Introduction to Artificial Neural Networks

Computing Methods for Experimental Physics and Data Analysis

Andrea.Rizzi@unipi.it - 09/11/2020

Timetable

1. Thursday, November 5th 9->11: Introduction to machine learning
   a. Basic concepts: loss, overfit, underfit
   b. Examples of linear regression, boosted decision trees
   c. Exercise with colab, numpy, scikit
2. Monday, November 9th 9->11: Deep Neural Networks
   a. Basic FeedForward networks and backpropagation
   b. Importance of depth, gradient descent, optimizers
   c. Reduction of complexity with invariance: RNN and CNN
   d. Generative Adversarial Networks
   e. Autoencoders
3. Monday, November 9th 16->18:
   a. Introduction to tools and first exercises
4. Thursday, November 12th 9->11:
   a. Exercise with CNN
5. Monday, November 16th 9->11: Graph Neural Network
6. Monday, November 16th 16->18: Graph Neural Network exercises
7. Monday, November 23rd : Exercise with CNN/LSTM/GN on an actual problem (in module 3)

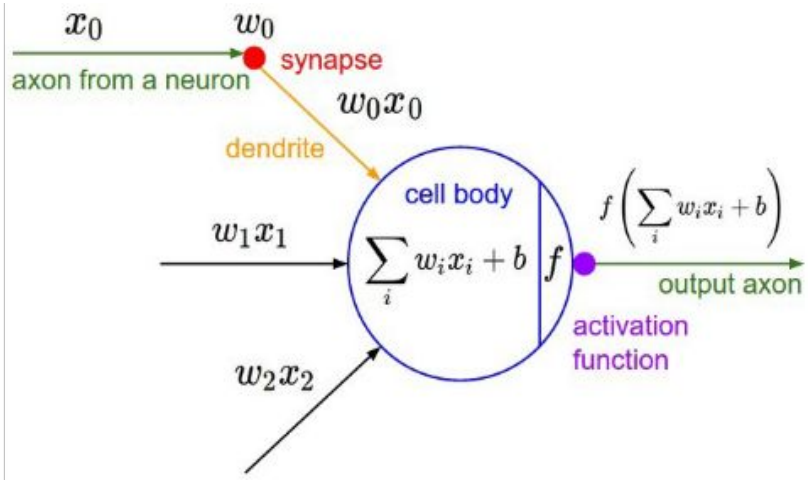
Recap of lecture 1

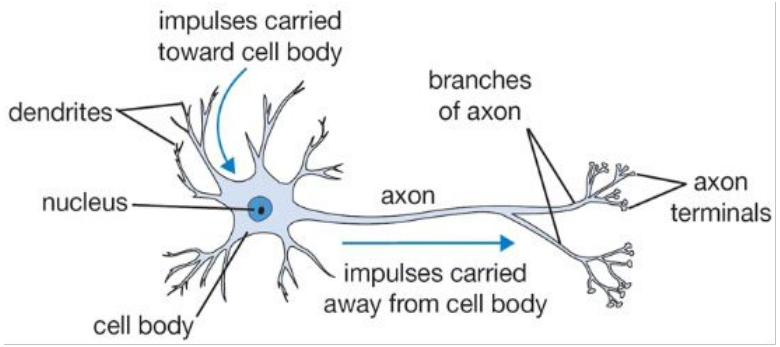
- ML techniques have common elements:
  - The function “f” to approximate
  - The model used to approximate “f” (e.g. polynomials functions)
  - The parameters of the model (e.g. the coefficients of the poly)
  - The hyper-parameters of the models (e.g. the grade of the polynomial, N=1 for linear)
  - The objective function (i.e. the loss such as MSE or binary cross entropy)
  - The variance-bias tradeoff (aka training vs generalization)
  - The regularization techniques
- Example of ML algorithms
  - Linear regression
  - PCA
  - Decision trees
    - Bagging vs boosting

Artificial Neural Networks

(Artificial) neural networks: the “Model”

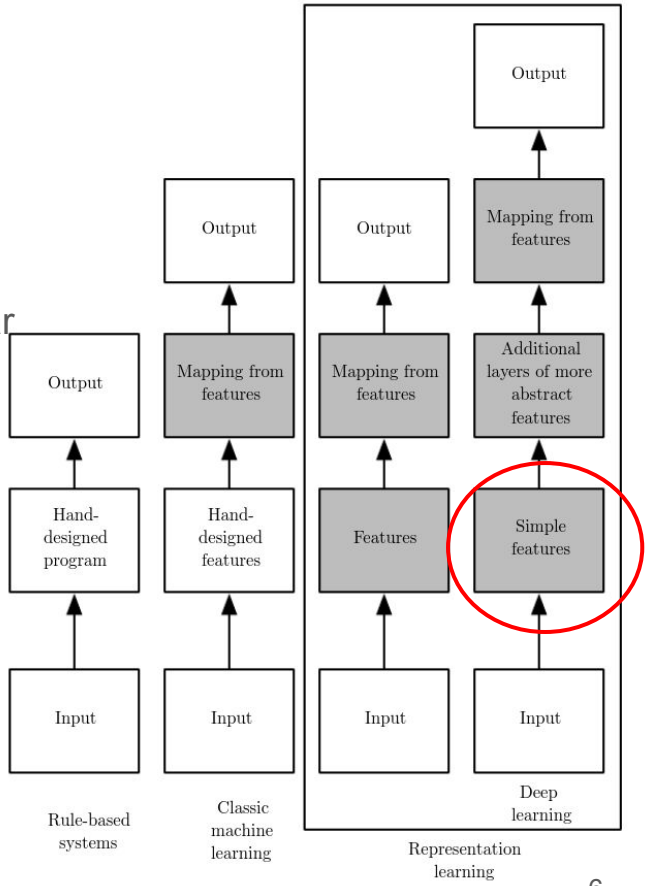
- Computation achieved with a network of elementary computing units (neurons)
- Each basic units, a neuron, has:
  - **Weighted** input connections to other neurons
  - A **non linear** activation function
  - An output value to pass to other neurons
- Biologically inspired to brain structure as a network of neuron
  - But artificial NN goal is not that of “simulating” a brain!





Brief history, highs and lows

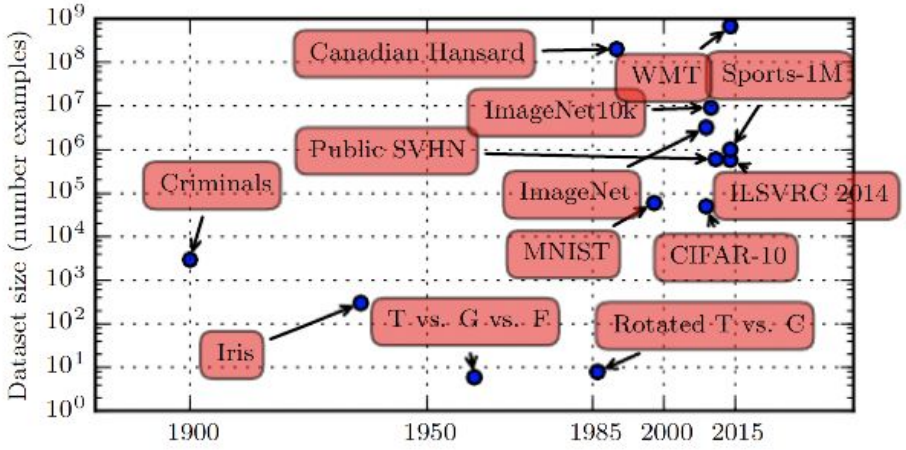
- First work originates back in ~1940-1960 “cybernetics”
  - Linear models
- Then called “connectionism” in ’80-’90
  - development of neural networks, backpropagation, non-linear activations (mostly sigmoid)
- High expectations, low achievements in the ‘90
  - A decade of stagnation
- New name, “Deep Learning”, from 2006
  - Deep architectures (see next slides)
  - Very active field in the past decade
  - Availability of GP-GPU game changing on typical “size”
  - Processing raw, low level, features
    - It doesn’t mean you **must** use “raw features” but that rather that you **can** use raw features!

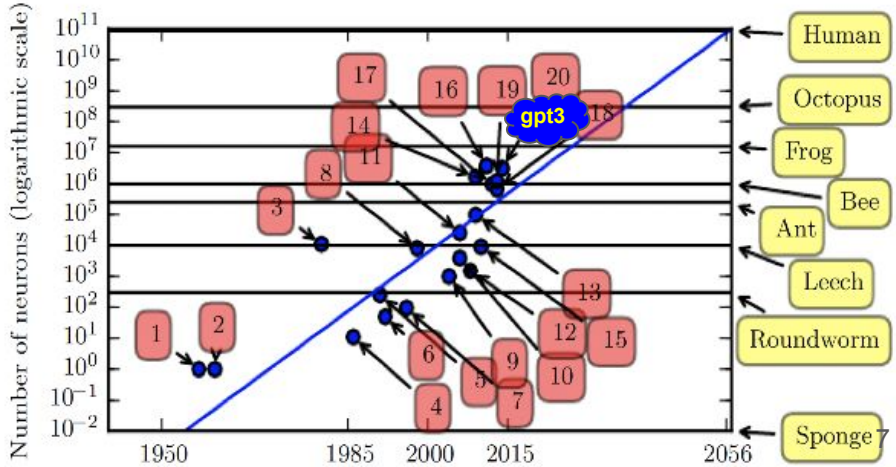


Complexity growth

- Dataset become larger and larger (“big data”)
  - Not just in “industry”, experimental scientific research is now producing multi PetaByte datasets
  - Digital era => everything can be “data”
- Increasing hardware performance

  => increasing complexity of the network (number of neurons and connections)
- Today (2020) largest ANN: OpenAI GPT3, 175 miliardi di “sinapsi”





Performance on classic problems

- Image classification and speech recognition are the typical problems where ML (and Neural Networks) failed in the 90’
- Now it beats humans ...

Speech recognition

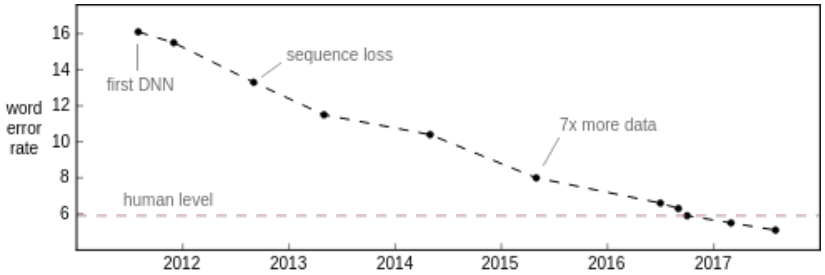
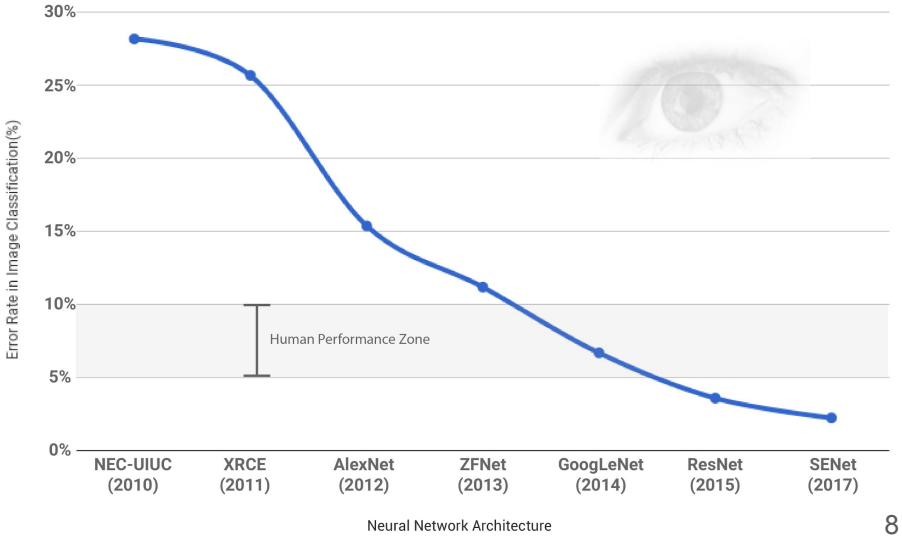
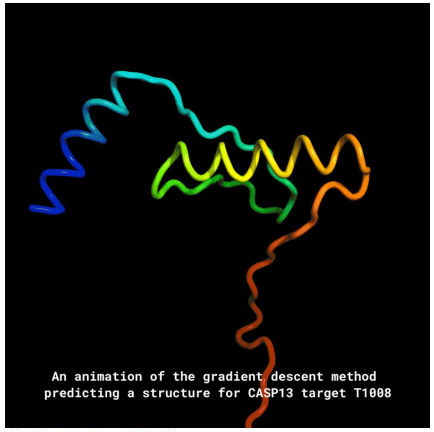


