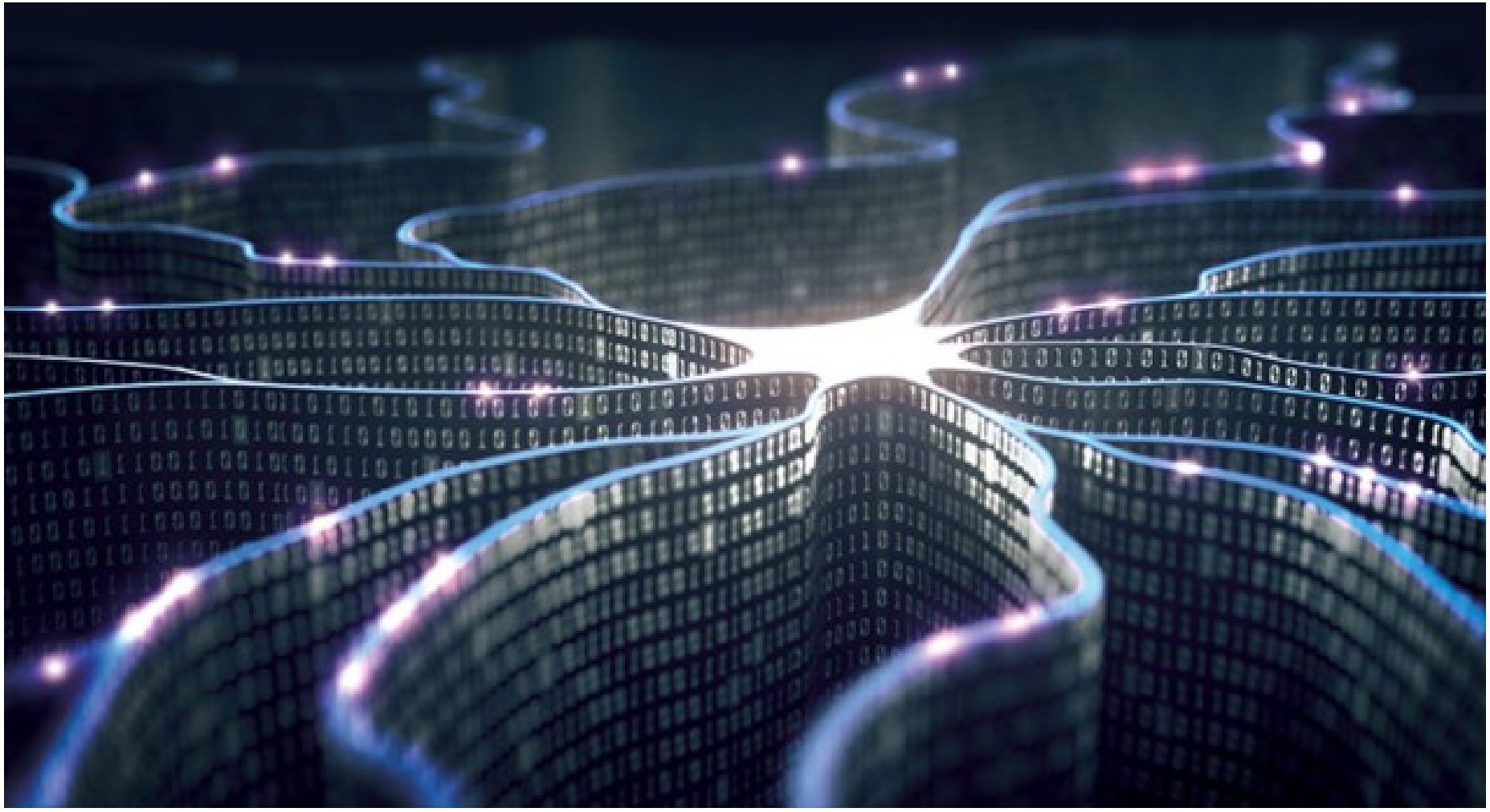
Artificial Neurons for HEP

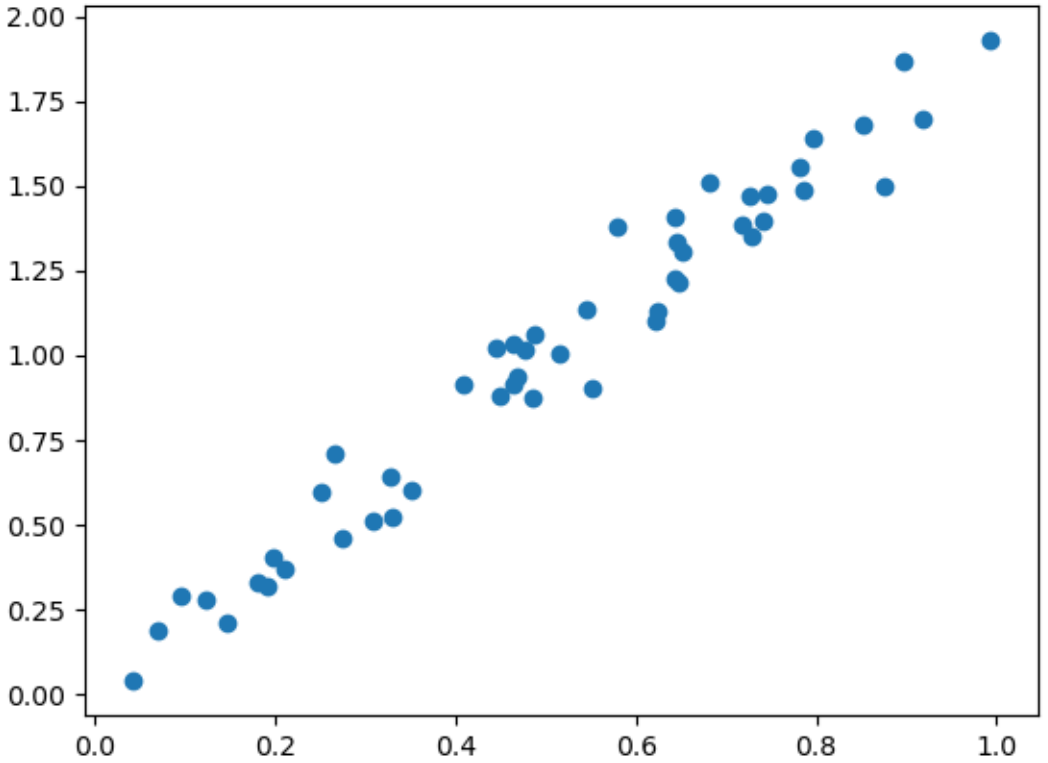


Frédéric Magniette - LLR

- **Problematics**
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

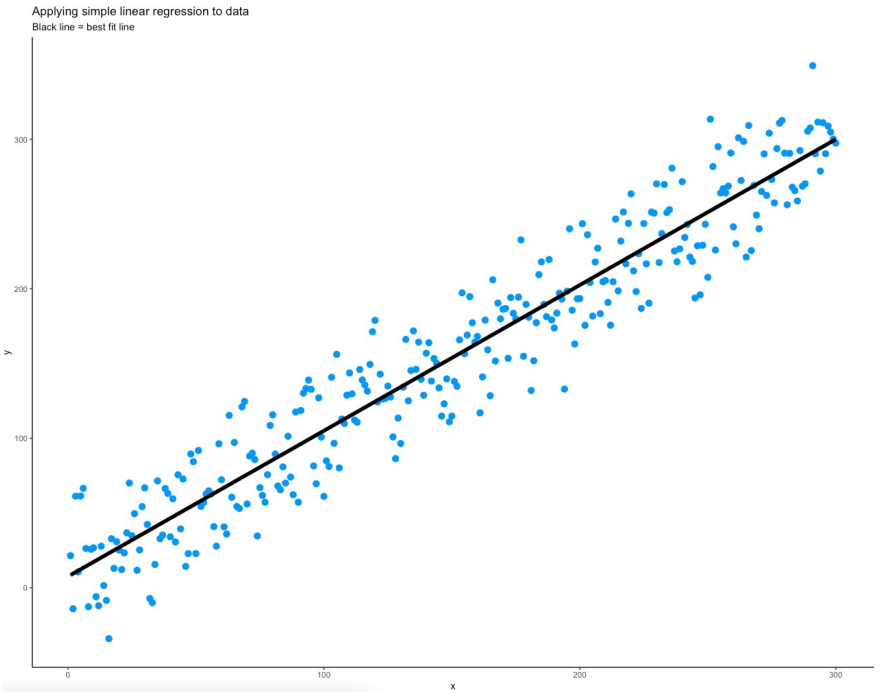
Problematics

x[0]=0.326785 y[0]=0.640017
x[1]=0.484787 y[1]=0.877112
x[2]=0.728836 y[2]=1.350927
x[3]=0.190499 y[3]=0.321072
x[4]=0.717005 y[4]=1.384066
x[5]=0.648116 y[5]=1.216906
x[6]=0.488057 y[6]=1.062203
x[7]=0.917032 y[7]=1.697487
x[8]=0.274938 y[8]=0.460703
x[9]=0.197535 y[9]=0.404545
x[10]=0.122173 y[10]=0.277121
x[11]=0.852632 y[11]=1.682158
x[12]=0.991762 y[12]=1.930109
...



- Supervised learning : n pairs of (x,y) samples are given
- What is the most probable y value for non-given x=0.356 ?

Modelization



- simple linear regression
- $\hat{y}=f(a,x)=a.x$
- x is the input
- a is the parameter (to determine)
- f is the model → a priori choice
- $\hat{y}$ is the estimated result
- y is the true value (know as ground truth)
- $L(y,\hat{y})$ is the loss function. For example $| y - \hat{y} |$
- How to evaluate a ?

Estimator

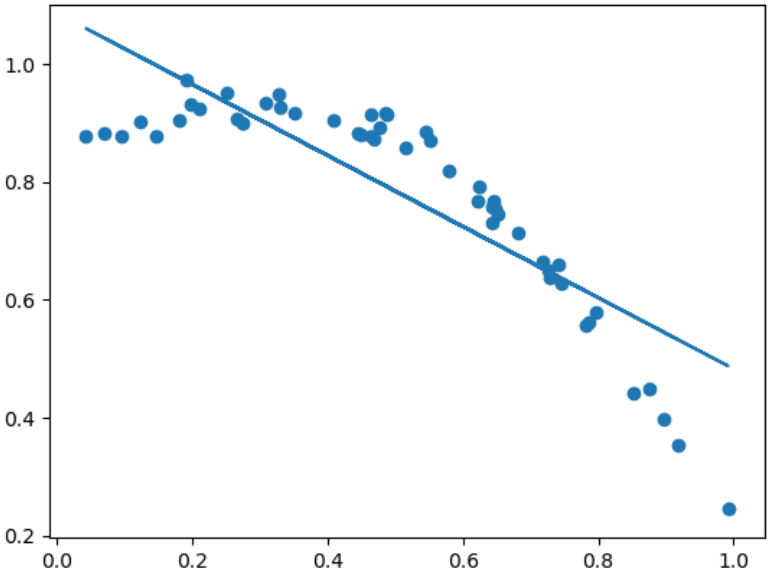
- For simple models, we can compute estimators of the parameters
- Example : estimator for a
- We expect the estimator to be asymptotically unbiased

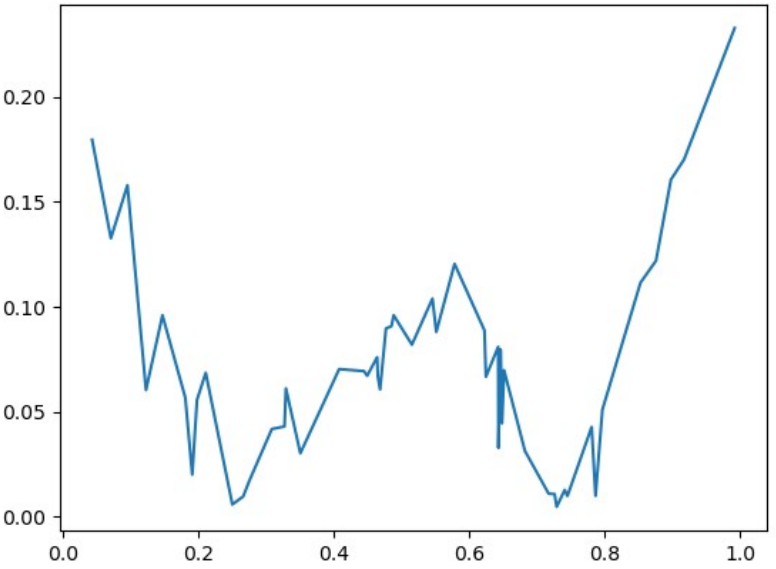
$$\hat{a}(x_i, y_i) = \frac{\sum_{i=0}^{n} y_i / x_i}{n}$$

```
>>>np.mean(y/x)
1.987
>>>0.356*1.987
0.707
```

Bias and model choice

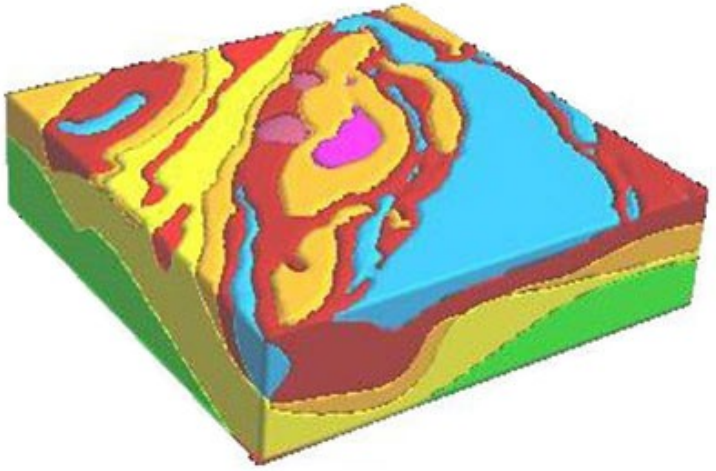
- If the model is not adapted, the error (bias) becomes important
- y=ax+b
- Not expressive enough
- Error=|y-ŷ|





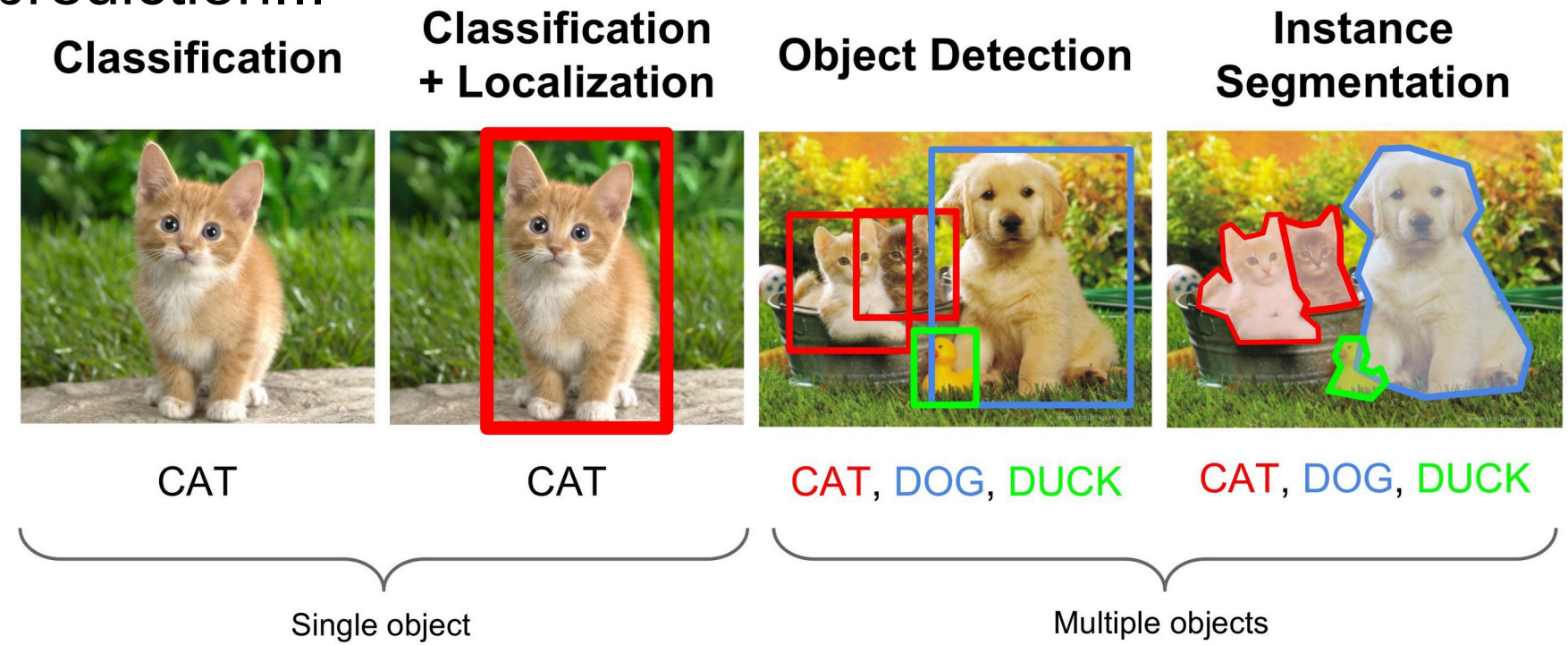
General form of models

- x is extended to vector X
- a is extended to vector Θ
- y is extended to vector Y
- $\hat{Y}=f(X,\Theta)$
- Loss function $L(\hat{Y},Y)$
- No estimators for finding Θ



Diversity of problematics

- Inputs: numerical measures, pixel color, 3D points, variables of interest, encoded or noised data, sequence of signs (text), sequence of continuous values (sound)…
- Outputs: continuous values (regression), probability of categorization (classification), segmentation and disentaglement, outlier detection, denoizing, sequence prediction...

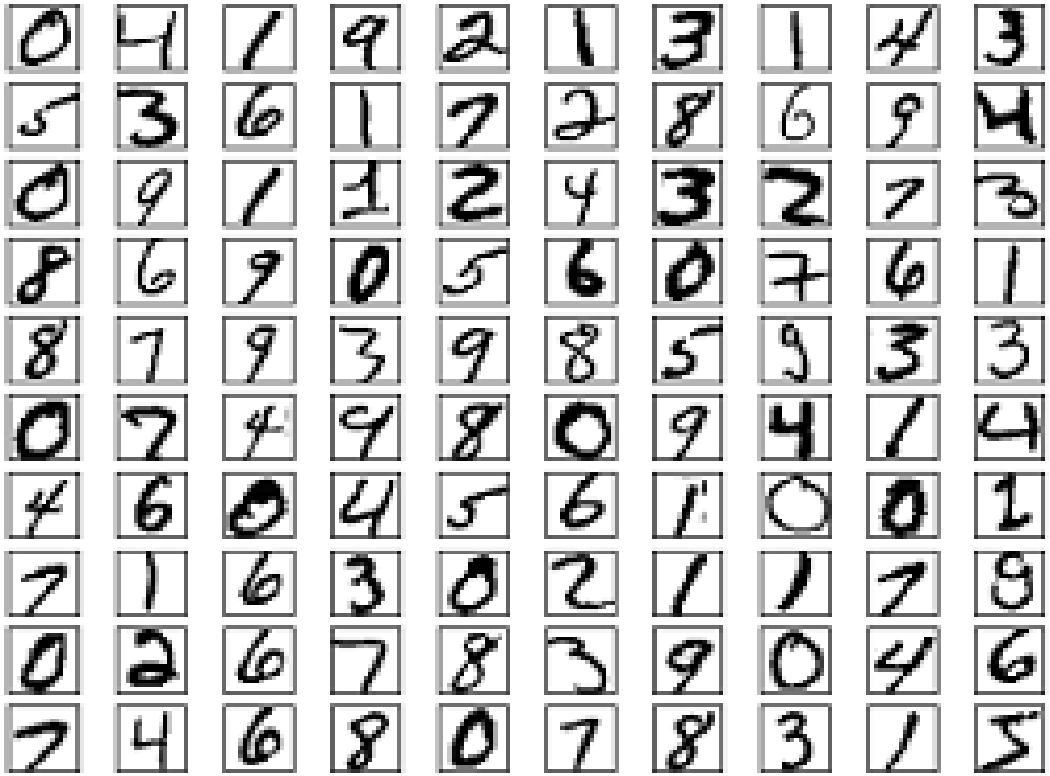


How to choose a good model ?

- What mathematical form ?
- How many parameters ?
- Is there universal models ?
- Is it possible to have good estimators for all my parameters ?

