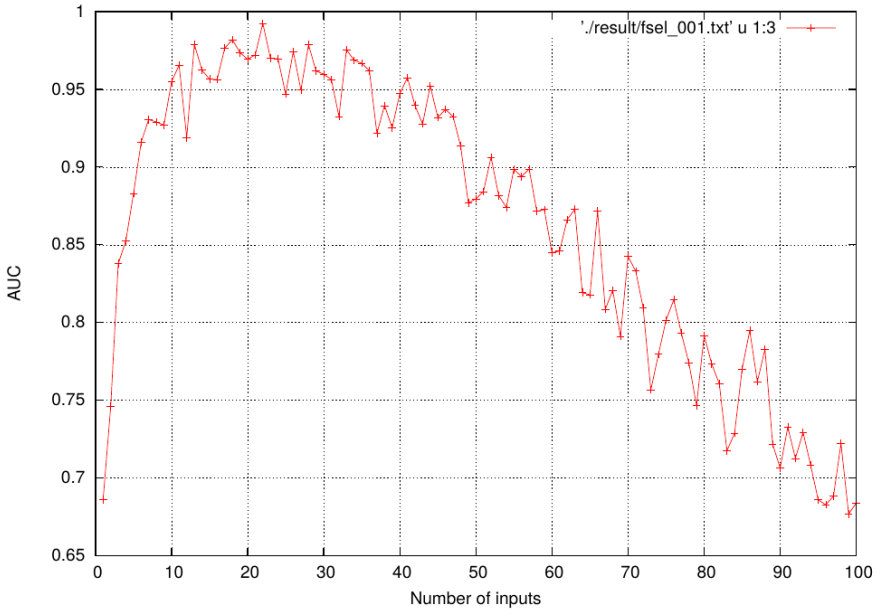
- Autoencoder (Pros/Cons?)
-
- Do nothing (Pros/Cons?)
-
- “Model driven feature selection (Pros/Cons?)
  (Example of backward elimination)

Example:

- Binary classification problem
- Small dataset (50)
- Large number of inputs (100)
- Use 5-fold cross-validation to find optimal number of inputs (backward elimination)
- Small model (5 hidden nodes)


'./result/fsel_001.txt' u 1:3
AUC
Number of inputs
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0
10
20
30
40
50
60
70
80
90
100

20 inputs is optimal!



The problem is that all variables
in this problem was random numbers!!

Inputs were random numbers
Target values were random numbers (0/1)

More pre-processing of input data

- Missing data imputation
- Encoding
- Dimensionality reduction
- Feature selection
- ...

Missing data imutation

Single imputation methods

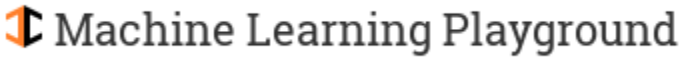
- Mean imputation (warning)
- KNN methods
- Autoencoder
- ”Many more”

Multiple imputation methods

- Random imputation
- MICE
- Variational Autoencoder
- “Many more”

Demo time

Machine learning playgrounds



http://ml-playground.com/

https://playground.tensorflow.org/

More details on Convolutional Neural Networks

To understand the construction of a CNN we need to introduce filters/kernels

$$I(i,j) = \text{Image}$$
$$K(m,n) = \text{Kernel}$$

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, i-n) K(m,n)$$

Filtered image

Here is a small numerical example where a 4x4 image matrix is filtered by a 2x2 kernel.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} * \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (-1+2+0+6) & (-2+3+0+7) & (-3+4+0+8) \\ (-5+6+0+10) & (-6+7+0+11) & (-7+8+0+12) \\ (-9+10+0+14) & (-10+11+0+15) & (-11+12+0+16) \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 8 & 9 \\ 11 & 12 & 13 \\ 15 & 16 & 17 \end{bmatrix}$$

The box average kernel

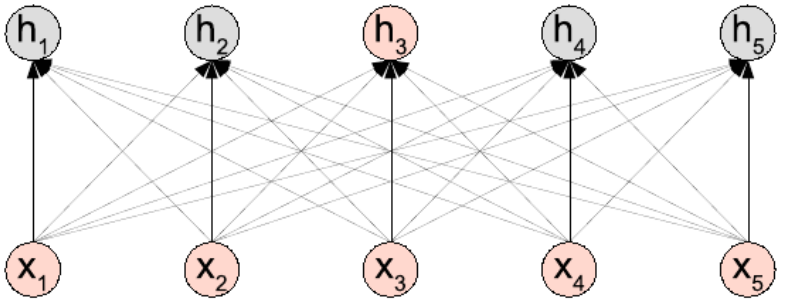
$$\text{box average} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Have fun with kernels!

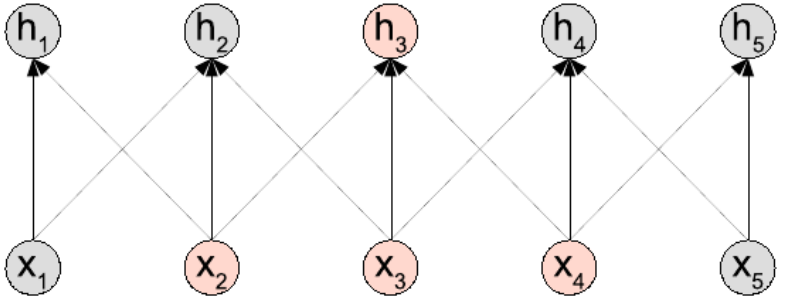
http://setosa.io/ev/image-kernels/

Sparse connectivity and parameter sharing

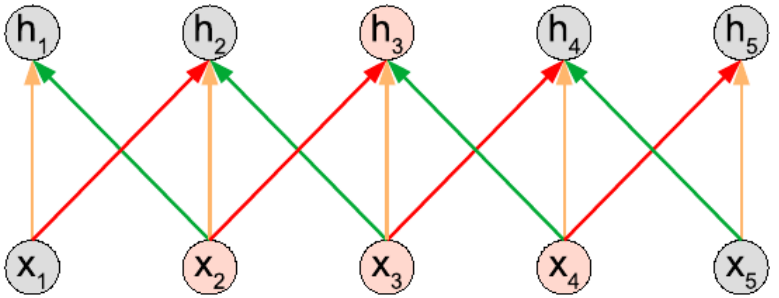
Fully connected MLP



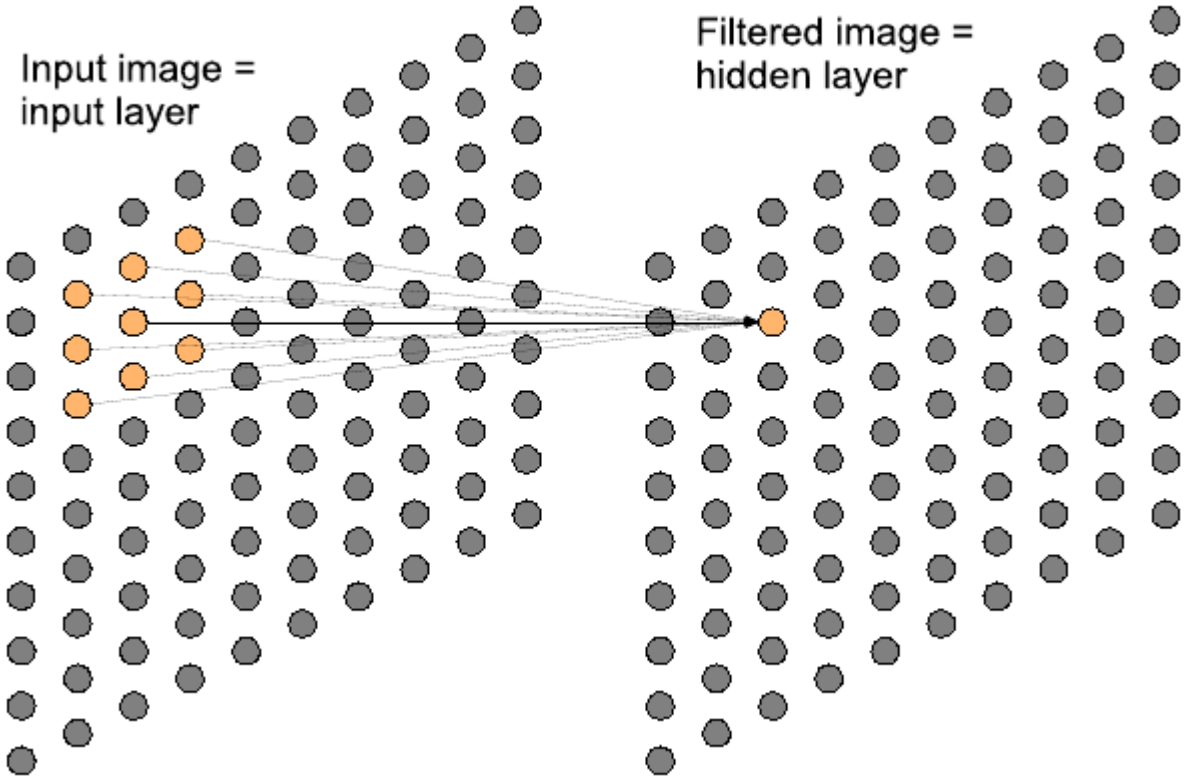
Sparsely connected



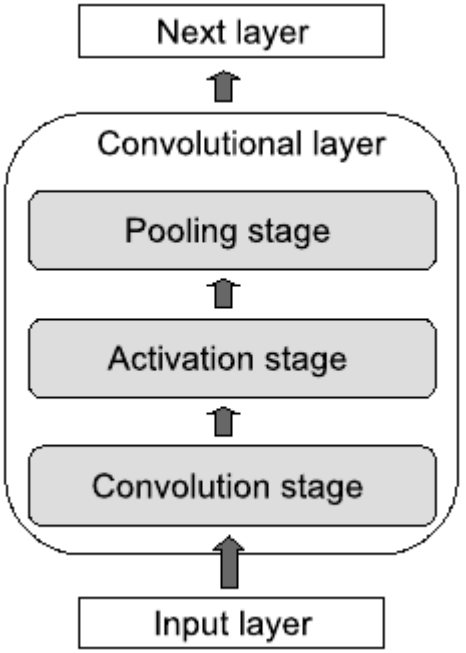
Sparsely connected +
shared weights = 1D kernel



For images the layers represent different (filtered) images

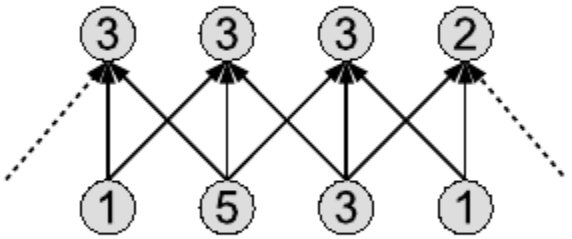


CNN building blocks



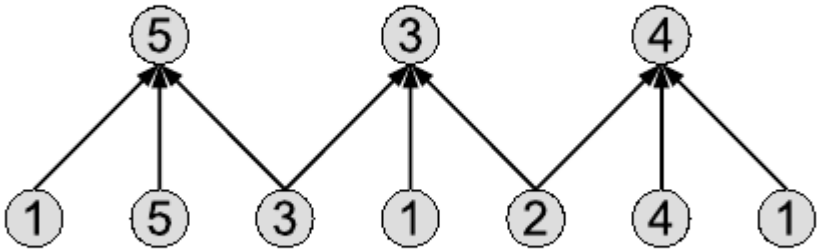
Most common

Average pooling layer



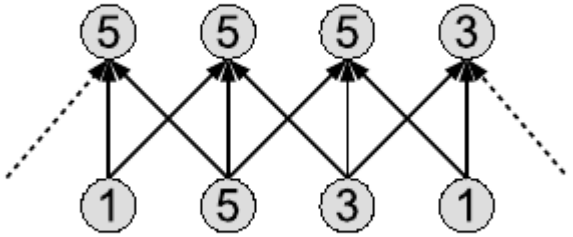
Hidden layer

Max pooling, with downsampling



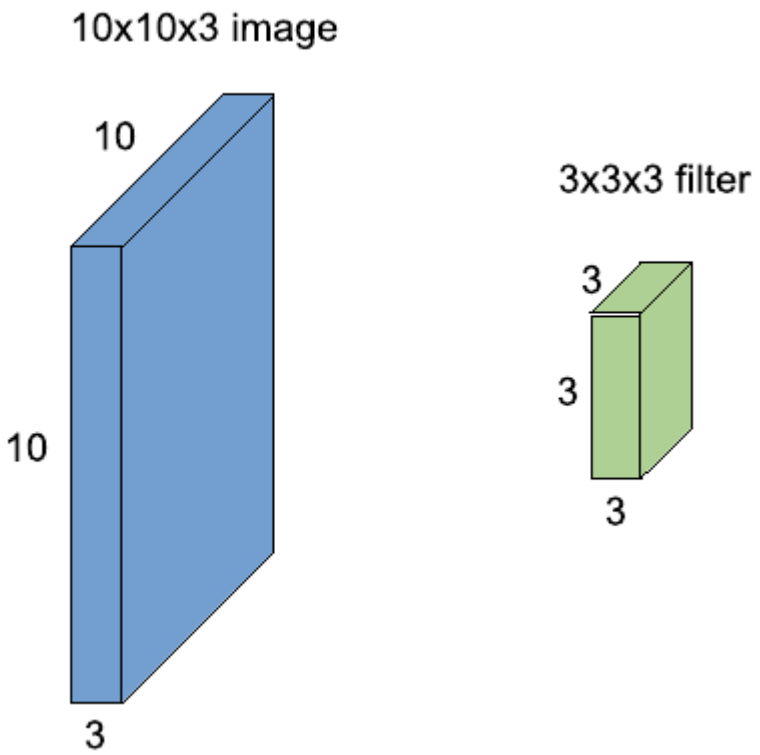
Hidden layer

Max pooling layer



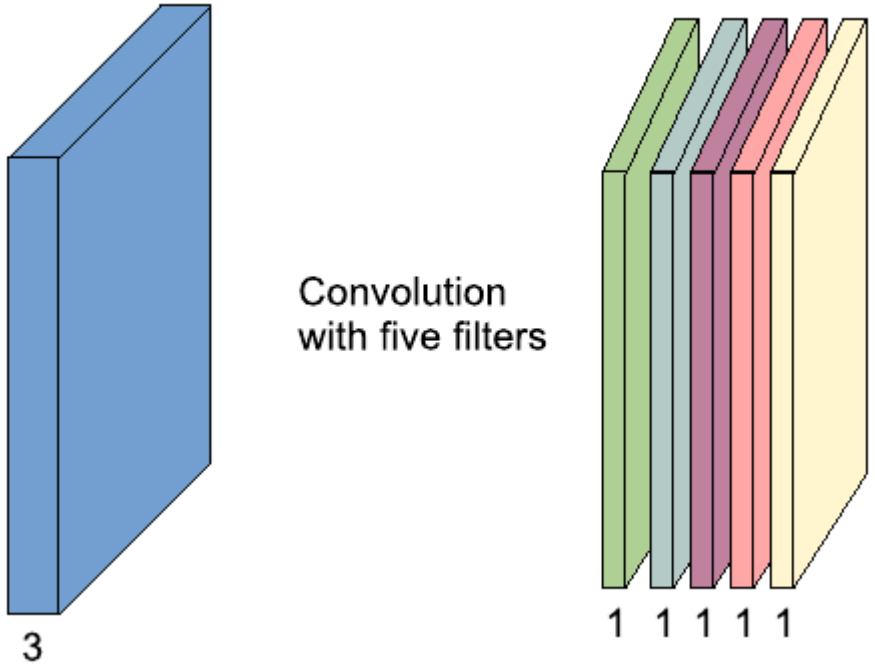
Hidden layer

Multi Channel images

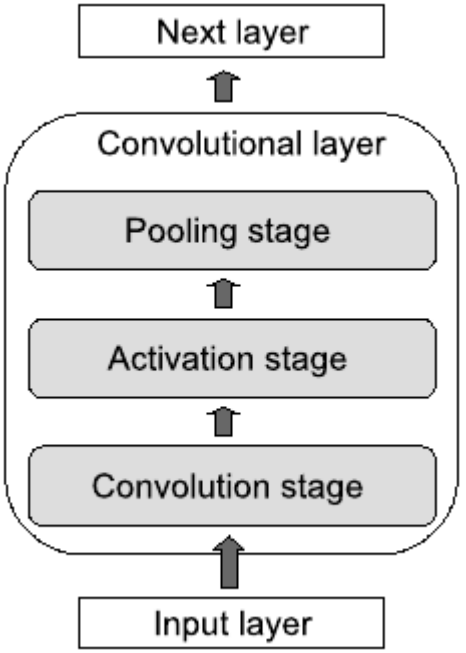


A 10x10x3 images and a filter of size 3x3x3. The important thing to remember is that filters always extends to the full depth on the input image.