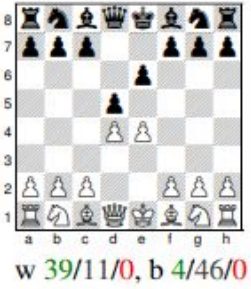
Image classification



My favorite performance examples

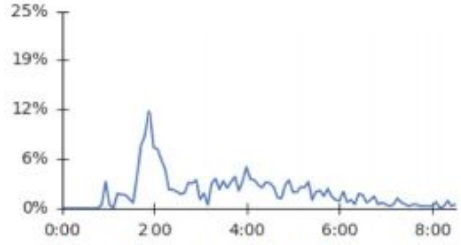
- Learning how to translate **without seeing a single translation example**, just having two independent monolingual corpora (https://arxiv.org/abs/1711.00043)
- Alpha-Fold: contest to predict protein folding, alphafold ranked first with 25 correct predictions out of 43 tests. The second ranked reached 3 out of 43.
- AlphaGo => AlphaZero: AlphaGo beat humans at “Go”, learning from human matches and know-how. Then AlphaZero learned from scratch. AlphaZero beats AlphaGo 100-0
- AlphaZero learned chess too, and beat the best existing chess program
- AI recently proved math theorems, 1200 of them
- Microsoft and Alibaba AIs beated humans in text understanding test (SQuAD)
- DeepFake: never ever believe what you see on a screen, even in videos



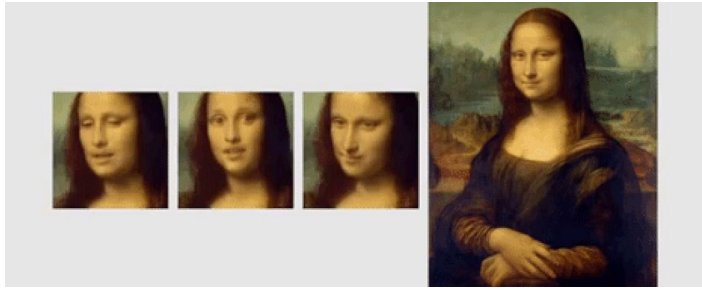


w 39/11/0, b 4/46/0

C00: French Defence



3.♘c3 ♘f6 e5 ♘d7 f4 c5 ♘f3 ♗e7



https://www.technologyreview.com/2020/11/03/1011616/ai-godfather-geoffrey-hinton-deep-learning-will-do-everything/

OpenAI GPT3

Generative Pre-trained Transformer

A 12M$ autocomplete (that is not really understanding what is talking about, but can still write better than most of us)

https://openai.com/blog/openai-api/

Solve for X:
X+40000=100000

X=50000

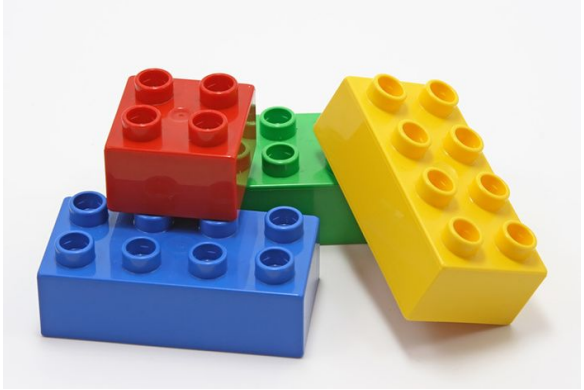
https://doi.org/10.1007/s11023-020-09548-1

Tanto gentile e tanto onesta pare
la donna mia, quand'ella altrui saluta,
ch'ogne lingua devèn, tremando, muta,
e li occhi no l'ardiscon di guardare.

Ella par che sia una cosa santa,
e in lei veder si può quanto bontà
di femmina è capace, e tanto valore
che già mai non fu da più bellezza.

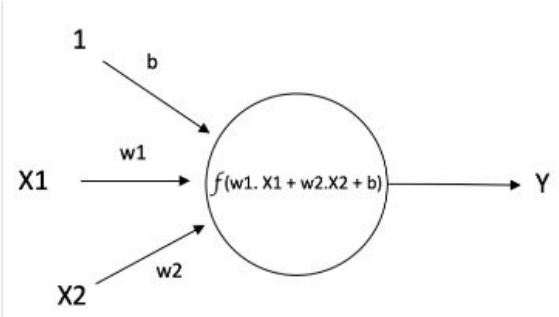
Ond'io per me la vo' sempre adorando,
et se mai la mente mia fosse offesa,
spero di far perdonando; et questo è quello
che fa ch'io son sì del tutto suo servitore.

Neural Nets Basic elements

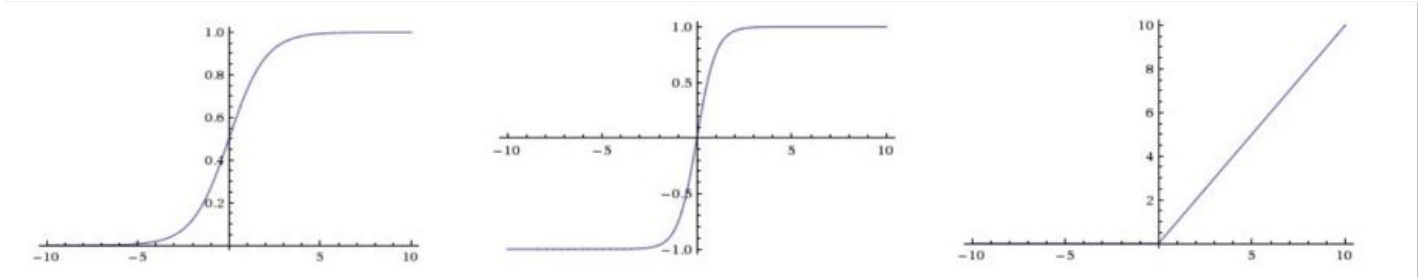


A neural network node: the artificial neuron

- The elementary processing unit, a neuron, can be seen as a node in a directed graph
- Inputs are **summed**, with **weights**, and an **activation function** is evaluated on such sum
- Nodes are typically also connected (with weight **b**) to an input “bias node” that has a fixed output value of 1
- Different activation functions can be used, common ones are: sigmoid, atan, relu (rectified linear unit)

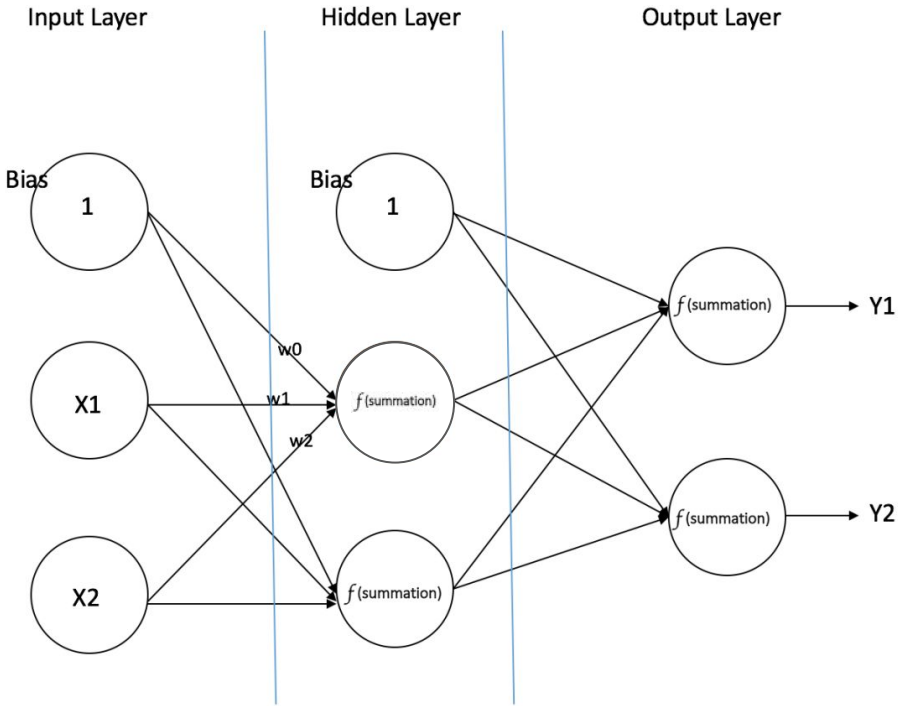


$$Y = f(w1.\,X1 + w2.X2 + b)$$



The MLP model

- The most common NN in the ‘90 was the Multi Layer Perceptron (MLP)
- Graph structure organized in “layers”
  - Input layer (nodes filled with input value)
  - Hidden layer
  - Output layer (node(s) where output is read out)
- Nodes are connected **only from one layer to the next** and **all possible connections** are present (known as “dense” or “fully connected” layer)
  - No intra-layer connections
  - No direct connections from input to output
- Size of input and output layers are fixed by the problem
- **Hyperparameters** are
  - The size of the hidden layers
  - The type of activation function
- The **parameters** to learn are the weights of the connections



Universal approximation theorem

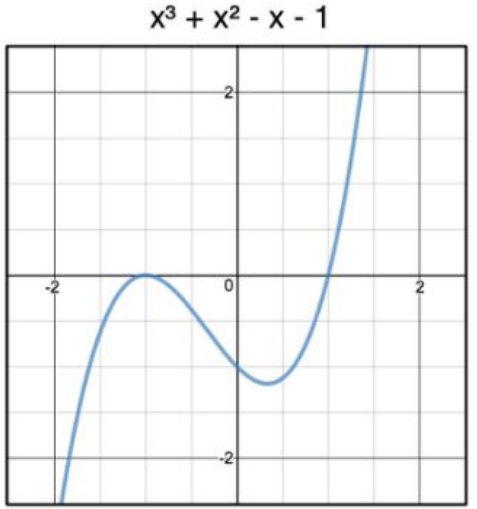
“One hidden layer is enough to *represent* (not *learn*) an approximation of any function to an arbitrary degree of accuracy” (I. Goodfellow et al. 2016)

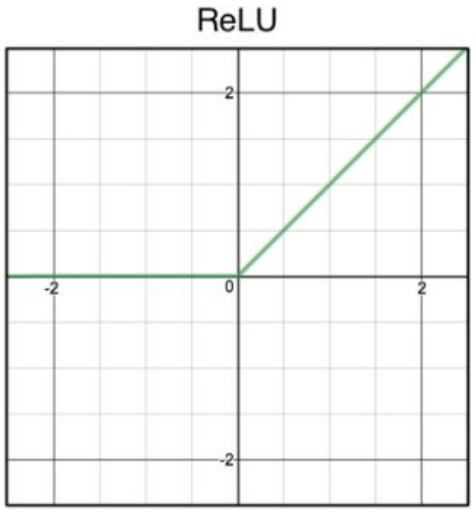
- You can approximate any function with arbitrary precision having **enough hidden nodes** and **the right weights**
- How do you get the right weights? You need a “training” for your network
  - The theorem does not say that one hidden layer (+ some training algorithm) is enough to find the optimal weights, just that they exists!
- Achieving some (even modest with some metric) level of accuracy may need an unmanageable hidden layer size
  - And may need an unreasonable number of “examples” to learn from

