Intro to Artificial Neural Networks

Lecture 3

Course of:

Signal and imaging acquisition and modelling in environment

13/03/2024

Federico De Guio - Matteo Fossati

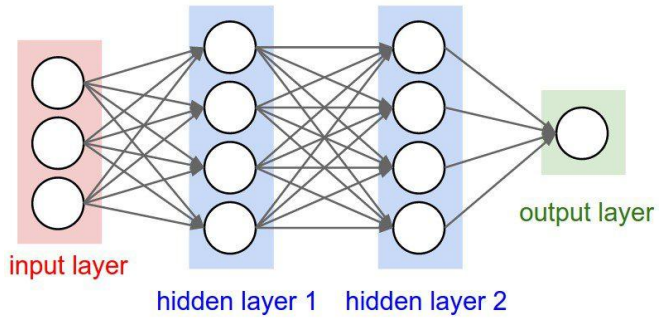
What is a ANN?

- A **mathematical model** able to approximate with high precision a **generic multi-dimensional function**:

$$f: R^n \rightarrow R^m: y = f(x) \longrightarrow \text{ANN}(x) = \hat{y}$$

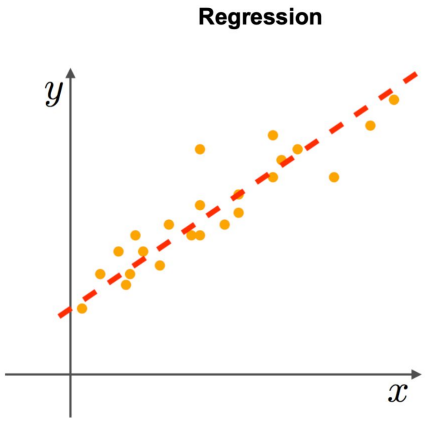
- Composition of functions (**layers**) connected in chains and described by **graphs**

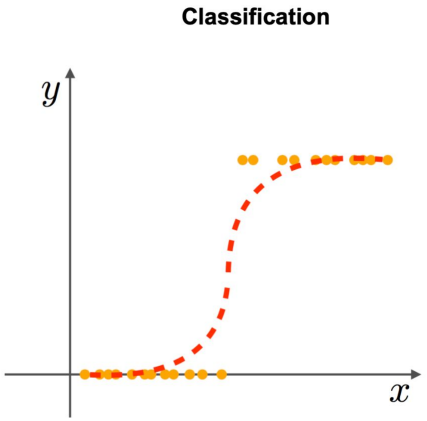
- **Characteristics**:
  - Interconnected group of **identical computation units**
  - Collective **actions performed in parallel** by the neurons
  - The net structure dynamically evolves during the training
  - **Non-linear response**
  - Potentially complex multi-layer topologies



What problems can an ANN solve?

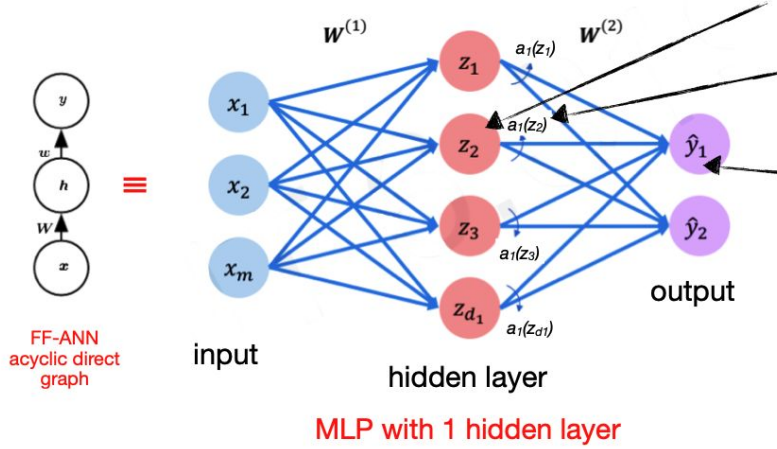
- **Regression:** model the relationship between variables
  - Linear vs non-linear regression, one vs multi-dimensional
- **Classification:** recognize and categorize the inputs
  - Two vs multi-category problems