MNIST a machine learning benchmark since 90's

- MNIST : Set of digitized handwritten digits
- Part of the NIST data set (National Institute of Standards and Technology)
- Used as benchmark for machine learning for years



Model comparison on MNSIT

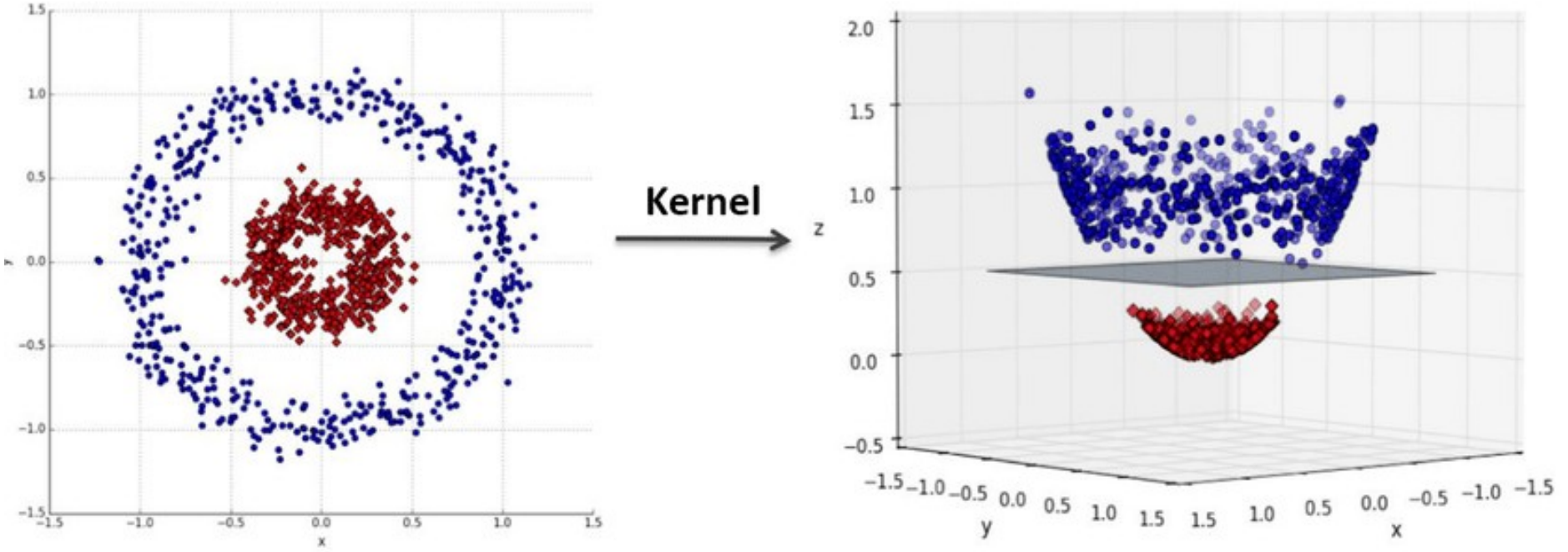
Method	Error Rate	Authors	Year
Linear Classifier	12	Lecun et al.	1998
2-layer NN, 300 hidden units	4.7	LeCun et al.	1998
KNN	0.63	Belongie et al.	2002
Virtual SVM, deg-9 poly, 2-pixel jittered	0.56	DeCoste and Scholkopf	2002
6-layer NN 784-2500-2000-1500-1000-500-10 [elastic distortions]	0.35	Ciresan et al.	2010
Convolutional NN, 1-20-P-40-P-150-10 [elastic distortions]	0.23	Ciresan et al.	2012
Human brain	0.2	Ciresan et al.	
Random multi-model Deep Learning	0.18	Kowsari et al.	2018
Branching/Merging CNN Homogeneous Vector Capsule	0.13	Byerly et al.	2020

From http://yann.lecun.com/exdb/mnist/

Presently, Neural Network is by far the best algorithm !

Kernel Methods

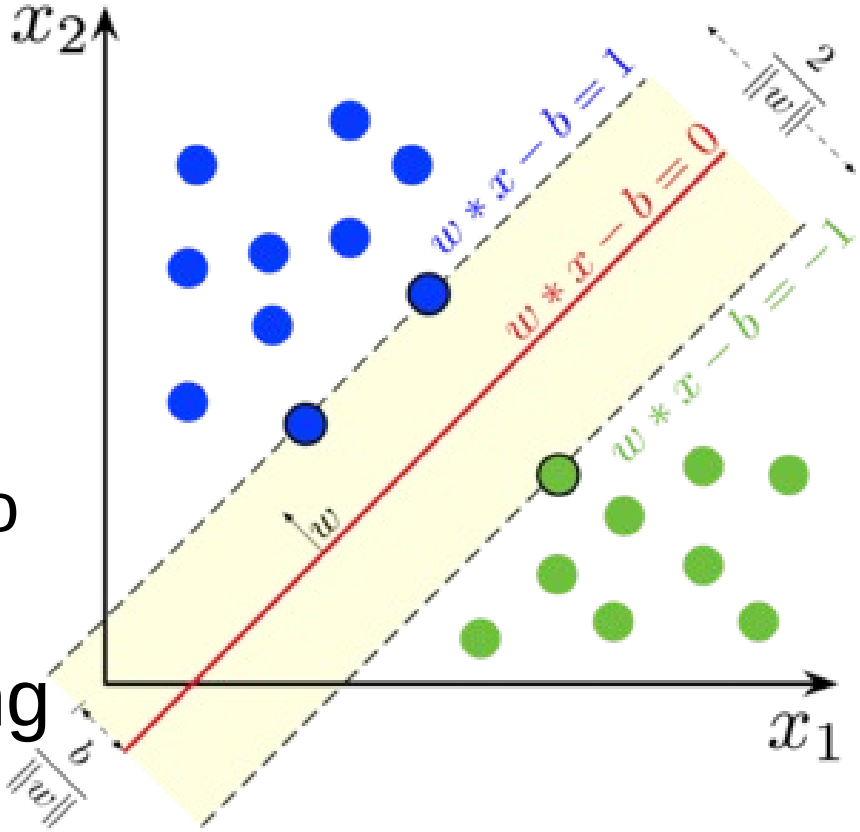
- Adapted to non linearly separable data
- Adding features to the original data, i.e. creating a new representation of the data in a space with more dimensions
- Applying a linear classifier (i.e. separating hyperplane)
- Example $\varphi(a,b)=(a,b,a^2+b^2)$:



Support Vector Machine (SVM)

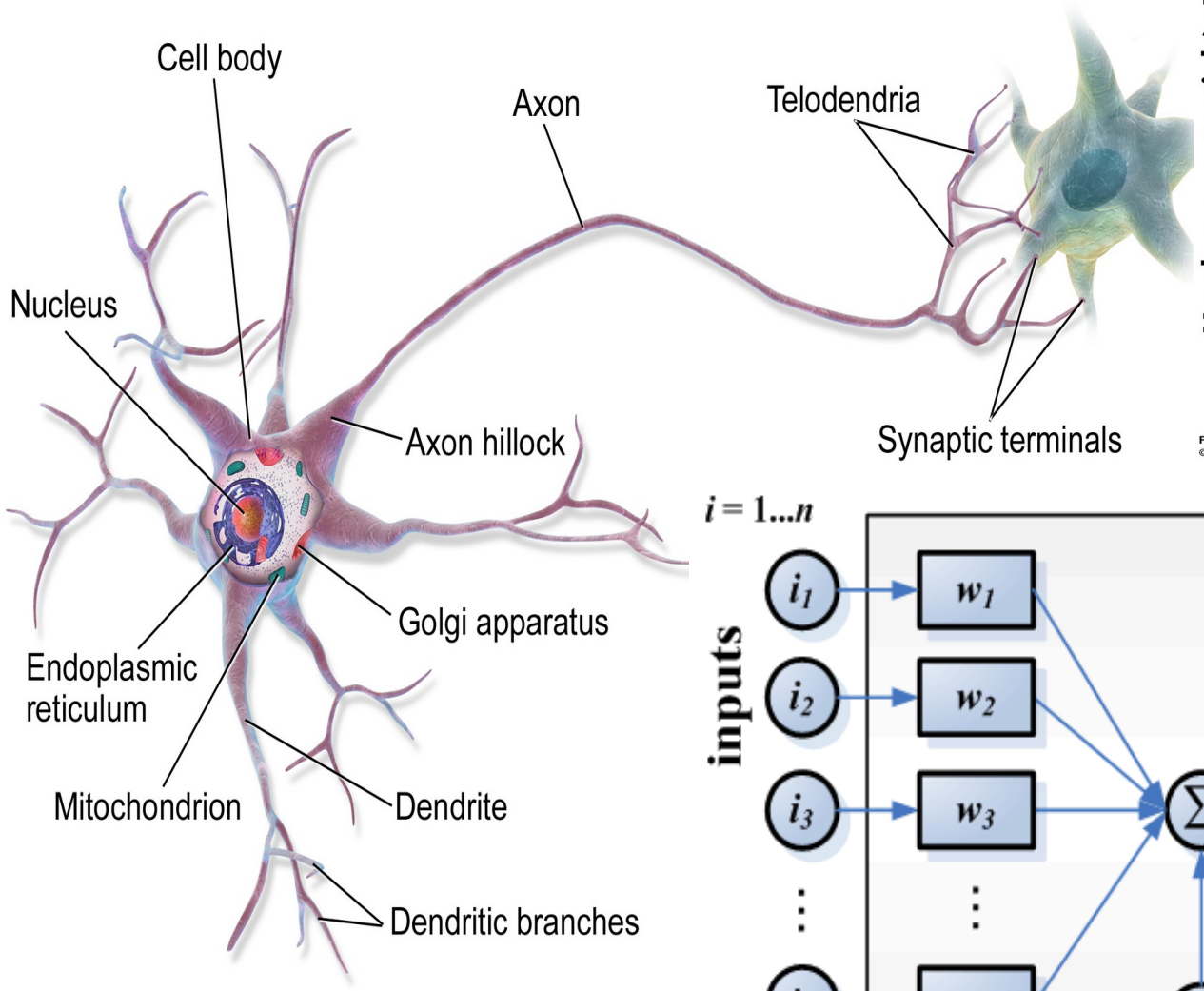
- Enrich data with kernel features
  - Polynomial kernel
    $K(x,y)=(x^T.y+1)^d$
  - Gaussian Kernel

    $K(x,y)=\exp(-\|x-y\|^2/2\sigma^2)$
- Find the support vectors
  - the points of each class closer to the other class
- Find the hyperplane maximizing the distance to the support vectors (hard-margin)



- Problematics
- **Neuron, perceptron and back-propagation**
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

From real to formal neuron



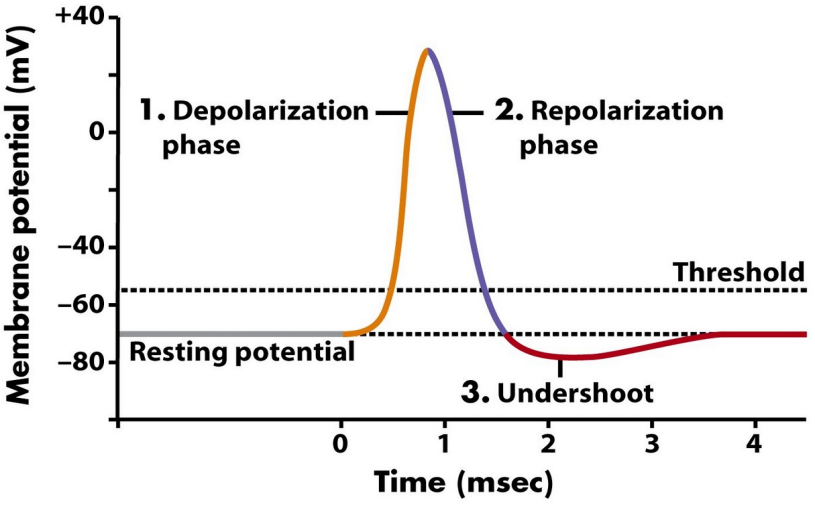
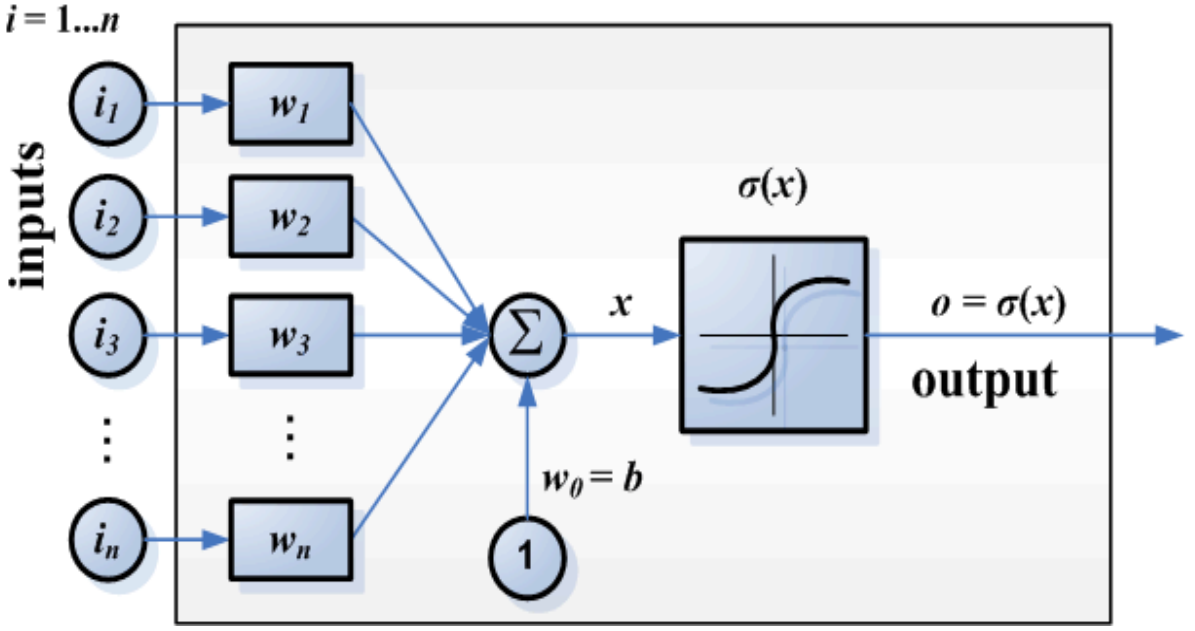
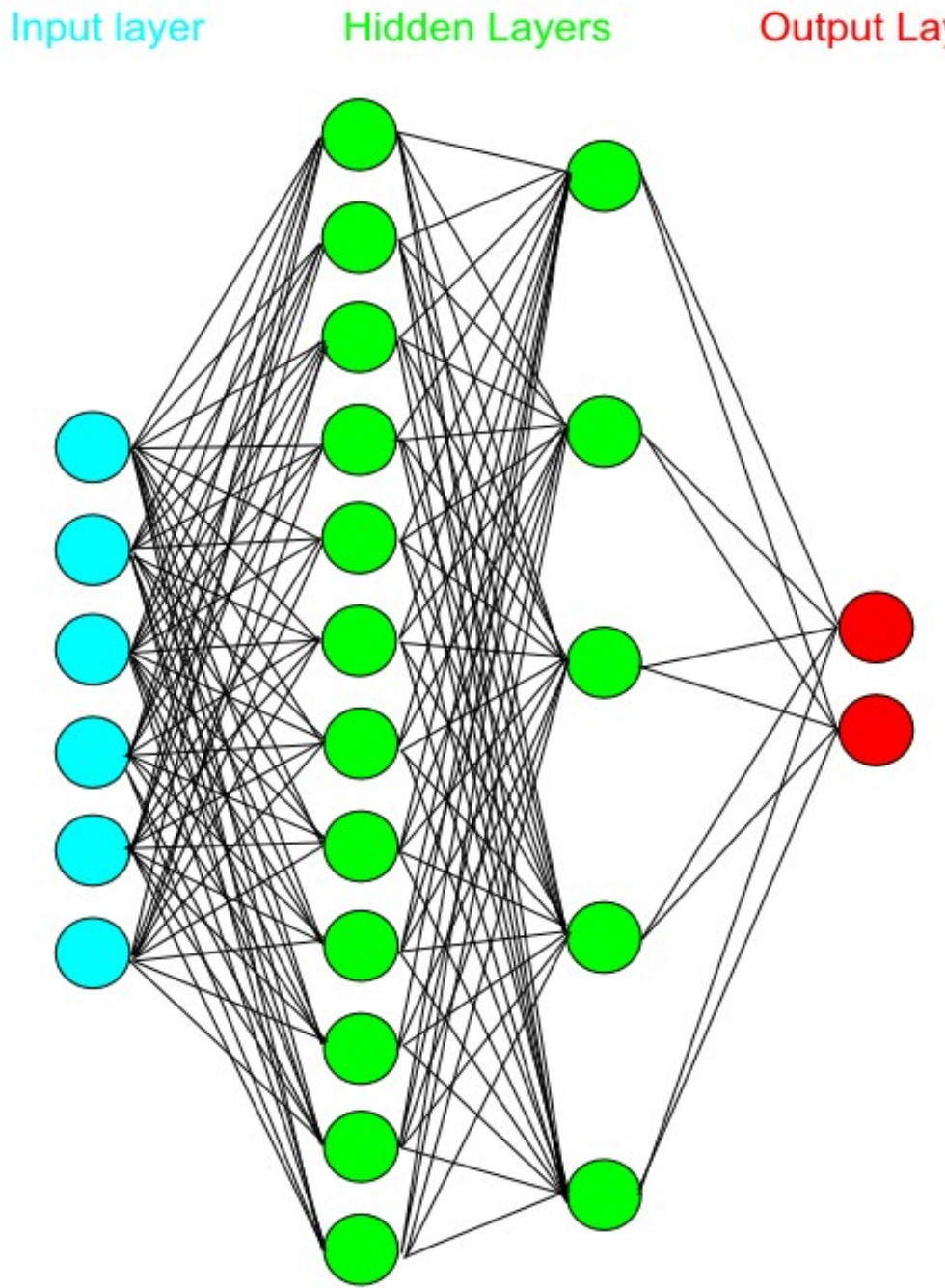


Figure 45-5 Biological Science, 2/e
© 2005 Pearson Prentice Hall, Inc.



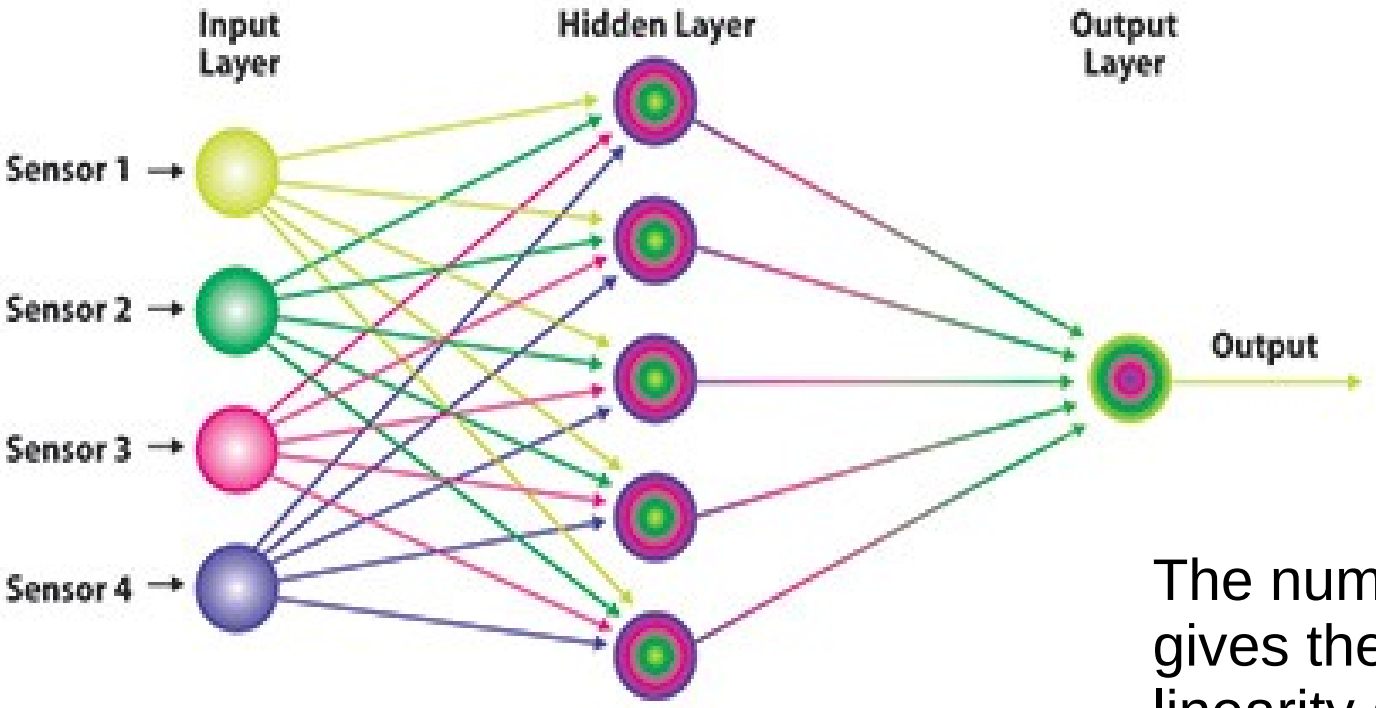
Warren McCulloch & Walter Pitts 1943

Multi-layer Perceptron



- Neural neurons
- Organized by layers
- Different size of layers
- Full connectivity between layers
- Input layer
- Output layer
- Hidden layers

How does it work ?