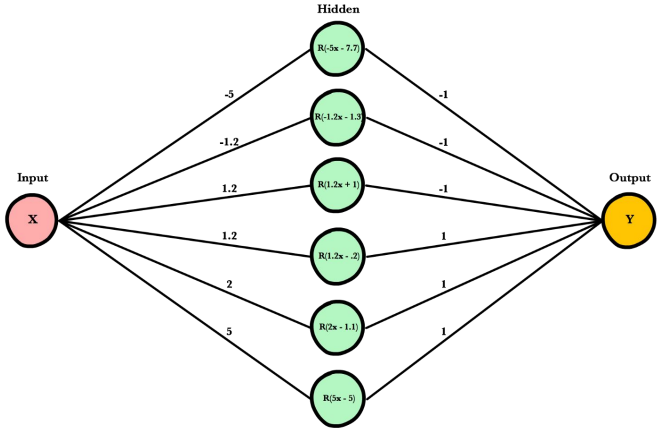
Example (1-D input)

Approximate this function

With a weighted sum of functions like this one

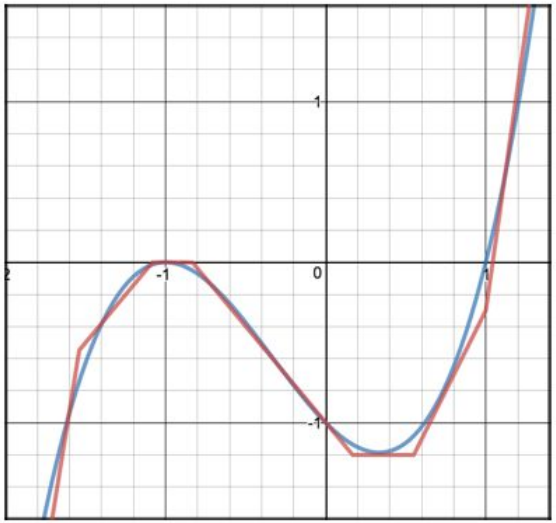






Example

- The Universal Approximation Theorem says that increasing #nodes I can increase the accuracy as much as I want
- More hidden nodes, higher “capacity” => more accuracy



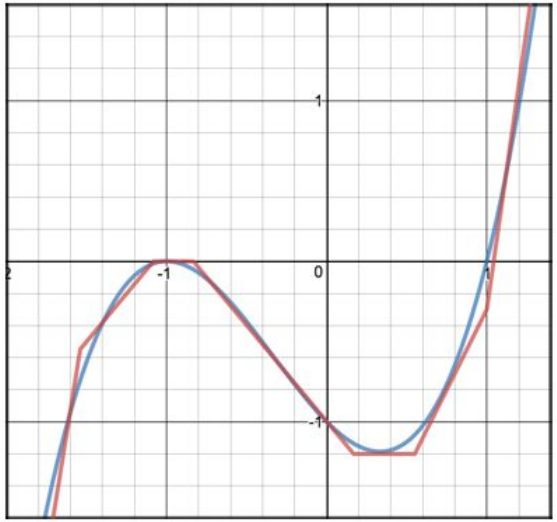
$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6

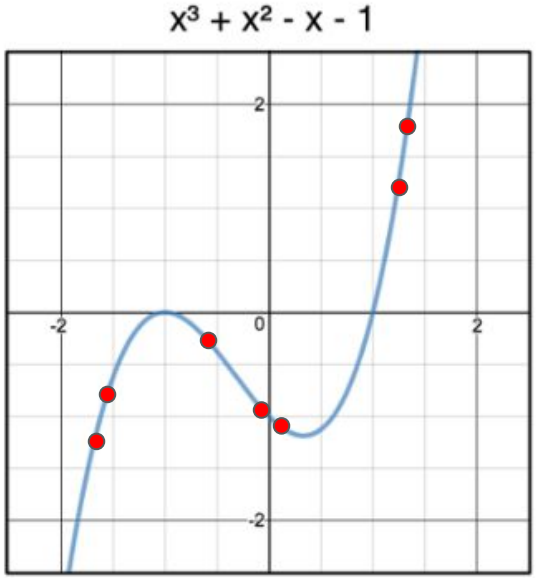
Training of an MLP

- How do I get the weights?
- Remember: we do not know the function we want to approximate, we only have some “samples”

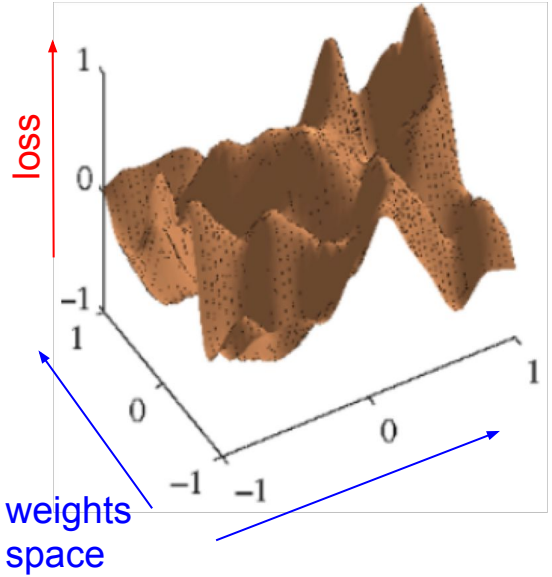


$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$



Training a NN

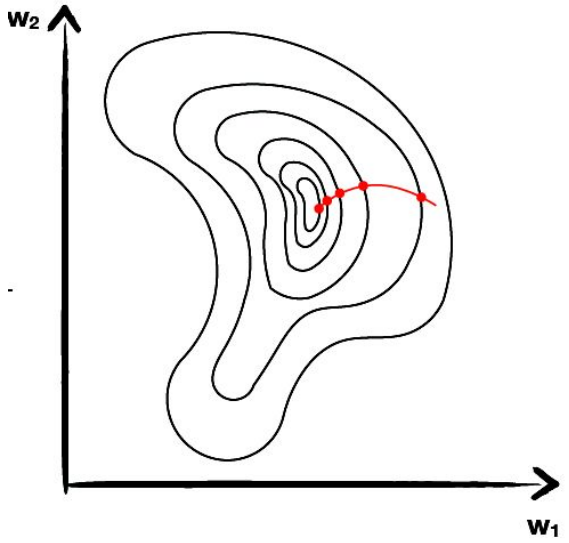


- The goal of training is to minimize the objective function (possibly both on the training and validation sample)
  - I.e. we want to minimize the loss as a function of the model parameters (i.e. the weights)
- For a MLP the basic idea is the following
  a. Start with random weights
  b. Compute the prediction for a given input ***x*** and check the difference with target *y* and the loss (repeat for a few example, aka “one batch”)
  c. Estimate an update for the weights that **reduces the loss**
  d. Iterate from point (b), repeating for all samples
  e. When the sample has been used completely (end of an epoch), iterate from (b) again on all samples
  f. Repeat for multiple epochs
- The important point is how to implement point (c) => (stochastic) gradient descent

How to find a minimum?

Gradient Descent

- We know the loss function value in a point in the weights phase space (e.g. the initial set of random weights, or the iteration N-1), computed numerically as the mean or the sum of the losses for each of our training examples
- We can compute the gradient of the loss function in that point, we expect the minimum on “the way down” hence we adjust our set of weights doing a “step” in the direction pointed by the gradient with a step size that is proportional to the length of the gradient

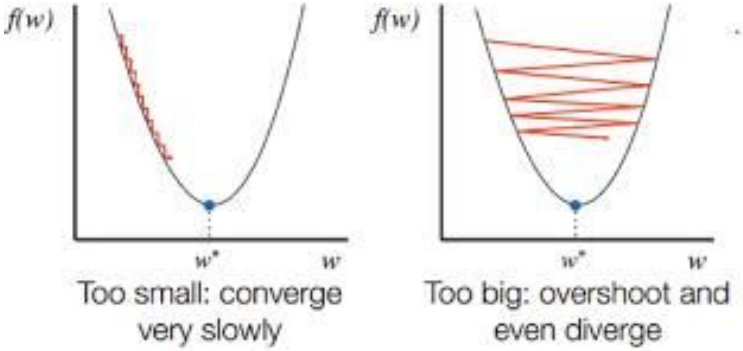


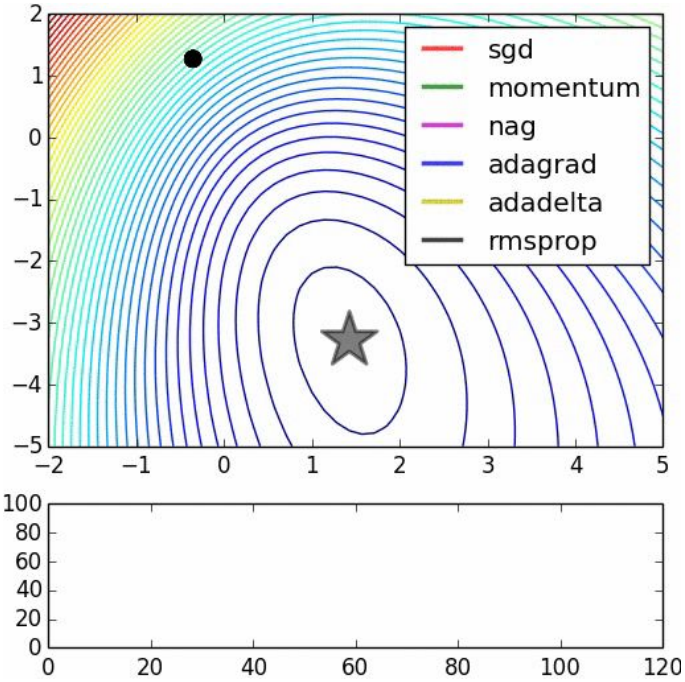
Stochastic Gradient Descent (SGD):

- Compute the gradient on “**batches**” of events rather than full sample
- The “noise” may help avoiding local minima

Not as simple as you would imagine

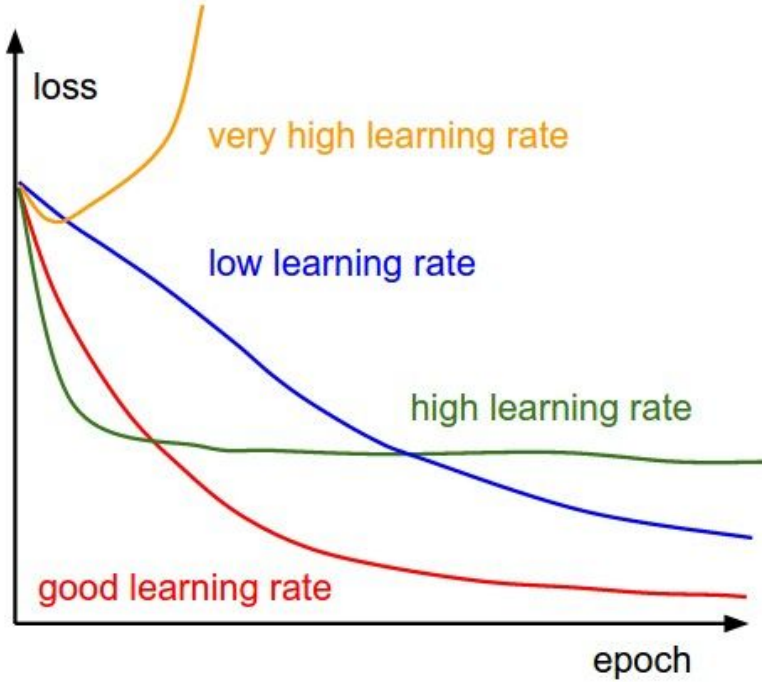
- A parameter named **learning rate** controls how big the step in the direction of the gradient is
  - A too large step may let you bounce back and forth on the walls of your “valley”
  - A too small step would make your descent lasting forever
- Several variants of SGD
  - Include “momentum” from previous gradient calculations (may help overcome local obstacles)
  - Reduce step size over time
  - Adadelta, Adagrad, Adam, and many more