The feed forward NN

- Have **hierarchical structures**: inputs enter from the left and flow to the right + no closed loops
- How they work:
  - Each **input is multiplied by a weight**
  - The **weighted values are summed**
  - A **bias is added**
  - The result is **passed to an activation function**



$$z_i = w_{0i}^{(1)} + \sum_{j=1}^{m} x_j w_{ji}^{(1)}$$ synaptic sum

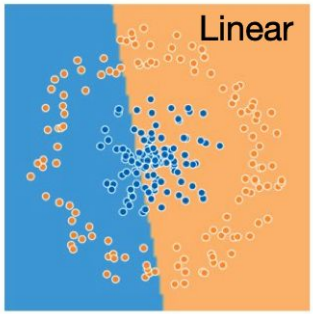
$$a_1(z_i)$$ activation

$$\hat{y}_i = a_2\left(w_{0i}^{(2)} + \sum_{j=1}^{d_1} a_1(z_j) w_{ji}^{(2)}\right)$$

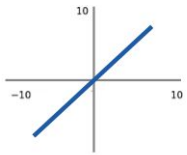
- NN behaviour determined by:
  - network topology (#layers, size of each layer, …)
  - weights w_{ij}
  - activation function of each layer

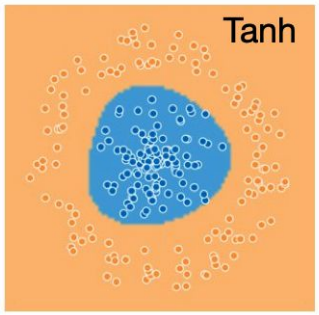
Linear vs non-linear activation functions

- **Non-linear activations** allow to learn complex and non-linear patterns

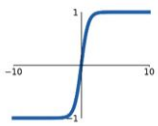


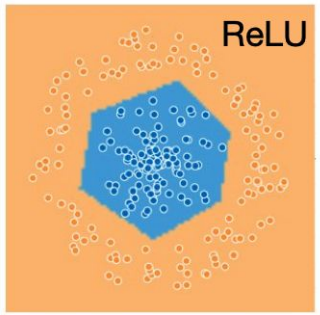
$a(z) = z$



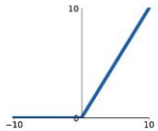


$a(z) = \tanh[z]$





$a(z) = \max[0,z]$

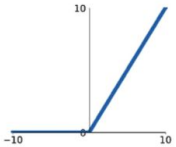


Choice of the activation function for the **hidden layers**

- In general, any **continuous and differentiable function** would be fine. In practice some functions work better than other for specific ANN architectures...

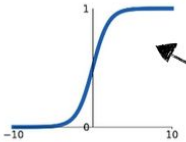
ReLU

$\max(0, x)$



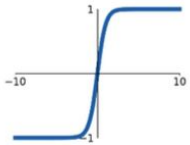
Sigmoid

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



tanh

$\tanh(x)$



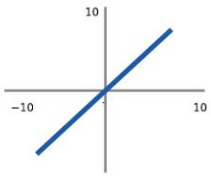
ReLu is the most popular:

- allows non linear dynamics
- faster convergence of the NN because doesn't saturate
- no vanishing gradient problem

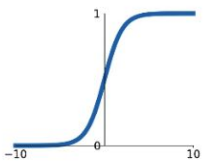
Should not be used in general for dense and convolutional layers:

- gradient vanishes away from x=0 → vanishing gradient problem
- sigmoid has output not centered in zero → affects SGD dynamic (zig-zag instabilities)