The number of hidden layer gives the level of non-linearity and the precision

Similar to power series
more development → more precision

Non linearity

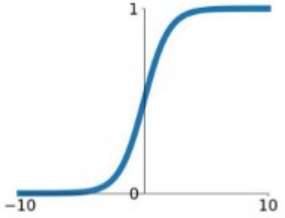
- Function σ
- Wished properties

- Non linear (if linear then equiv. to monolayer)
- Derivable everywhere
- Non null
- Centered on 0
- Monotonic
- Smooth (monotonic derivative)

Activation Functions

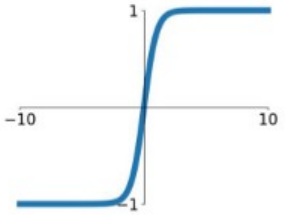
Sigmoid

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



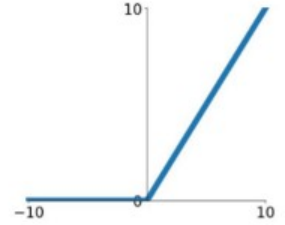
tanh

$$\tanh(x)$$



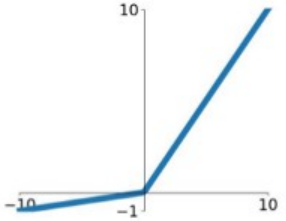
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

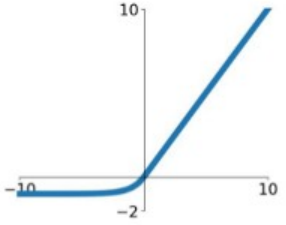


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Multi-layer perceptron formalism

- mono-layer (matricial form) $\hat{Y} = \sigma(W^T X + B)$
- multi-layer

$$\hat{Y} = \sigma(W_1^T \sigma(W_2^T \ldots \sigma(W_n^T X + B_n) + \cdots + B_2) + B_1)$$

- Depth of deep learning is the number of σ applications
- W_i and B_i are the parameters (Weights and Biases)
- X are the inputs
- No general form → universal approximator
- No estimator for parameters
- How to evaluate the optimal parameters ?

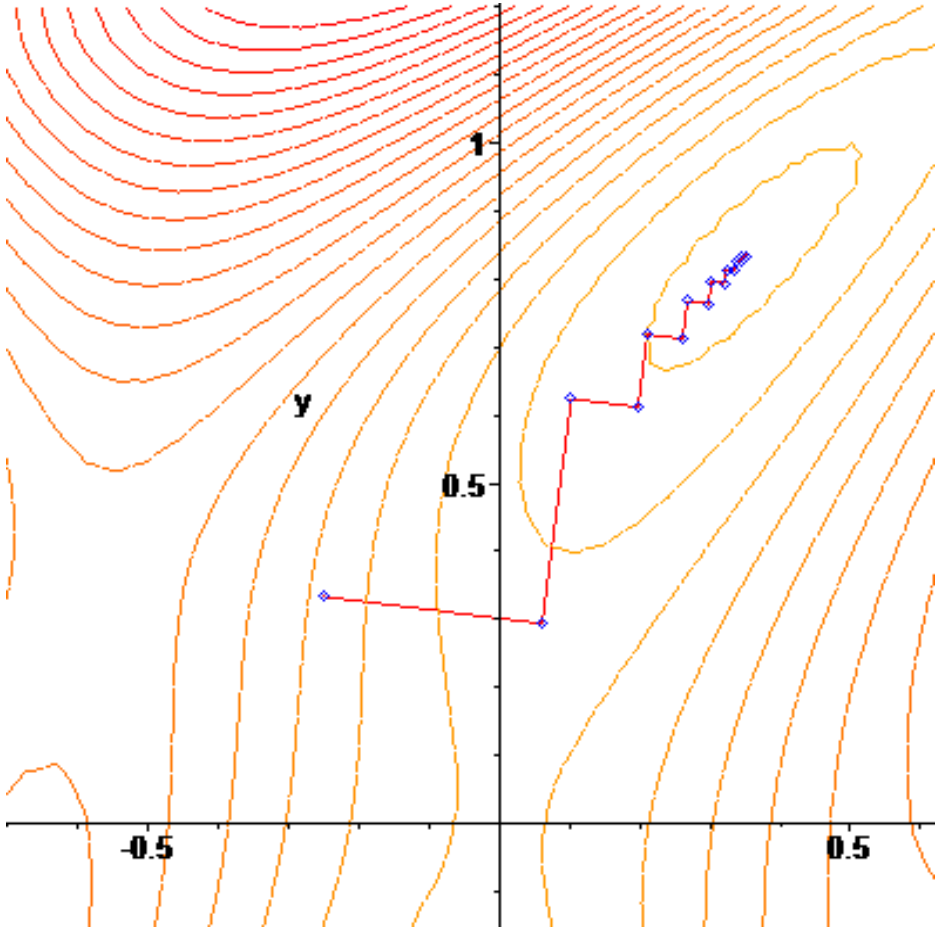
Optimization as a Blind Walk

- « Following the slope » method
- Only local knowledge of the field required
- Known as gradient descent algorithm class
- Proposed by Cauchy in 1847

Gradient Descent

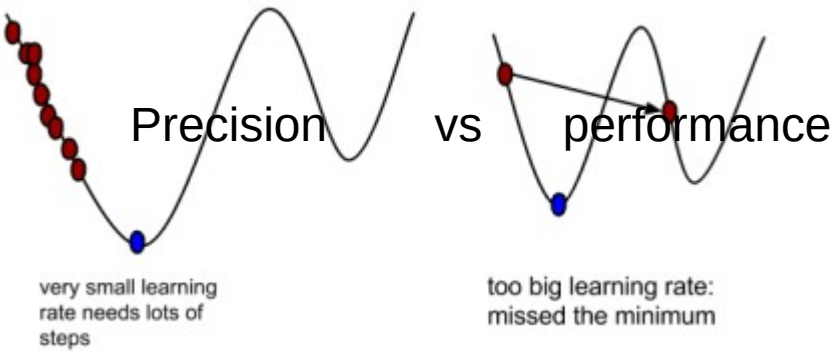
Generalizing the slope method to any number of parameters by calculating the gradient vector

$$\nabla J(\Theta) = \left\langle \frac{\partial J}{\partial \Theta_1}, \frac{\partial J}{\partial \Theta_2}, \ldots, \frac{\partial J}{\partial \Theta_n} \right\rangle$$

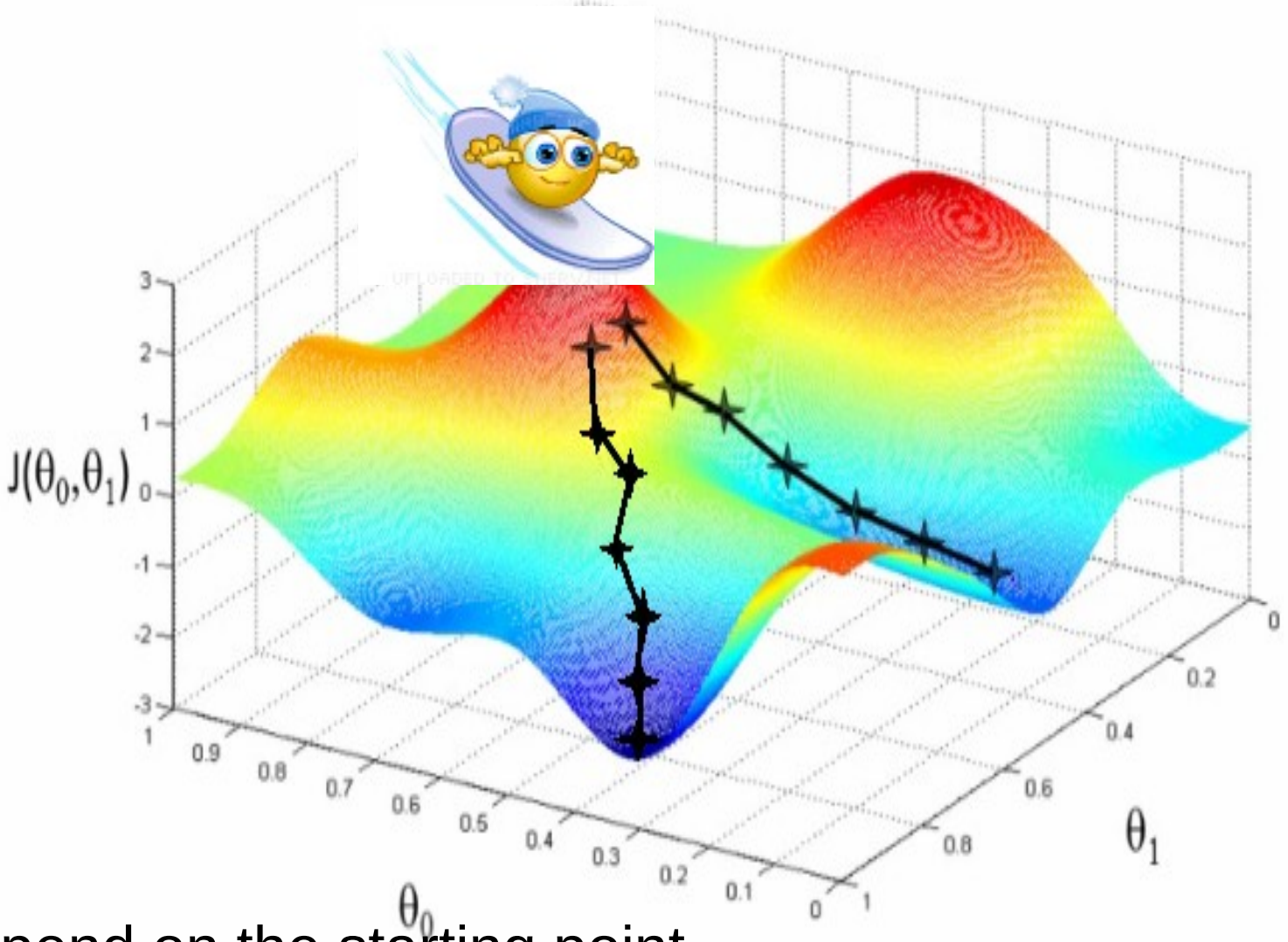


$$\Theta = \Theta - \alpha \nabla J(\Theta)$$

α : step size



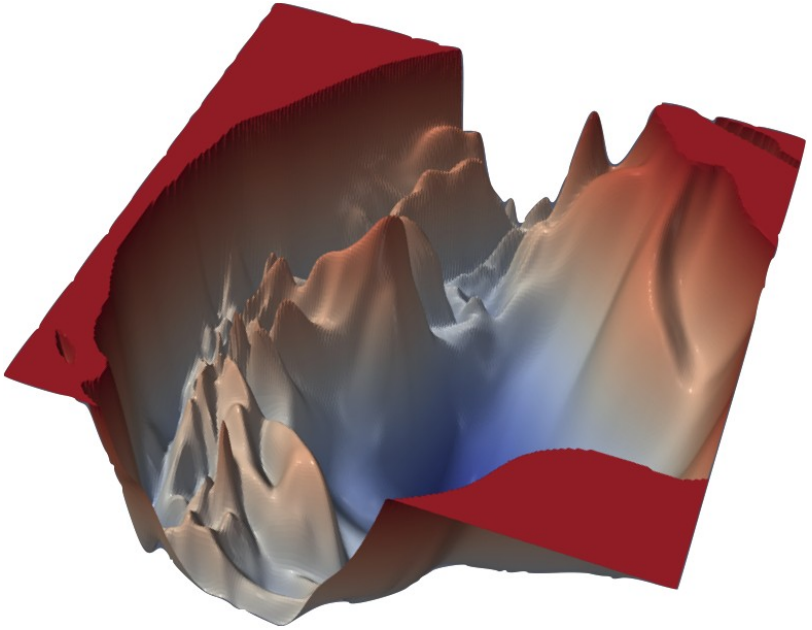
Gradient Descent & Convexity



- Depend on the starting point
  → require convexity (unique minima)
- Practical solution : multiple random starts

Backpropagation

- Based on gradient descent
- Function to optimize loss function L(Y,Ŷ)
- First version 1974, Paul Werbos, gradient retro-propagation algorithm
- Calculating partial derivatives over weigths and biases

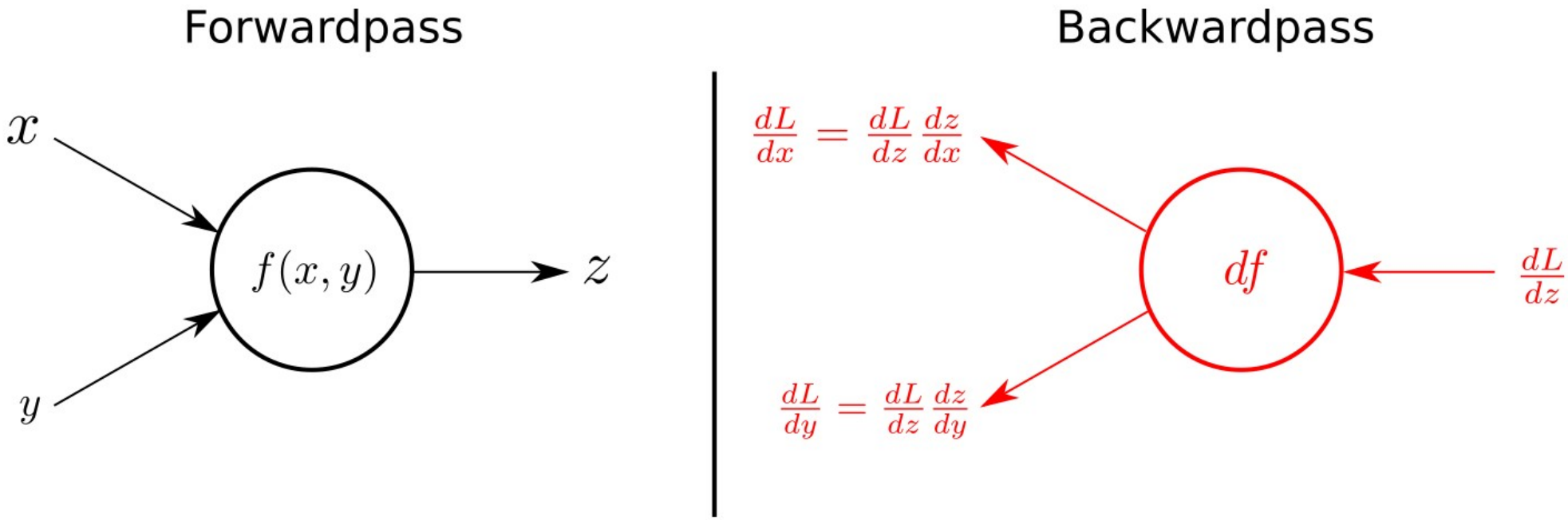


Li & al, « Visualizing the loss landscape of neural nets, 2018, 1712.09913

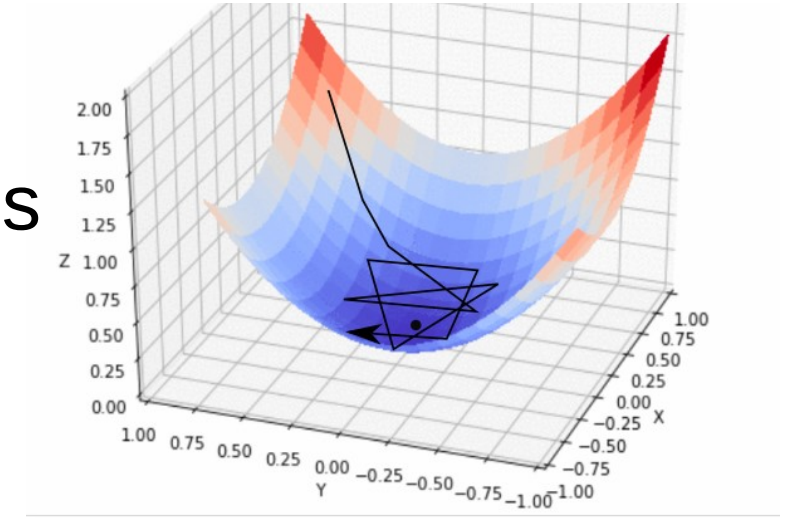
At each iteration, for each parameters w_i

$$w_i = w_i - \alpha . \frac{\partial L}{\partial w_i}$$

Backpropagation implementation by automatic differentiation



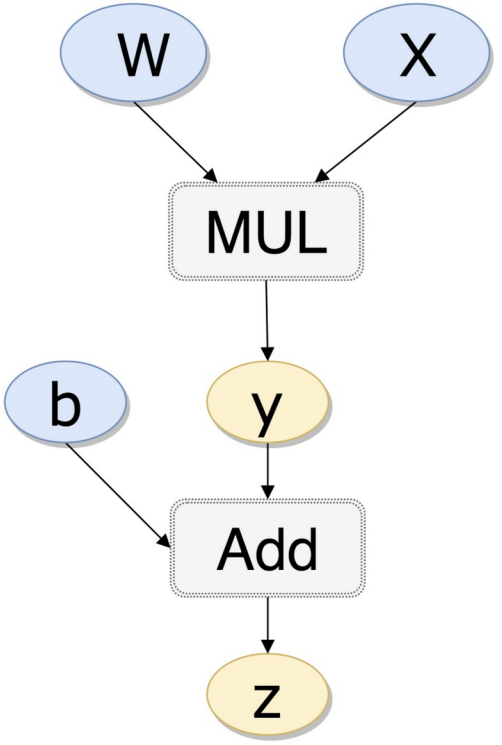
- Gradient descent
- Propagation of derivative of loss function
- Chain rule



Automatic Differentiation Example

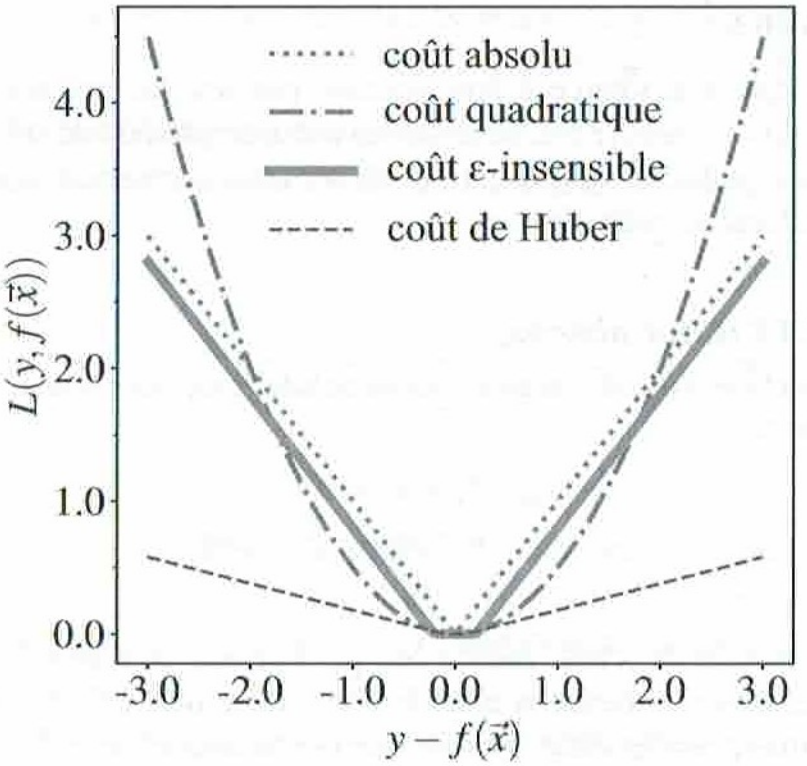
- The derivative of an expression is build as chain rule multiple application
- Dynamic and automatic execution graph building

$$\frac{\partial z}{\partial y} = 1$$
$$\frac{\partial z}{\partial b} = 1$$
$$\frac{\partial y}{\partial x} = w$$
$$\frac{\partial y}{\partial w} = x$$
$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y} = w$$
$$\frac{\partial z}{\partial w} = \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial y} = x$$