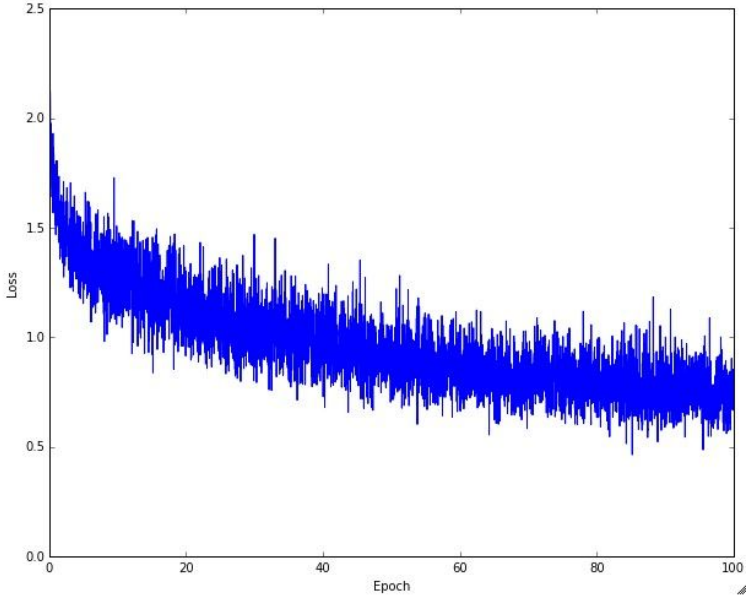


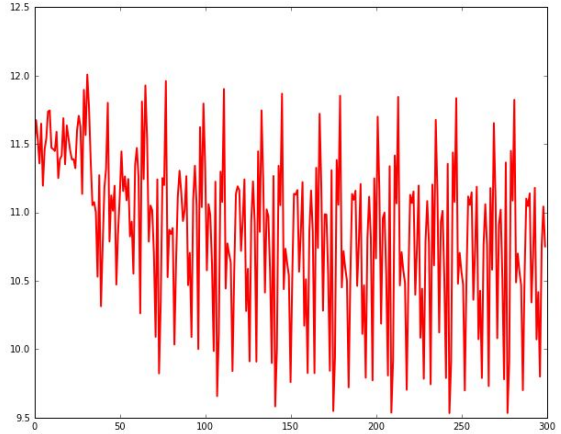
Learning rate, epochs and batches

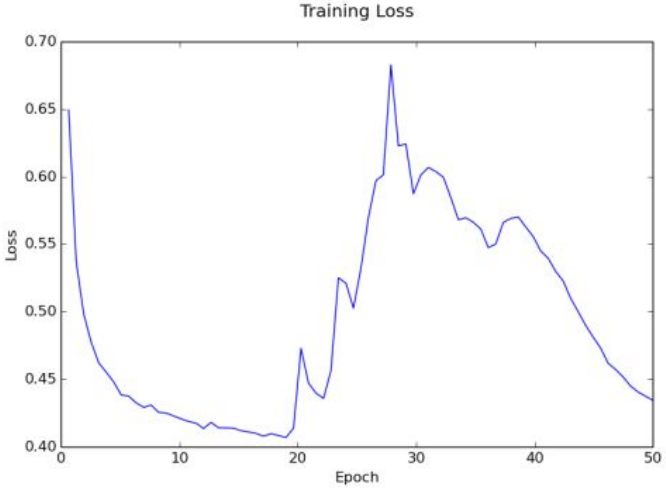
- The gradient update (in SGD) is repeated for each “batch” of events
- A full pass of the whole dataset (i.e. all batches) is called an **epoch**
- A typical training foresee iteration on multiple epochs
- The size of the update step can be controlled with a multiplicative factor called “**learning rate**”
    - Learning rate can be adapted over time



In reality

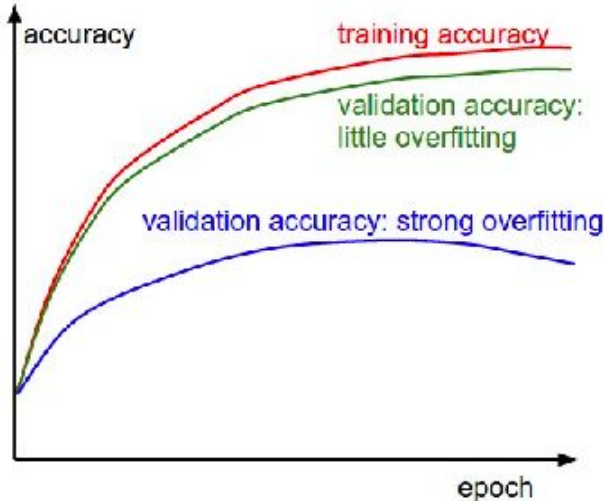






Training and overfitting

- As discussed earlier if the capacity is large enough the network could “overfit” on the training dataset
- Have a separate, stat independent, validation/generalization sample
- Evaluate performance (with “loss” or with other metrics) on the validation sample
- Training results depends on many choices
  - Size of batches (amount of “noise”)
  - Learning rate (how much you move along the gradient at each iteration)
  - Gradient Descent algorithm
  - Capacity of the network



Neural Networks, computers and mathematics

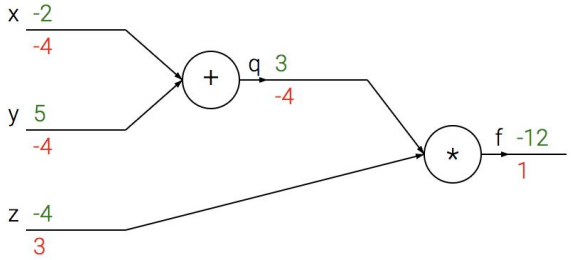
- ANN use a fairly limited and simple set of operations
  - Many operation are simply represented with linear algebra
  - Non linear function are typically applied, repeated, to multiple inputs (hence can be “vectorized”)
- Datasets are represented as multidimensional tensors
  - The actual size of an index is usually called “shape” and is a tuple with dimension of each index
  - The first index is the one running the “number of sample in the dataset”, and is sometimes omitted when describing a neural network
- Classification with multiple category is often converted in the “categorical” representation
  - I.e. rather than labelling with a scalar “y” (with 0=horse, 1=dog, 2=cat, 3=bird, …) a vector **y** is used with as many components as the category (with [1,0,0,0]=horse, [0,1,0,0]=dog, etc..)
- Tools exist to describe mathematically the network structure that are optimized for fast computations on CPU/GPU/TPU

Back-propagation

Calculating the gradient in complex networks could be computationally expensive:

- Some expression appear repeated, hence we should avoid recomputing them
- Back-propagation method allow to efficiently compute the derivatives wrt each of the weights
  - Start from the last node and apply derivative chain rule going backward
  - At each step the computation depends only on the already computed derivative and the values of the node outputs (computed already in the NN evaluation, aka forward pass)

$$f(x, y, z) = (x + y)z$$



$$q = x + y \qquad f = qz$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1 \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

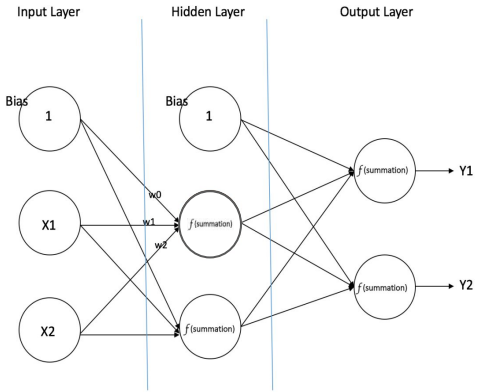
Deep networks

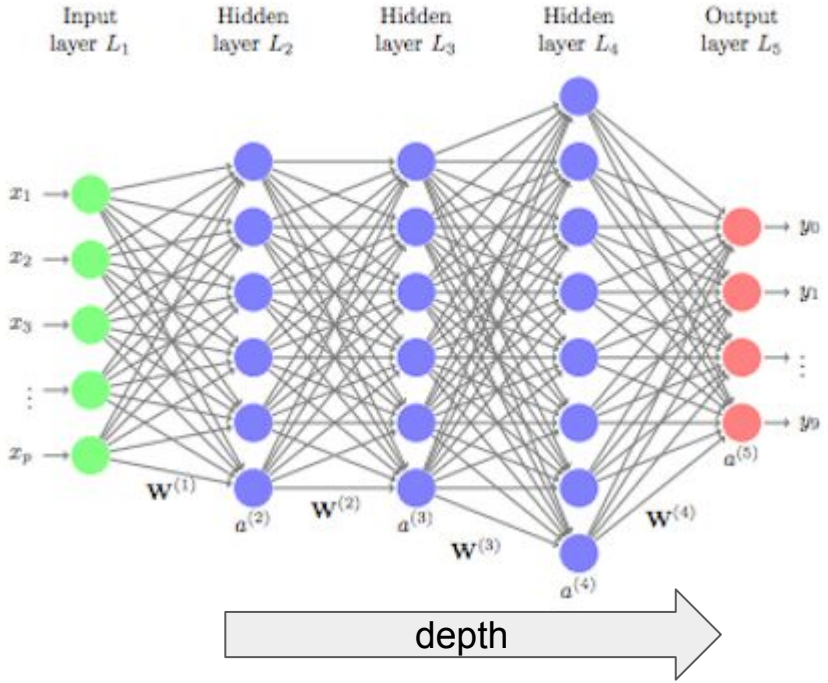
Deep Feed Forward networks

The simplest extension to the MLP is to just add more hidden layers

Other names of this network architecture

- Deep Feed Forward network
- Deep Dense network, i.e. made (only) of **Dense**(ly) connected layers





Why going deeper?

Hold on… wasn’t there a theorem saying that MLP is good enough ? Yes but…

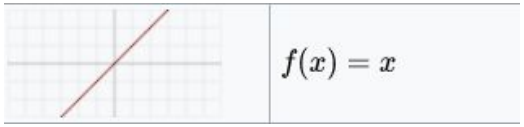
- Amount of nodes to represent complex functions can be too high
- Learning the weights on finite samples could be too difficult

Advantages of Deep architectures

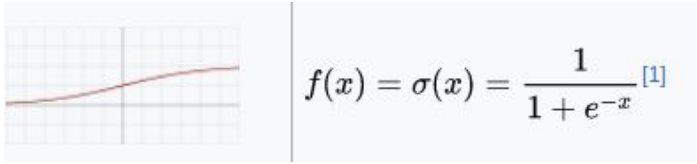
- Hierarchical structure can allow easier “abstraction” by the network with early layers computing low level features and deeper layers representing more abstract properties
- Number of neurons and connections needed to represent the same function highly reduced in many realistic cases

Activation functions

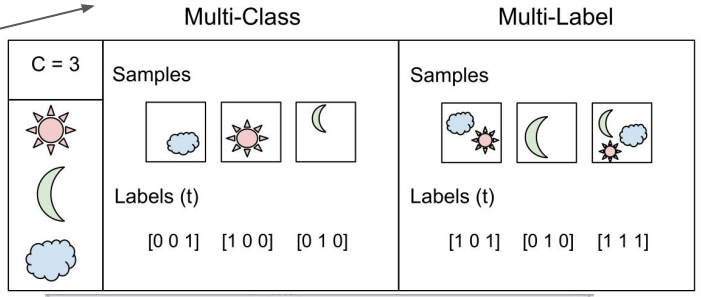
- Linear:
  - Cannot be used in hidden layer has the derivative is constant (does not depend on the inputs, cannot perform gradient descent)
  - Useful for output nodes in **regression** problems
- Sigmoid:
  - Used in the past in the hidden layer (leads to vanishing gradient problem)
  - Useful in **classification** problem with a single output or **multi-label** classifiers
- Softmax:
  - Useful in **classification** problems with one-hot encoded **multi-class**
- Rectified Linear Unit (ReLU)
  - The workhorse for hidden layers activations
  - Variants: Leaky-ReLU, Swish