Choice of the activation function for the **output layer**



- **Identity (linear):** standard choice for regression tasks

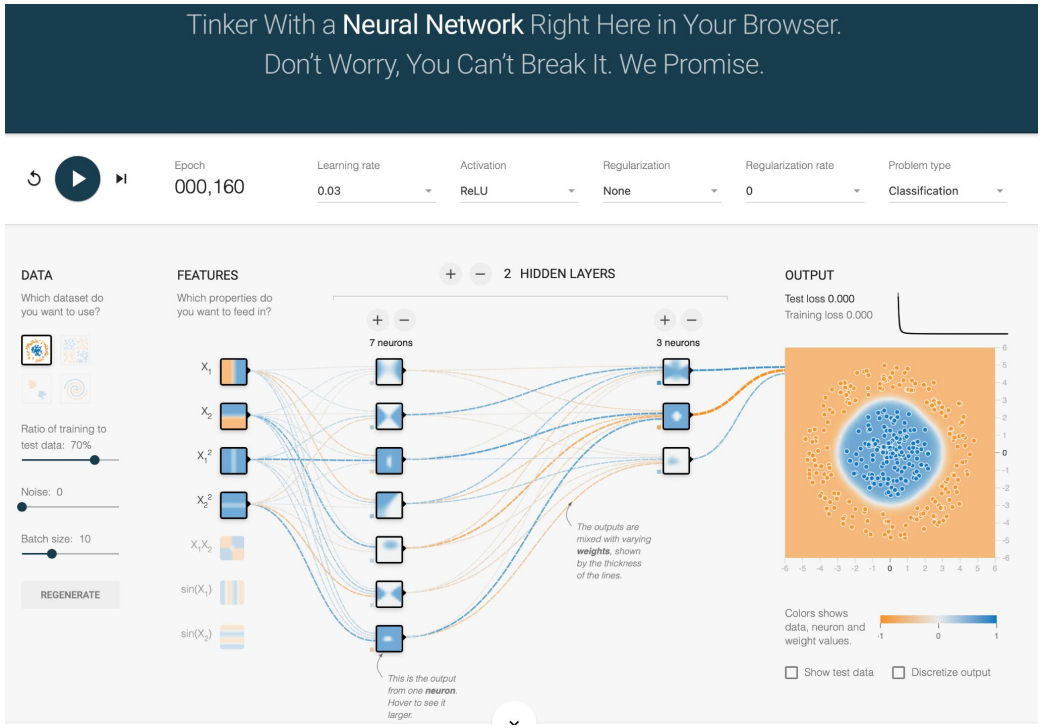


- **Sigmoid:** typically used in binary classification problems (2 classes) with a single output neuron or multilabel (multiple mutually inclusive classes) or sometime when the output features are numbers in (0,1)

$$y_i = \frac{e^{z_j}}{\sum_{j=1}^{n} e^{z_j}}$$

- **Softmax: $R^n \rightarrow [0,1]^n$**
  - soft version of the argmax output
  - often used in multi-class classification tasks (with mutually exclusive classes) - output of each neuron $\in$ (0,1) and interpretable as a probability ($\sum y^i=1$)

Try yourself on https://playground.tensorflow.org

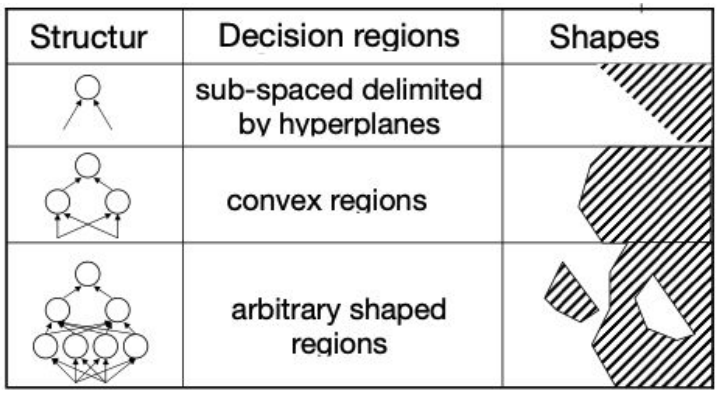


ANN as universal approximators

- It can be demonstrated that a feed-forward network with a **single hidden layer** containing a **finite number of neurons** with **non-linear activations** can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function

$$F(x) = \sum c_i a(w_{0i} + \mathbf{w}^t \mathbf{x})$$

$$\int_{R^n} ||f(x) - F(x)||_p dx < \epsilon$$



IMPORTANT: the theorem doesn't say anything about the effective possibility to learn **in a simple way** the parameters of the model, all the DNN practice boils down to finding optimal and efficient techniques to solve this problem ...