Regression loss function

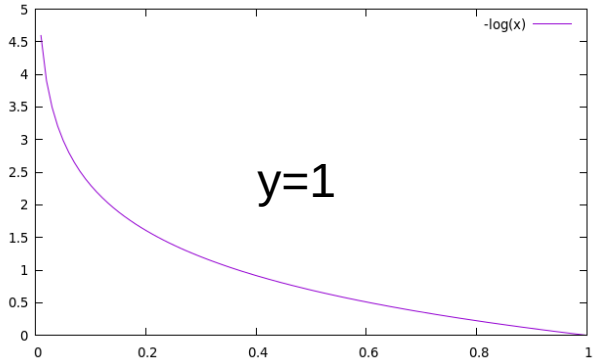
- Base of retro-propagation
- Used to quantify the difference between target and obtained result
- Plenty of different loss functions
  - Absolute error or L1
    - Not derivable → not used in NN
  - Mean square error or L2
    - Most used
    - Too Sensitive to large errors

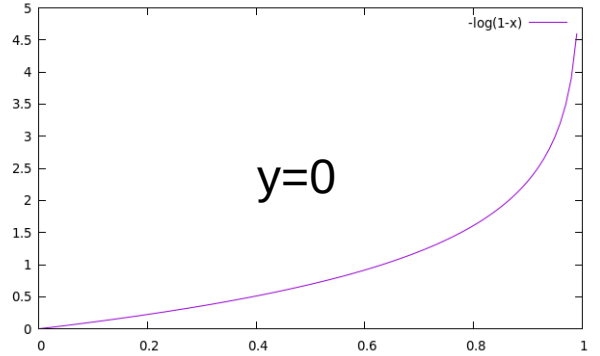


Classification Loss function

- L2 can be used (with probability vs 0/1) but not ideal
- Binary Cross-entropy (BCE) dedicated to binary classification (value must be between 0 & 1)

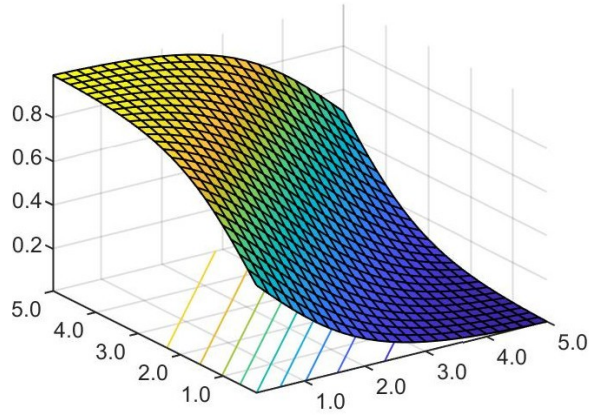
$$L(y, \hat{y}) = -(y.log(\hat{y}) + (1 - y).log(1 - \hat{y}))$$





- Softmax (normalized exponential) for multi-class classification

$$L(y, \hat{y}) = -\sum_{c=1}^{C} \delta(y, c) log(\hat{y})$$



The loss function shape

- perceptron ⟷ spherical spin-glass model
- theoritical results reuse
  - $\#min_{loc} \alpha e^{dim}$
  - $\#Bad\_min_{loc} \alpha e^{-dim}$
  - Good local minimum : $J(min_{loc}) - J(min_{glob}) \leq \epsilon$
  - Funnel global shape
- Global minimum is overfitting
- Deep learning (dim is big) gives better results

Lecun & al, The loss surface of multi-layer networks, 2015, 1412.0233

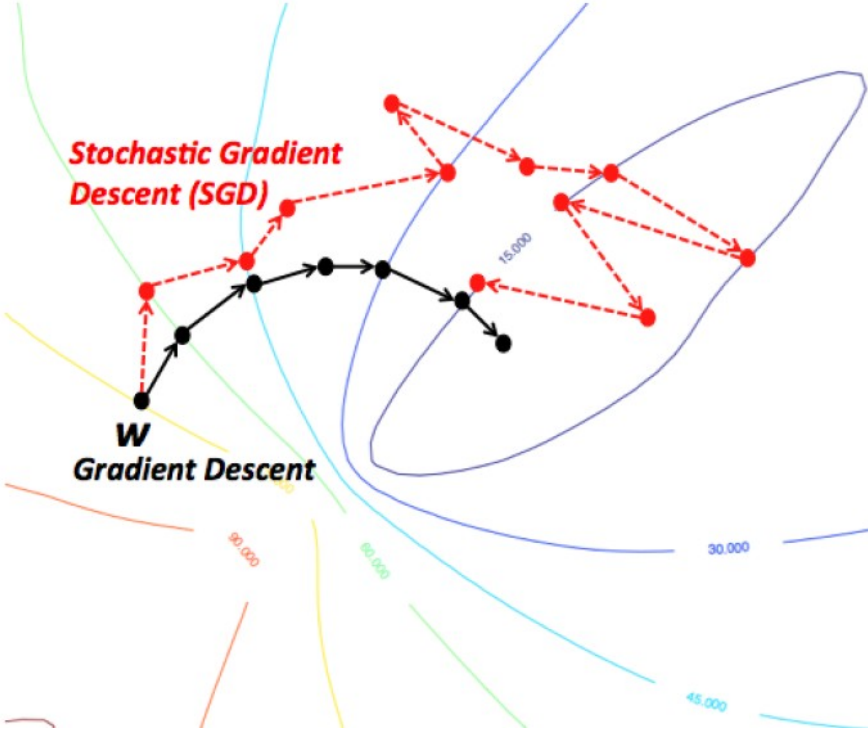
Loss function denoted J in the sequel

Stochastic gradient descent

Principle

- Computes the gradient one data at a time
- Works on sum-structured objective function

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J(\theta_t, x(i), y(i))$$

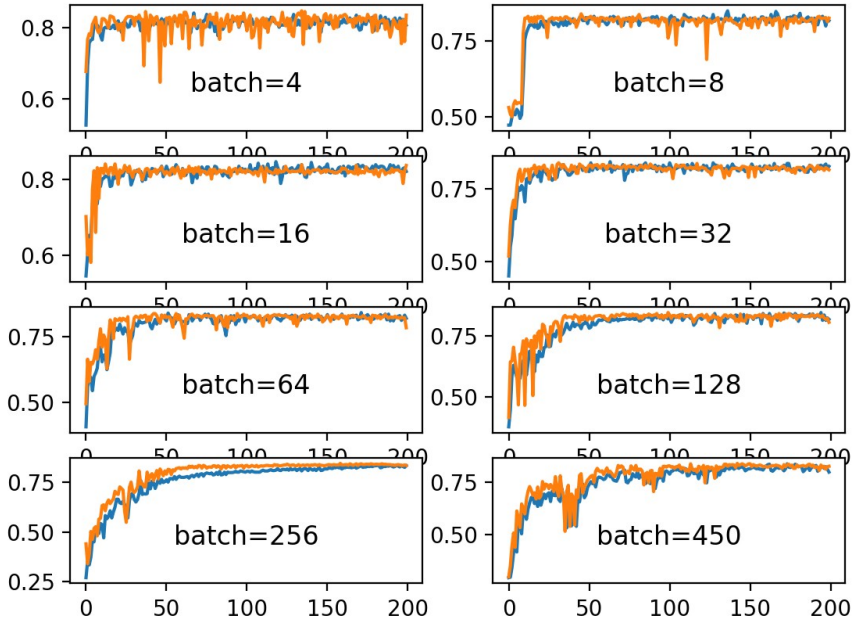



```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for d in data:
    params_grad = evaluate_gradient(loss_function, d, params)
    params = params - learning_rate * params_grad
```

Pros
- Converges faster
- Require no memory

Cons
- High variance
- May shoot after achieving global minima
- Need learning rate adaptation

Mini-Batch Gradient Descent



Principle

- Computes gradient on batches of m data
- Reduces variance

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J(\theta_t, x(i; i+m), y(i; i+m))$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data,batch_size):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Pros
- Reduce variance
- Require medium amount of memory

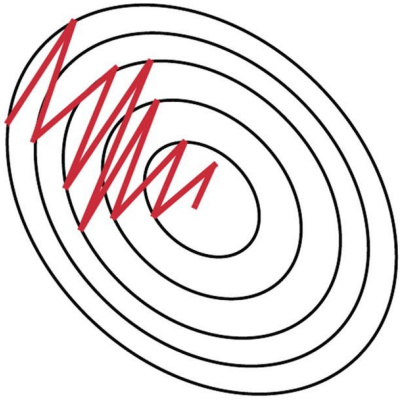
Cons
- Need memory again
- Local minimas

Momentum

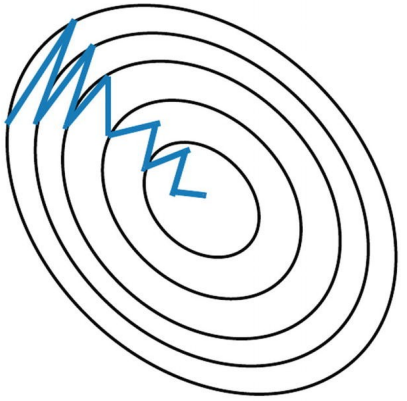
Notation : $g_t = \nabla_\theta J(\theta_t)$

Principle

- Softens the convergence of SGD by keeping a momentum



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

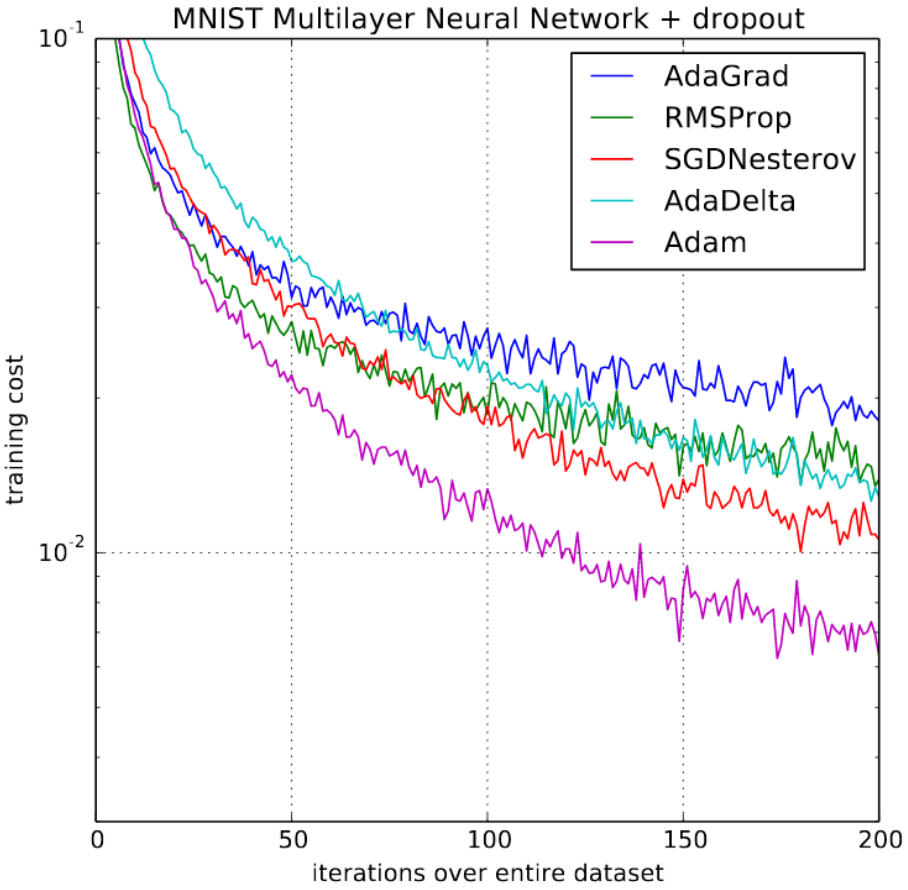
$$\theta_{t+1} = \theta_t - V_t \qquad V_t = \gamma V_{t-1} + \alpha g_t \qquad \gamma_{def} = 0.9$$

```
V=0
for i in range(nb_epochs):
  np.random.shuffle(data)
  for d in data:
    params_grad = evaluate_gradient(loss_function, d, params)
    V=γ * V + leaning_rate * params_grad
    params = params - V
```

Adam : Adaptive Moment Estimation

A method for stochastic optimization, Kingma & Ba, 2017, 1412.6980

- One learning rate per parameter
- Slow down adagrad a bit to get better precision
- Use the two first moments to adjust parameter (mean and uncentered variance)
- Gradient is replaced by a sliding window moment
- Properties
  - Actual step size bounded by learning rate parameter
  - Step size is invariant to magnitude of gradient



Adam (2)

1st & 2nd order moments
Initialized at 0 → biased at the beginning

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Evolution rule

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Unbiased estimators

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

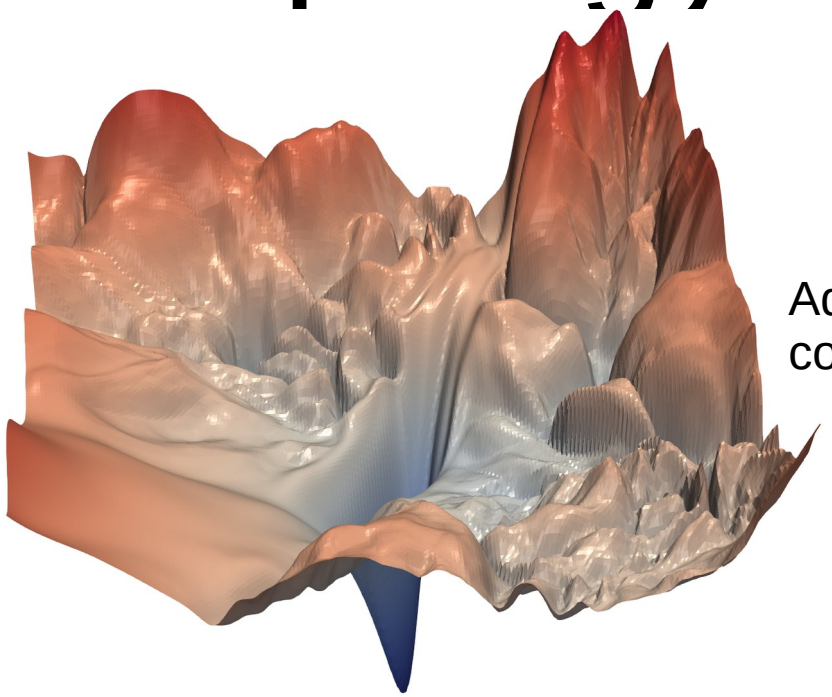
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Good default settings

- α=0.001
- β1=0.9
- β2=0.999
- ε=10−8

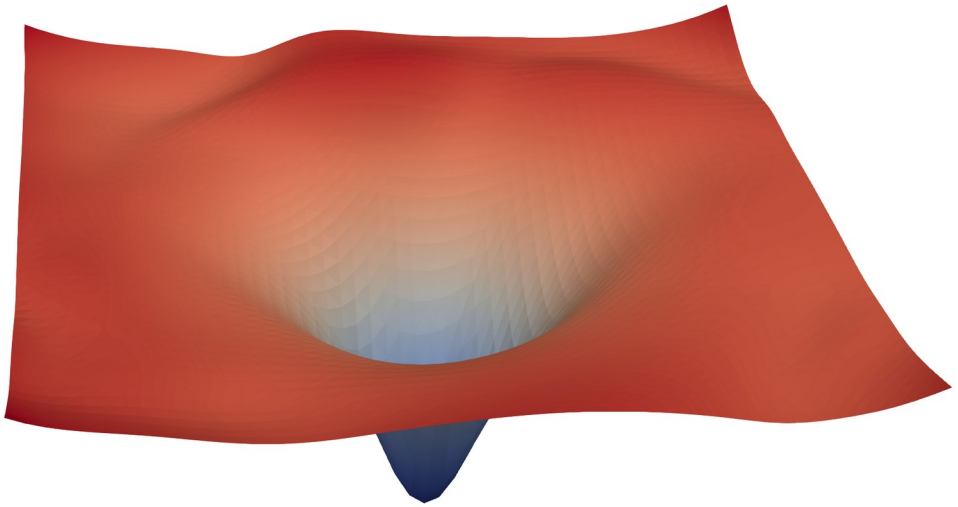
```
for t in range(nb_iters):
    g = compute_gradient(w)
    m = beta_1 * m + (1 - beta_1) * g
    v = beta_2 * v + (1 - beta_2) * np.power(g, 2)
    m_hat = m / (1 - np.power(beta_1, t))
    v_hat = v / (1 - np.power(beta_2, t))
    w = w - learning_rate * m_hat / (np.sqrt(v_hat) +
epsilon)
```

Topology influence on loss



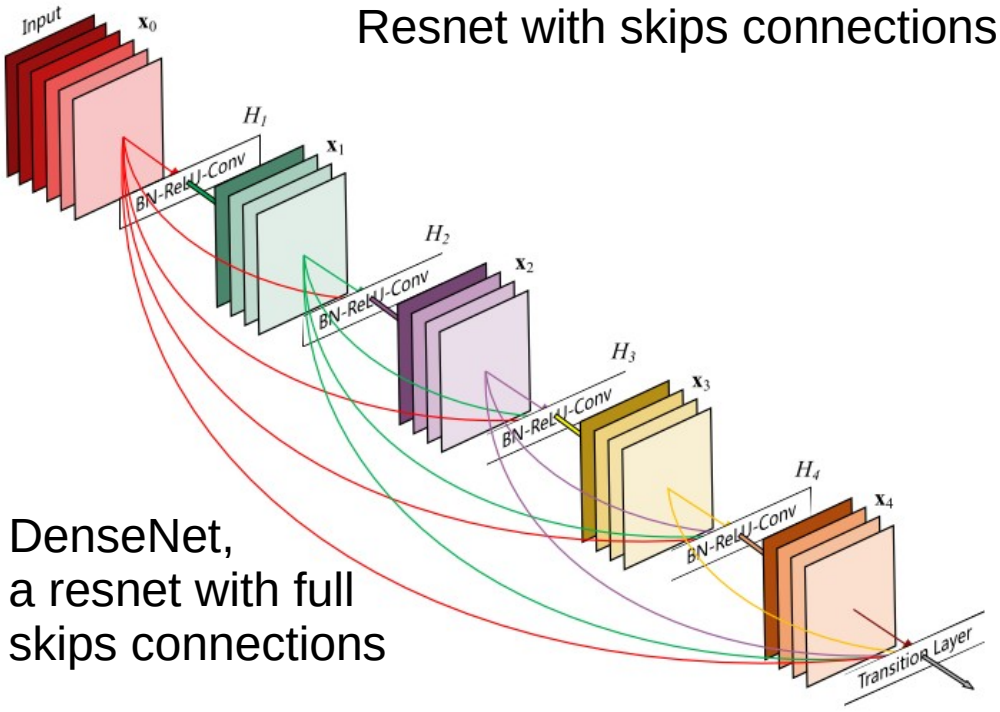
Resnet (very deep convolutional NN)

Adding skips → connections



Resnet with skips connections

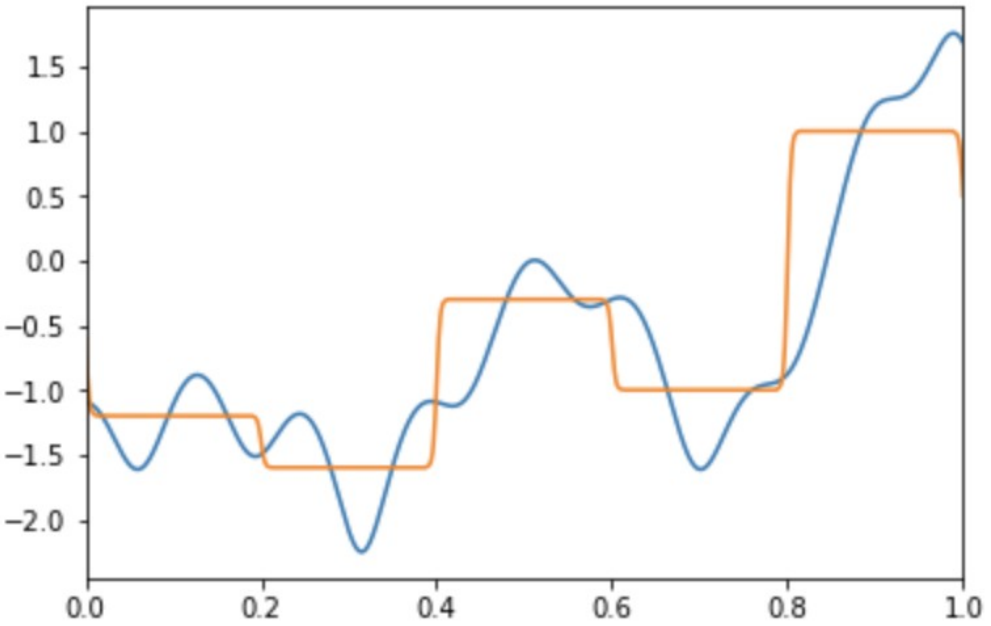
Topology influences dramatically the loss surface shape



DenseNet,
a resnet with full skips connections

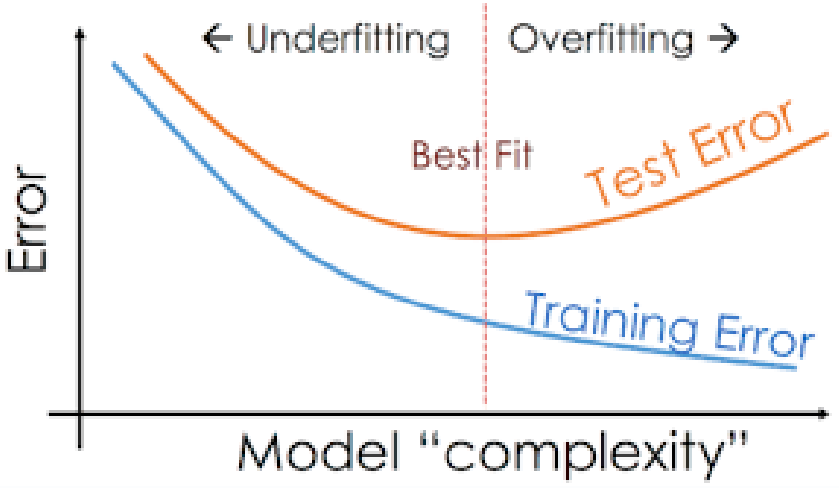
What kind of function can I train ?