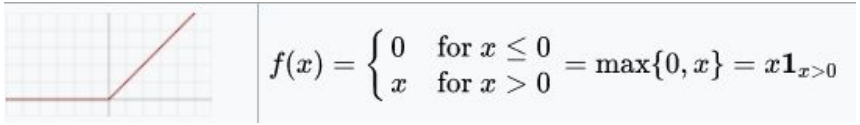
$$f(x) = x$$



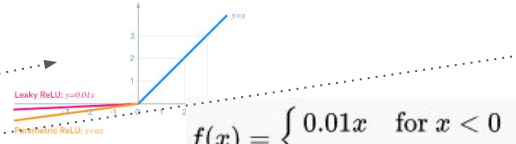
$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$ [1]



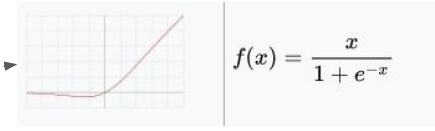
$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad \text{for } i = 1, \ldots, J$$



$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$$

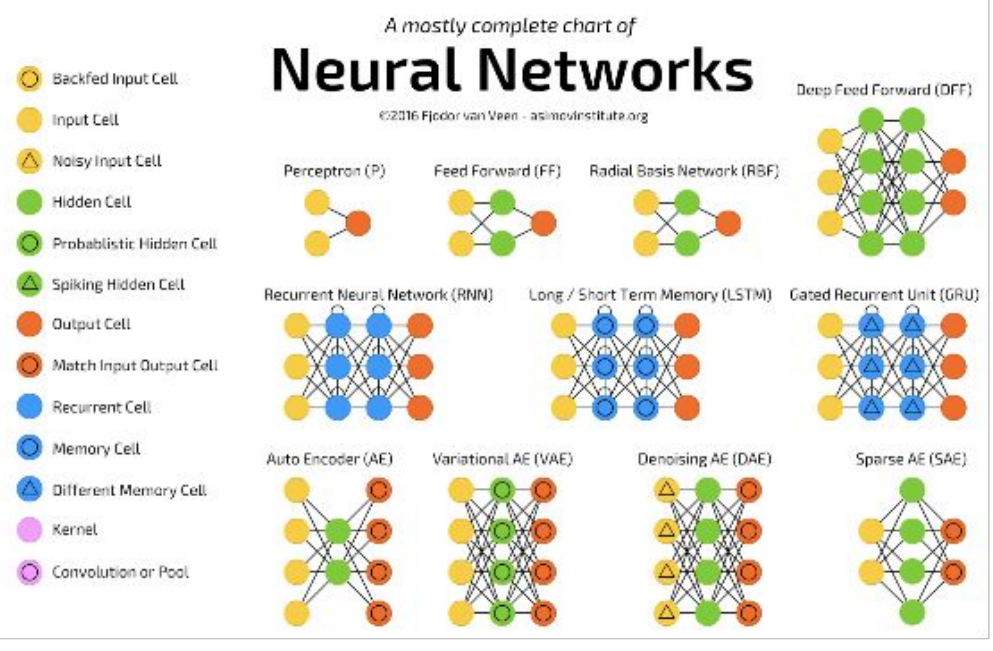


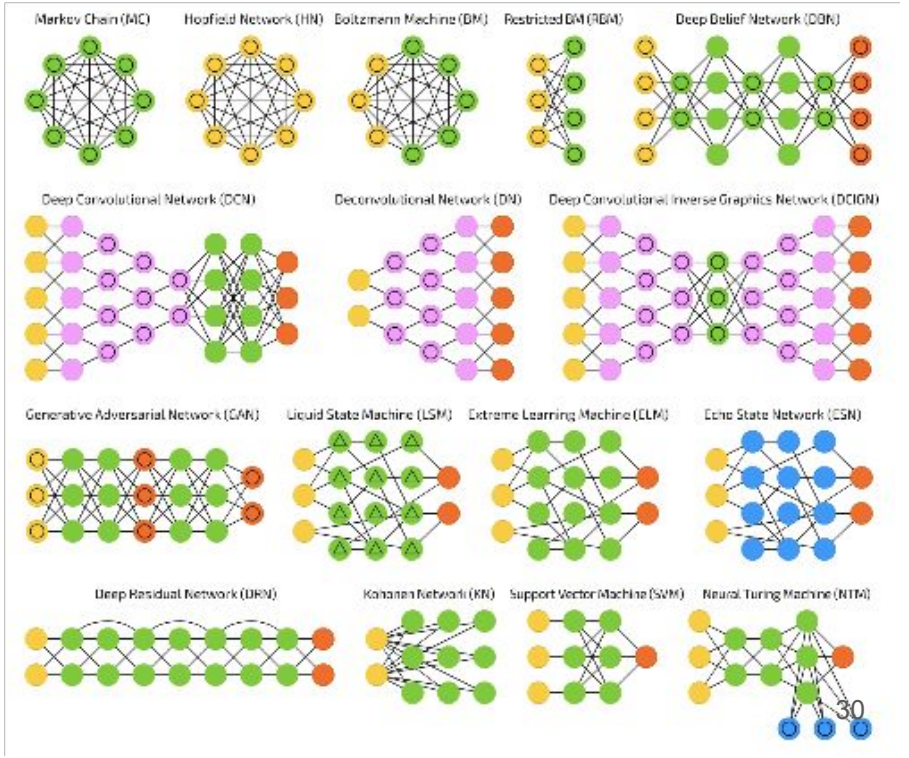
$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



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

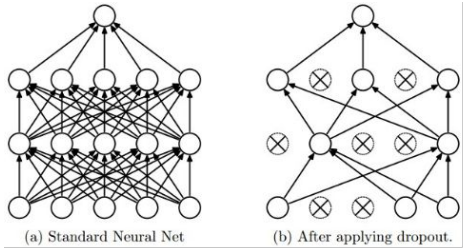
Deep architectures

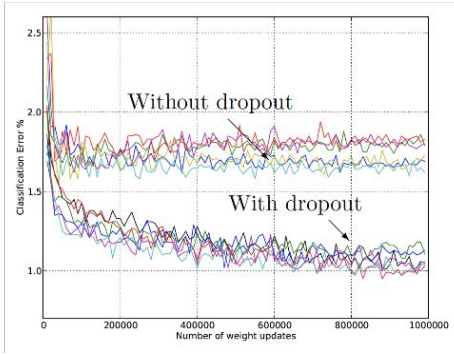




Dropout and regularization methods

- NN training is a numerical process
- Often the number of samples is limited hence the gradient accuracy is not great
- Several regularization methods exists to avoid being dominated by stochastic effects
  - Caps to the weights (so that individual nodes cannot be worth more than some amount)
  - **Dropout** techniques: during the training a fraction of nodes is discarded, randomly, at each iteration
    - NN more robust to noise
    - Effectively “augmenting” the input dataset

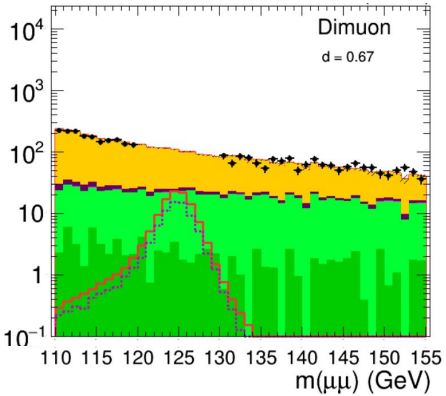


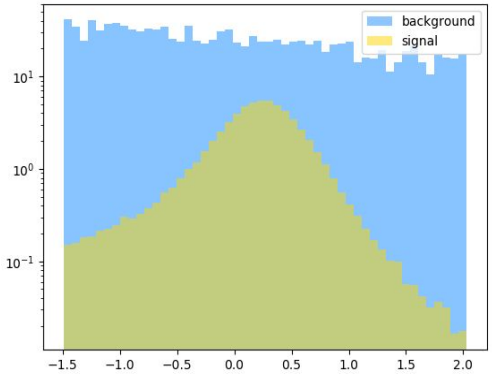


(Batch) normalization

- Input features have typically different ranges, means, variance
- It is generally useful to “normalize” the input distribution
  - Mean zero
  - Variance 1
- Often it could be practical to compute the normalization on individual batches rather than full sample
  - Batch vs full sample ? may depend on your use case

A typical observable, e.g. the invariant mass of a pairs of leptons





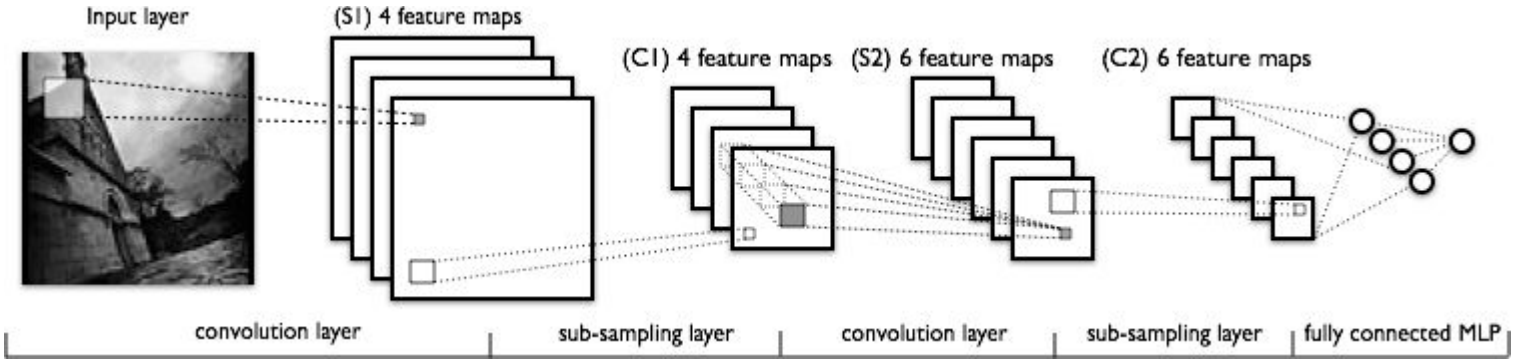
Normalized version

Exploit invariance and locality

- Suppose you want to count windows in a 800x600 picture with houses
  - With an MLP or DFF you have 800x600x3(RGB)=1.4M inputs
  - Each node process independently some part of the image
  - The initial “Dense” connection should converge to something with lot of “zero” weights because far away pixel points have no reason to be considered at the same time in order to detect **local** features
  - => the problem cannot be managed this way
- But the problem is translation invariant!
  - “Windows” are local features, you can just analyze a patch of the image **(locality)**
  - A window is a window no matter if it is top left or bottom right of your image **(Invariance)**
  - And actually windows are made of even more local features (some borders/frame, some uniform area, a squared shape)

Can we exploit problem invariance?