The training process: learning the parameters

- Training consists in **adjusting the parameters according to a given cost function** (loss)
- The loss is a differentiable function that measures the **distance** wrt the prediction
  - **Weights and biases:** randomly initialized and "adjusted" using gradient descent with back-propagation
  - **Hyperparameters**: "adjusted" using heuristic approaches (manual trial&error, grid or random search, autoML, ...)

- For each event the output of the model $\hat{y}(x^{(i)})$ is calculated and compared with the expected target $y^{(i)}$ by means of an appropriate loss function:

$$L(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L_i\left(y^{(i)}, \hat{y}^{(i)}(x^{(i)}\,|\,\mathbf{w})\right)$$

example: MSE

$$L_i\left(y^{(i)}, \hat{y}^{(i)}(x^{(i)}\,|\,\mathbf{w})\right) = \frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}(x^{(i)}\,|\,\mathbf{w})\right)^2$$

Target

Model output given ***w***

Loss function can be chosen.
- MSE is just one choice

- Select the weights that minimize L: $\mathbf{w}^* = \underset{\mathbf{w}}{argmin}[L(\mathbf{w})]$
- → Find the minimum with GD techniques: $\mathbf{w}_{(t+1)} = \mathbf{w}_{(t)} - \eta\,\nabla_w L(T\,|\,\mathbf{w})$

Backpropagation

- **The backpropagation consists of:**
  - Updating weights using the **gradient of the loss**
  - **Propagating the gradient backwards** through the network
  - Applying recursively the rule of derivation (chain rule)

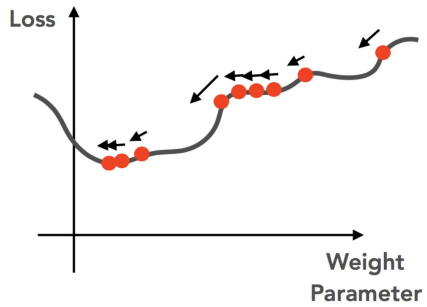
- At each iteration:
  - **Forward phase:** the weights are fixed and the input vector is propagated to the output neurons **(function signal)**

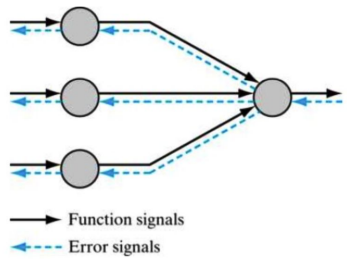
  - **Backward phase:** the error is calculated by comparing the output with the target and the result is propagated back **(error signal)**

$$\frac{\partial L(\mathbf{w})}{\partial w_1} = \frac{\partial L(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial L(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}$$