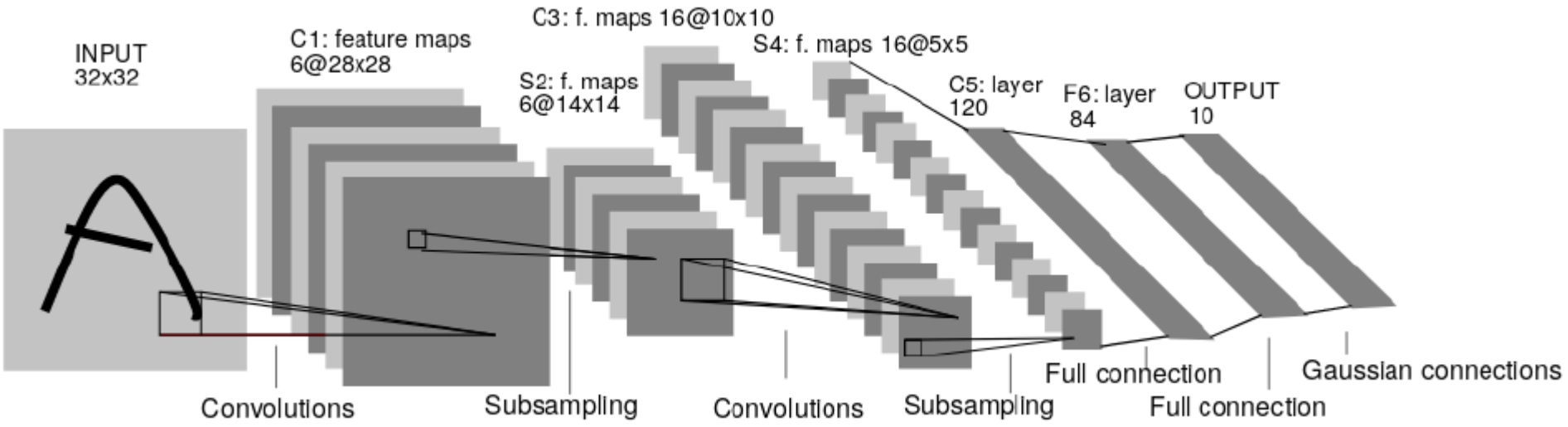
Each filter will produce one filtered image (channel)



Again the building blocks

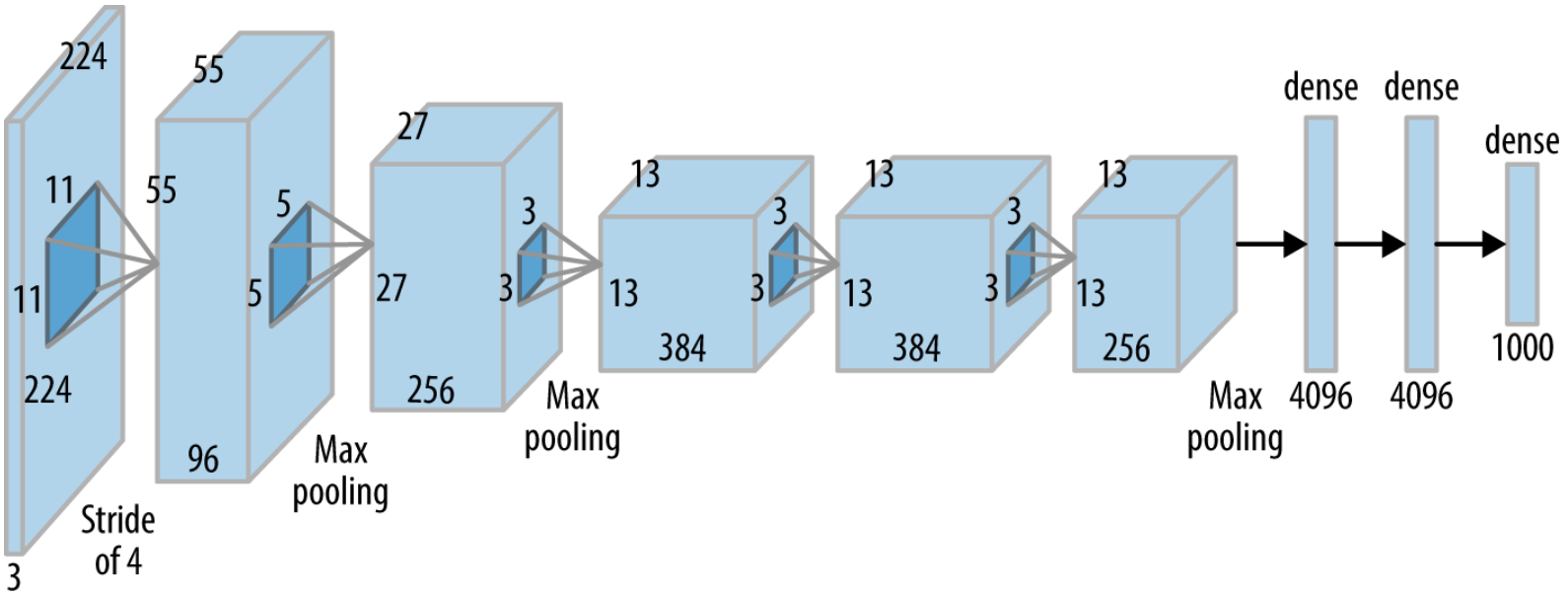


One of the very first CNNs (Lecun, 1998). This one is called LeNet-5.



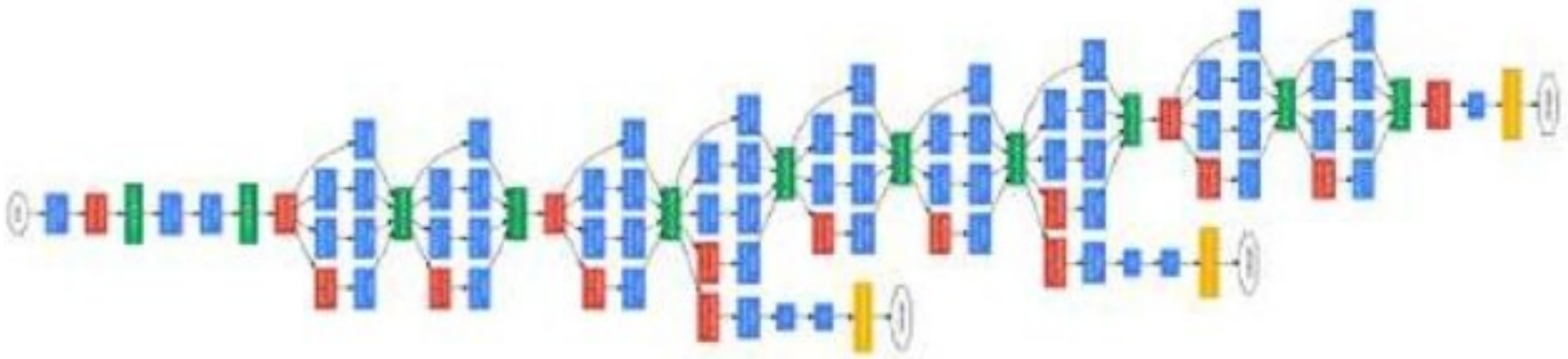
Some examples of “famous” CNNs

Alexnet, Winner of ILSVRC competition 2012.

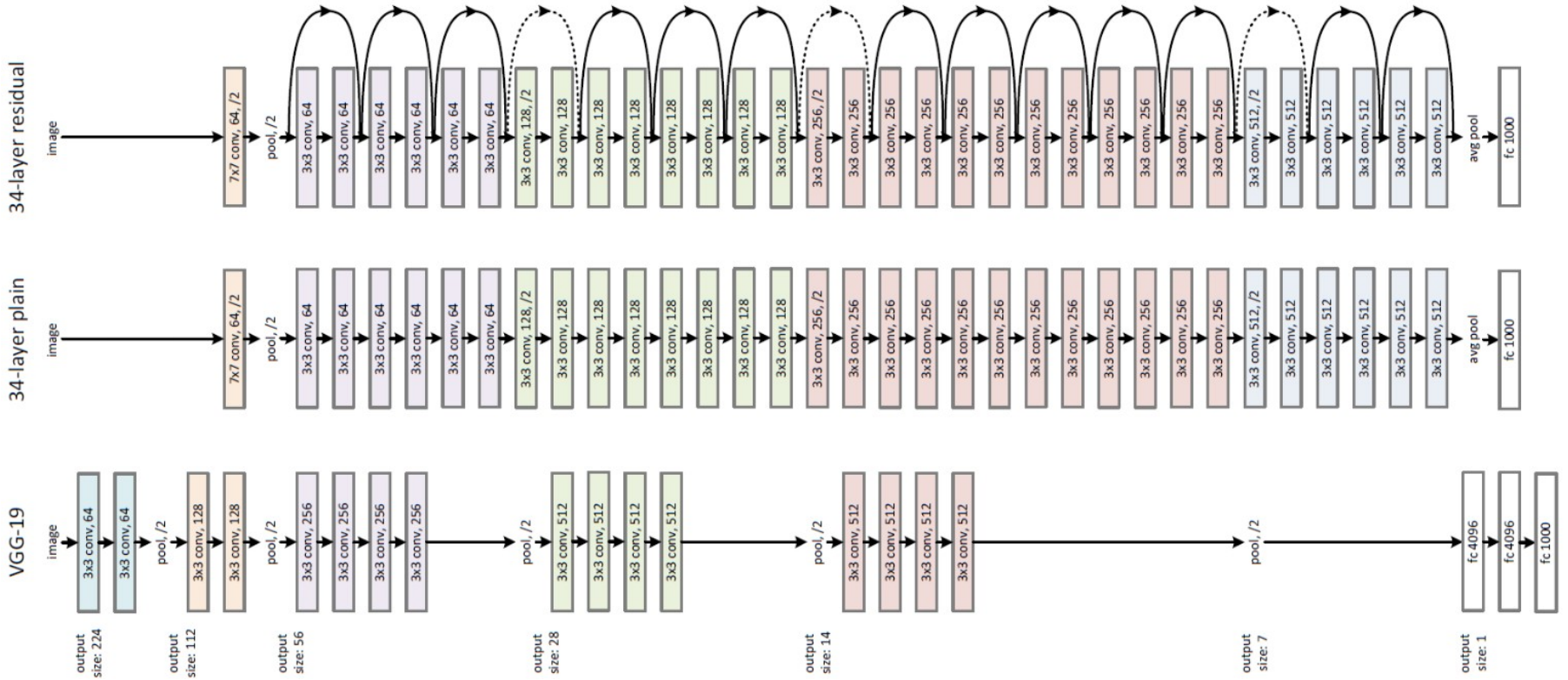


ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

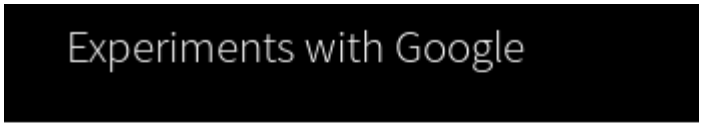
GoogleNet, Winner of ILSVRC competition 2014.



Microsoft ResNet, Winner of ILSVRC competition 2015.



https://cs.stanford.edu/people/karpathy/convnetjs/



COLLECTION

AI Experiments

https://experiments.withgoogle.com/collection/ai

More details on Recurrent Neural Networks

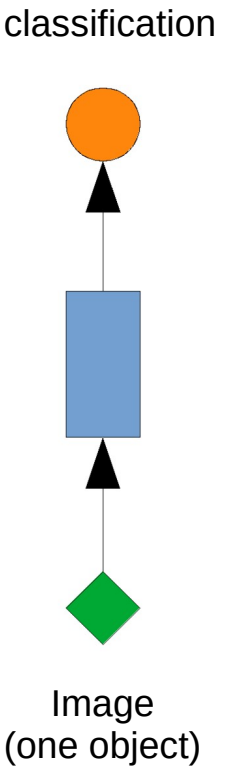
- Common for all neural network models we have studied so far is the lack of feedback connections. We will now study networks with such connections!

- Recurrent networks are typically used when we are dealing with sequence data. It can be text data, speech data, image data or numerical times series data coming from eg. sensors or stock markets. And combinations of all these!

- The feedback connections are used to capture the short and long term temporal dependencies in the data.

What is it that recurrent networks offer?

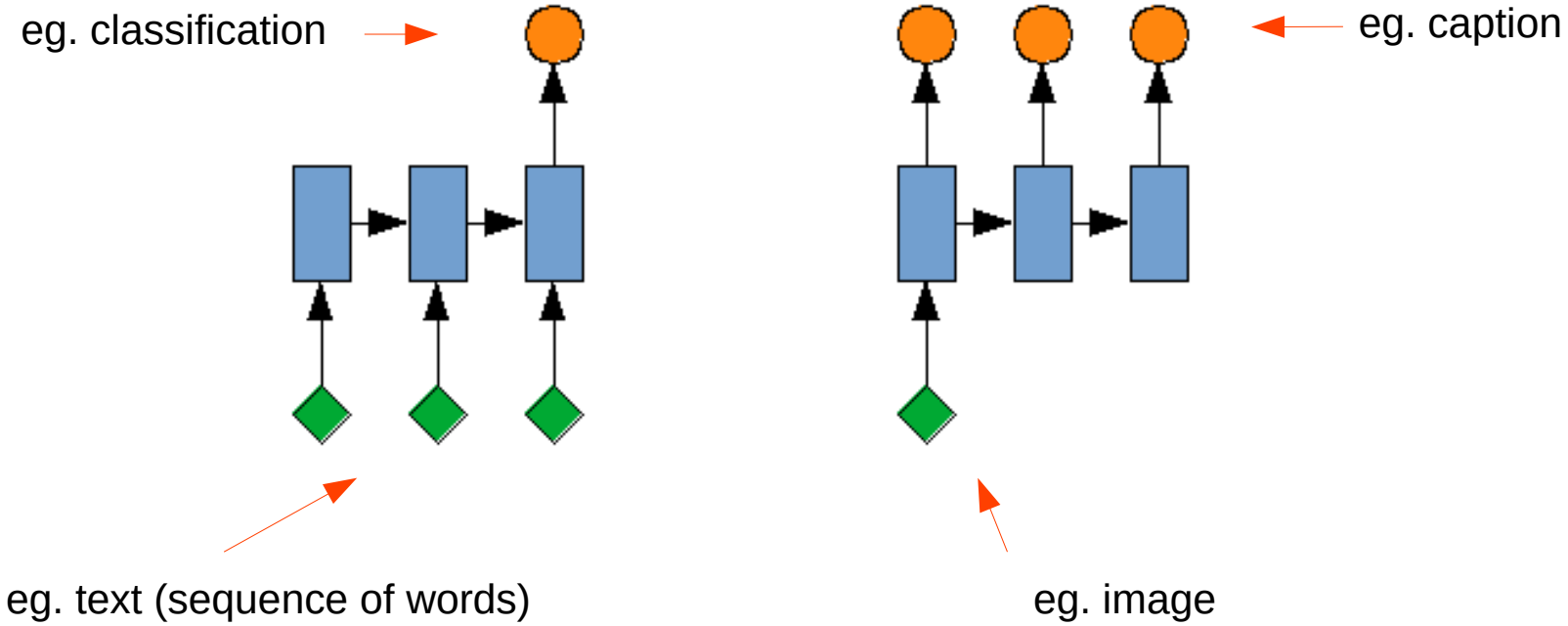
Sofar, one-to-one problems!



With recurrent networks

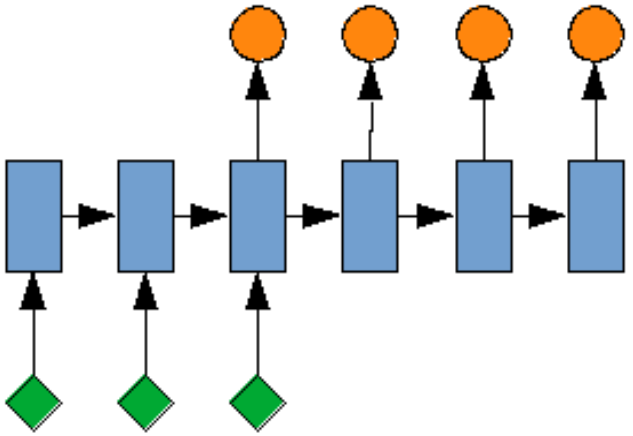
many-to-one

one-to-many



many-to-many

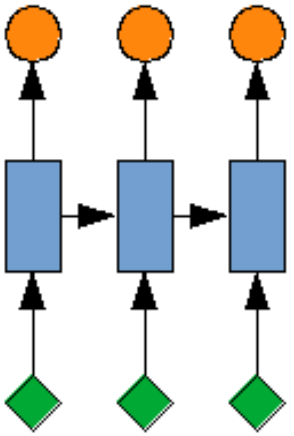
eg. text



eg. text

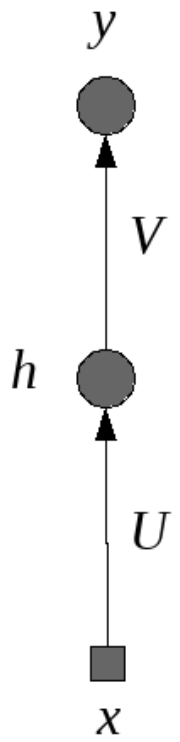
many-to-many

eg. another sequence

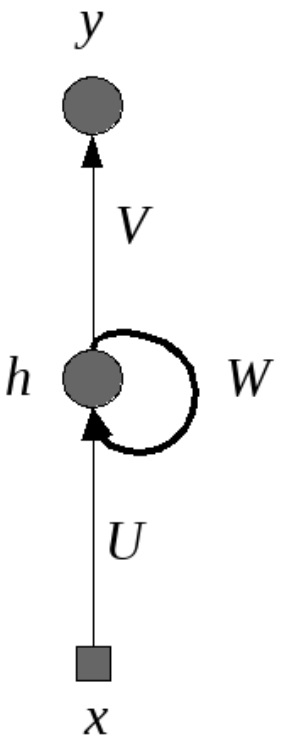


eg. sequence

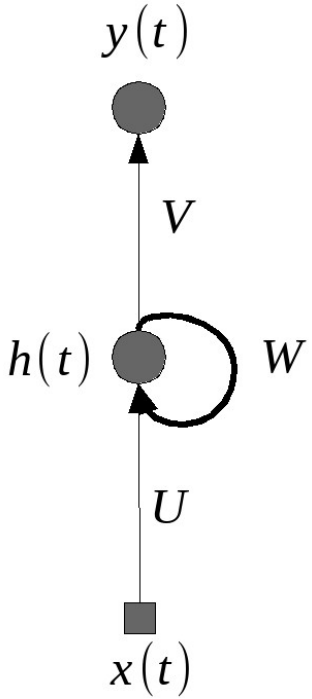
Simple recurrent networks



A simple 1-1-1 MLP



This network has a feedback connection from the hidden output feeding into itself.



Given: $x(0), x(1), ..., x(T)$

$$y(t) = g_{\mathrm{o}}\left(Vh(t)\right)$$
$$h(t) = g_{\mathrm{h}}\left(Ux(t) + Wh(t-1)\right)$$

Let's be explicit for a few time steps!

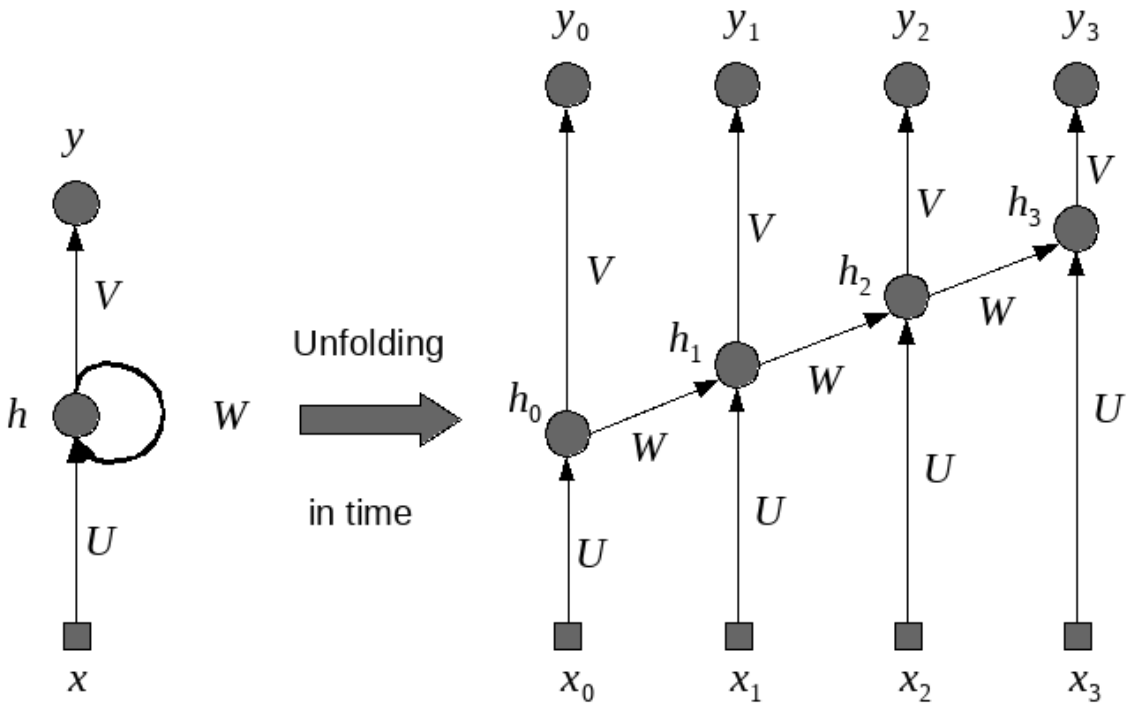
$$(\text{to avoid clutter:} \qquad x(t) \to x_t, \;\; h(t) \to h_t)$$

$$y(0) = g_{\mathrm{o}}\big[Vh_0\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_0\big)\big] \qquad (\text{using the initial condition } h(-1) = 0)$$

$$y(1) = g_{\mathrm{o}}\big[Vh_1\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\big]$$

$$\begin{aligned} y(2) &= g_{\mathrm{o}}\big[Vh_2\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wh_1\big)\big] = \\ &= g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big)\big] = \\ &= g_{\mathrm{o}}\Big[Vg_{\mathrm{h}}\Big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\Big)\Big] \end{aligned}$$

Do we recognize this structure?



$$y(2) = g_{\mathrm{o}}\Big[V g_{\mathrm{h}}\Big(Ux_2 + W g_{\mathrm{h}}\big(Ux_1 + W g_{\mathrm{h}}\big(Ux_0\big)\big)\Big)\Big]$$

Unfolding in time!