- Convolutional neural networks (CNN) attempt to exploit invariance against spatial translations
  - Smaller networks
  - Acting on a single patch of the image
  - Stacking multiple such Convolutional Layers one after the other

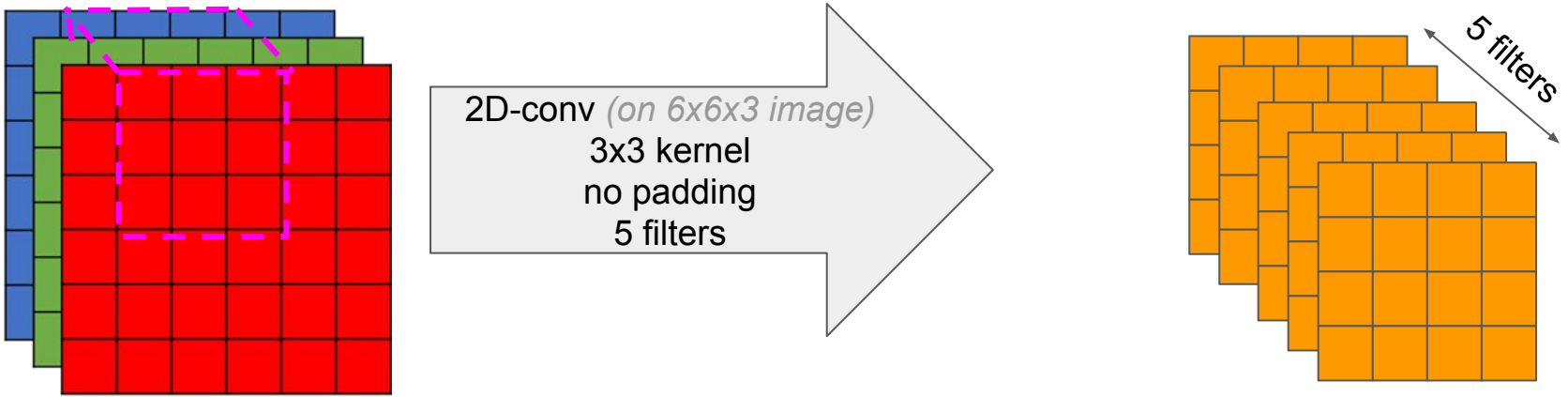


Limitations

- The linear algebra formalism we use can handle nicely images, hence implement nicely CNN (translation invariance along x and y)
- There are more invariances out there!
  - Rotation
  - Scale
  - Luminosity
  - … you name it…
- So currently the networks have to learn them all
  - We can do tricks to increase the number of samples in our datasets with augmentation techniques (i.e. apply random transformations of scale, rotation etc..)
  - “Built-in” invariance (such as the x-y one) has the advantage of reducing by orders of magnitude the number of weights to learn

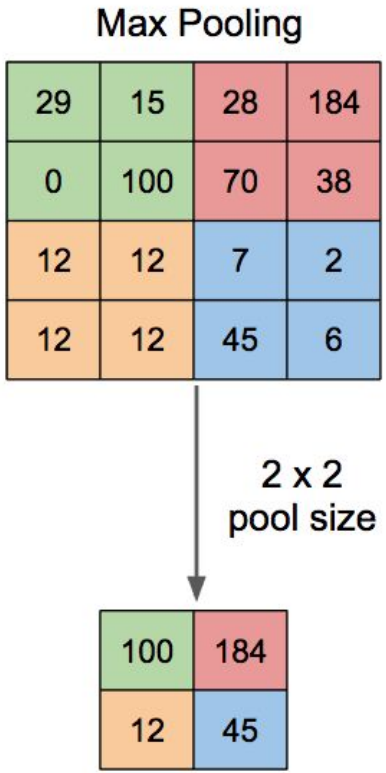
Understanding the dimensions of the convolution

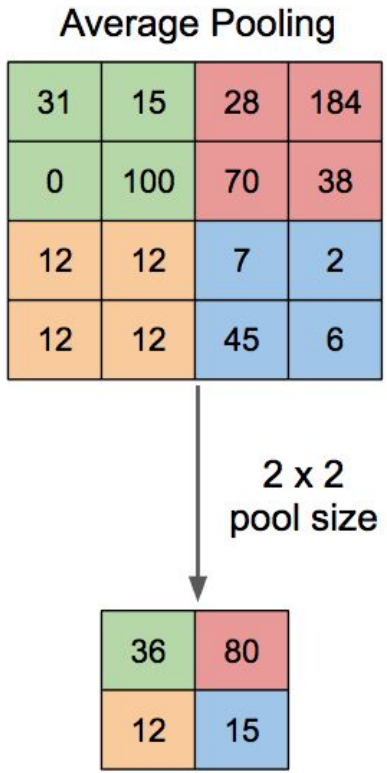
- Convolution can be 1D, 2D, 3D
- Kernel size, typically square (MxM) with M odd (but can be any shape)
- *Padding*: how to we handle borders? We can do only “valid” windows (no padding) or process borders as if there were zeros (or other values) outside
- Each “point” in the 1D, 2D, 3D matrix can have multiple features (e.g. R,G,B)
- Each Convolutional layer have mutiple outputs (filters) for every “patch” it scans on (one optimized to detect if the patch is uniformly filled, one looking for vertical lines, etc..)



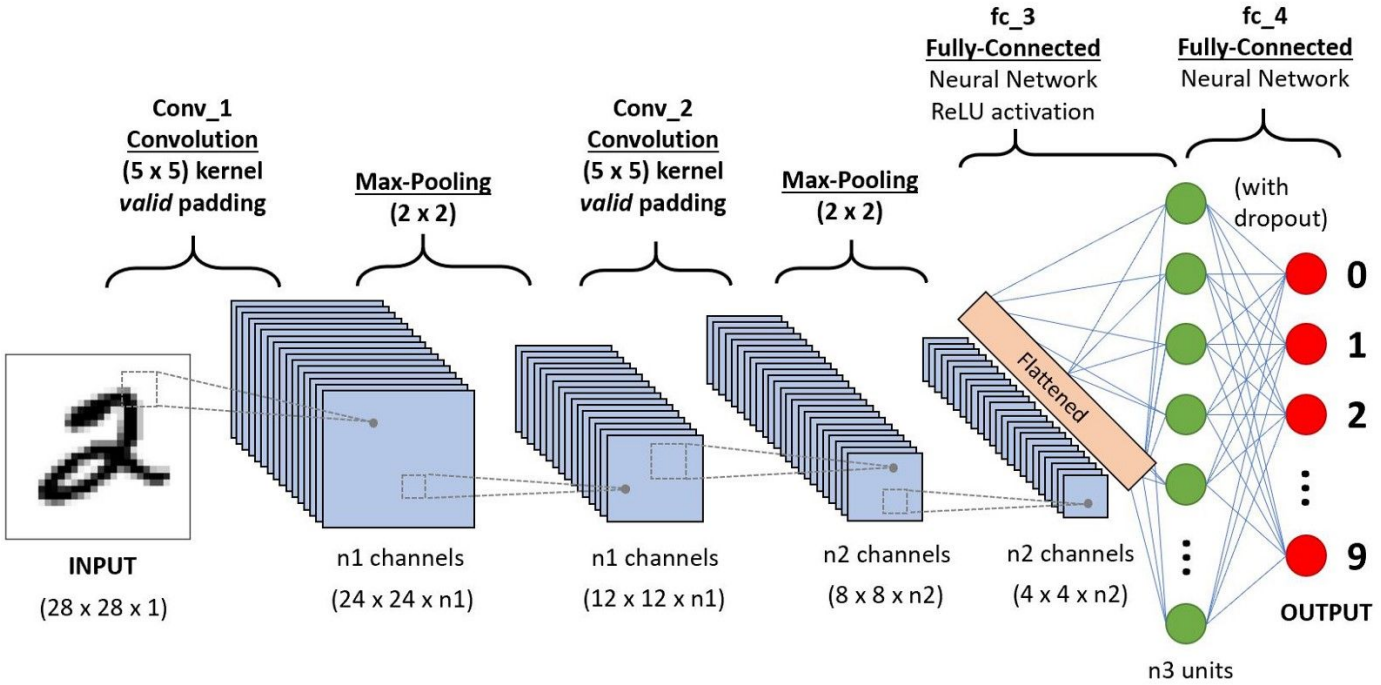
Pooling

- Pooling layers are simply finding maxima or computing average in patches of some convolution layer output
- Pooling is used to reduce the space dimensionality after a convolutional layer
  - The Conv “filters” look for features (e.g. a filter may look for cats eyes)
  - The Pooling layer checks if in a given region some filtered fired (there was a cut eye somewhere in this broad region)



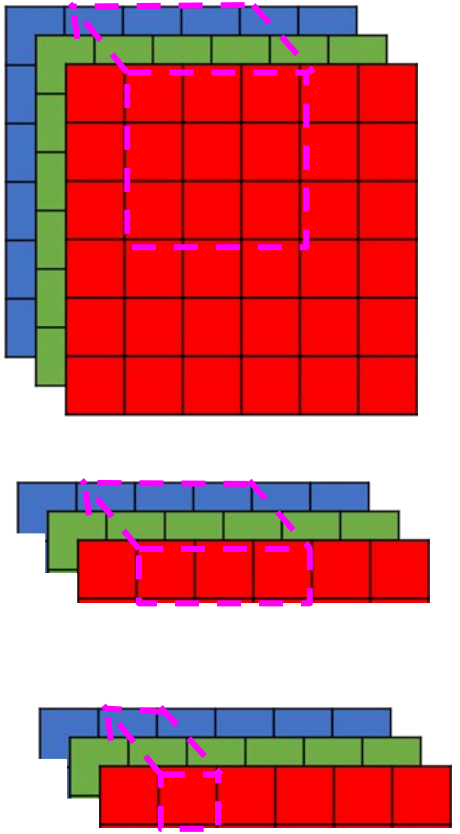


Typical CNN architecture



More on convolution

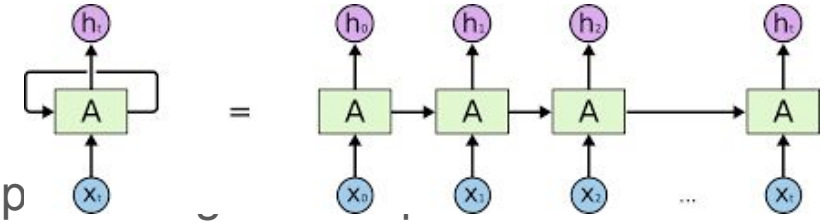
- Convolution is a way to correlate **local** input information and to reduce the NN size by sharing the weights of the nodes across all repeated patches.
- What if I have multiple objects, with no local correlation, but with multiple features (like R,G,B channels) and I want to process them all in the same way?
  - 1x1 convolution!
  - Conv1D is usually enough (as the x-y coordinates have no meaning here)
- Example :
  - Particles in a detector with information about 4-vector, tracking hits, calorimeter deposits, p-ID etc… and want to preprocess them one by one before using them for some higher level task



Exploiting time invariance

- Some problems are “time invariant”
  - E.g. recognize words in a sentence (written or spoken)
- Order matters and some causality is implied in the sequence
- Length of the inputs or the output may not be fixed

Recurrent Networks (RNN)

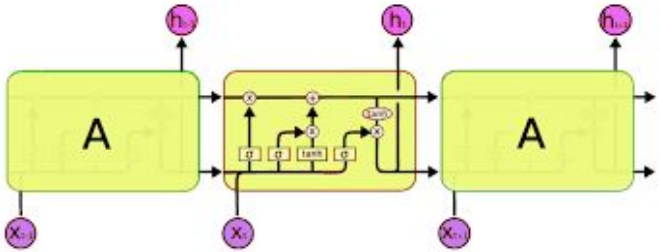


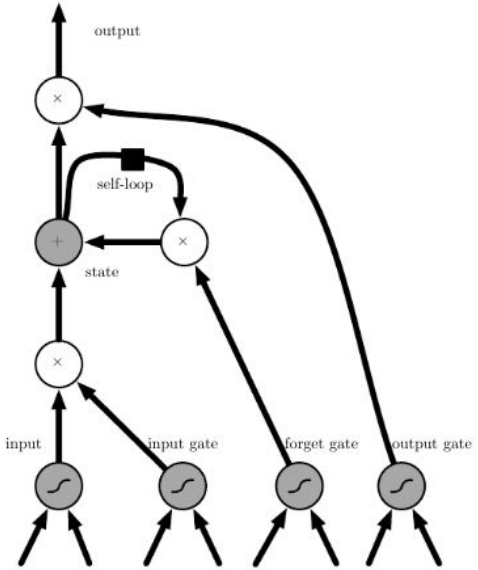
- Iterative networks with output p
  - Allow some “memory” of the previous inputs and/or some internal “state” of what the network understood so far in the sequence
- Most commonly used RNN are LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit)