available at the output

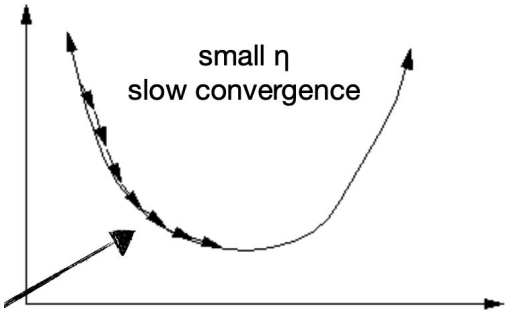
analytically calculable

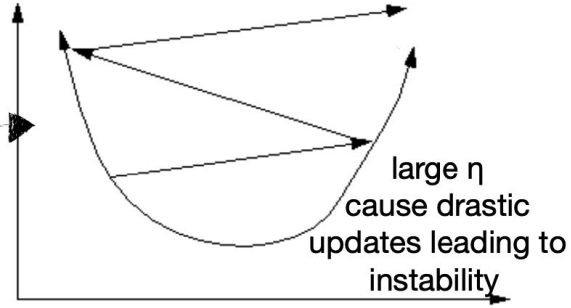




Loss function and learning rate

- The gradient descent method is an **iterative procedure**
- At each iteration weights are updated according to: **$w(t+1) = w(t) - \eta \cdot \nabla L(w(t))$**
- **η is called learning rate** and defines the magnitude of the vector modification
- η affects the **speed and quality of convergence** toward a minimum

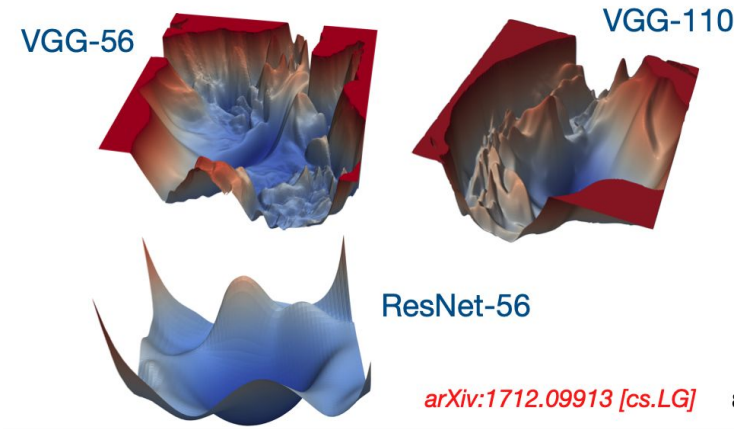




Why going deep?

- One layer is in principle sufficient to approximate any function with arbitrary precision
- **Deep architectures are much more efficient** at representing a larger class of mapping functions

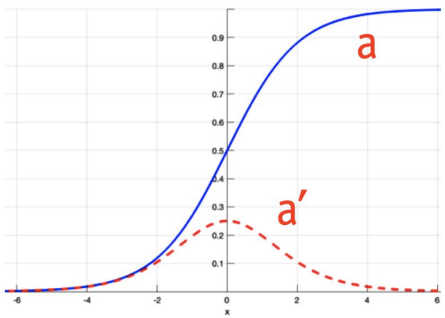
  - DNN require a **smaller number of parameters**
  - **Sub-features can be used in parallel** for multiple tasks within the same model
  - Overparameterization seems to have beneficial effects in smoothing the loss function → **easier convergence**



Vanishing gradient

- Going deep can be difficult: **the first layers of a deep NN fail to learn efficiently**

- **Reason:** during backprop, n derivatives of the activation functions will be multiplied together. If they are small the gradient will decrease as we propagate through the model until it eventually vanishes
- **Solutions:**
  - use activation functions which do not produce small derivatives: i.e. ReLU, LeakyReLU, Selu, …
  - use batch normalisation layers in which the input is normalised before to be processed by the layer in order not to reach regions of the activation function where derivatives are small



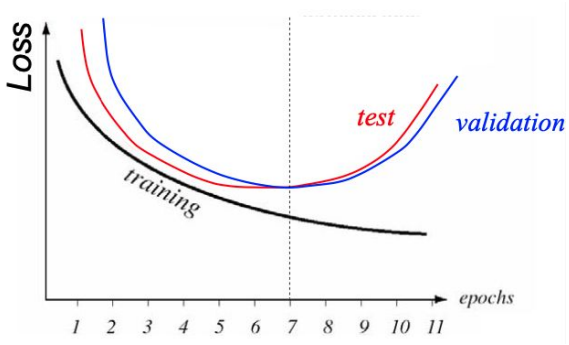
Learning curves

- At the **beginning** of the training phase the weights are randomly initialized → **the error on the training set is typically large**