The unfolded network is an MLP with 4 hidden layers, sparsely connected and shared weights.

Backpropagation through time (BPTT)
(= basically training the unfolded MLP)

Two numerical problems can occur:

- Vanishing gradients
- Exploding gradients

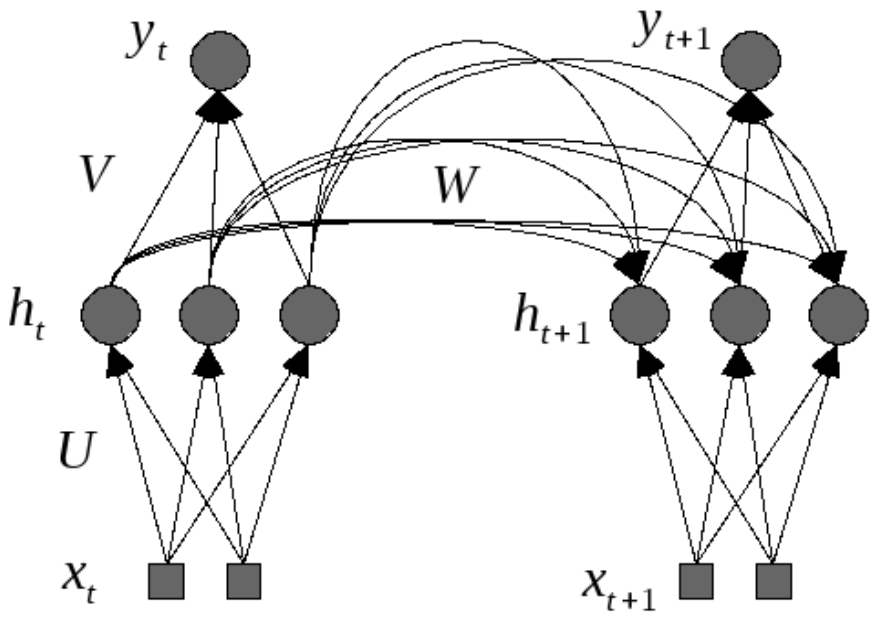
Where vanishing gradients are probably most common!

As a result it is difficult to learn long term dependencies!

Solution: Change the behavior of the hidden node as to have a more or less constant value of the recursive derivative.

→ **LSTMs**

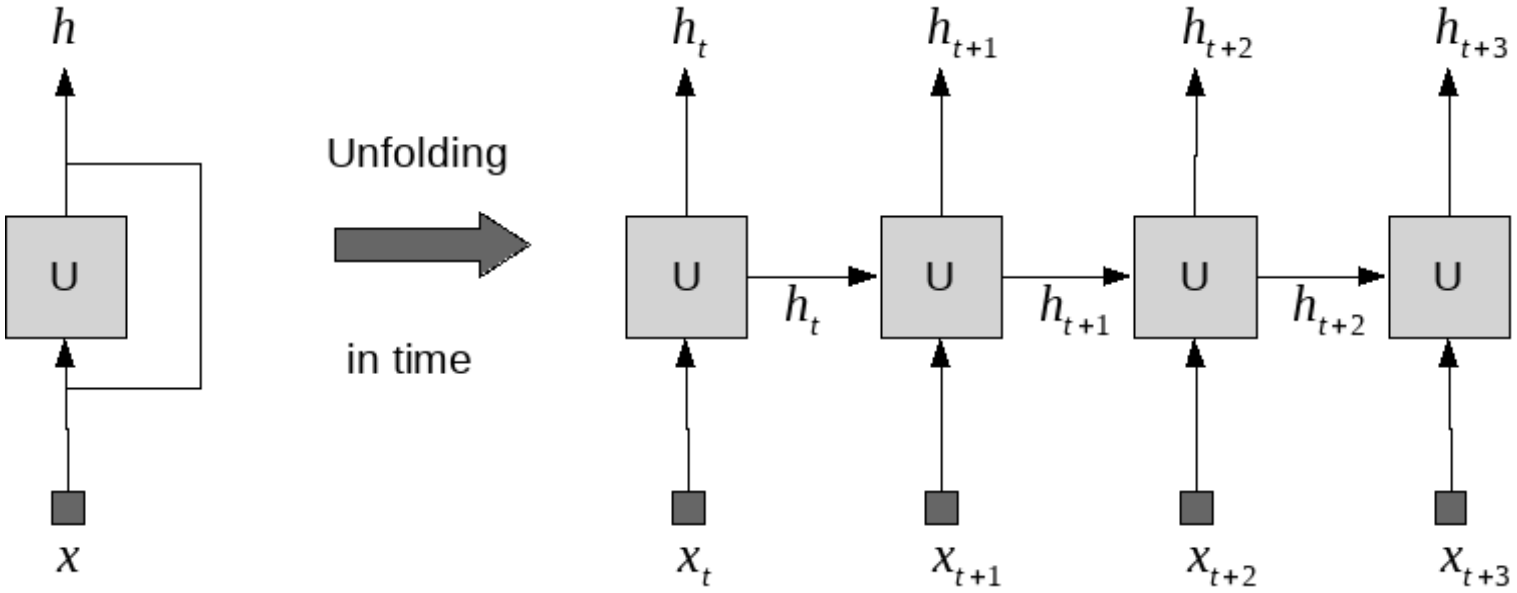
Before that let's just conclude by noting that the simple recurrent network can be more complicated than what we have showed here. As an example:



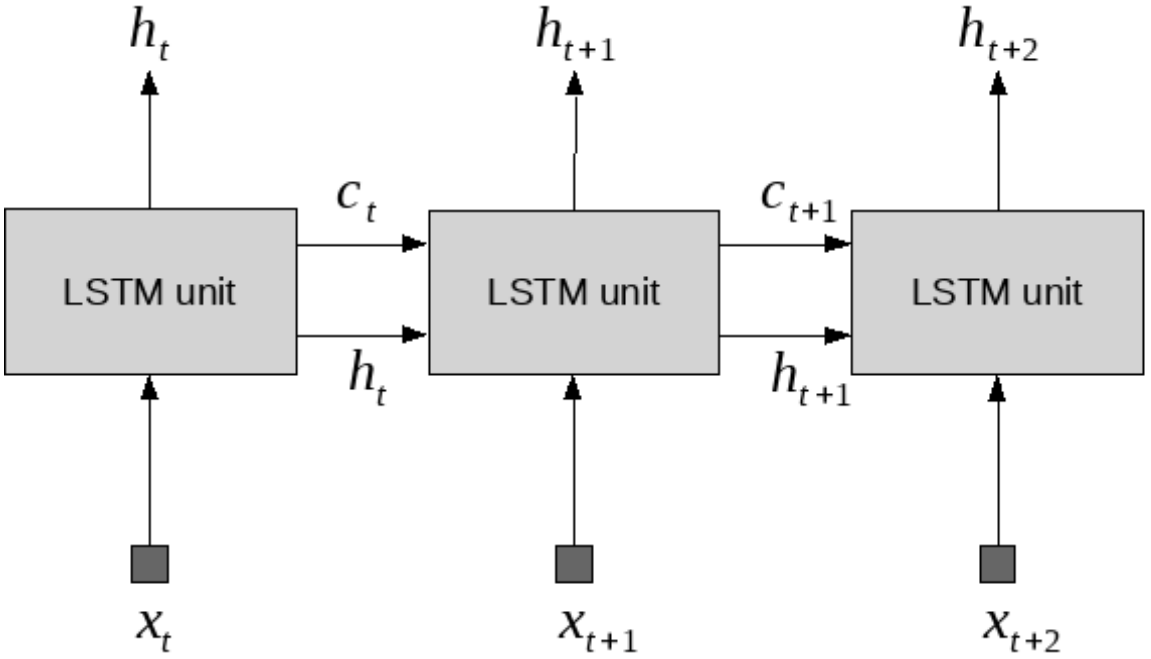
Two input, three hidden nodes. Unfolded one time step!

Long short-term memories (LSTM)

Previously we had networks like this, where the “U” box represents the simple recurrent unit.

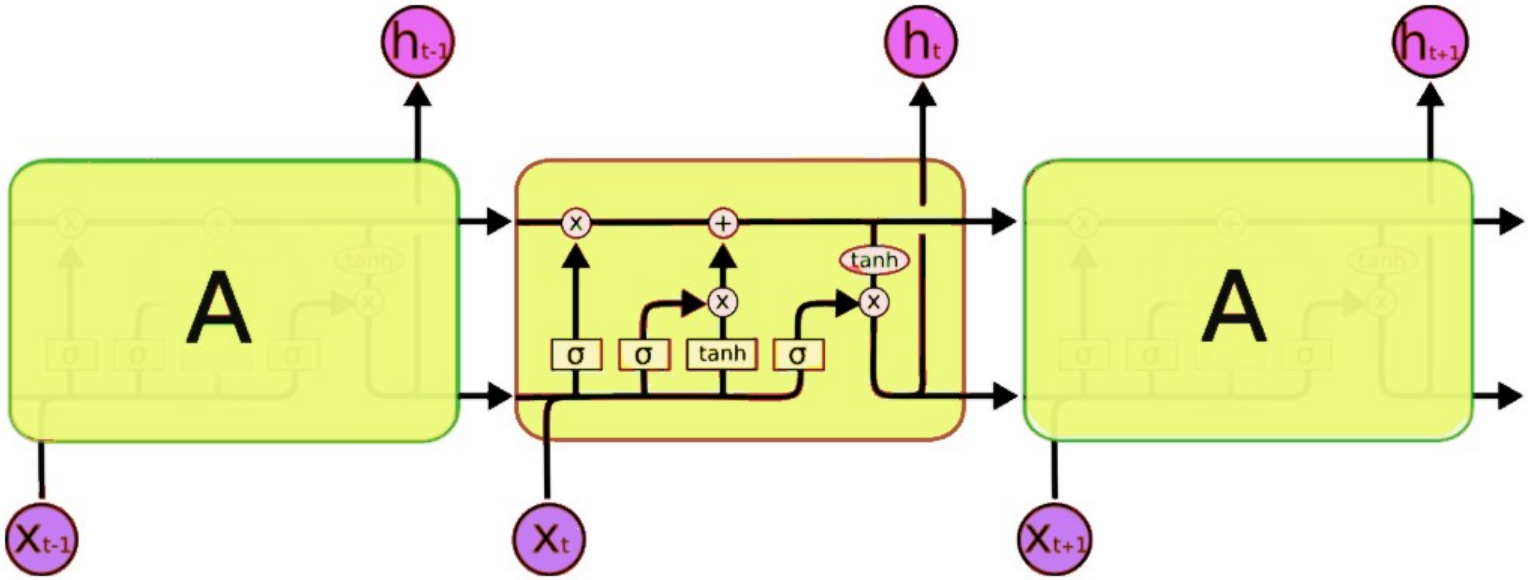


For a LSTM network we simply replace the “U” box with a LSTM box and add a new connection,



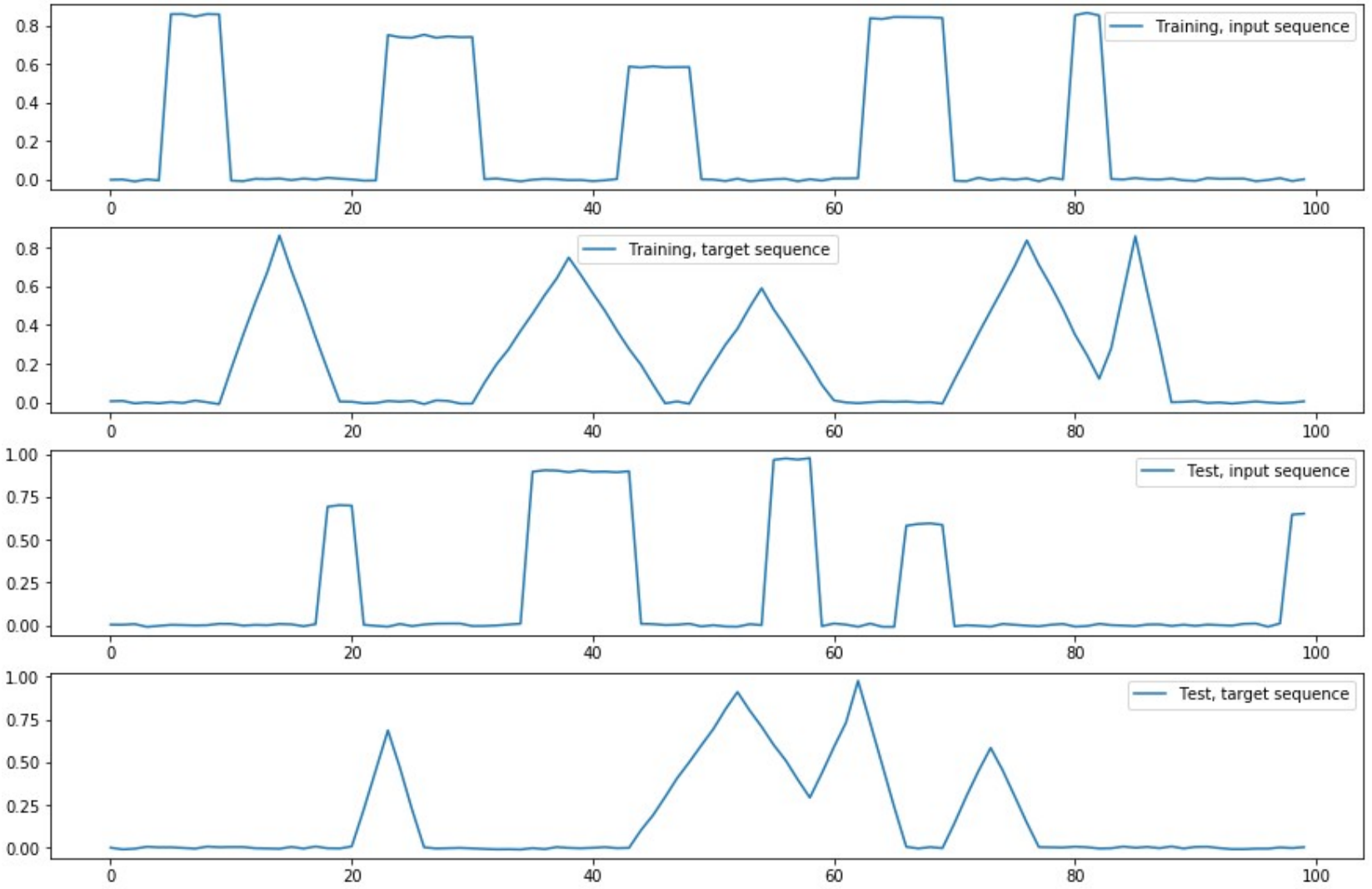
So in a sense it is similar to the previous network (apart from the ”c” value. However the LSTM box is more complicated than the “U” box!