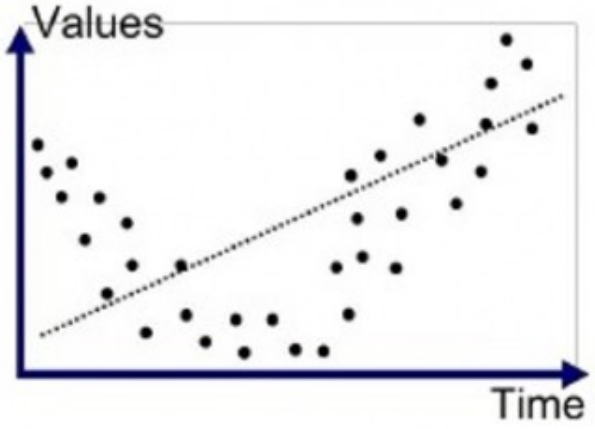
- Any continuous multivariate function on R (Hornik et ak 1989) → universal approximator
- Extended to R^n (Sun and Cheney 1992)
- Extension to classification problems (Cybenko 1989) → universal classifiers
- Caution : the theorem gives no clue about learnability



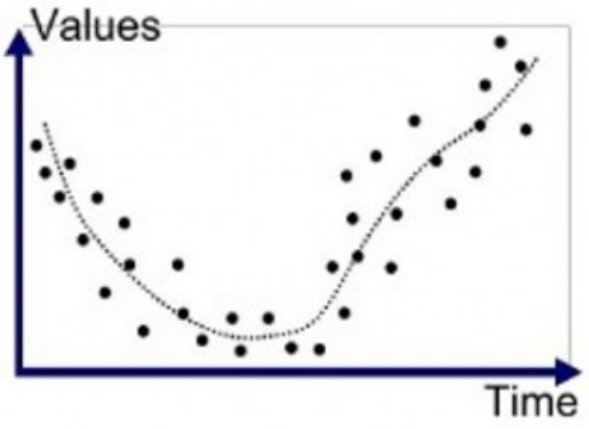
Training and over-fitting

- While training, the model can over-fit the data
- Too adapted, no generalization power
- Must be tested with a test data set
- Solutions
  - Increase the data set
  - Penalize the number of parameters

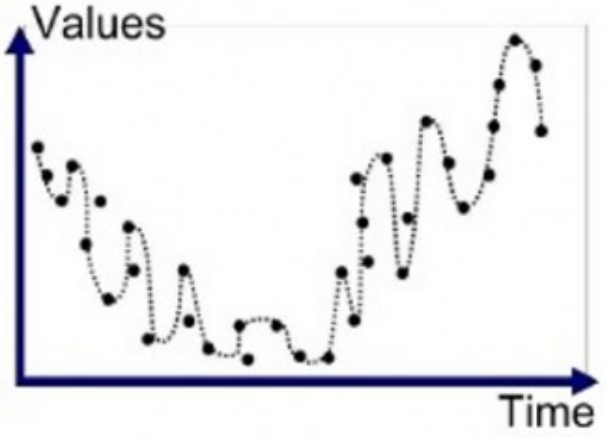




Underfitted



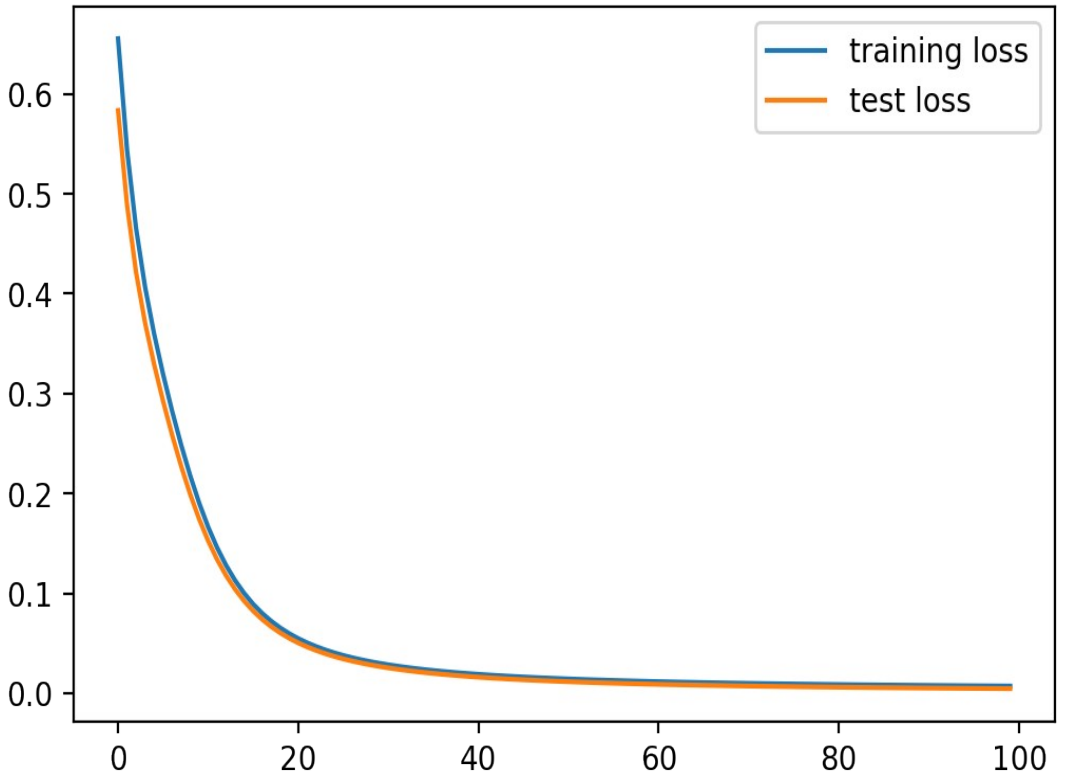
Good Fit/Robust



Overfitted

Non convergence

- Sometimes the gradient descent does not converge
  - Not enough convex loss function (too simple architecture, not enough parameters)
  - Touchy correlation between input and output
- Solutions
  - Increase model complexity
  - Normalize data as inputs or by batch



The perfect convergence

Batch normalization

- Normalize input of every hidden layer

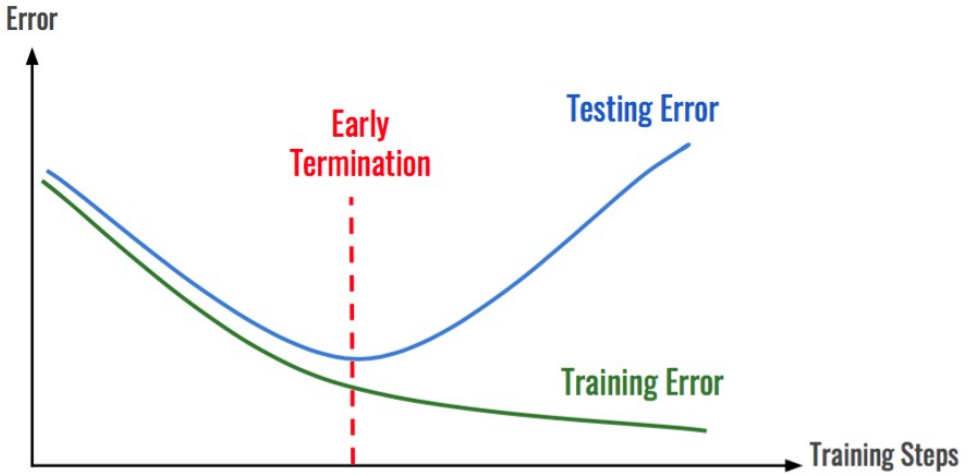
$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

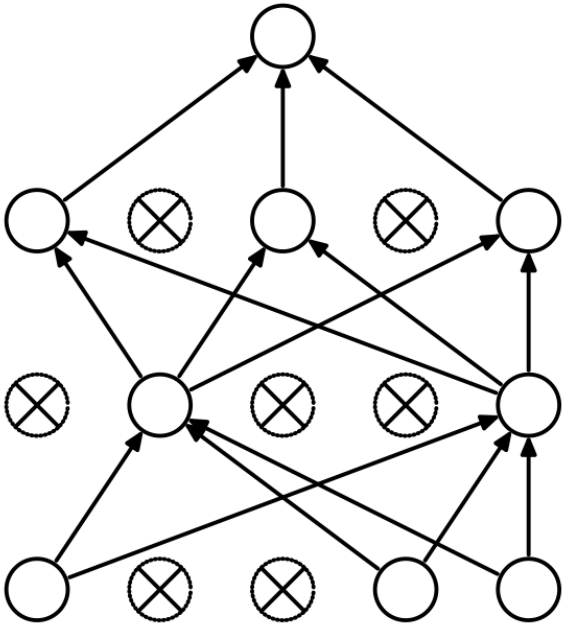
Ioffe & al, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015, 1502.03167

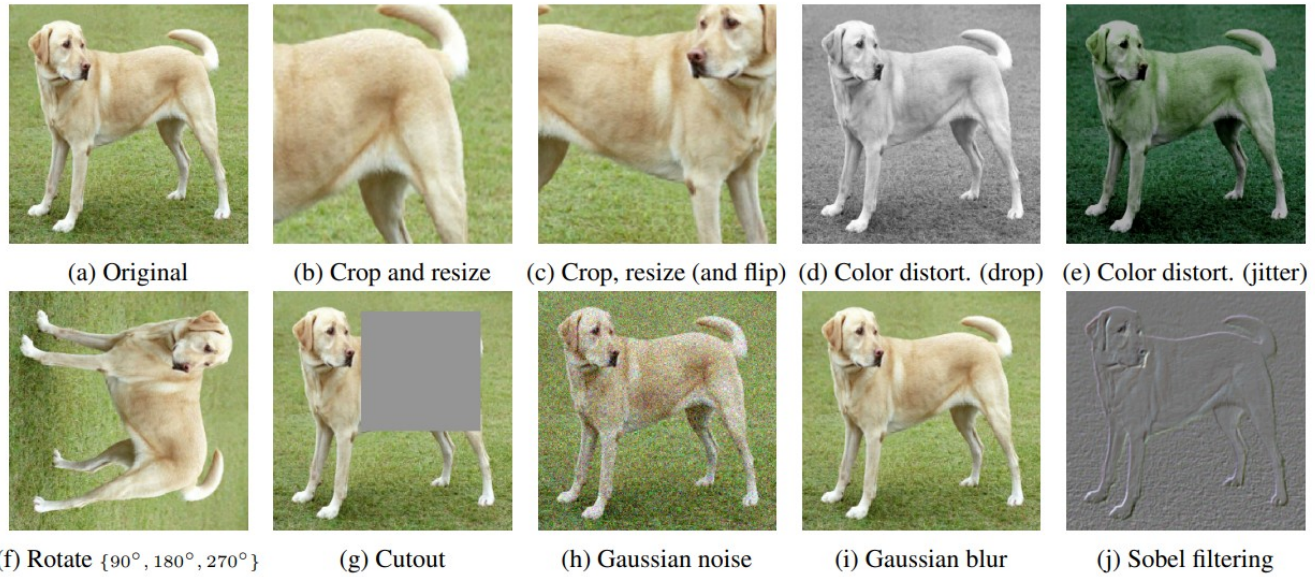
- Mean and variance are calculated for every batch
- Data are adjusted accordingly
- Make the layers a little bit more independant
- Improve speed, performance and stability

Regularization : increase generalization power

- Reduce the number of parameters → bias
- Dropout
- Data augmentation
- Early stopping



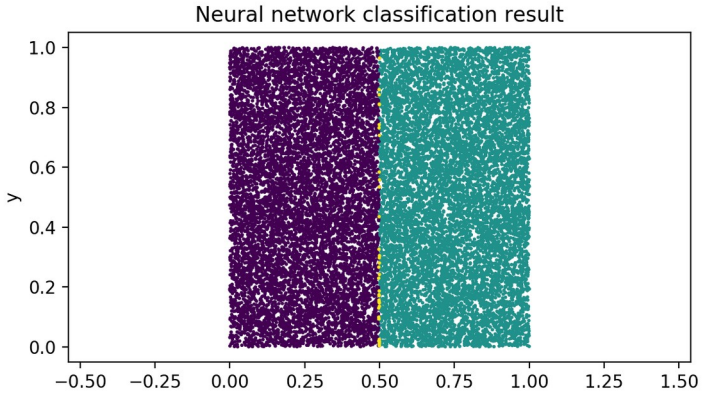


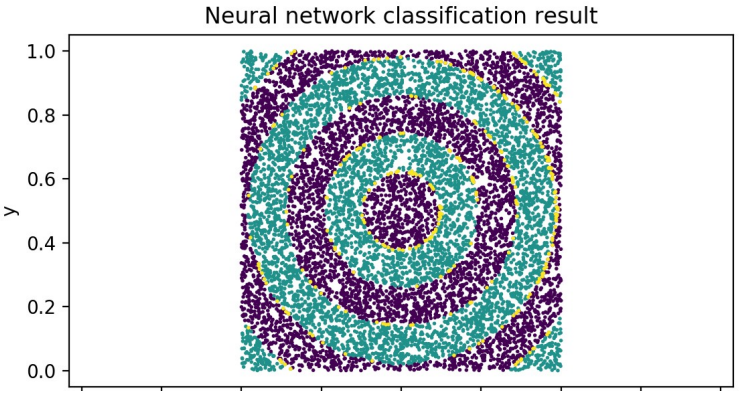


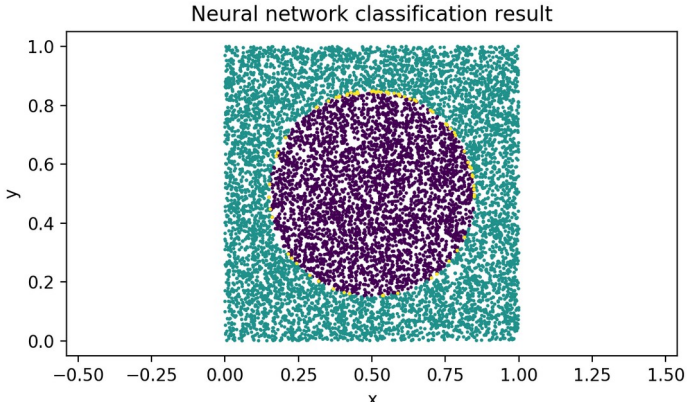
Implementations

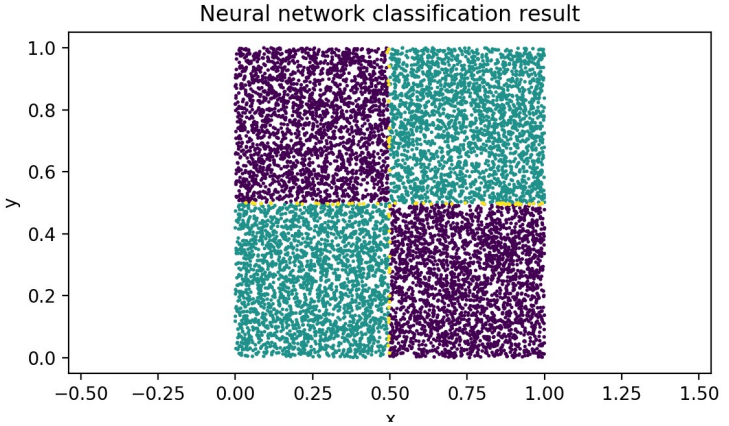
- TensorFlow / Keras (Google)
  - Black box, used in industry
- PyTorch (Facebook)
  - used in academics, plenty of experimental libraries
- Scikit-learn (Python)
  - underlying tools (statistics)
- GPU support from most of libraries
- Common format : ONNX

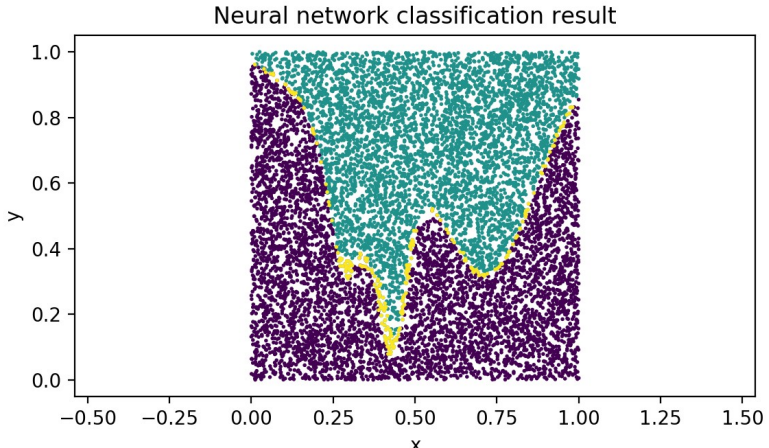
Perceptron in action

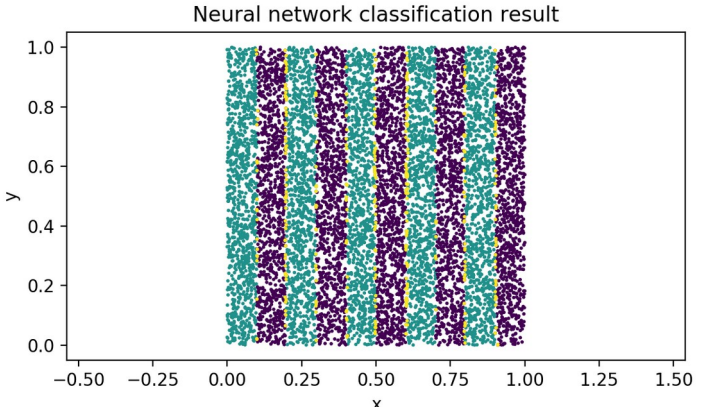






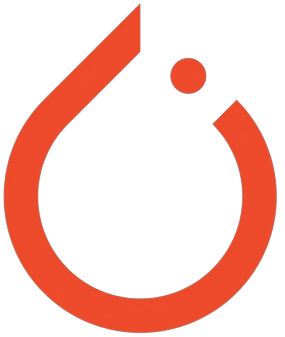




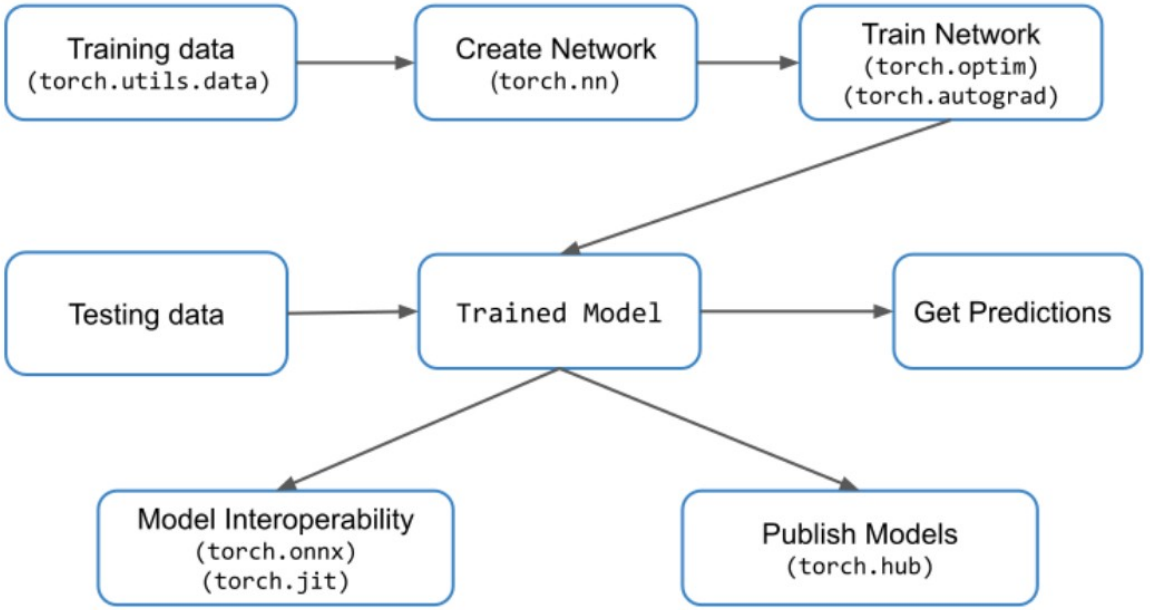


- Problematics
- Neuron, perceptron and back-propagation
- **PyTorch Hands on**
- Convolutional networks
- Auto-encoders
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