LSTM and GRU

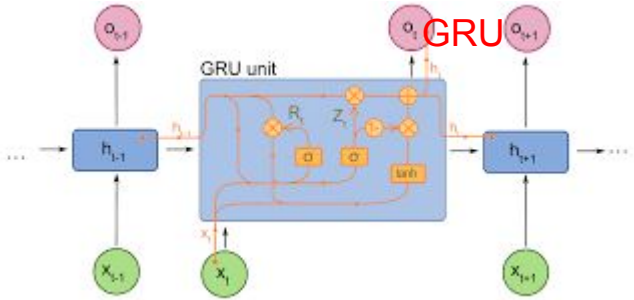
- LSTM and GRU are RNN units with additional features to *control* their “memory”
- “Gates” allow to control (keep or drop) *input*, *output* and *internal state*
- The advantage of gated units is that they *can forget* so that when processing a sequence they focus on the relevant part (e.g. when processing a text we may know that each time we encounter a space the word is over)

LSTM



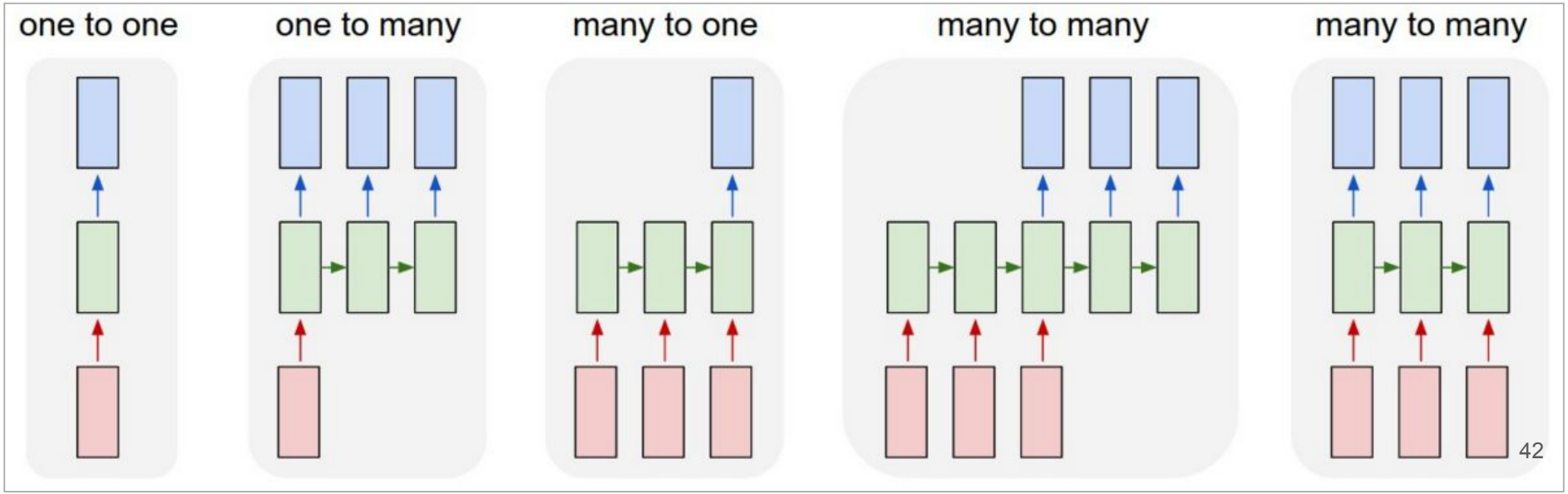


GRU



Different ways of processing time series

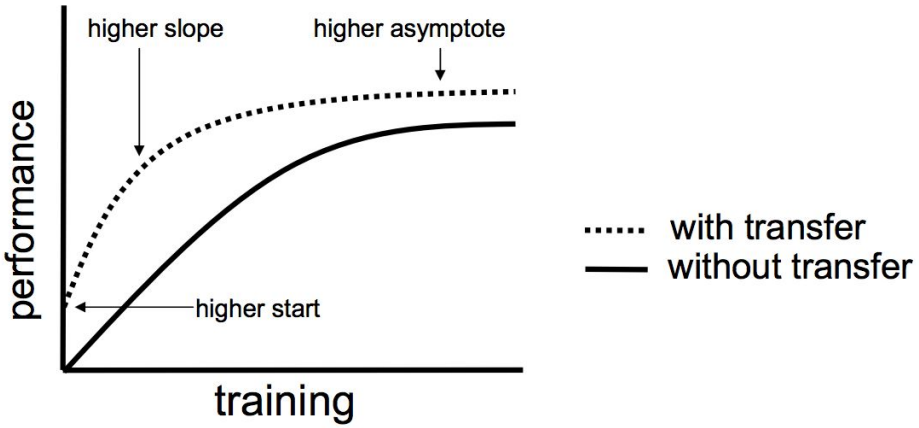
- Recurrent Networks can be used to implement networks with variable number of inputs and outputs
  - Encoding, Decoding, Sequence2Sequence



Transfer learning

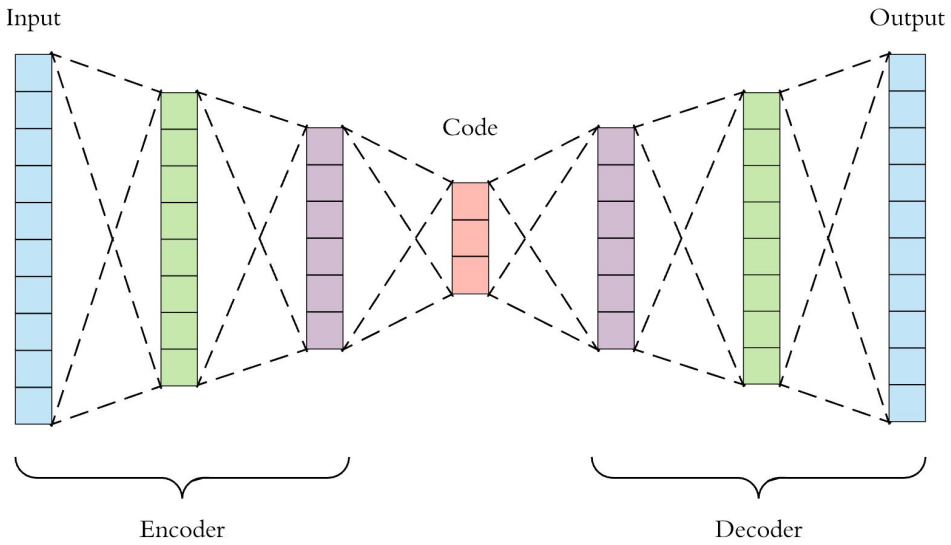
Transfer learning is a technique to reuse a network training for a task to perform **another task** with reduced retraining

- E.g. a Conv2D network meant for image processing have initial layers processing “local features”... that is not very domain specific (if you trained on flowers images it may work on animals too)
- Very useful when the available sample of the proper domain is small
    - E.g. annotated medical images are harder to get than labelled real world pictures



Example of unsupervised architecture: autoencoders

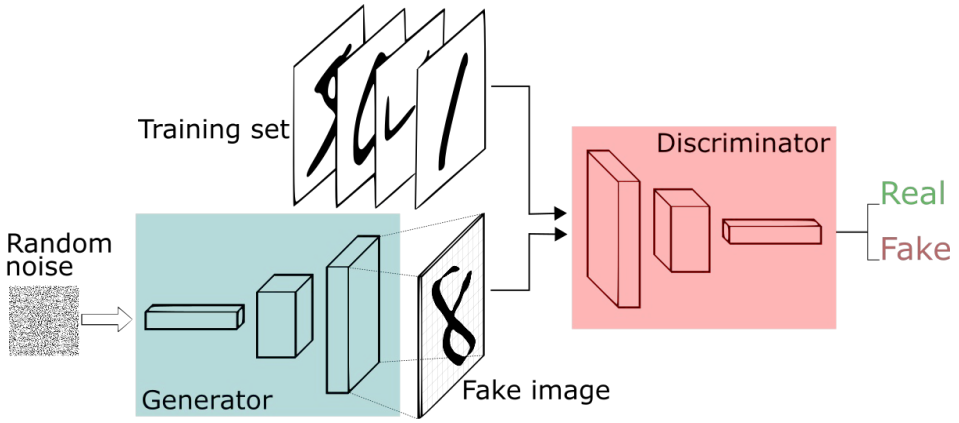
- An example architecture that doesn't need labelled data is the "autoencoder"
- The network is shaped so that one of the inner layer has a much smaller dimension than the input
- The target is an **output matching the input**
- Being able to reproduce the input from the "coded" information in the inner layer means that the network was able to extract the important features of the input



Generative Adversarial Netwokrs

How about generating realistic samples ?

- E.g. generate pictures of animals or showbiz like faces
- Or something more useful in (HE)physics such as generating hadronic shower of a quark/gluon



GAN works with two independent networks:

- A generator
- A discriminator

- The **discriminator** separates samples of the training dataset from samples generated by the **generator**
- No label is needed in the training set as we know where each sample comes from
- **Generator** loss is controlled by the **discriminator** being able to recognize the fake

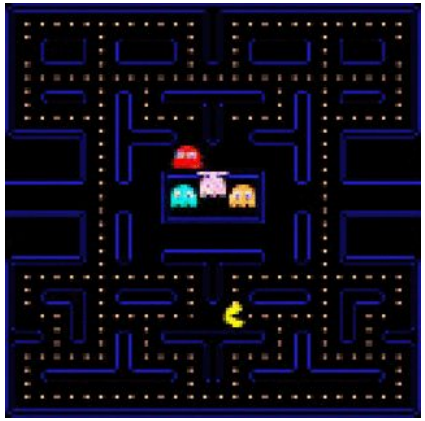
GAN progress

2014: “dogs with three heads”

2018: coherent generation of faces

See also https://thispersondoesnotexist.com/

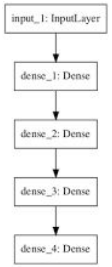
2019: re-create a **playable** video game just by looking at videos of an existing one (so far PacMan)

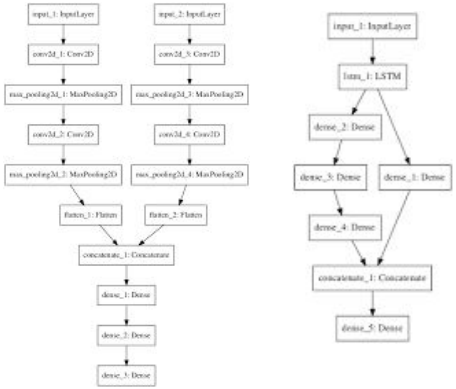


DNN Tools

Keras

- Keras is a python library that allow to build, train and evaluate NN with many modern technologies
- Keras supports multiple backends for actual calculations
- Two different syntax are usable to build the network architecture
  - Sequential: simple linear “stack” of layers
  - Model (functional API): create more complex topologies
- Multiple type of “Layers” are supported
  - Dense: the classic fully connected layer of a FF network
  - Convolutional layers
  - Recurrent layers
- Multiple type of activation functions
- Various optimizers and gradient descent techniques





Other common tools

Common alternative to keras

- Pytorch
- Sonnet
- Direct usage of TensorFlow (or other backends such as Theano, Torch, …)
  - Need to write yourself some of the basics of NN training
  - Especially useful to develop new ideas (e.g. a new descent technique, a new type of basic unit/layer)