- During the learning, the error decreases and **reaches a plateau** that depends on:
  - The size of the training set
  - The NN architecture
  - The initial value of the weights
  - …

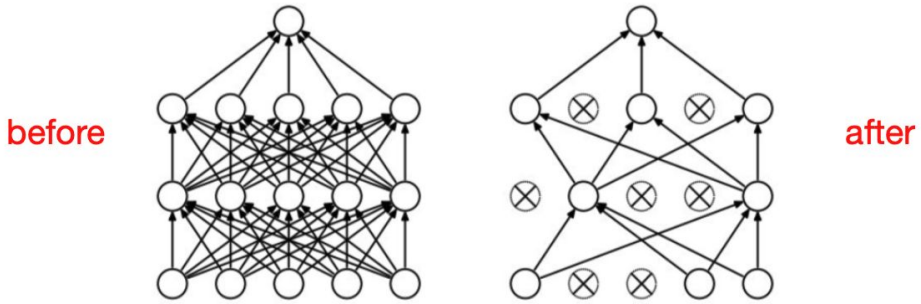
- The effectiveness of the learning process is visualized with the **learning and accuracy curves**

- **Overfitting** can be an issue: network too specialized on a given problem



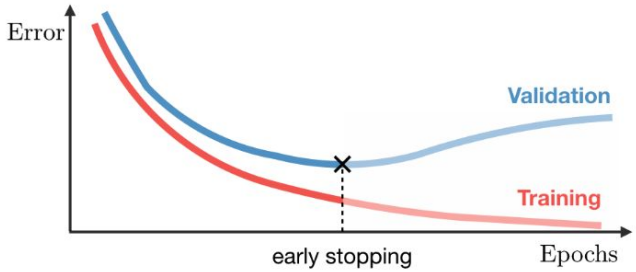
Dropout

- Powerful technique to **prevent overfitting in DNNs**
- It shuts down randomly a fraction of the connections
- **Forces the model not to rely excessively on particular sets of features**

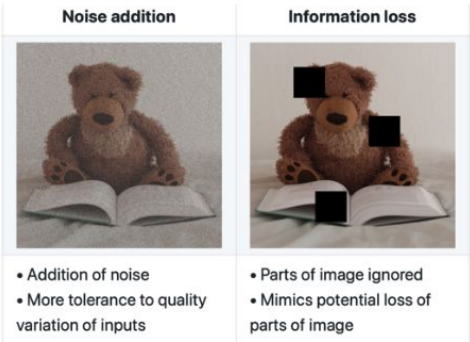


Early stopping and noise injection

- **Early stopping:** imposes constraints on the error reduction on the training set
  - The training process is stopped as soon as the loss on the validation sample reaches a plateau or start to increase



- **Noise injection/information loss:** makes it more difficult for the network to learn specific characteristics of the input features
  - random flip of labels
  - random occlusion of pixels or feature bits
  - adding withe/colored/gaussian noise to the features ...

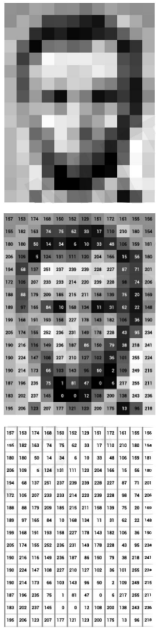


Data augmentation

- One of the best ways to make an ML algorithm to generalize better is to **train it on larger and more expressive data**
- If the dataset size is limited → **artificially increase** the dimension of the training set by applying transformations that preserve the "physics" of the data/problem

Original	Flip	Rotation	Random crop	Color shift	Contrast change
					
• Image without any modification	• Flipped with respect to an axis for which the meaning of the image is preserved	• Rotation with a slight angle • Simulates incorrect horizon calibration	• Random focus on one part of the image • Several random crops can be done in a row	• Nuances of RGB is slightly changed • Captures noise that can occur with light exposure	• Luminosity changes • Controls difference in exposition due to time of day