Description

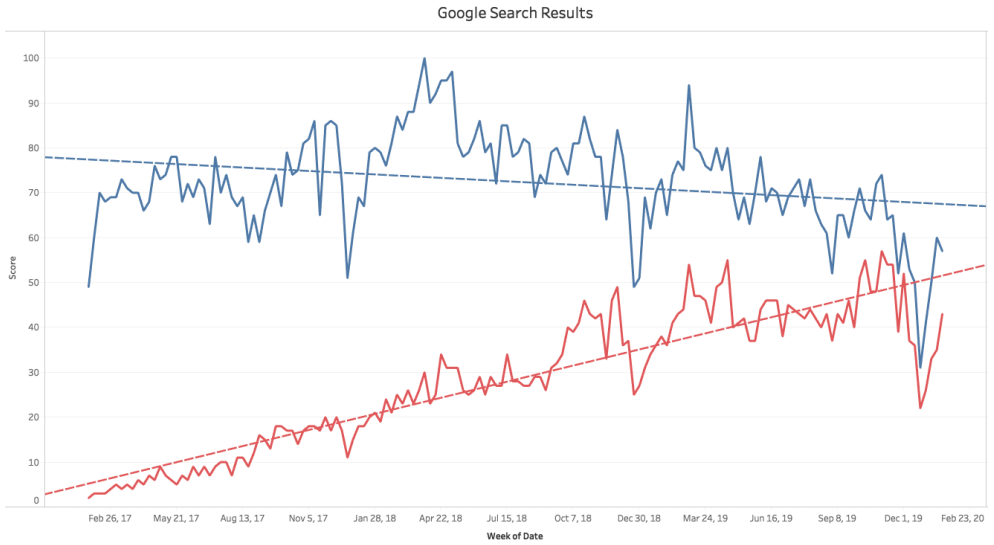


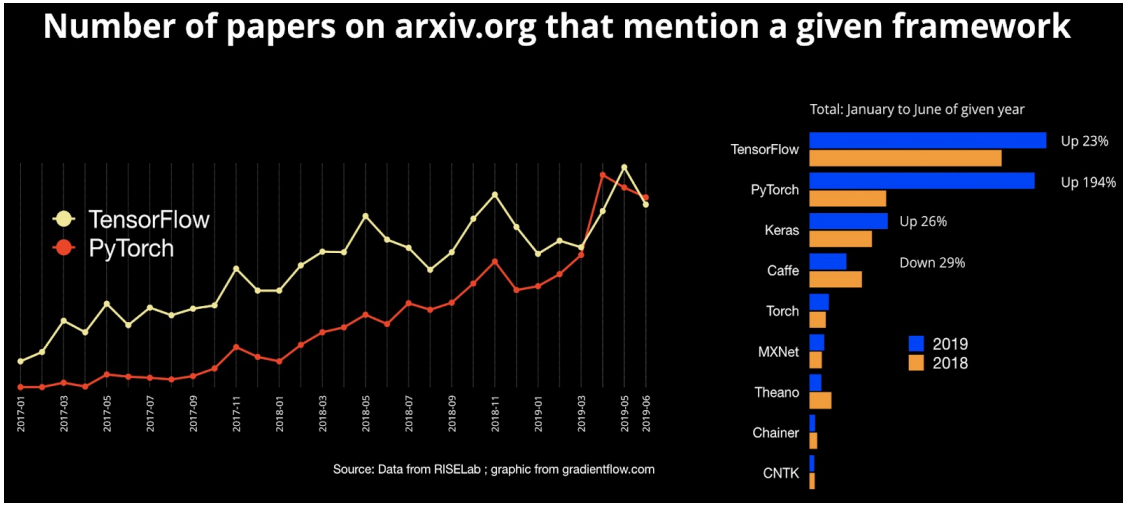
- Deep learning general framework since January 2016
- Easy to use API
- Integrated into Python science stack
- Imperative programming
- Easy to use Multi-GPU
- Allow to build custom data loaders
- Allow to build arbitrary complex networks



Pytorch vs TensorFlow

- Mandatory to follow AI community novelties
- More dedicated to data science
- Less used in industry





Comparison with TF

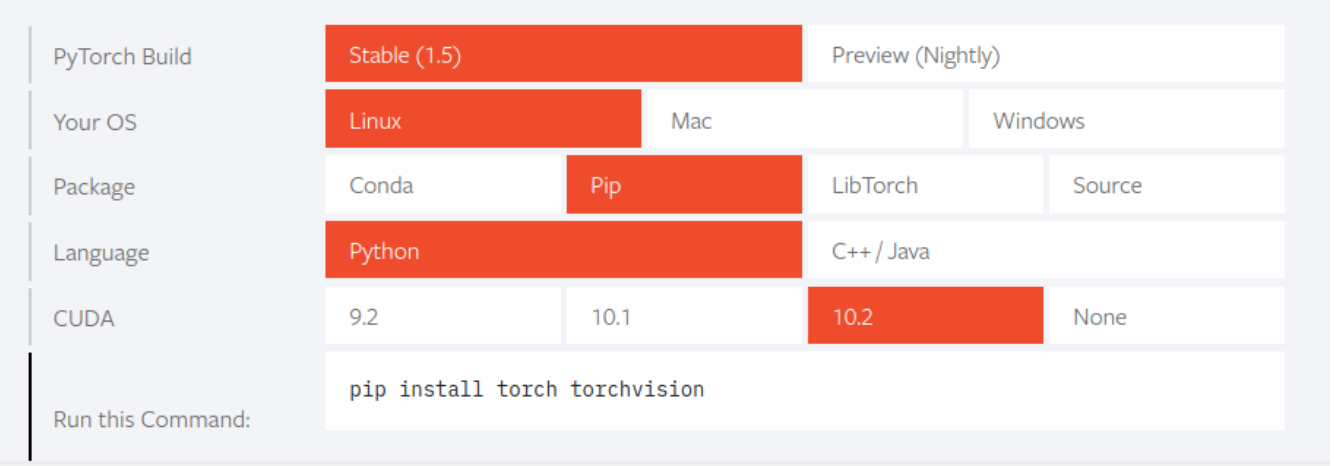
- Torch needs more deep details
  - Dataloaders, validation
  - Fit loop
- Torch is easier to debug
  - simple pythonic formulas ≠ object
  - pure imperative prog. (step by step)
- Torch is harder to learn : steeper learning curve
- Torch allow better customization

Installation

- On official site https://pytorch.org, install command generator
- Easier is pip

```
pip install torch torchvision
```

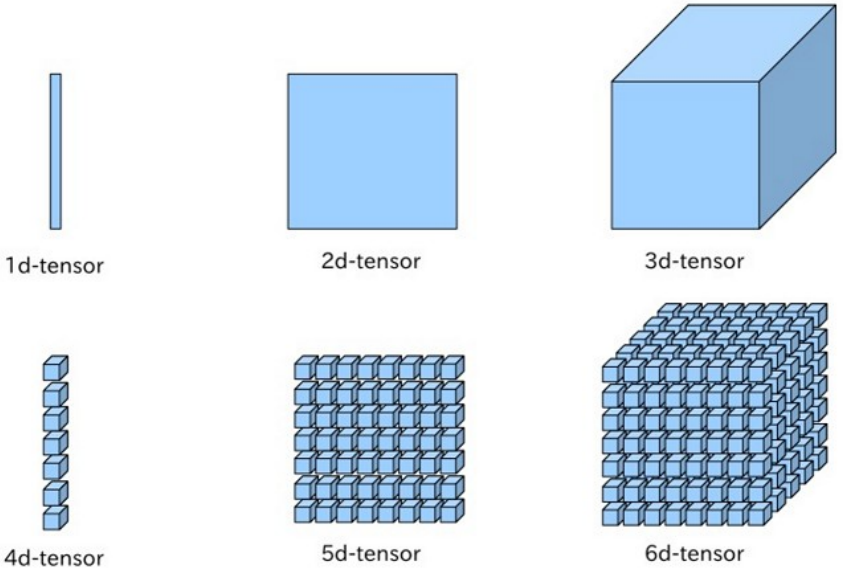
- Module is named torch in Python (not pytorch)




```
$python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
```

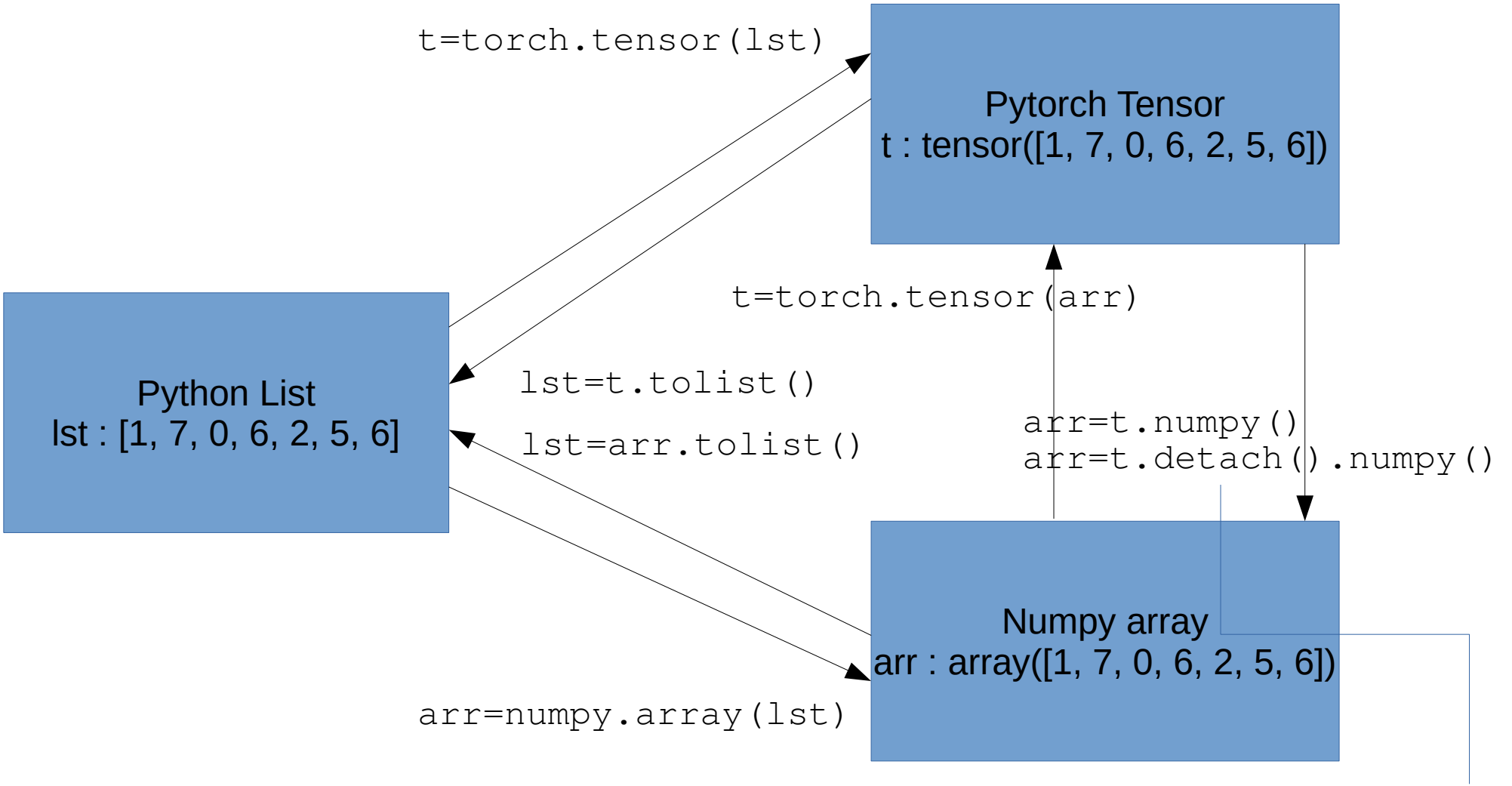
Tensors

- Multi-dimensional arrays
- Similar to numpy's ndarrays
- Can be used on GPU
- Typed but auto-detection available




```
>>> torch.FloatTensor([2])
tensor([2.])
>>> torch.tensor([2.0])
tensor([2.])
>>> torch.tensor([2],dtype=float)
tensor([2.], dtype=torch.float64)
>>> torch.rand((2,5))
tensor([[0.1385, 0.1119, 0.0556, 0.9772, 0.6497],
        [0.9943, 0.4124, 0.9334, 0.1790, 0.2562]])
>>> a=torch.ones((3,3))
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
>>> a.shape
torch.Size([3, 3])
>>> a[0,0].item()
1.0
```

Python list ↔ Numpy ↔ Pytorch tensors




```
In case of autograd :
RuntimeError: Can't call numpy() on Variable
that requires grad. Use var.detach().numpy() instead.
```

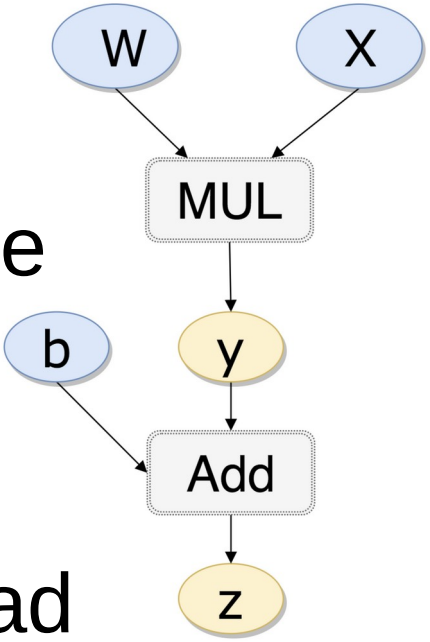
Tensor’s operations

- More than 200+ mathematical operations
- Pythonic integration of this operators
- Automatic derivatives included via autograd

```
>>> a=torch.tensor([[2.0,3.0],[4.0,5.0]])
>>> b=torch.ones(2,2)
>>> a+b
tensor([[3., 4.],
        [5., 6.]])
>>> a*b #element wise multiplication (Hadamard product)
tensor([[2., 3.],
        [4., 5.]])
>>> torch.mm(a,b) #matrix multiplication
tensor([[5., 5.],
        [9., 9.]])
>>> torch.tanh(a)
tensor([[0.9640, 0.9951],
        [0.9993, 0.9999]])
>>> torch.sin(torch.ones((3,3)))
tensor([[0.8415, 0.8415, 0.8415],
        [0.8415, 0.8415, 0.8415],
        [0.8415, 0.8415, 0.8415]])
```

Autograd

- Automatic calculation of partial derivative
- Backward is the function computing the derivatives
- Available as a field of the variables x.grad




```
>>> w=torch.tensor([4.0],requires_grad=True)
>>> x=torch.tensor([3.0],requires_grad=True)
>>> b=torch.tensor([2.0],requires_grad=True)
>>> z=b+x*w
>>> z.backward()
>>> print(b.grad)
tensor([1.])
>>> print(w.grad)
tensor([3.])
>>> print(x.grad)
tensor([4.])
```

$$\frac{\partial z}{\partial y} = 1$$

$$\frac{\partial z}{\partial b} = 1$$

$$\frac{\partial y}{\partial x} = w$$

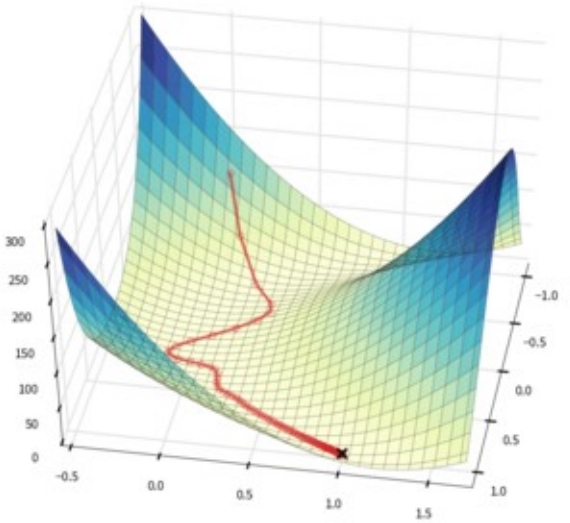
$$\frac{\partial y}{\partial w} = x$$

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y} = w$$

$$\frac{\partial z}{\partial w} = \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial y} = x$$

Optimizer

- Torch implements all standard optimizers based on autograd
- First parameter : a set of autograd-able variables
- Optimizer hyper-parameters as named parameters
- Two important functions
  - zero_grad() : clear the gradient of all variables used by the optimizer (w.grad=0)
  - step() : apply the optimizer step based on the autograd backward function



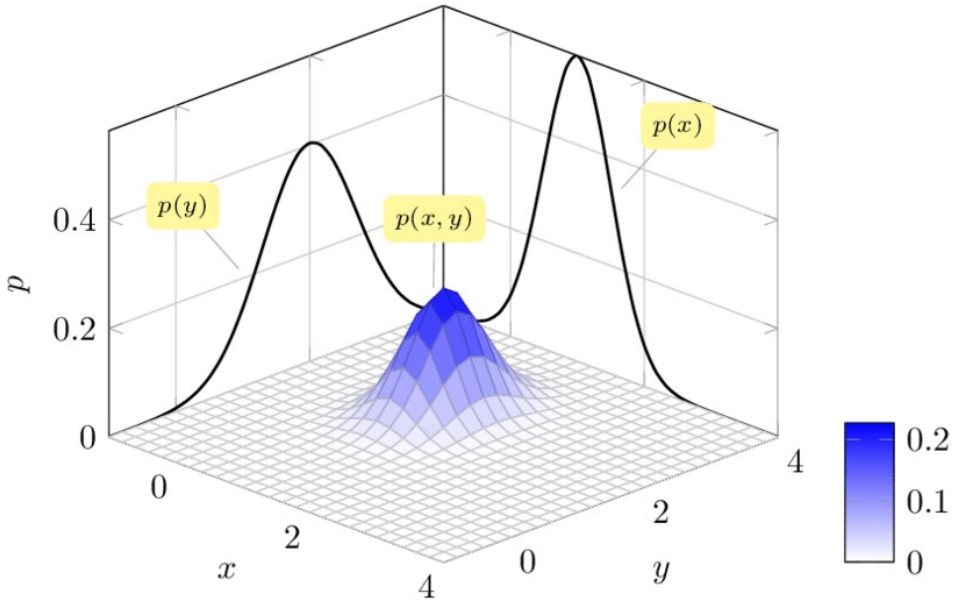
$$w = w - \alpha . \frac{\partial L}{\partial w}$$