There are many illustrations of the LSTM node, here is one:

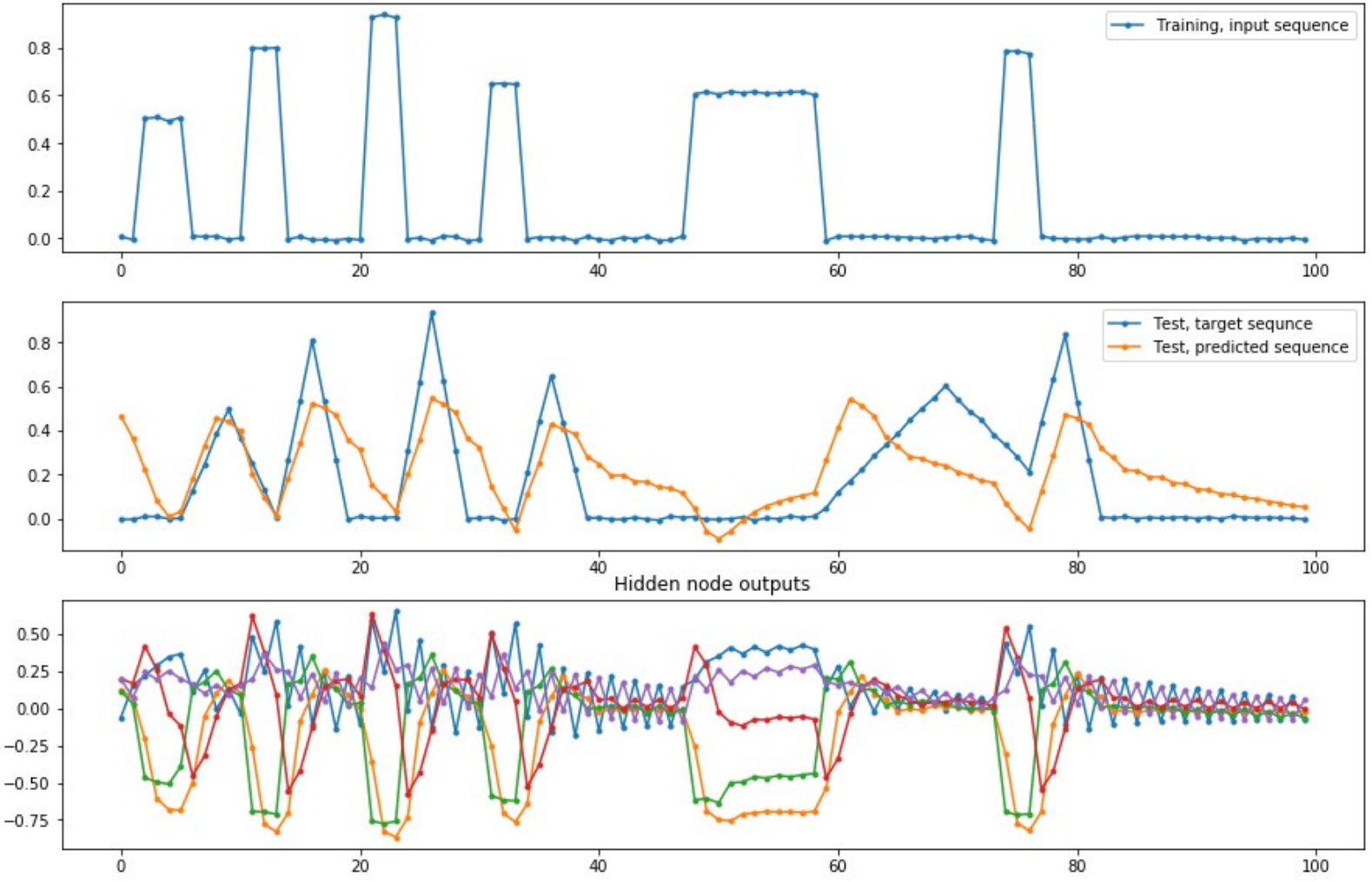


From a good blog:
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Numerical example! RNN as a pulse converter

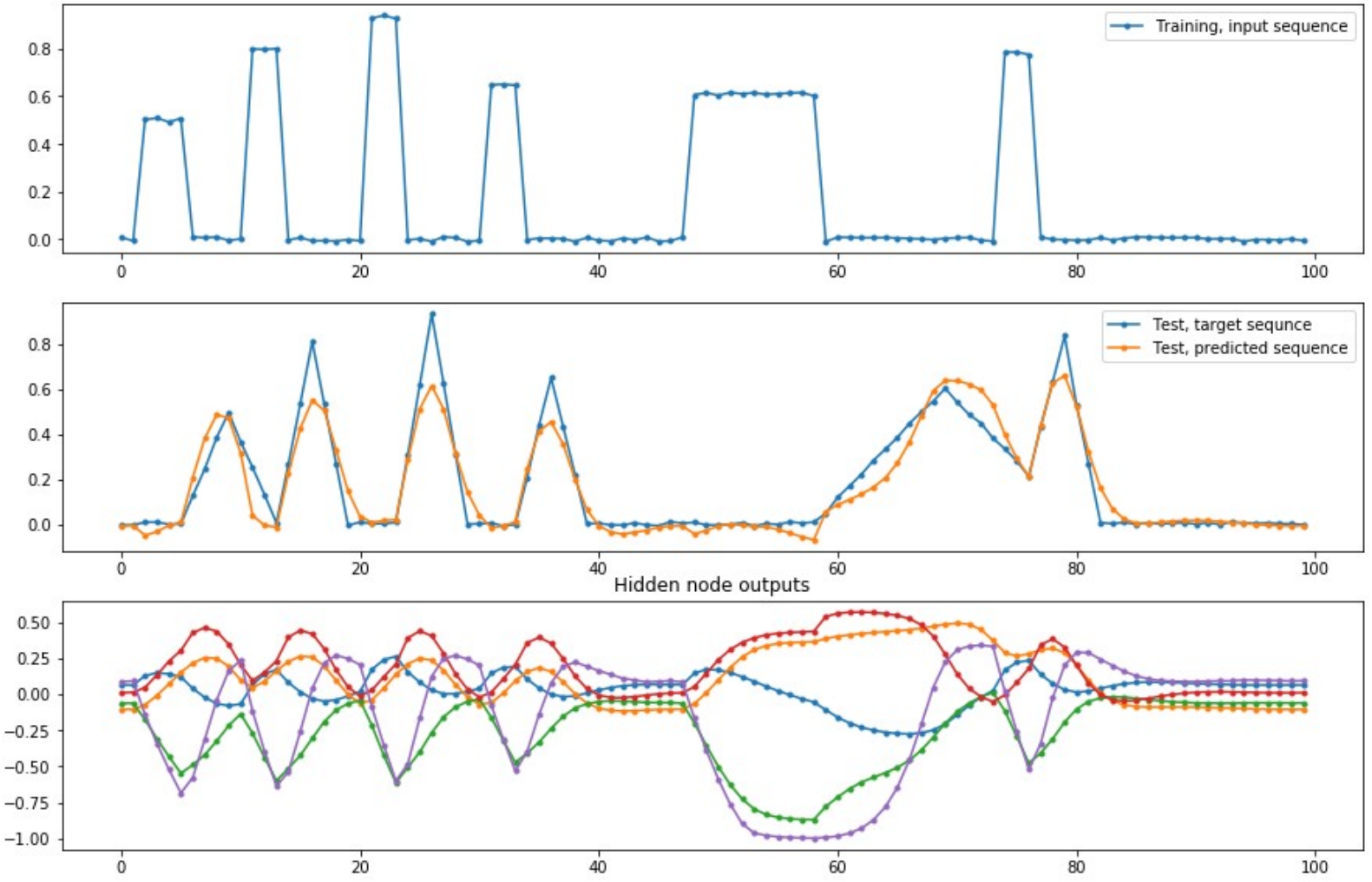


Simple recurrent network with 5 hidden nodes (no LSTM)



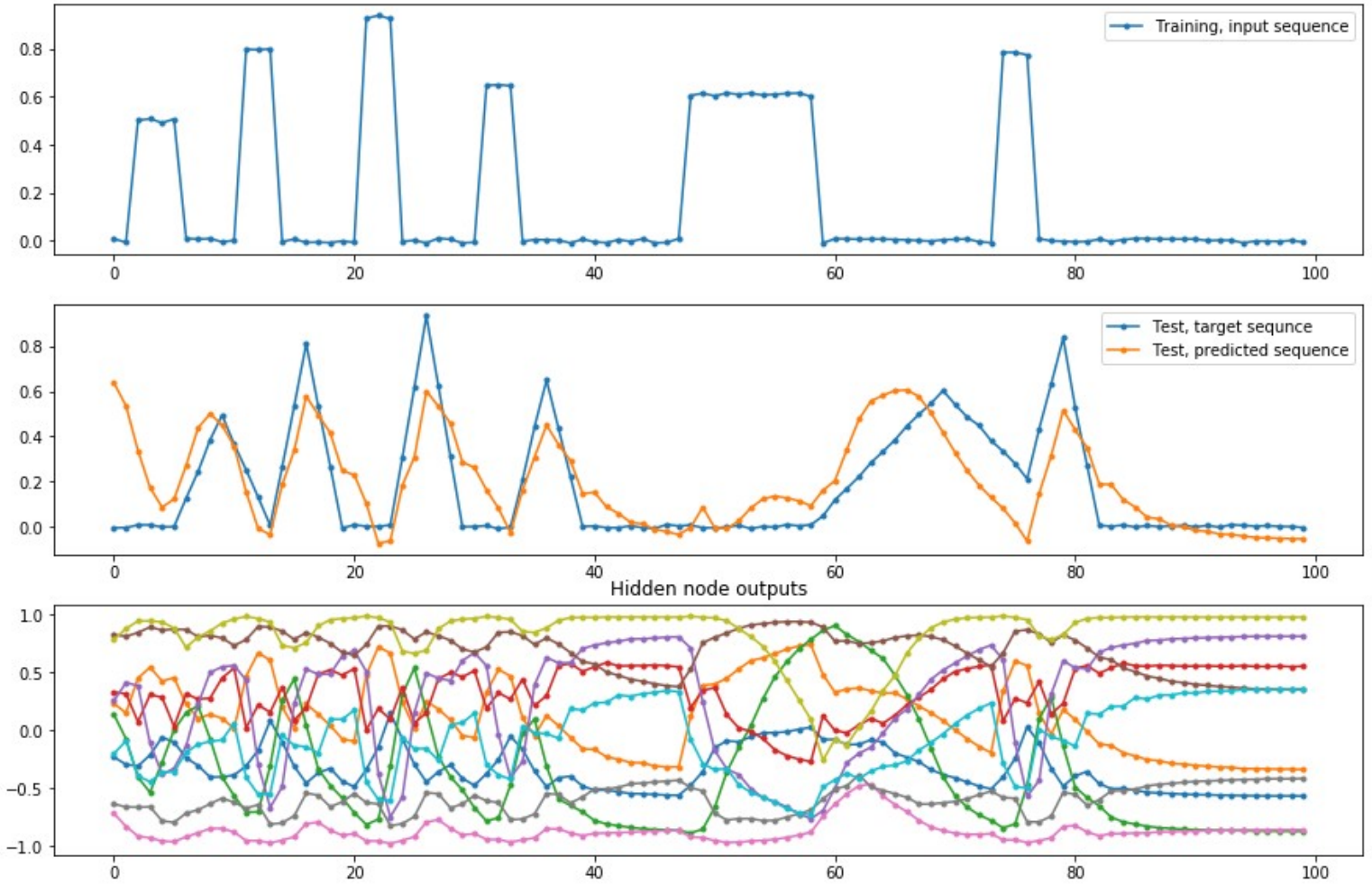
Test error: 0.47

LSTM network with 5 hidden nodes!



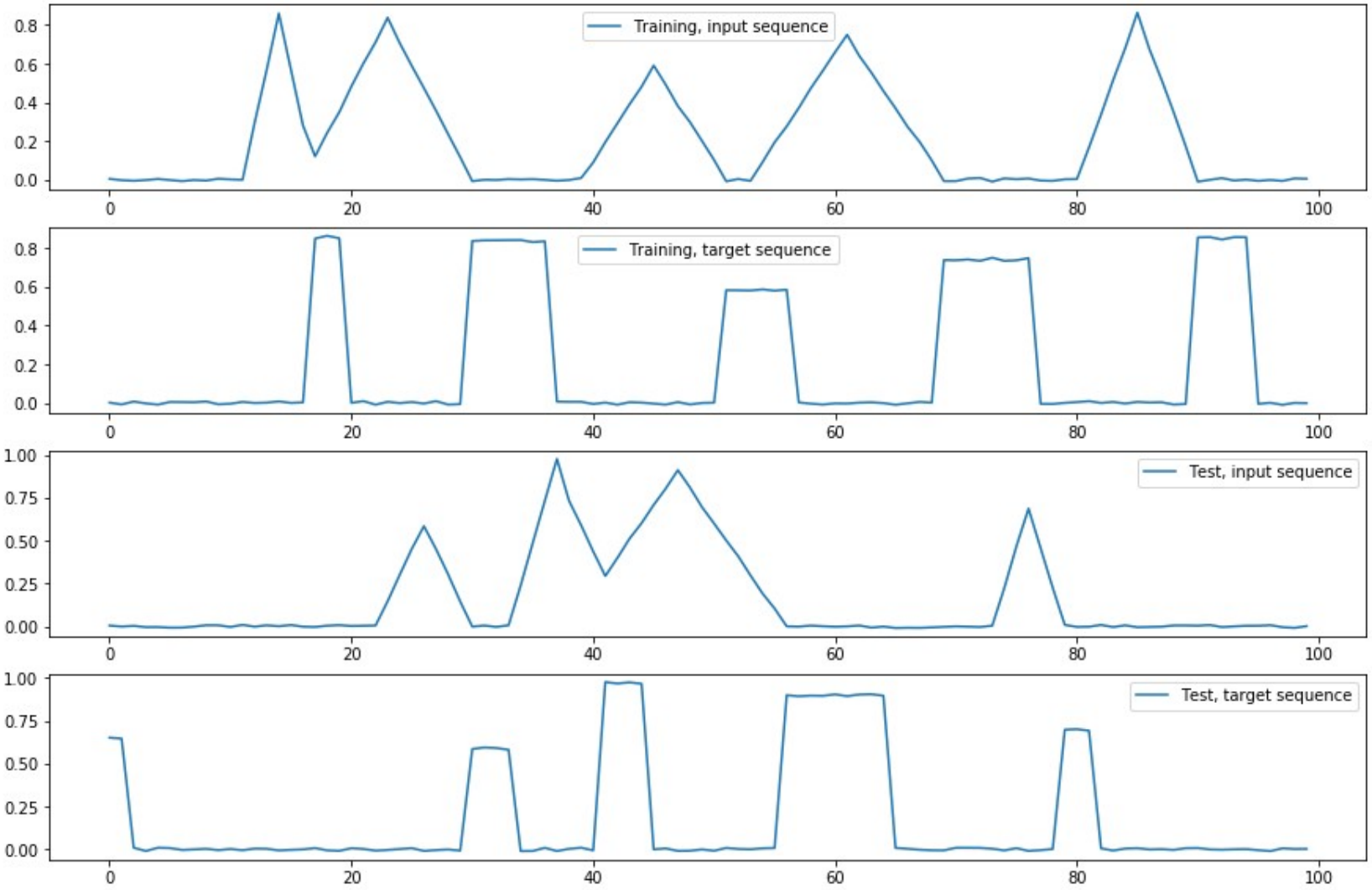
Test error: 0.08

Simple recurrent network with 10 hidden nodes!



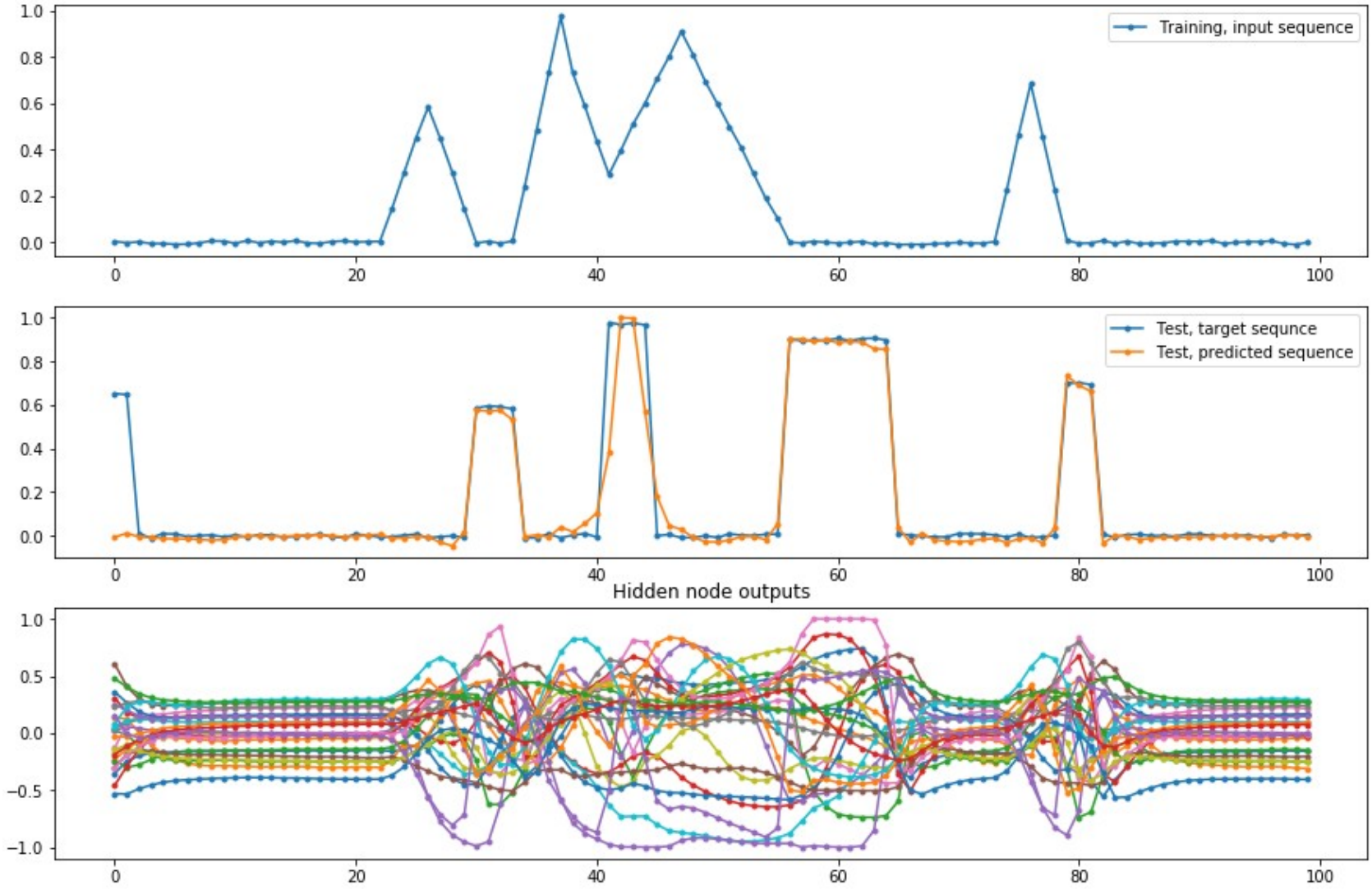
Test error: 0.27

The inverse problem is more difficult!



LSTM network with 25 hidden nodes!

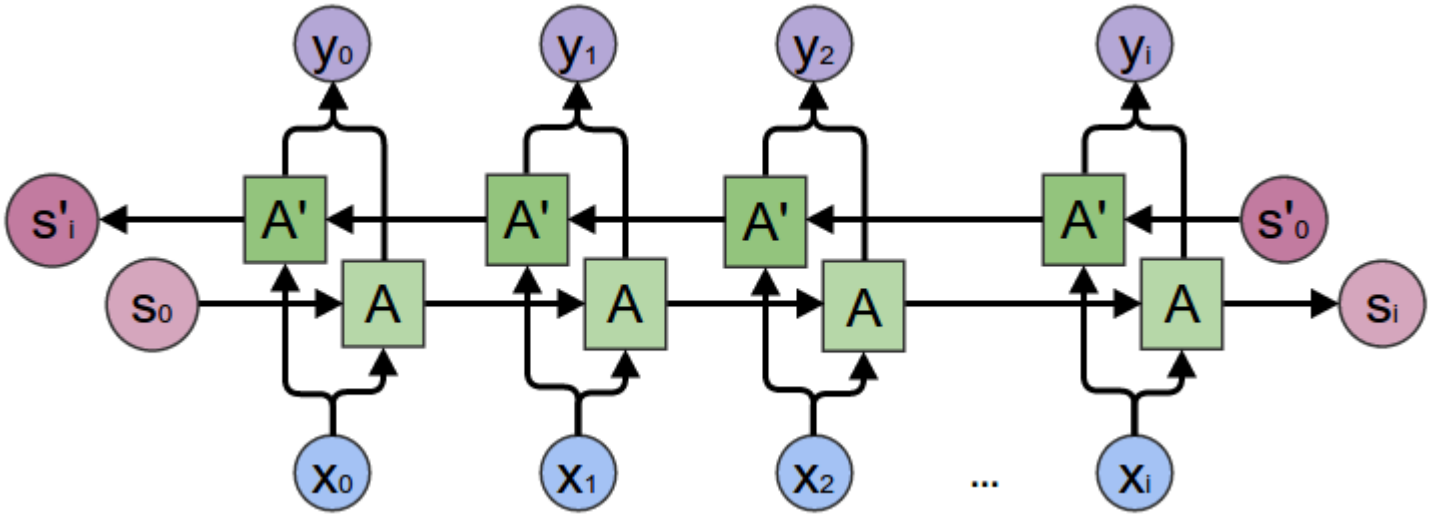
(2726 trainable weights!)