Keras Sequential example

```
# first neural network with keras tutorial
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

Keras “Model” Functional API

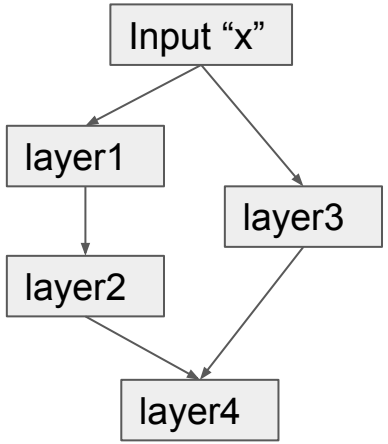
A NN can be seen as the composition of multiple functions (one per layer), e.g.

- A simple stack of layers is: $y=f_5(f_4(f_3(f_2(f_1(\boldsymbol{x})))))$
- A more complex structure could be something like

$$y=f_4(\, f_2(f_1(\boldsymbol{x})),f_3\,(\boldsymbol{x})\,)$$

- The functional API allow to express the idea that each layer is evaluate on the output of a previous layer, i.e.

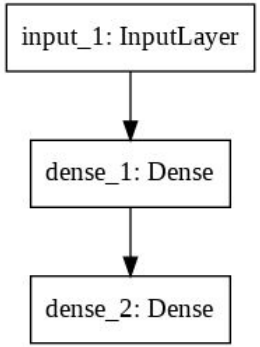
```
x = Input()
layer1=FirstLayerType(parameters) (x)
layer2=SecondLayerType(parameters) (layer1)
layer3=ThirdLayerType(parameters) (x)
layer4=FourthLayerType(parameters)([layer2,layer3])
```

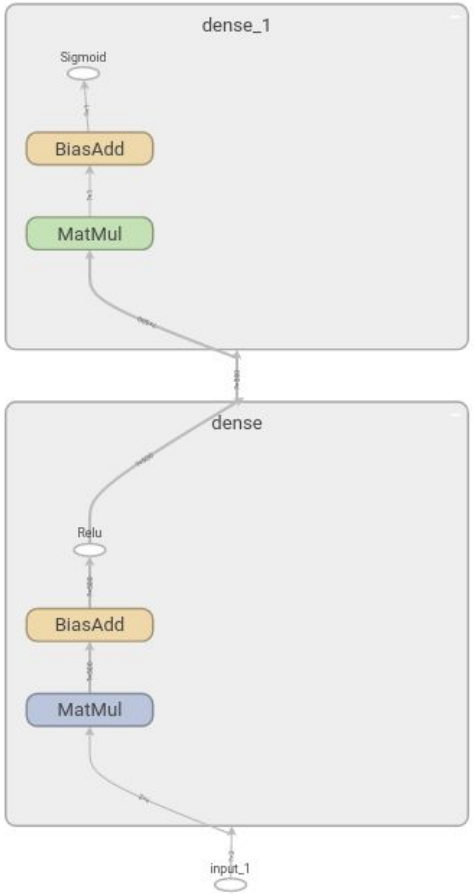


An MLP in keras

```
from keras.models import Model
from keras.layers import Input, Dense
x = Input(shape=(32,))
hid = Dense(32, activation="relu")(x)
out = Dense(1, activation="sigmoid")(hid)
model = Model(inputs=x, outputs=out)


model.summary()
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```



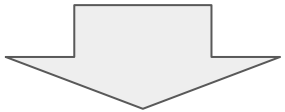



```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,089
Trainable params: 1,089
Non-trainable params: 0
_________________________________________________________________
```

From the ~1995 to ~2010

```
from keras.models import Model
from keras.layers import Input, Dense

x = Input(shape=(32,))
hid = Dense(32, activation="sigmoid")(x)
out = Dense(1, activation="sigmoid")(hid)
model = Model(inputs=x, outputs=out)
```



```
from keras.models import Model
from keras.layers import Input, Dense

x = Input(shape=(32,))
b = Dense(32,activation="relu")(a)
c = Dense(32,activation="relu")(b)
d = Dense(32,activation="relu")(c)
e = Dense(32,,activation="sigmoid")(d)
model = Model(inputs=x, outputs=e)
```

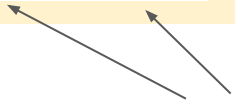
Training a model with Keras

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```



Those are numpy arrays with your data

Keras layers

Keras basic layers

- Basic layers
  - Inputs
  - Dense
  - Activation
  - Dropout
- Convolutional layers
  - Conv1D/2D/3D
  - ConvTranspose or “Deconvolution”
  - UpSampling and ZeroPadding
  - MaxPooling, AveragePooling
  - Flatten
- More stuff
  - Recursive layers
  - ...check the keras docs...

Callbacks

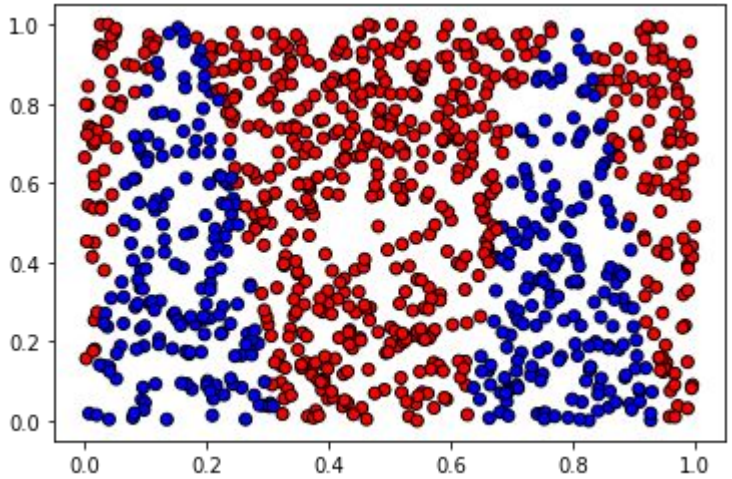
- During training some “callbacks” can be passed to the fit function
  - E.g. to monitor the progress of the training
  - To adapt the training
    - Stop if no improvements in the last N epochs
    - Change learning rate (reduce) if no improvements in the last M epochs
  - Some callbacks are predefined in keras, other can be user implemented

```
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
# train
history = model.fit(X_train, y_train, epochs=n_epochs, batch_size=batch_size, verbose = 2,
                validation_data=(X_test, y_test),
                callbacks = [
                EarlyStopping(monitor='val_loss', patience=10, verbose=1),
                ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=2, verbose=1)
                ])
```

Assignment 1

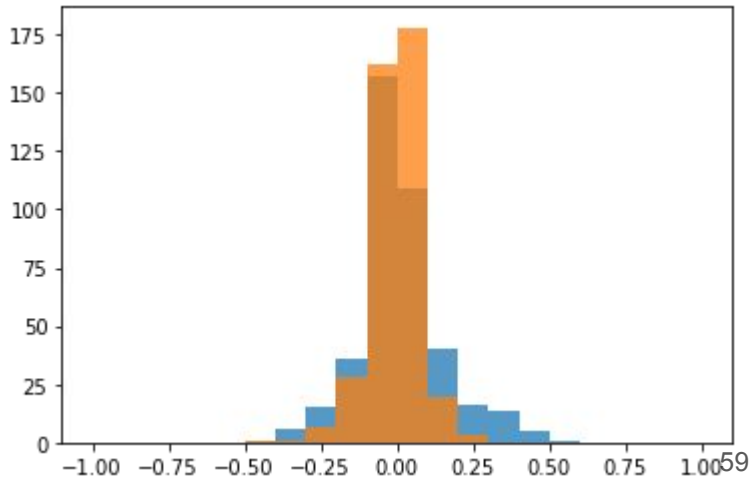
- Partition a 2D region with a simple function that returns true vs false having x1,x2 as arguments
  - E.g. x1>x2 or x1*2>x2 or x1 > 1/x2 or … whatever...
- Generate 1000 samples
- Create a classifier with a MLP or a DNN with similar number of parameters that approximate the function above

Start from this notebook:
Exercise 1



Assignment 2

- Let's try to implement a **regression** with DNN in keras
- Invent a function of x1,x2,x3,x4,x5
- Generate some data
- Create a Feed Forward model (with 1 or more hidden layers)
- What should we change compared to the classification problem?
  - What is the loss function to use?
  - Which activation function in last layer?
- Try to make a histogram of the residuals on the validation sample
  - residuals=(y_predicted - y_truth)

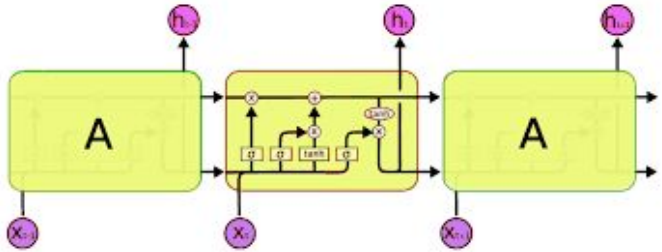


Start from this notebook: Exercise 2

Solution

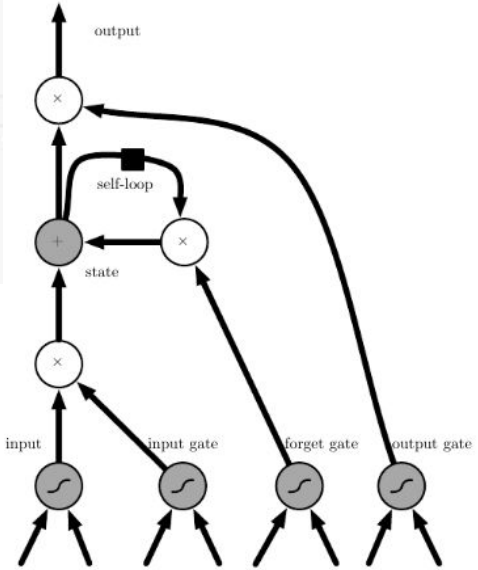
More on LSTM

- LSTM layers in keras can return
  - Just the output of the last iteration
  - The whole sequence of output
  - The gated output of the memory
  - The cell state

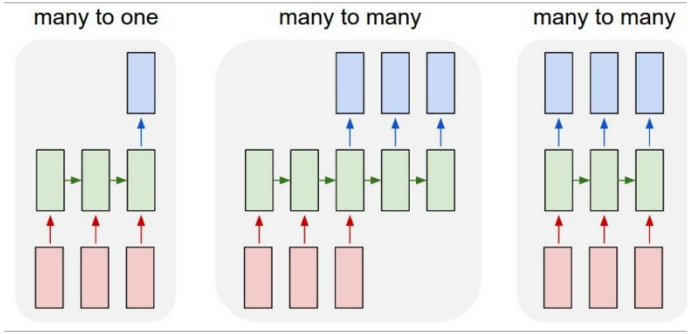



```
from keras.models import Model
from keras.layers import LSTM
from numpy import array
inputs1 = Input(shape=(5, 1))
lstm1, state_h, state_c = LSTM(1, return_state=True, return_sequences=True)(inputs1)
model = Model(inputs=inputs1, outputs=[lstm1, state_h, state_c])
data = array([0.1, 0.2, 0.3, 0.4, 0.5]).reshape((1,5,1))
print(model.predict(data))
```

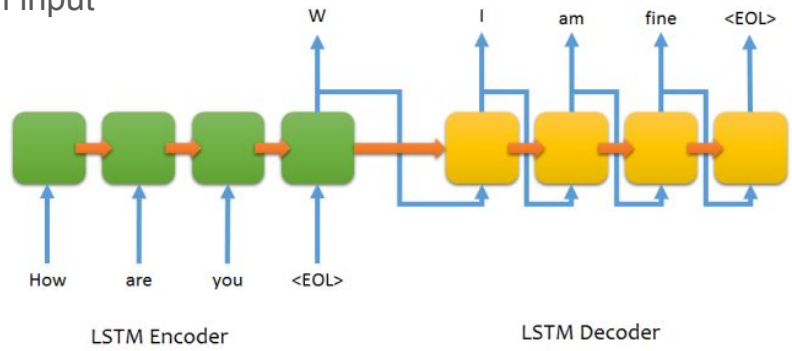
```
[array([[[0.02106816],
         [0.05576485],
         [0.09626514],
         [0.13520567],
         [0.16713278]]], dtype=float32),
 array([[0.16713278]], dtype=float32),
 array([[0.41894686]], dtype=float32)]
```



Using LSTM



- Many to one configuration:
  - Just use a LSTM layer with default config
    - No need to know the full sequence
    - Optionally request also the cell state
- Many to Many (syncronous)
  - Set return_sequence=True to get exactly one output for each input
- Many to many (async, different length)
  - Need two LSTM: A encoder + a decoder
  - *Sequence2Sequence* or *Encode-Decode* architecture
  - The cell state of the encoder can be used as initial state for the decoder
  - Need to define a STOP character to receive when the decoding sequence is over



- Inputs with variable length should be “padded”
  - Masking layers exist in keras to avoid “learning from padding”
  - Reversing the sentence order (so that padding is at the beginning also helps)
  - Often with LSTM useful to provide most important information at the end