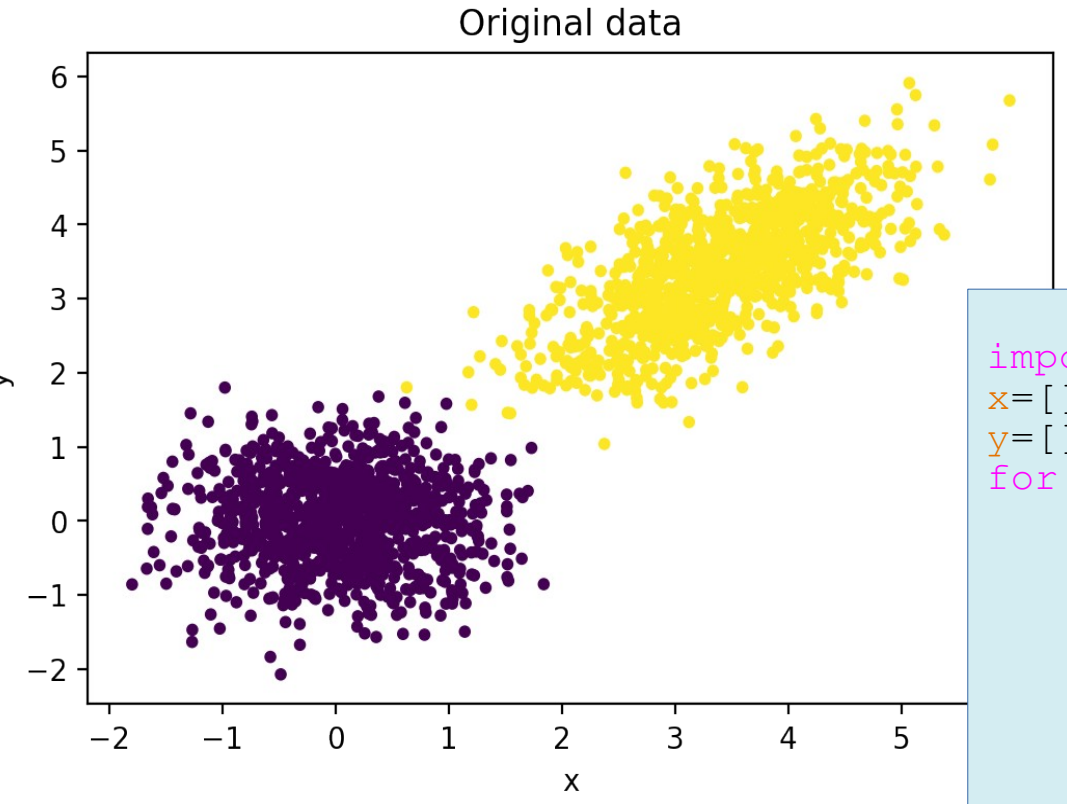
```
import torch.optim as optim

optimizer=optim.SGD(model.parameters(),lr=0.01,momentum=0.9)
optimizer=optim.Adam([var1, var2],lr=0.0001)

optimizer.zero_grad()
loss=loss_fn(autograd_variables,other_variables)
loss.backward()
optimizer.step()
```


q

First MLP training

- Train a MLP to distinguish two multivariate normal distributions of points
- Generate two data list
  - x: coordinates of points
  - y: label of points (boolean)

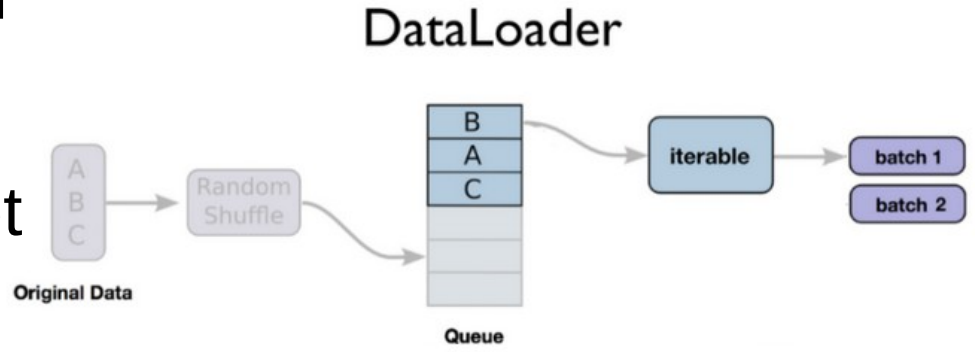





```
import numpy as np
x=[]
y=[]
for i in range(1000):
    t=np.random.multivariate_normal([0,0],
                [[0.4,0],[0,0.4]])
    x.append(t)
    y.append(0)
    t=np.random.multivariate_normal([3.5,3.5],
                [[0.6,0.4],[0.4,0.6]])
    x.append(t)
    y.append(1)
```

Creating the dataloader

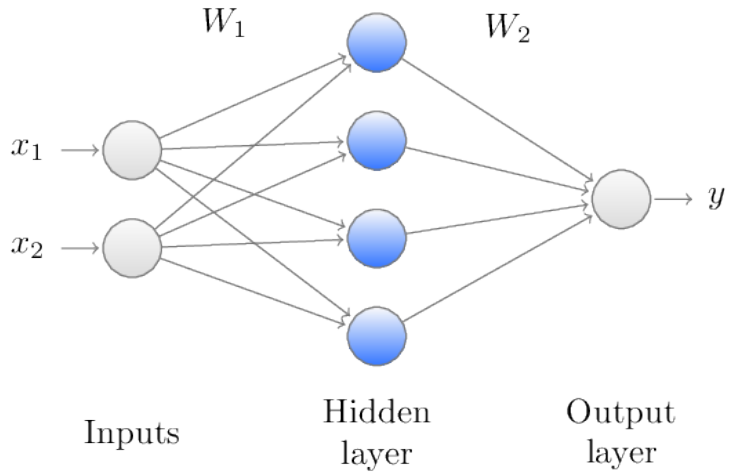
- Data must be inserted in a tensor
- Dataset : x/y concatenation
- Split function for train/test dataset
- Dataloader : batch grouping + optional shuffling




```
from torch.utils import data

#Convert data from python list to tensors
tensor_x=torch.Tensor(x)
tensor_y=torch.Tensor(y)
dset=data.TensorDataset(tensor_x,tensor_y)

#split data for training/testing
train_size=int(0.9*len(dset))  #90% for training
test_size=len(dset)-train_size #10% for testing
train_dset,test_dset=data.random_split(dset,[train_size,test_size])

#create the two data loaders
train_loader=data.DataLoader(train_dset,batch_size=64,shuffle=False)
test_loader=data.DataLoader(test_dset,batch_size=64,shuffle=False)
```

Network Class Definition

- creating the layers in init
- Defining the forward computation in forward function
- Backward function is useless thanks to autograd




```
class mlp(torch.nn.Module):
        def __init__(self,input_size,hidden_size_1,hidden_size_2):
            super(mlp,self).__init__()
            self.relu=torch.nn.ReLU()
            self.sigmoid=torch.nn.Sigmoid()
            self.fc1=torch.nn.Linear(input_size,hidden_size_1)
            self.fc2=torch.nn.Linear(hidden_size_1,hidden_size_2)
            self.fc3=torch.nn.Linear(hidden_size_2,1)
        def forward(self,x):
            x=self.fc1(x)
            x=self.relu(x)
            x=self.fc2(x)
            x=self.relu(x)
            x=self.fc3(x)
            x=self.sigmoid(x)
            return x
```

Instanciating the network

- 3 objects to instanciate/define
  - Network (call to __init__ of our class)
  - Loss function (often called criterion)
  - Optimizer (lr is learning rate)

```
model=mlp(2,50,25)
loss=torch.nn.BCELoss()
optimizer=torch.optim.SGD(model.parameters(),
                          lr=0.01)
```

Basic training

- get x/y from loader
- compute the forward + loss
- compute backward and apply optimizer step

```
nb_epoch=100
model.train() #activate autograd
for epoch in range(nb_epoch):
    for x_train,y_train in train_loader:
        optimizer.zero_grad()
        ypred=model.forward(x_train)
        batch_loss=loss(ypred.squeeze(),y_train)
        batch_loss.backward() #calculate derivatives
        optimizer.step() #apply the optimizer step
```

q

Test Network on test dataset

- To do regularly to check overfitting
- Sum all the batch loss and calculate mean

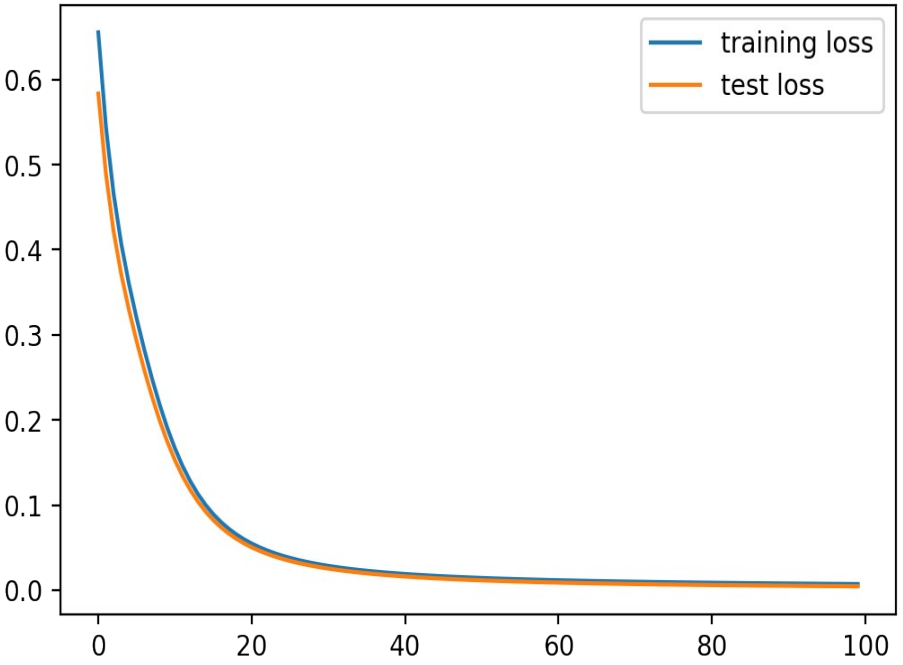
```
test_cum_loss=0.0
model.eval() #deactivate autograd
for x_test,y_test in test_loader:
    y_pred=model.forward(x_test)
    batch_loss=criterion(y_pred.squeeze(),y_test)
    test_cum_loss+=batch_loss.item()

test_loss=test_cum_loss/len(test_loader)
print("Test loss: %f"%(test_loss))
```

Plot loss function

- Very usefull to detect over-fitting
- Check that test loss does not increase

```
ptr_loss=[]
pte_loss=[]
pepoch=[]
for epoch in range(nb_epoch):
    ... learn model as seen ...
    ptr_loss.append(tr_loss)
    pte_loss.append(te_loss)
    pepoch.append(epoch)
```

```
import matplotlib.pyplot as plt

plt.plot(pepoch,ptr_loss,label="training loss")
plt.plot(pepoch,pte_loss,label="test loss")
plt.legend()
plt.show()
```

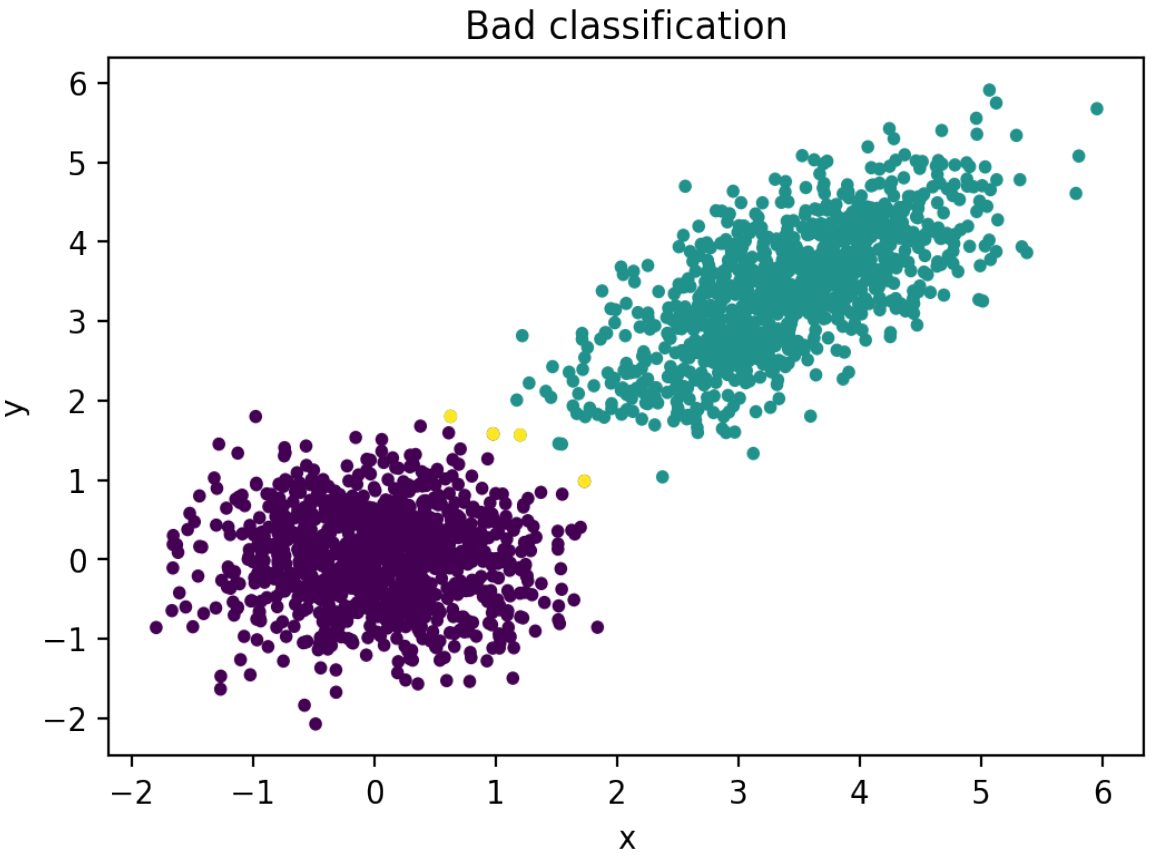


Plot classification errors

```
px=[]
py=[]
c=[]
for i in range(len(x)):
    px.append(x[i][0])
    py.append(x[i][1])
    c.append(y[i])
```

```
bad_class=0
for i in range(len(x)):
    y_pred=model.forward(tensor_x[i])
    y=y_pred.detach().numpy()[0]
    if abs(y-y[i])>0.4:
        px.append(x[i][0])
        py.append(x[i][1])
        c.append(2)
        bad_class+=1

print("nb of bad classif : %d"
        %(bad_class))
```

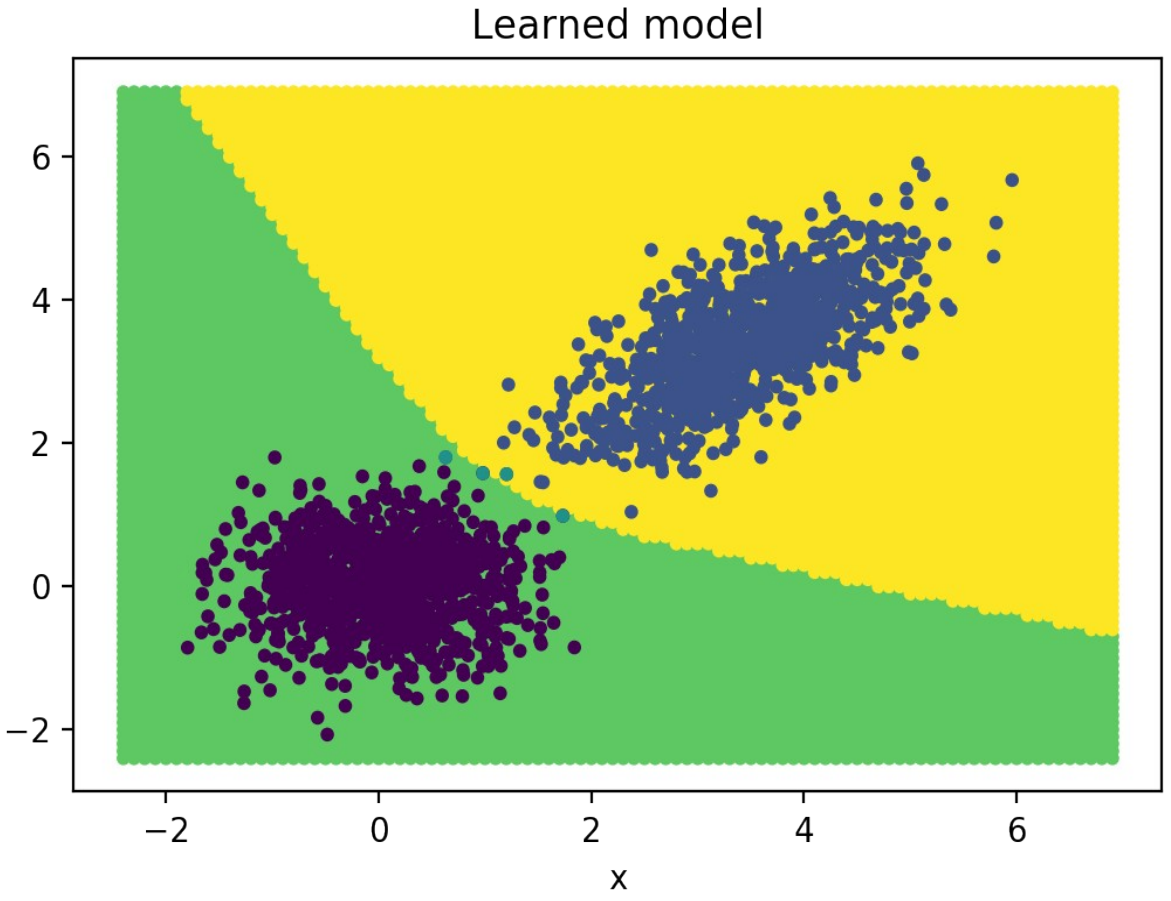
```
import matplotlib.pyplot as plt

plt.scatter(px,py,c=c)
plt.title("Data for torch intro")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Visualize learned model

```
mpx=[]
mpy=[]
mc=[]
model.eval()
#grid search
for i in np.arange(-2,7,0.1):
  for j in np.arange(-2,7,0.1):
    xl=[i,j]
    ty=model.forward(torch.tensor(xl))
    y=ty.detach().numpy()[0]
    mpx.append(i)
    mpy.append(j)
    if y<0.5:
      mc.append(3)
    else:
      mc.append(4)
#add data points
mpx+=px
mpy+=py
mc+=c
```

```
plt.scatter(px,py,c=c,s=10)
plt.title("Grid search")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



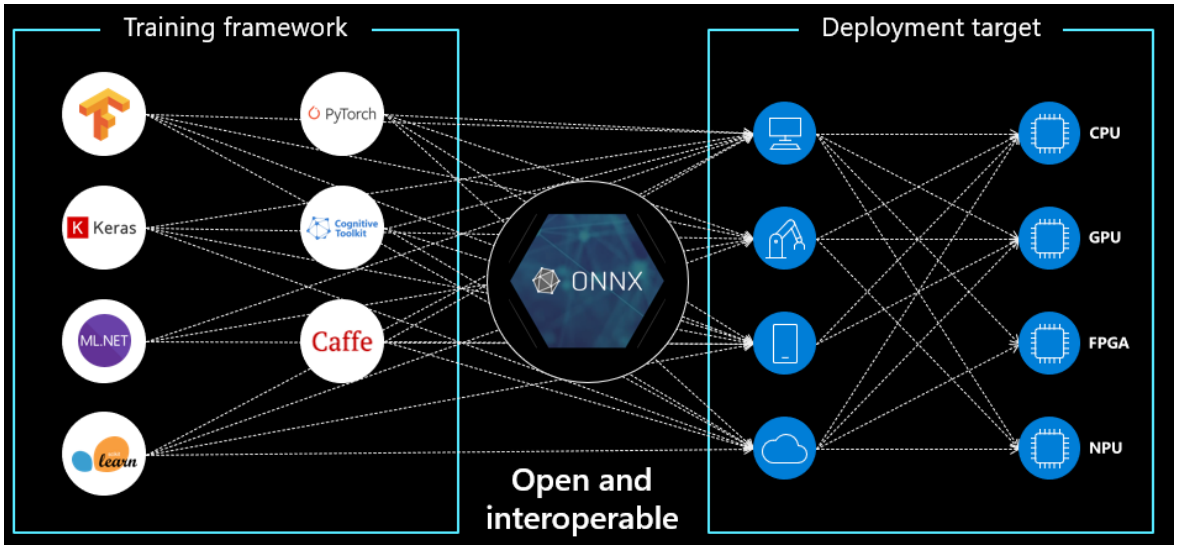
q

Disk Operations

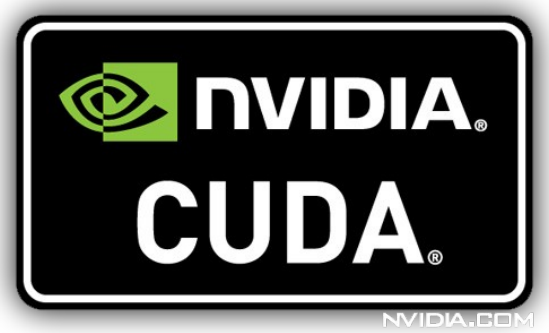
- Possibility to save the whole network to a file
- pickle like serialization
- Allows to re-train the network : just apply more epochs
- Allows to externalize test and forward uses
- Allows also ONNX format (Open Neural Network eXchange)
- Guarantee interoperability with other frameworks

```
torch.save(model,"mymodel.pth")
model=torch.load("mymodel.pth")
```

```
dummy_input=torch.randn(batchsize,
  channel,height,width)
torch.onnx.export(model,
  dummy_input,#indicate input shape
  "mymodel.onnx")
```