Test error: 0.07

Bidirectional LSTMs

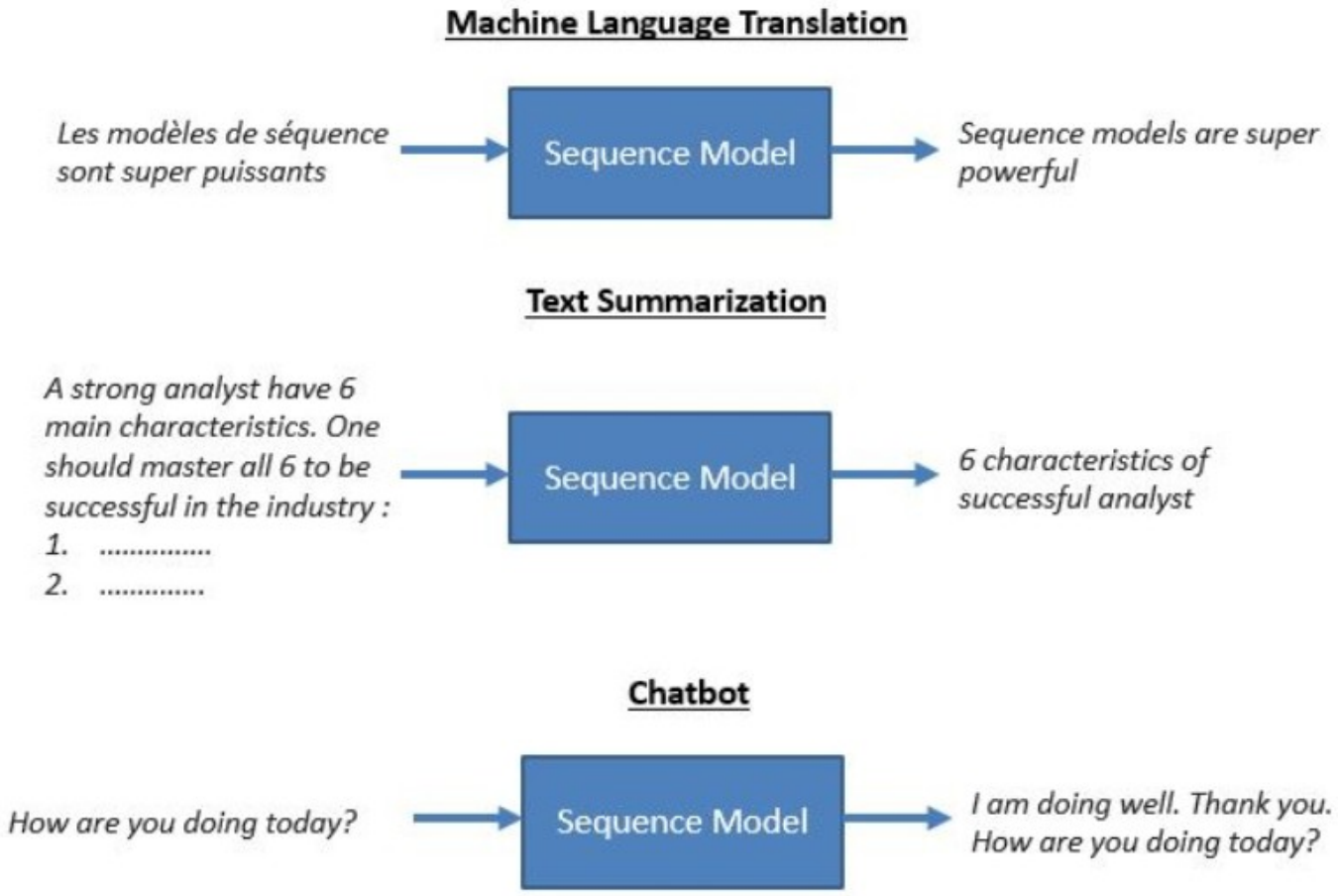
Putting two independent LSTMs together, one for the forward sequence and one for the reverse sequence.



Useful when the context of the input is needed.

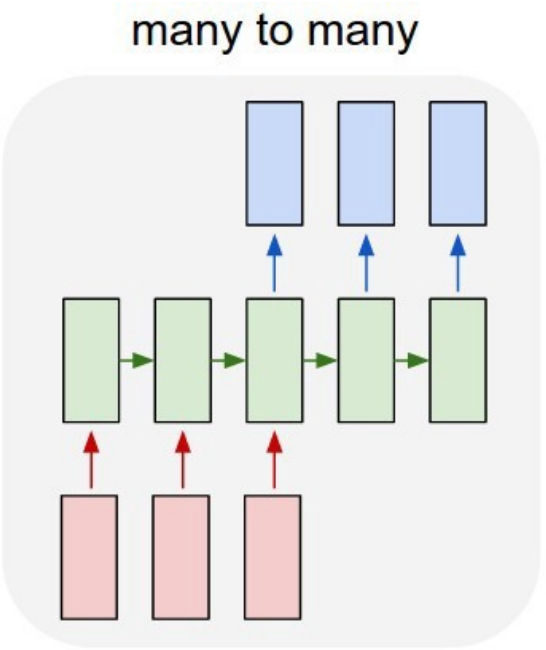
Sequence to Sequence (seq2seq) models are a special class of recurrent neural network typically used to solve language problems like

- Machine Translation
- Question Answering
- Chat-bots
- Text Summarization
- ...

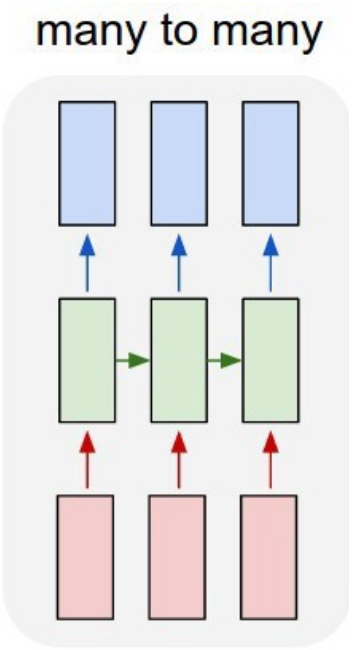


https://www.analyticsvidhya.com/blog/2018/04/sequence-modelling-an-introduction-with-practical-use-cases/

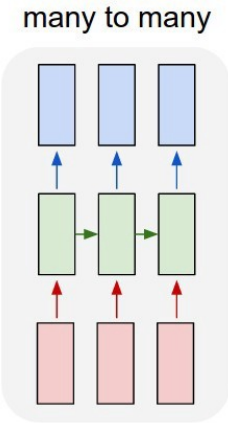
Sequence to sequence models



Building a “representation” vector



“Direct” translation



Use this architecture for a simple language model!

Task: given a sequence of characters, predict the next character!

Example: One, two, thre?

Example: My name i?

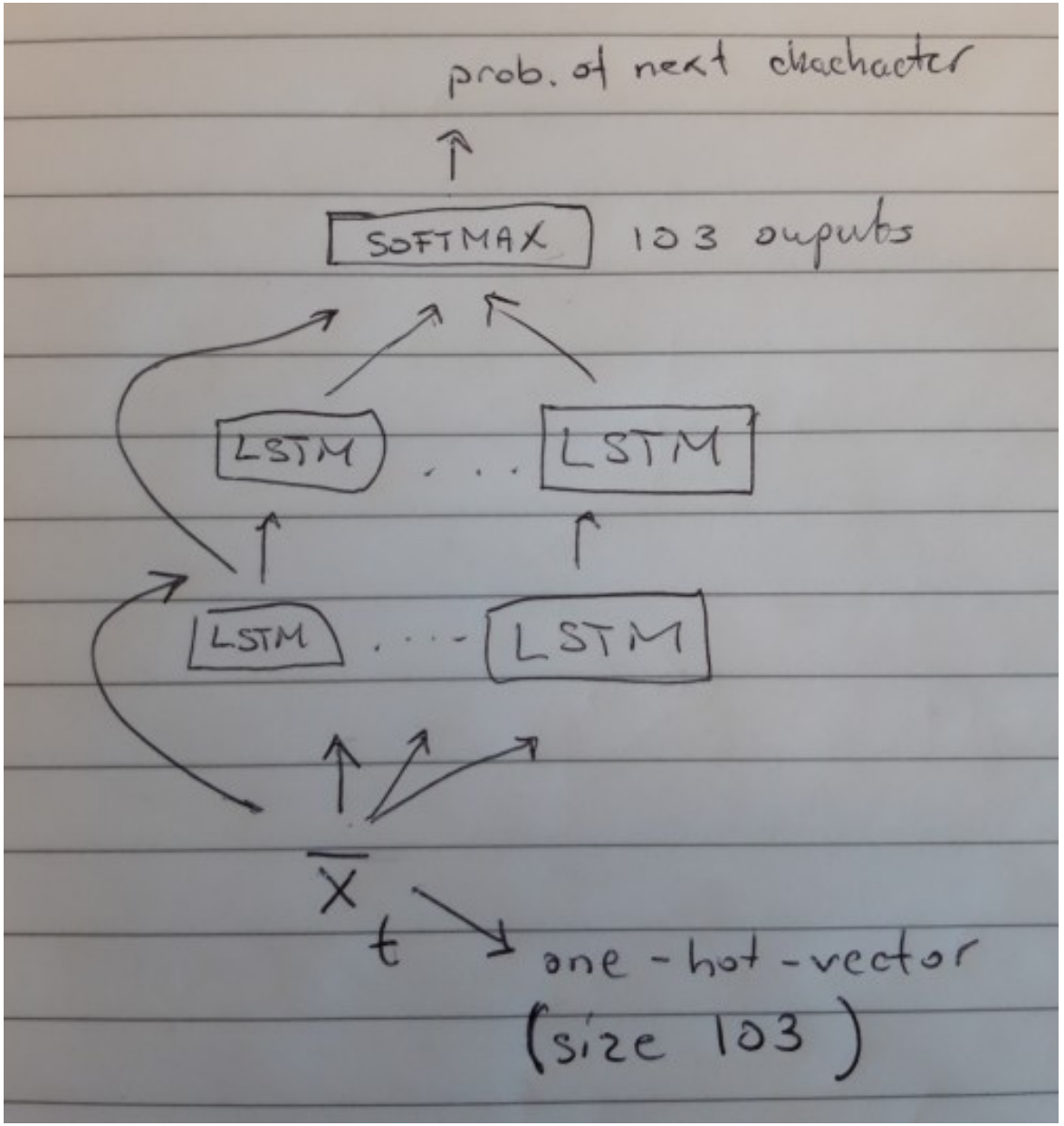
A more fun example!

Use all of the source code of **Tensorflow** (some earlier version).
This is C++ code!

About 14 million characters in the sequence!

There are 103 different characters used!

The model



Skip layer connections

Two layers of 1024 LSTMs each

About 13 million parameters

Since the output are probabilities for the 103 different characters, we can sample from the model:

Five characters (A, B, C, D, E)

As a simple example [0.1, 0.3, 0.5, 0.1, 0.0]

The probability for ‘C’ is 50%, for ‘B’ 30% and so on.

The sampled character is then used as the new input to the (trained) model.

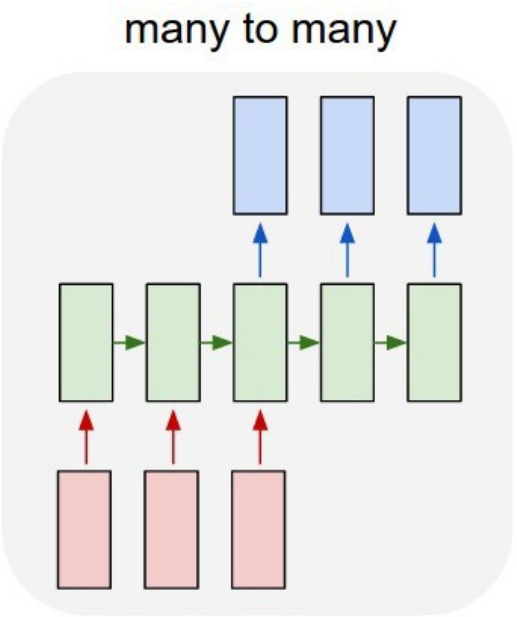
Seed: “void “

```
void AddShapeMemoryTranspose(const Key& key, const AllocatorAttributes attr) {
  VLOG(2) << "Instanding worker threads with dues to set cuSolveDDe gradient for: " << variant->devices[0]->env_;
  def.label_fills.push_back(TestMultipleWrites());
  std::unique_ptr<thire> locks(0);
  done(e->src(), cinfo->comp_device_context);
  return Status::OK();
}
Status GetStatusResponse::NewAppendable(string* stable_compress_ptr, class LoggingPand& BaseRendezvous,
                        const CreateSing& conten
```

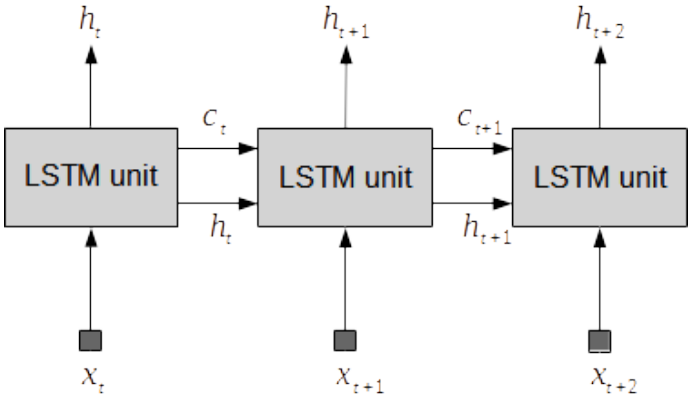
Seed: “printf(“

```
            printf("%s",
                                  strings::StrCat(true), ">");
              }
              if (!s.ok()) {
                for (Node* n = 0; node : merge.nodes n& y_noded && (n->dst_node == train_num ==
                    CreateNodeDef(SIGEDMB, DT_FLOAT))  // Dsty in different device:
                for (int i = 0; i < N; ++i) {
                  (*(start_times[idx].ptr = nullptr) || (i == 1) ? Padding == Padding::VALID:
                  (*is_log);
                }
                output_memory_usage = strings::StrCat(prefix, ", successfully.");
                }
              }
              list_type_list.set_select(tf_stat.m_def)
```

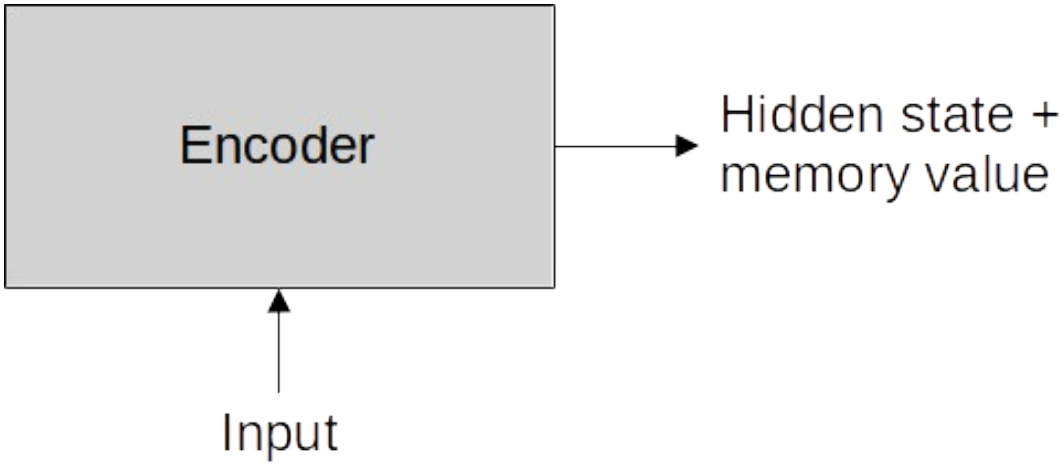
“Modern” Sequence to sequence (seq2seq) models