Many different ANN architectures



- FFNN are **universal models**
  - However too much flexibility can results in arbitrarily complex models
  - Hard to optimize a huge number of parameters
  - hard to achieve a good level of generalisation

- → **Task independent priors** are introduced in modern DNN architectures
  - Priors are inferred from general structures observed in data
  - One example are **Convolutional Neural Networks which operate directly on images**
  - **More about CNNs next week!**

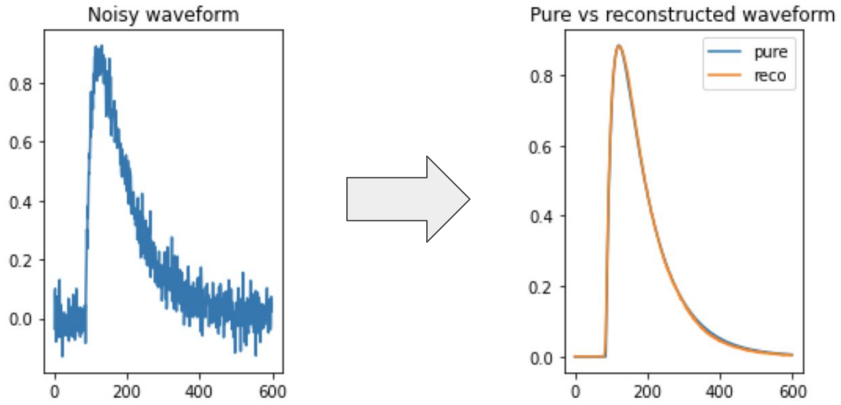
Convolutional NN is one of these specific DNN architecture designed to excel in image recognition tasks

Today’s exercise:

Build a denoising neural network

Definition of the problem

- Use the pulse library generated last time and train a DNN to remove the noise from the pulses
- What are the main parameters of a SiPM pulse like this?



- Pulse library available here

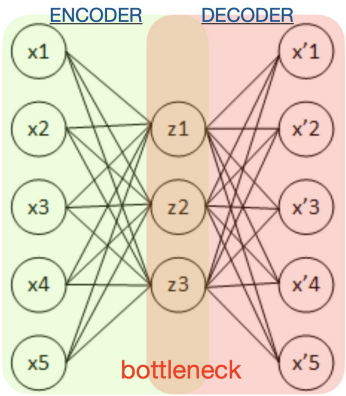
What ANN architecture could be used and why?

The Auto Encoder (AE)

- Goal: build a model that identifies **fundamental characteristics in the input data**
  - In our case we aim to extract from the noisy pulse: raise time, decay time, time offset, etc

- Combine an **encoder** that converts input data in a different representation, with a **decoder** that converts the new representation back to the original input

- One option is:
  - Noisy pulse as the input
  - Pure pulse as the target
  - The DNN learns the relevant features of the pulse



Before we start:
quick introduction to Keras
with TensorFlow as backend

Live example

The simplest architecture: sequential model (FFNN)

Keras

- **Define a sequential model**

```
model = Sequential()
mode.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

- **Compilation**

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuray'])
```

- **Training**

```
model = model.fit(data, one_hot_labels,
epoch=10, batch_size=32)
```

- **Predition**

```
Y = model.predict(X)
```

Why Keras:
- User friendly with high-level APIs
- Quick to get started
- Coding less lines for machine learning model construction/training/testing

Case study

- Diagnose the diabetes disease based on generic information on the patient
- The available info is a table such as:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
545	8	186	90	35	225	34.5	0.423	37	1
471	0	137	70	38	0	33.2	0.170	22	0
21	8	99	84	0	0	35.4	0.388	50	0
46	1	146	56	0	0	29.7	0.564	29	0
755	1	128	88	39	110	36.5	1.057	37	1

- Here is the full dataset
- Let's give it a try!