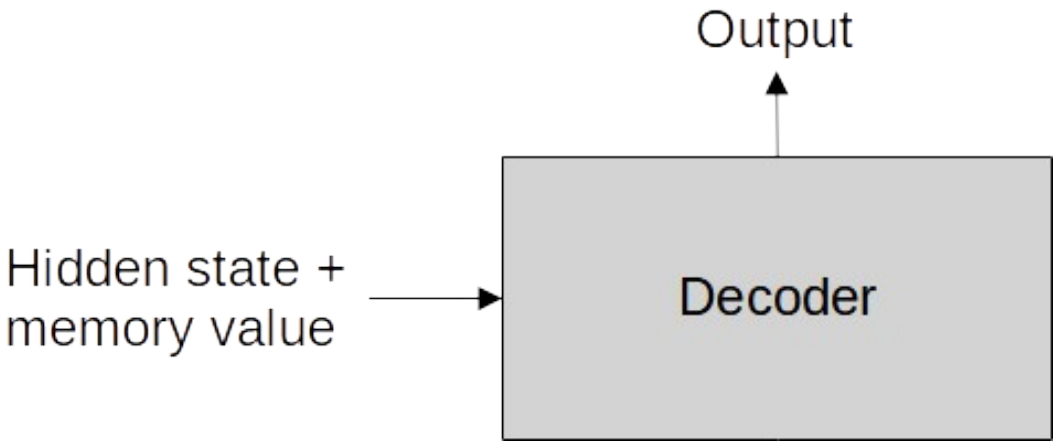


Seq2seq models are very often based on encoder-decoder architectures

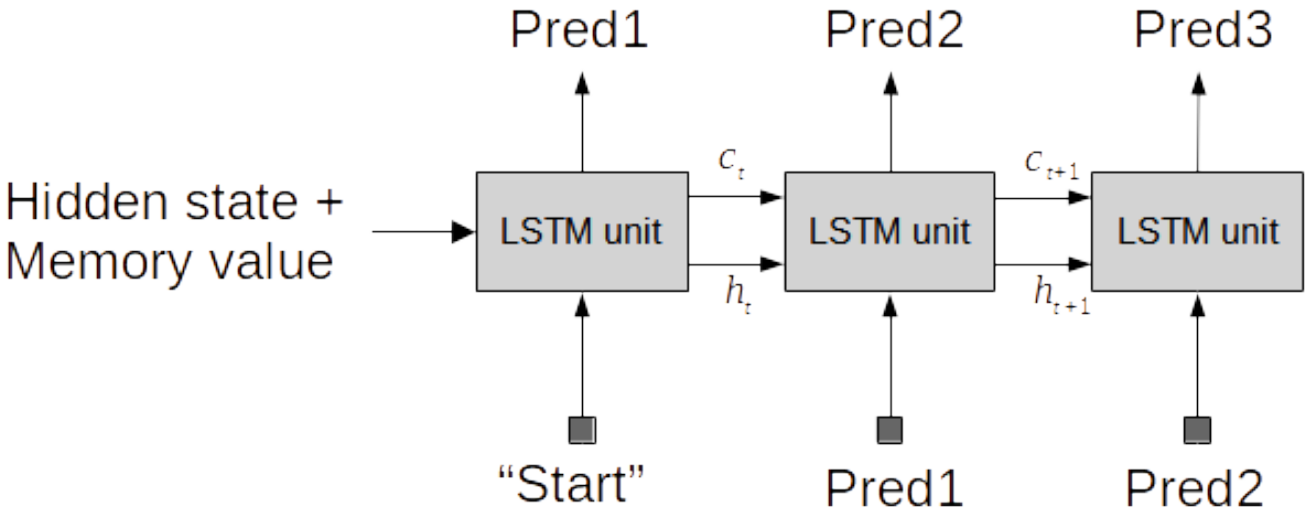


We can summarize this as an encoder

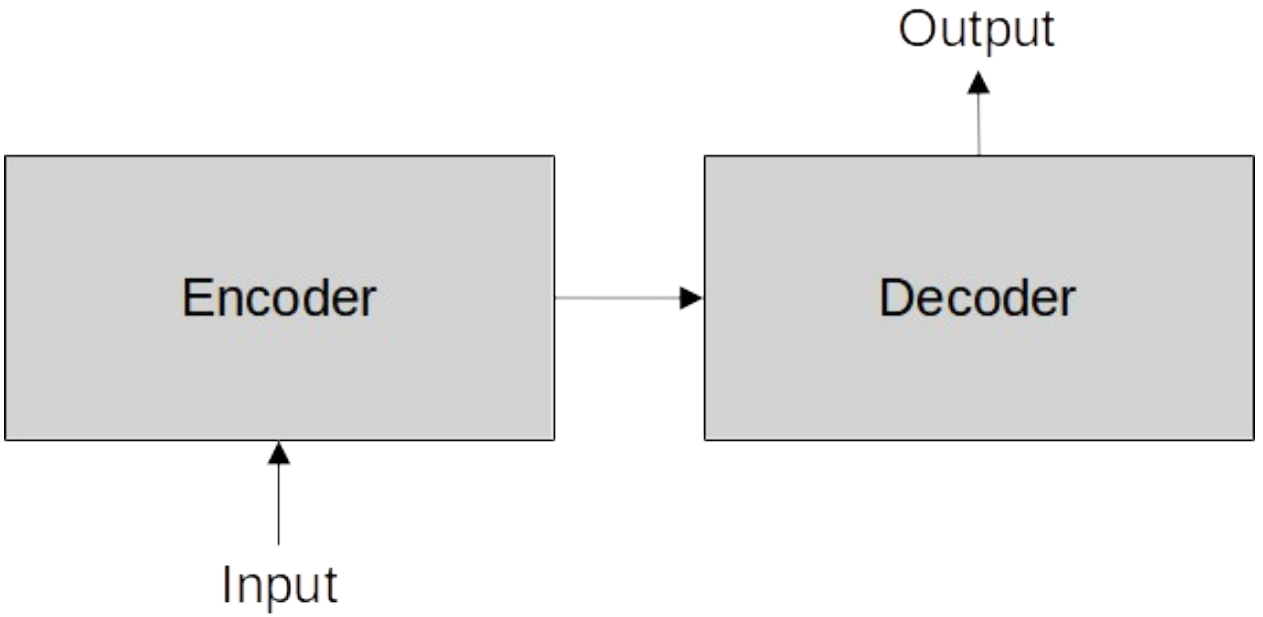




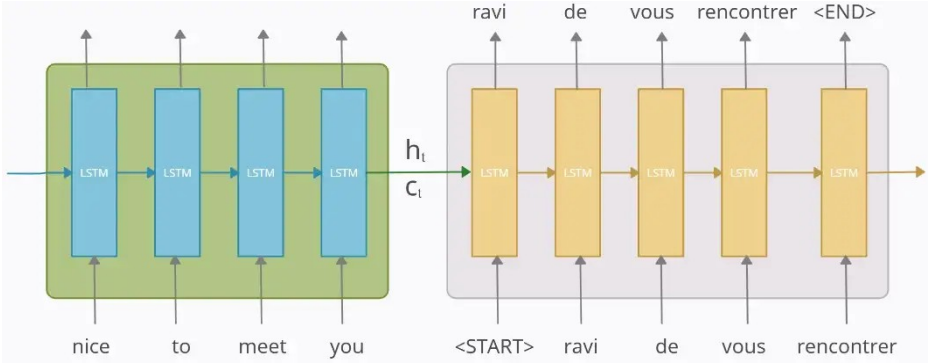
We can have a decoder that can take be initialized with a hidden state and a memory value. The decoder can then generate a sequence.

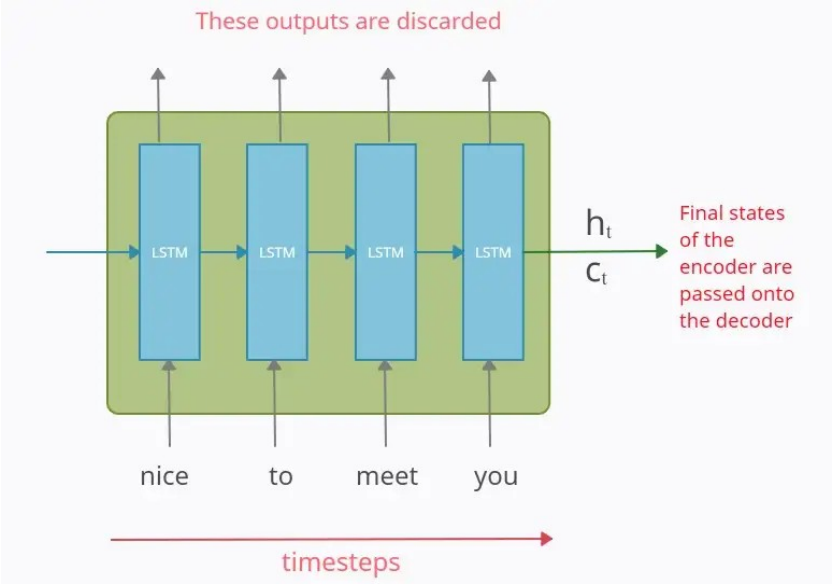


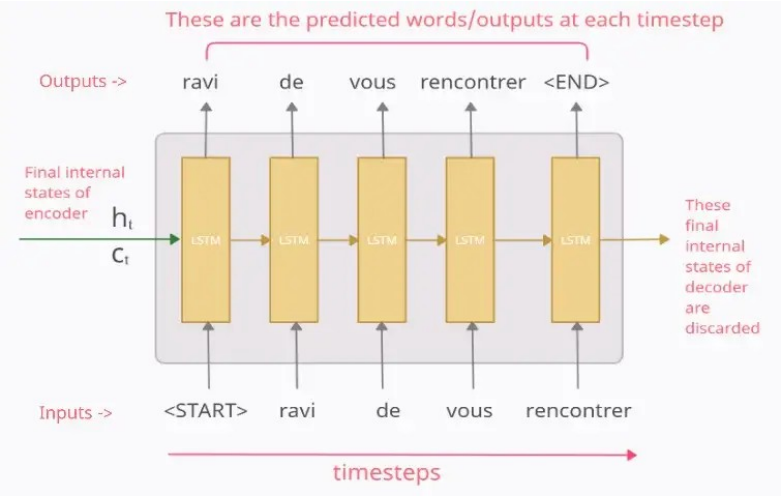
Encoder-decoder system



More details





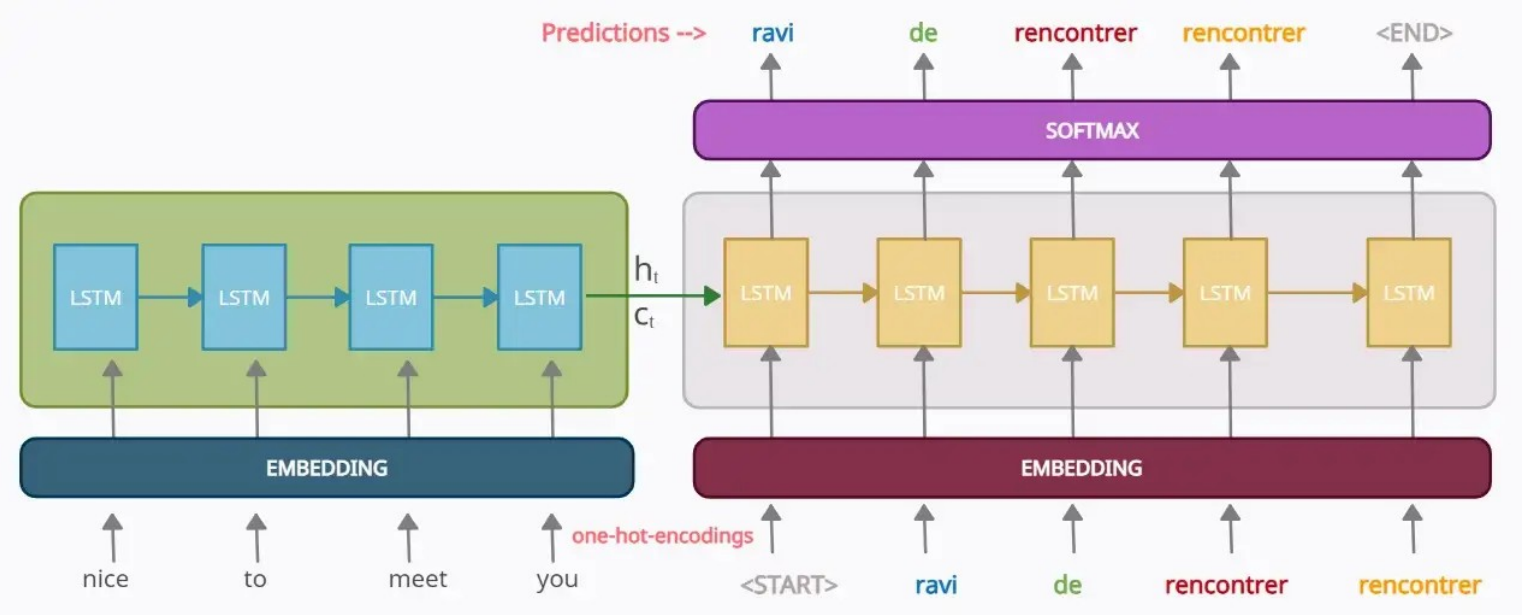


https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

How to train the seq2seq models?

- How to represent text?
- Embedding layers!
- Loss function?
- Teacher forcing!

Final view of the testing phase



https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

The Sutskever paper (2014)

Sequence to Sequence Learning with Neural Networks

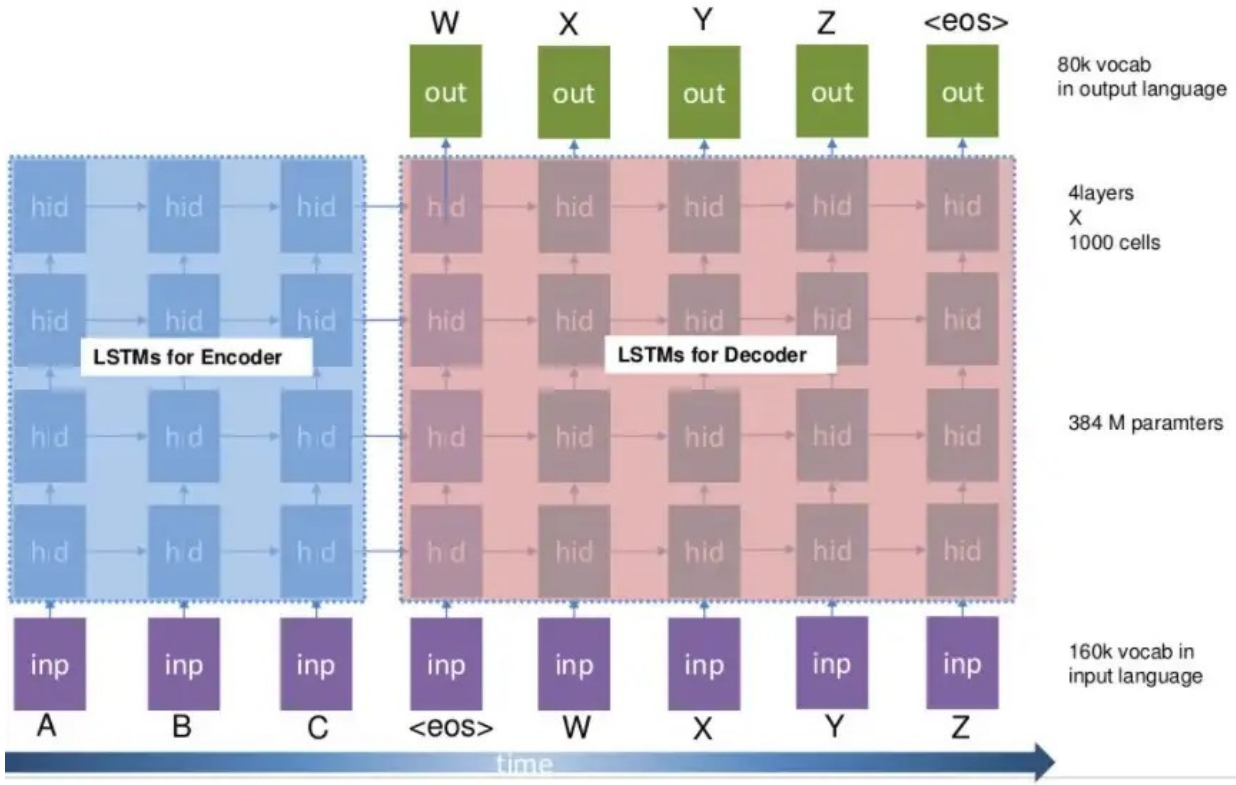
Ilya Sutskever Google ilyasu@google.com

Oriol Vinyals Google vinyals@google.com

Quoc V. Le Google qvl@google.com

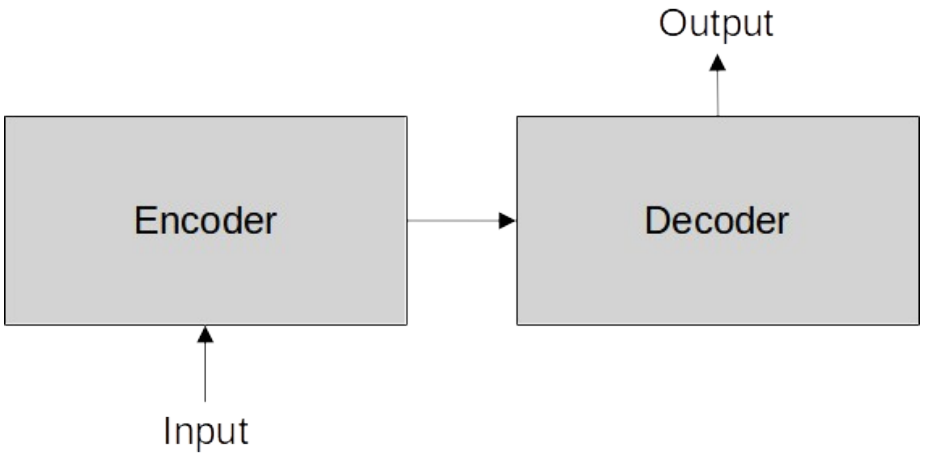
We used the WMT'14 English to French dataset. We trained our models on a subset of 12M sentences consisting of 348M French words and 304M English words, which is a clean "selected" subset from [29]. We chose this translation task and this specific training set subset because of the public availability of a tokenized training and test set together with 1000-best lists from the baseline SMT [29].

As typical neural language models rely on a vector representation for each word, we used a fixed vocabulary for both languages. We used 160,000 of the most frequent words for the source language and 80,000 of the most frequent words for the target language. Every out-of-vocabulary word was replaced with a special "UNK" token.

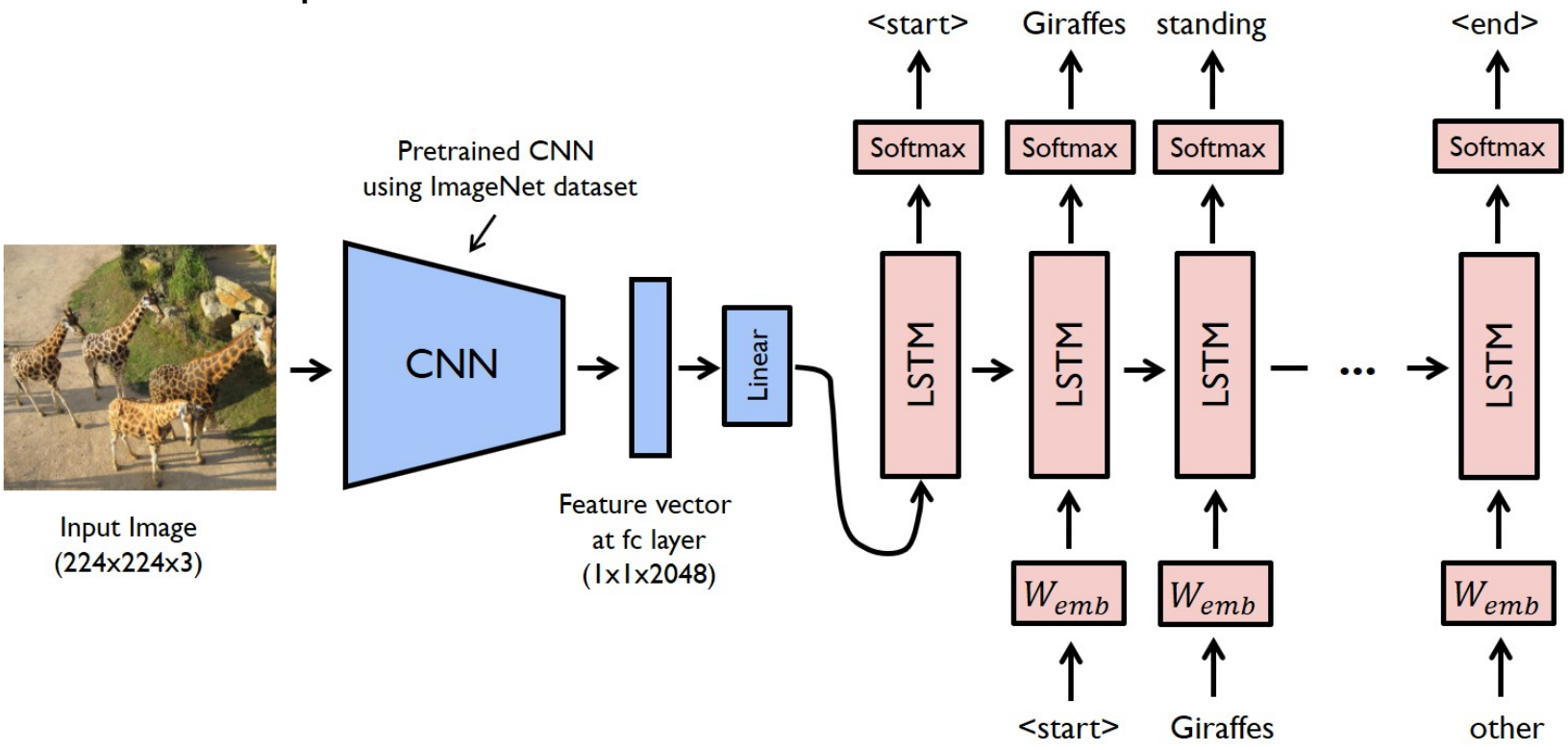


https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

What if the encoder is a CNN?



Example!



These architectures combine the CNN for dealing with images and LSTM for the sequnce part. Examples:

Activity recognition: image sequence to classification
Image description: image to text sequence (caption)
Video description: video sequence to sequence (description)

Two “early” papers:

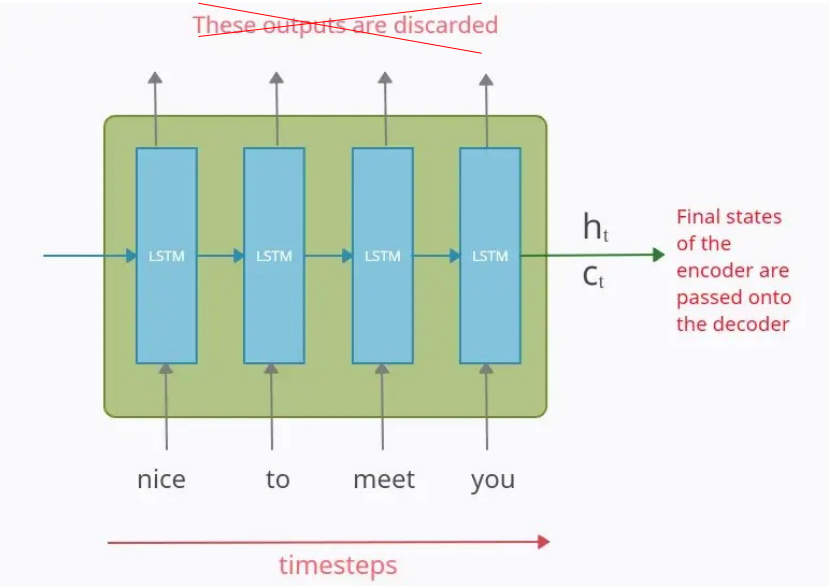
Show and Tell: A Neural Image Caption Generator (Link)

Long-term Recurrent Convolutional Networks for Visual Recognition and Description (Link)

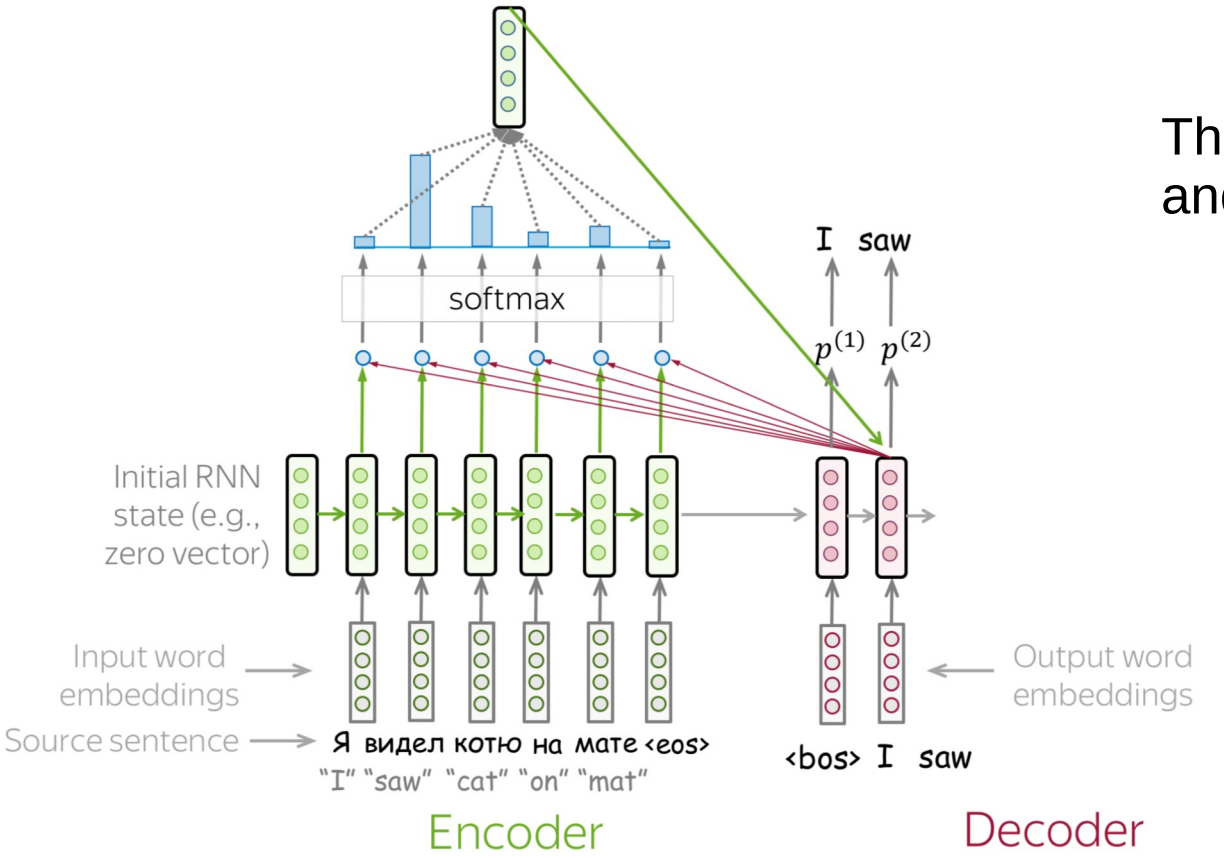
Adding ***Attention*** to seq2seq models

- So far, the encoder compresses the whole source sentence into a **single vector**
- We are **not using** the previous hidden states of the encoder, only the last one

  We will now change that and use all encoder states!!



An illustration

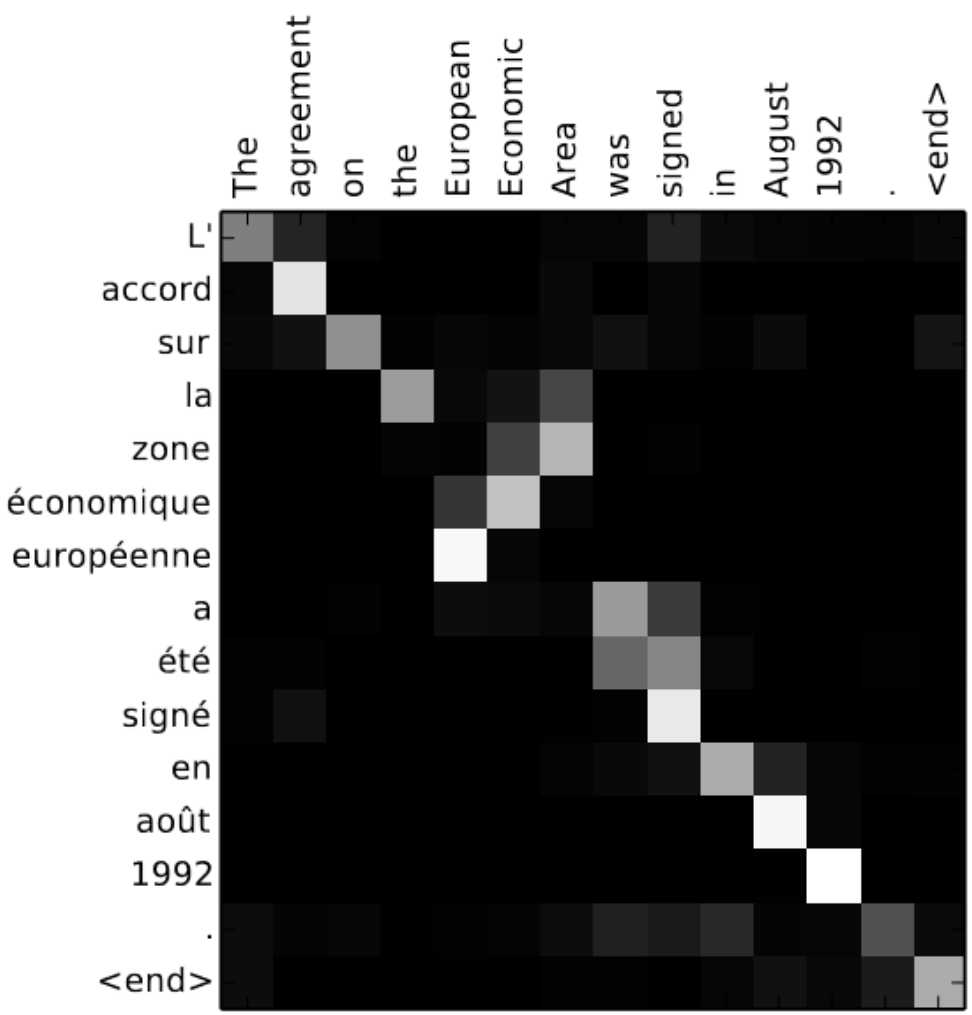


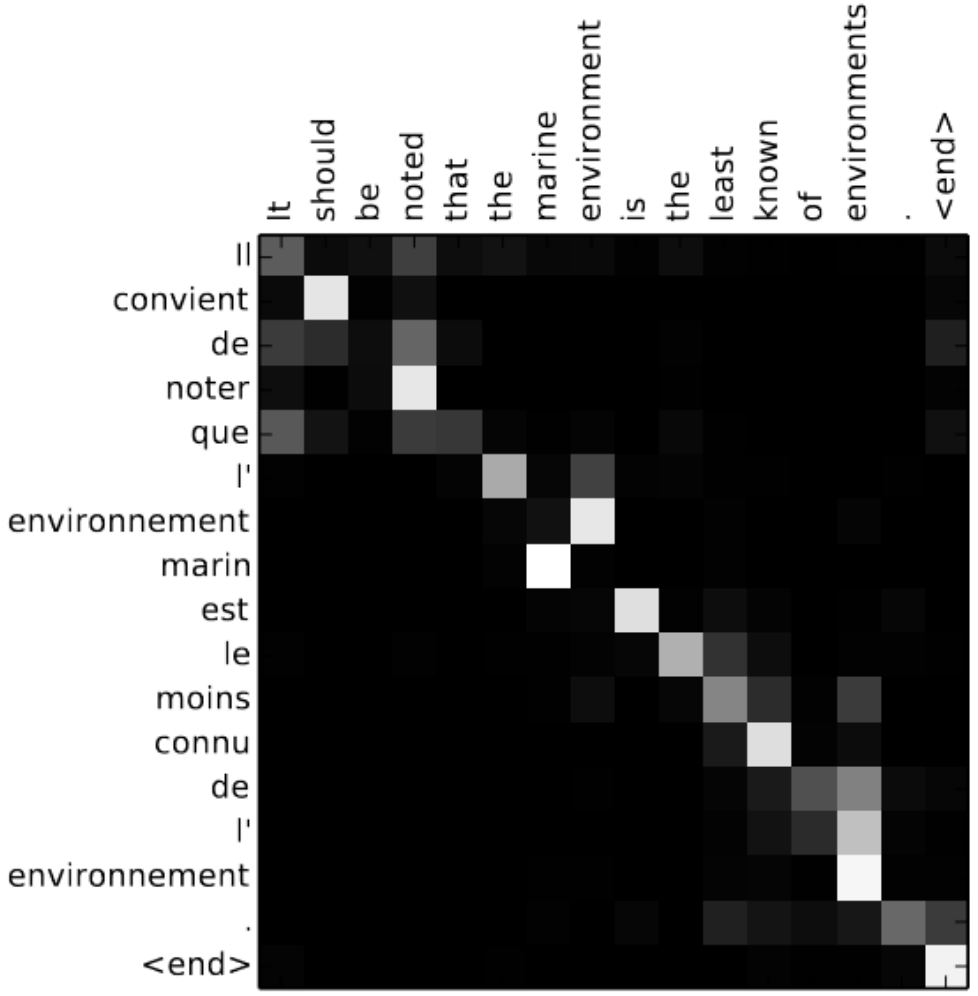
There are many implementations and they differ in the details!

NOTE: The attention “part” is all differentiable. You can learn all “weights” in an end-to-end fashion.

https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Attention weights can be visualized!
(From the paper)





Transformers

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

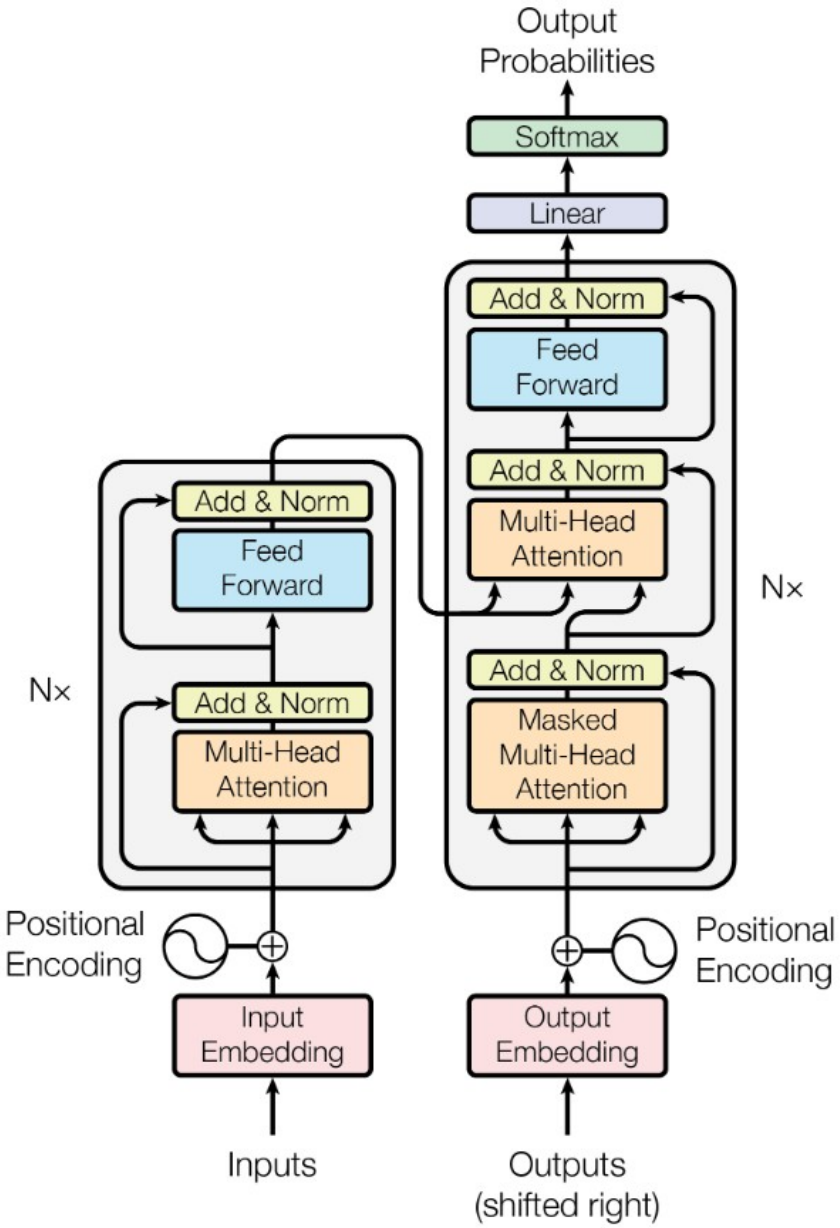
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

- The transformer model is a seq2seq model
- It uses attention in the encoder as well as the decoder
- It is not based on RNNs



The model!

Recommended reading:
https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html#transformer_intro

What is a large language model?

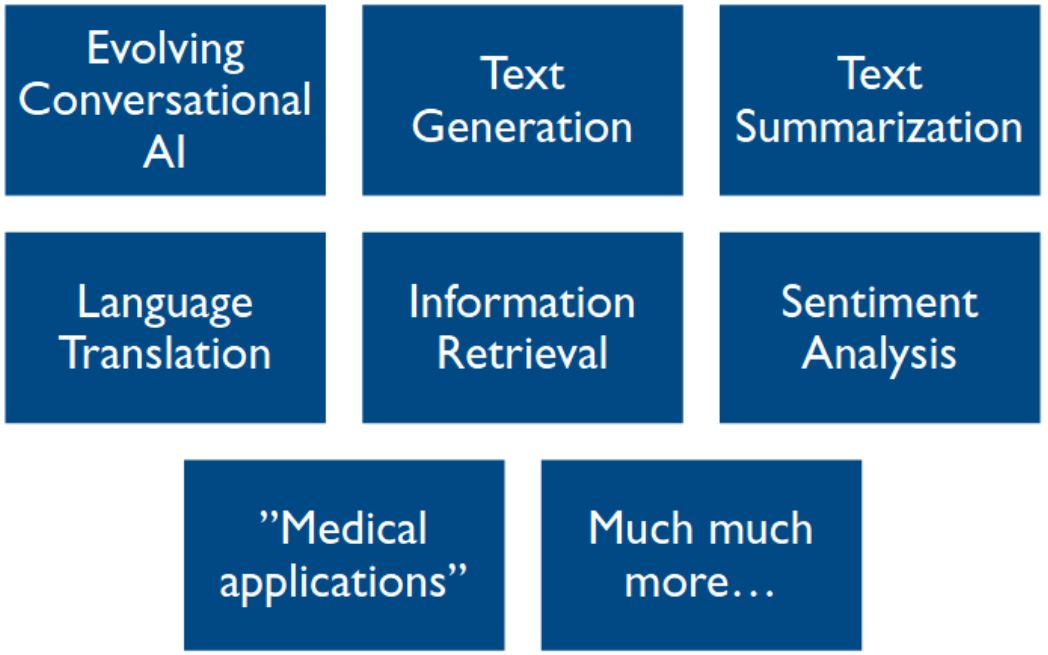
A **large language model** (**LLM**) is a type of language model notable for its ability to achieve general-purpose language understanding and generation. LLMs acquire these abilities by using massive amounts of data to learn billions of parameters during training and consuming large computational resources during their training and operation.[1] LLMs are artificial neural networks (mainly transformers[2]) and are (pre-)trained using self-supervised learning and semi-supervised learning.

Wikipedia

- Large Language Models (LLMs) are advanced AI systems designed to understand and generate human-like language.

- They use vast amounts of data to learn patterns and relationships in language, enabling them to answer questions, create text, translate languages, and perform various language tasks.

Language tasks



The concepts of:

- Pre-training
- Fine-tuning

Large amount of unlabeled text

Smaller datasets with well defined lables

My training data.

Halland is a province in Sweden. People living in Halland are called hallänningar.
Skåne is a province in Sweden. People living in Skåne are called skåningar.
Lund och Malmö are two cities in Skåne.

MLM task. Predict missing words:

Halland is a _____ in Sweden. People _____ in Halland are called hallänningar.

Next sentence prediction task:

Halland is a province in Sweden. → People living in Halland are called hallänningar.

Halland is a province in Sweden. → ~~Lund och Malmö are two cities in Skåne.~~

Predict the next token:

T1 T2 T3 T4 T5 T6 T7 T2 T8 T9 T?

How does Transformers fit into this?

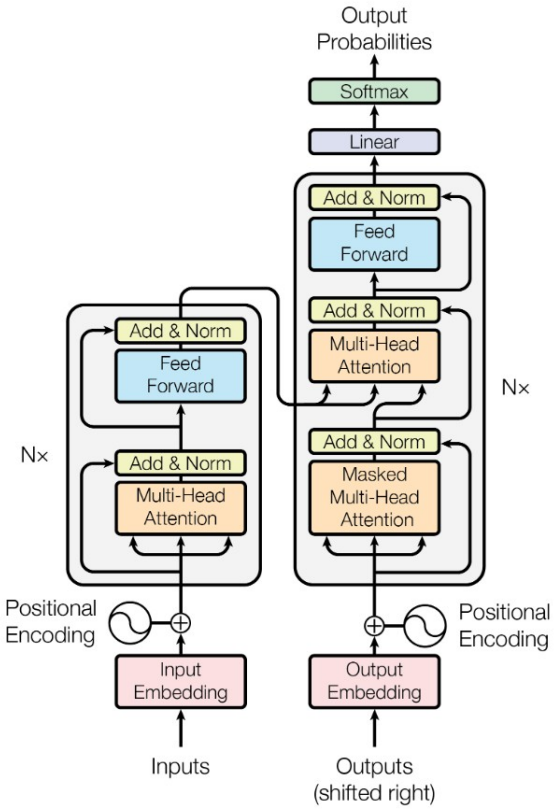
Two examples:

BERT (2018)

Encoder only architecture

Used MLM + next sentence prediction tasks for pre-training

Can be fine-tuned for a large variety of applications



GPT-1 (2018)

Decoder only architecture

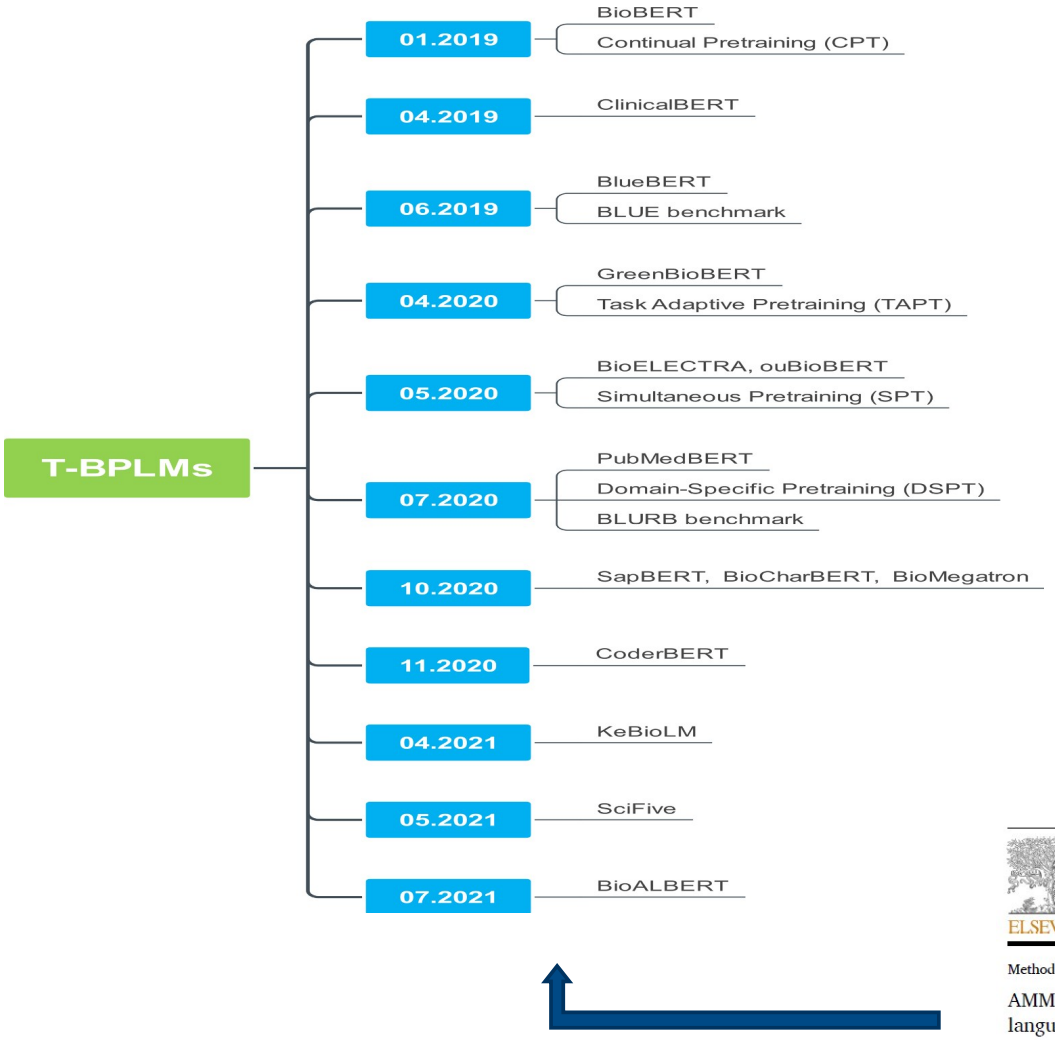
Used next-token prediction tasks for pre-training

Application:
e.g. ChatGPT

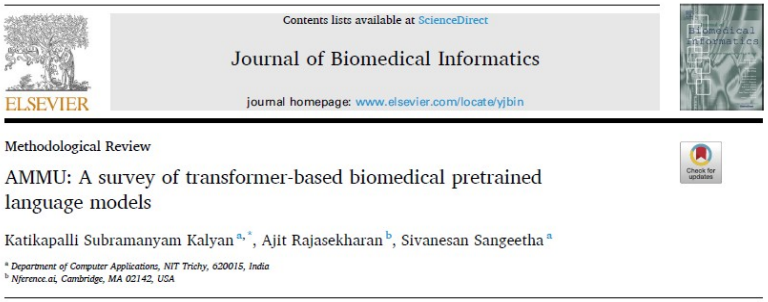
There are many LLMs out there…

- GPT 3.5 (OpenAI used by chatGPT)
- GPT-4 (OpenAI used by chatGPT)
- LaMBDA 2 (Google, used by Bard)
- LLaMA 2 (Meta, available for research, 7-70B)
- Falcon (Technology Innovation Institute, open source)
- Cohere
- PaLM 2 (Google, 340B)
- Med-PaLM (Google, fined-tuned on medical data)
- Claude v1
- Alpaca
- Vicuna-33B (LMSYS, open source model)
- Guanaco (Open source models, developed from LLaMA, 7-65B)
- RedPajama
- T5 (Google, encoder-decoder system, many offsprings)
- Stable Beluga
- MPT-30B (MosaicML, open-source model)
- 30B-Lazarus (CalderaAI, based on LLaMA))
- WizardLM
- BLOOM 176B
- Jurassic-1-Jumbo
- MT-NLG
- Wu Dao 2.0 (Beijing Academy of Artificial Intelligence, 1.75T)
- Dolly 2.0
- …

Are there any LLMs for biomedical data?



A review from 2022 lists 40+ such (pre-trained) models



Contents lists available at ScienceDirect

Journal of Biomedical Informatics

journal homepage: www.elsevier.com/locate/yjbin

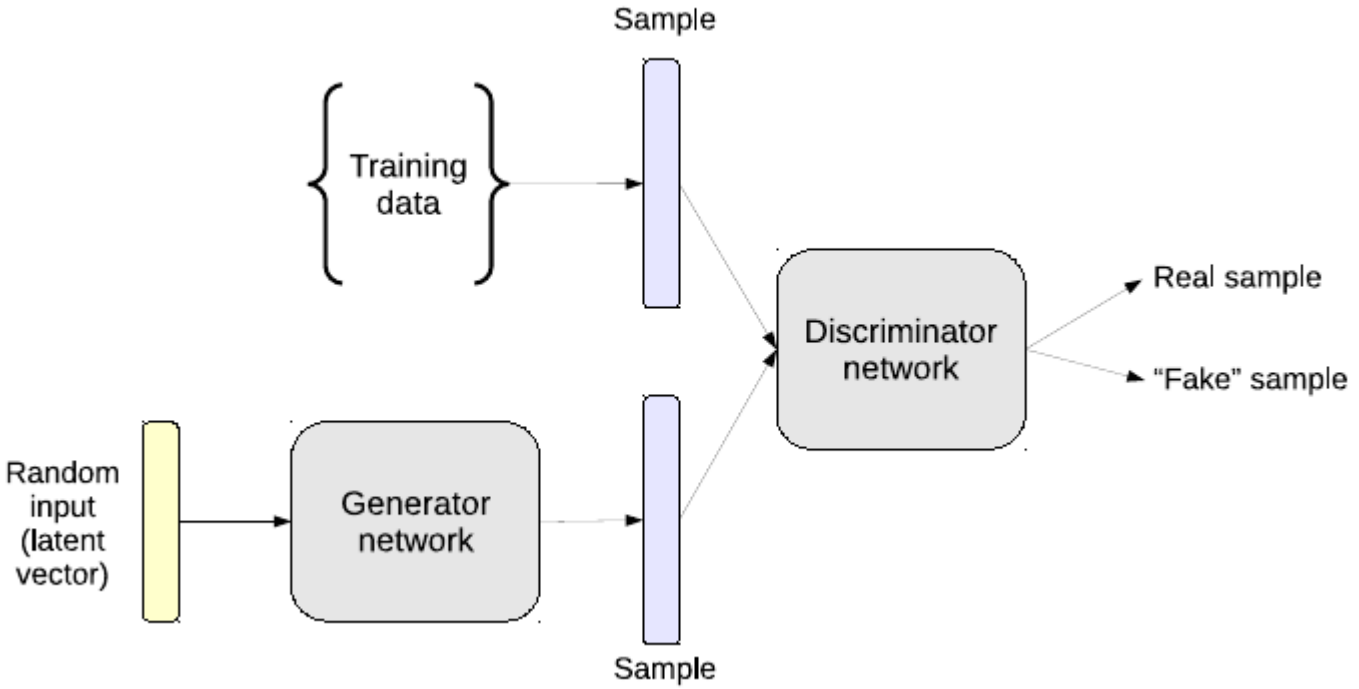
Methodological Review

AMMU: A survey of transformer-based biomedical pretrained language models

Katikapalli Subramanyam Kalyan [a,*], Ajit Rajasekharan [b], Sivanesan Sangeetha [a]

[a] Department of Computer Applications, NIT Trichy, 620015, India
[b] Nference.ai, Cambridge, MA 02142, USA

More details on GANs



Training

$$V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

$$\min_{G} \max_{D} V(D,G)$$

Generated images

Large Scale GAN Training for High Fidelity Natural Image Synthesis

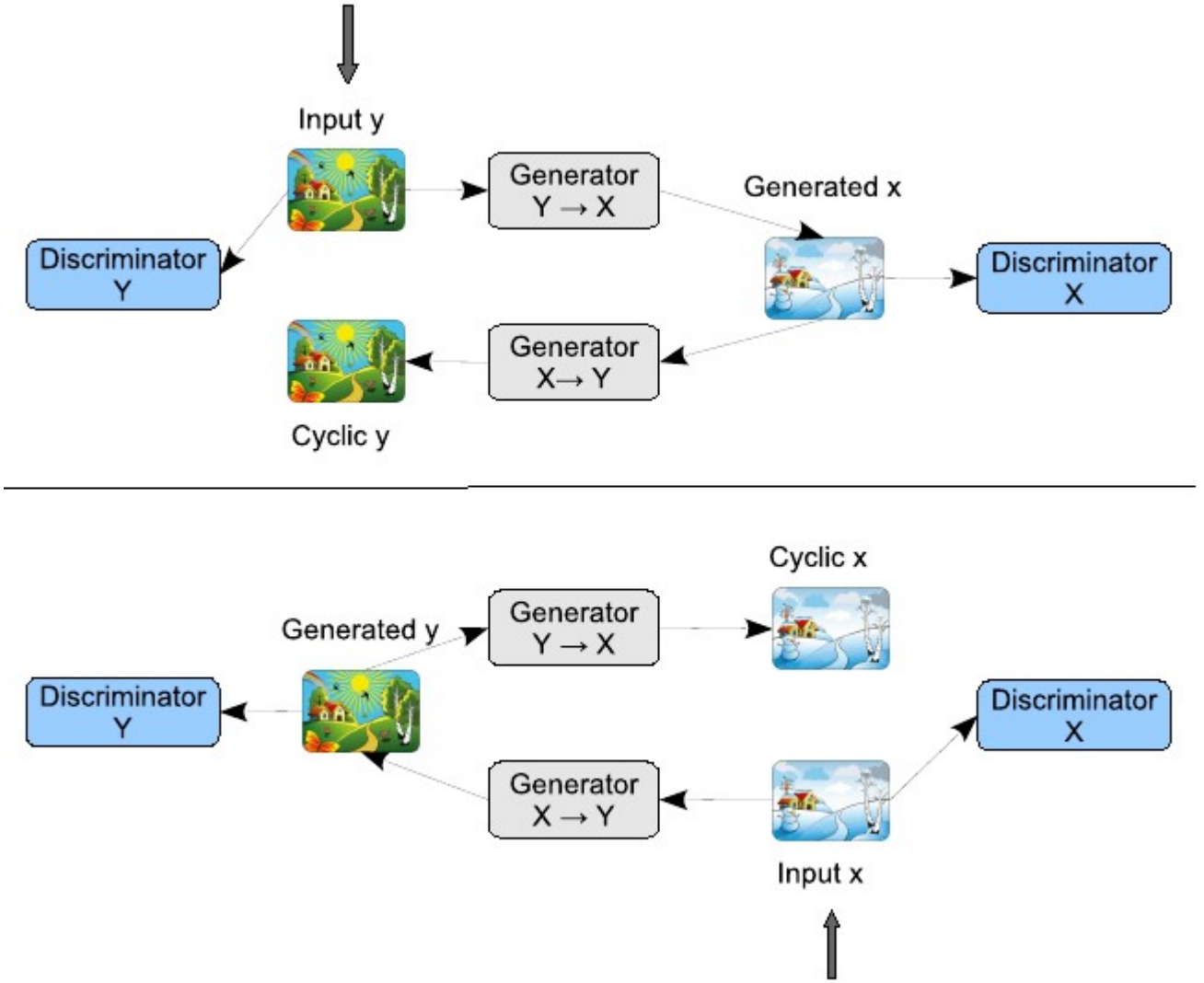
Cycle GAN

Two generators:

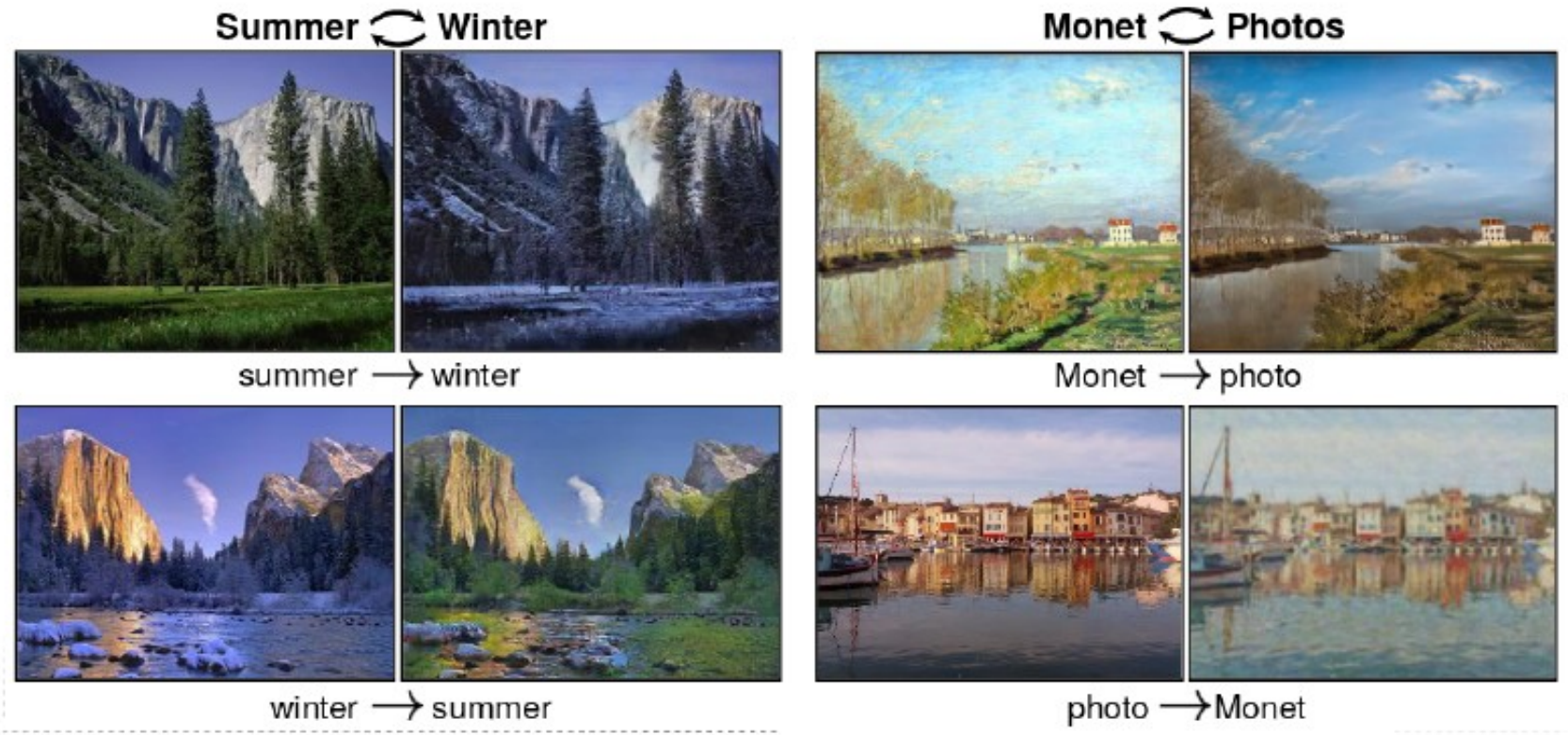
G: X → Y
F: Y → X

Two Discriminators:

Dx and Dy



Cycle GAN


Summer ⇄ Winter
Monet ⇄ Photos
summer → winter
Monet → photo
winter → summer
photo → Monet

Cycle GAN