Cuda support



- Native support of Cuda
- Detection of computing device
- Data and model need to be explicitely migrated to device
- Possibility to declare tensors directly on a device

```
if torch.cuda.is_available():
    device=torch.device("cuda:0")
else:
    device=torch.device("cpu")
print("using device %s"%(device))
```

```
model.to(device)
for i,data in trainloader:
    inputs,labels=data
    inputs.to(device)
    labels.to(device)
```

```
t=torch.tensor([1.],device=device)
```

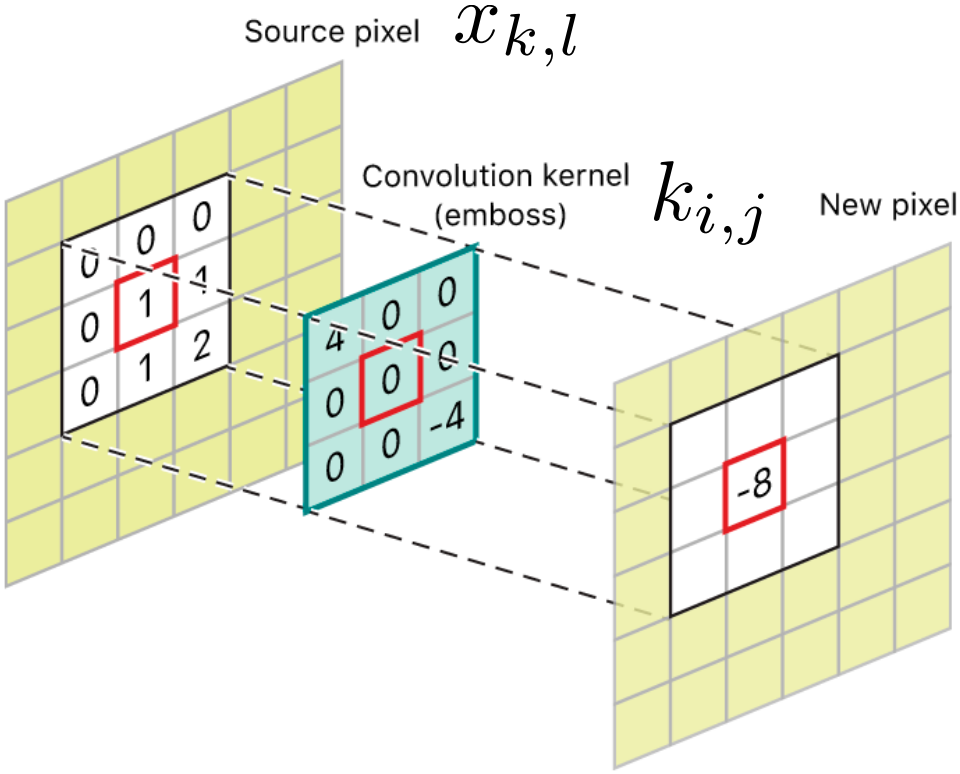
- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- **Convolutional networks**
- Auto-encoders
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

Convolutional Networks

- Tool to reduce the dimensionality of inputs
- Yann Lecun 1989
- Adapted to images

- inspired by the primary visual cortex
- Based on two operations
  - Convolution
  - Pooling

Convolution operation



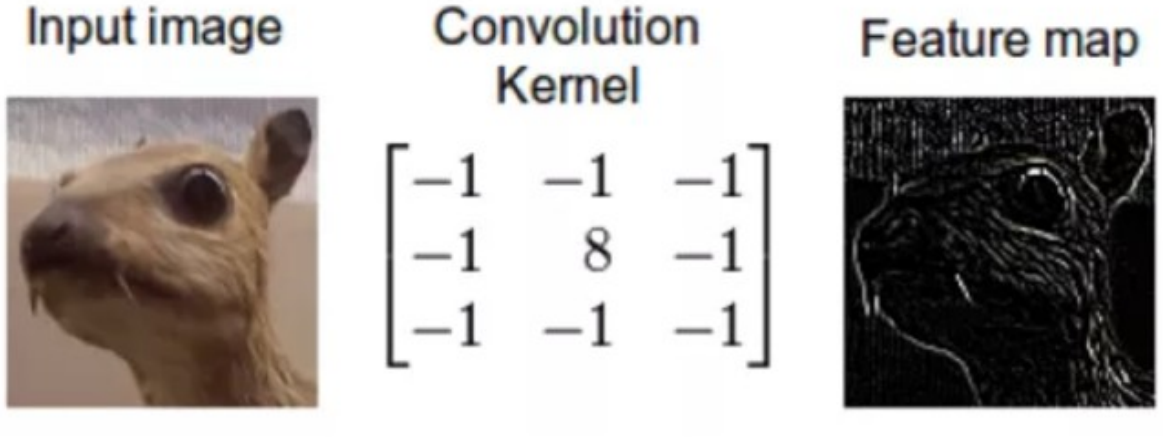
$$x'_{k,l} = \sum_{i,j} x_{k+i,l+j} * k_{i,j}$$

- Apply the same kernel to all the pixels of the image
- Kernel is **learnable**
- Filter is shared over the whole picture
- Idea : creating maps of features (one kernel per feature)

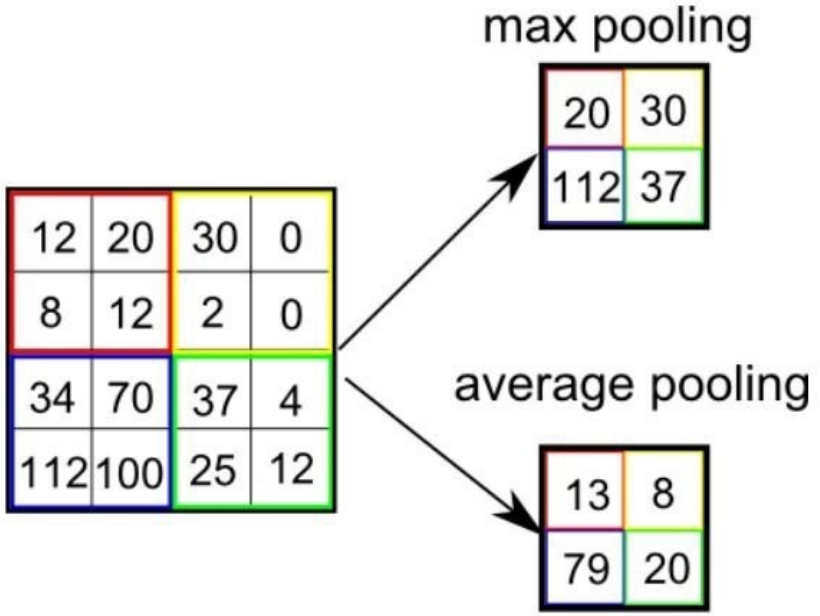
$$k_{i,j}^{t+1} = k_{ij}^{t} - \alpha \frac{\partial L}{\partial k_{ij}^{t}}$$

Effect of convolution kernel

- Create feature maps to describe the image pixel per pixel
- typically, increasing number of features at each step
- Every feature represent a filter on the picture or on a previous feature map

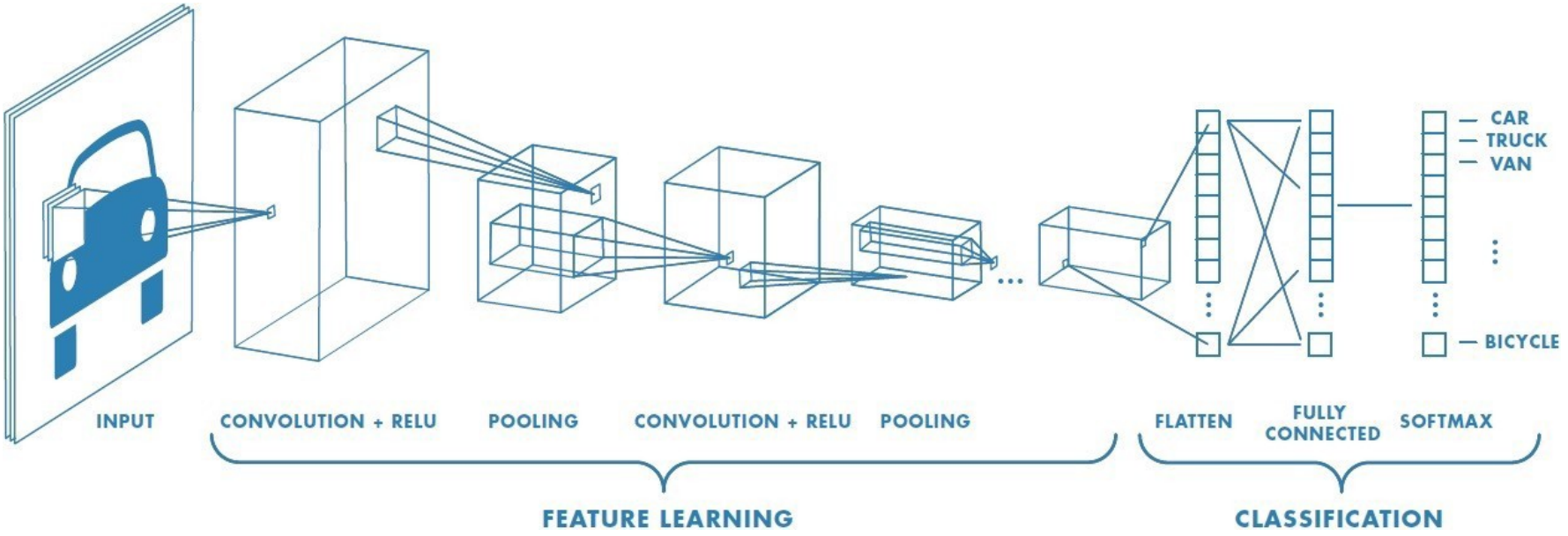


Pooling



- Reduce the dimensionality of the feature maps
- Move to higher level of abstraction
- Max pool is widely used

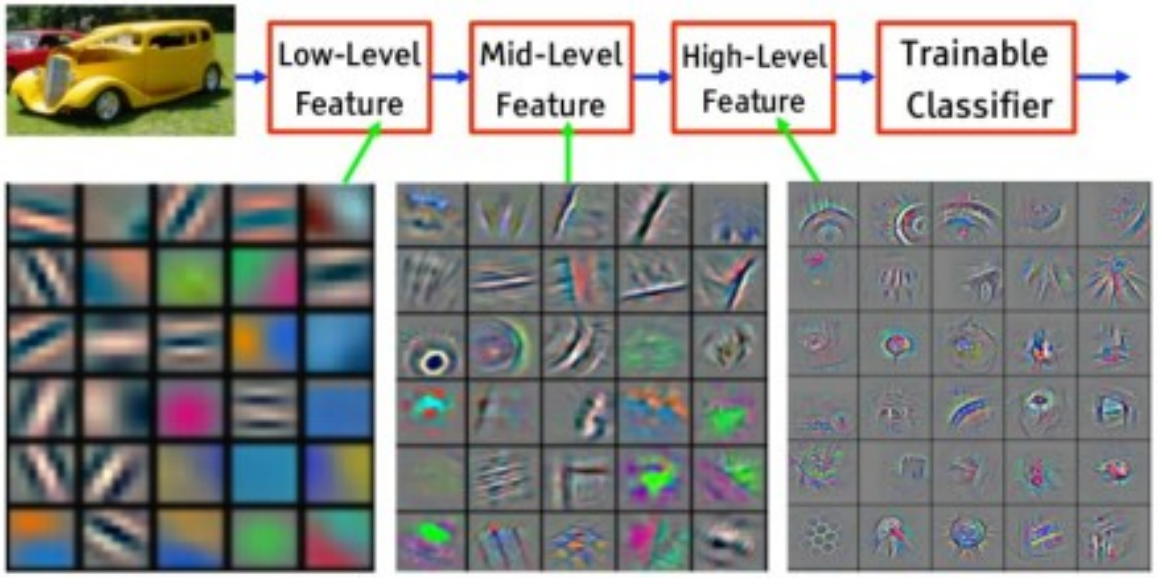
Convolutional network

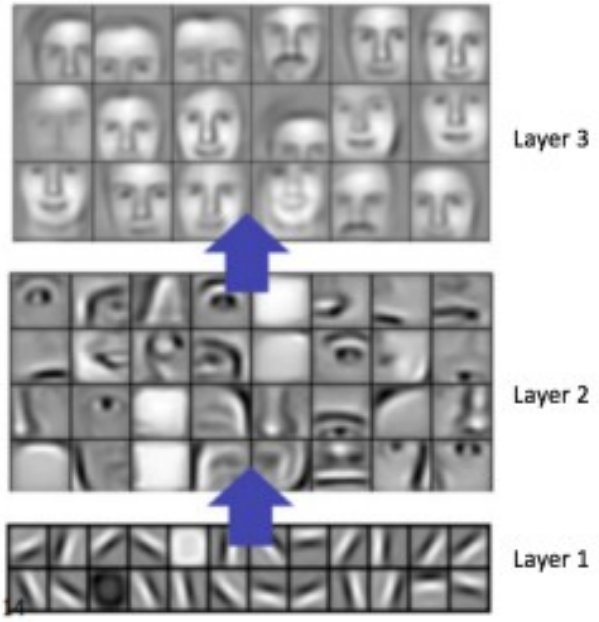


- Network structure :
  - Alternance of convolution & pooling
  - Flattering (sometimes called readout)
  - Multi-layer perceptron

How it works ?

- Feature maps aggregates more and more details to converges to high level recognition patterns
- on natural images, first kernels always correspond to Gabor filters
- pooling filters only important features (for the classification)
- Flattened high-level feature map is input for multi-layer perceptron



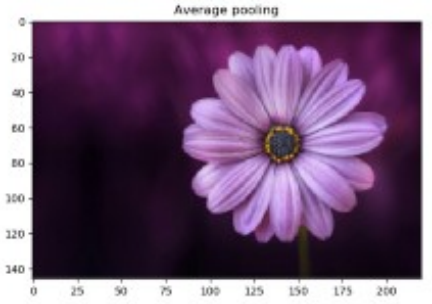


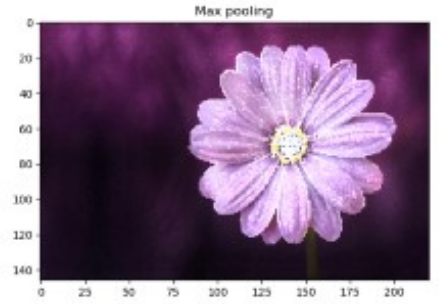
Why does it work ?

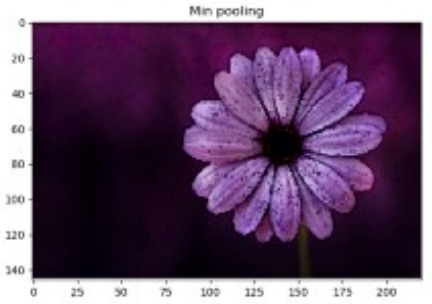
- The two operations derive naturally from local space Euclidean nature
  - Euclidean space → global translation-invariance (stationarity) → convolution
  - local translation-invariance → pooling

Cat

Cat

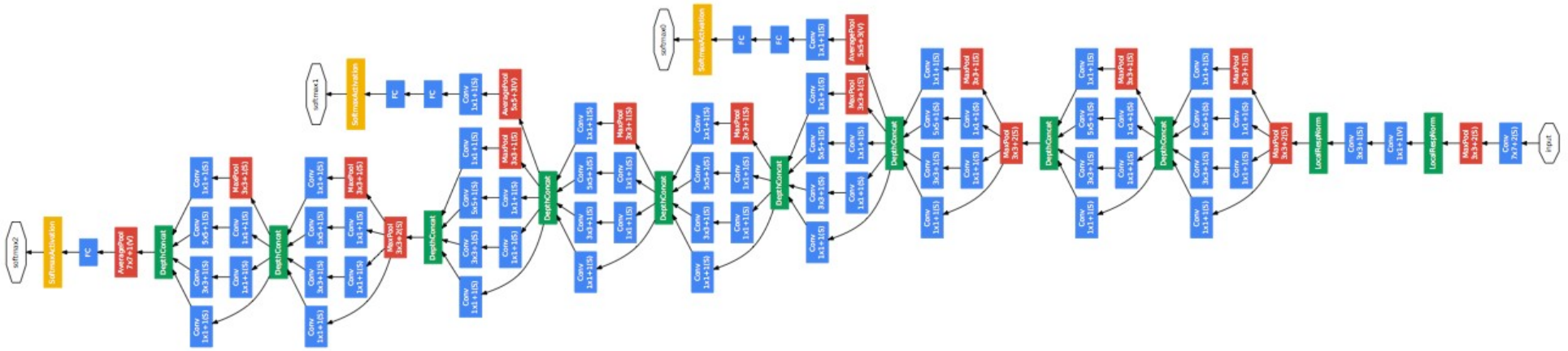






- Dream complexity
  - O(1) parameters per filter (independant of image size)
  - O(n) complexity in time per layer (n=#pixels)

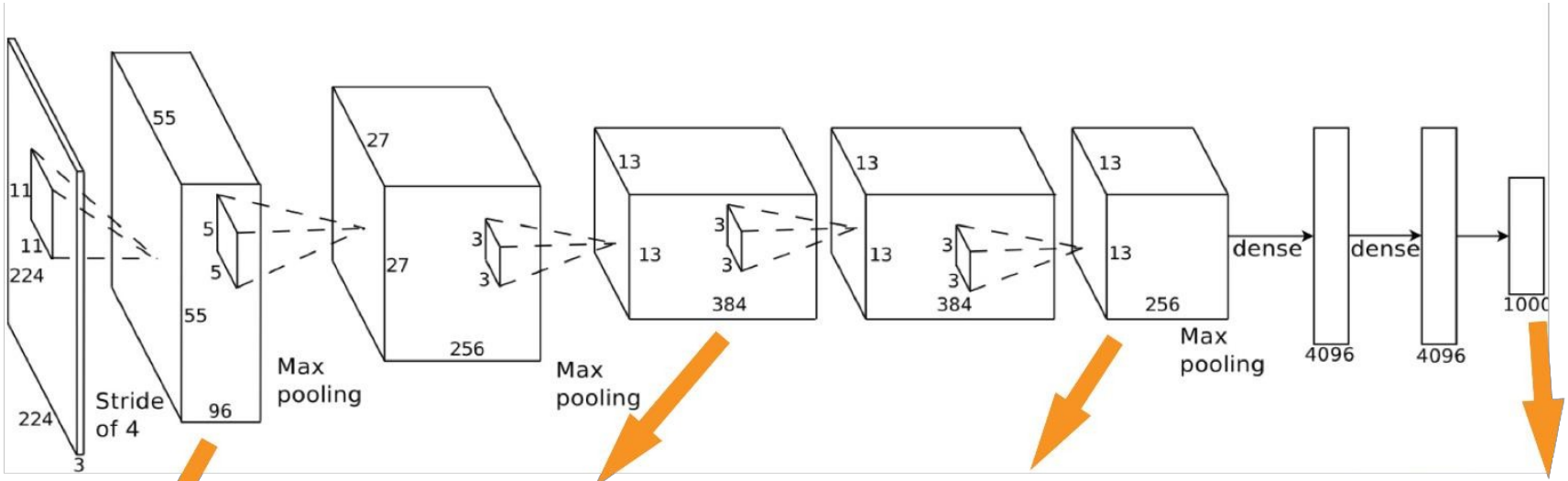
Conv. Neural network



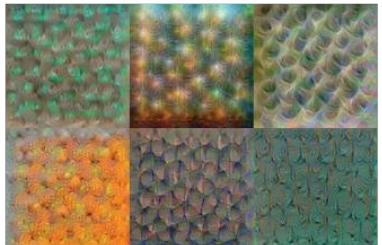
- Modern neural network → large assembly of building blocks : neurons, convolution kernels, poolers
- Training : optimizing all the parameters to minimize the loss function

AlexNet

- Designed by Alex Krizhevsky and his team
- 84.7 % accuracy on ImageNet challenge 2012



Conv 1: Edge+Blob



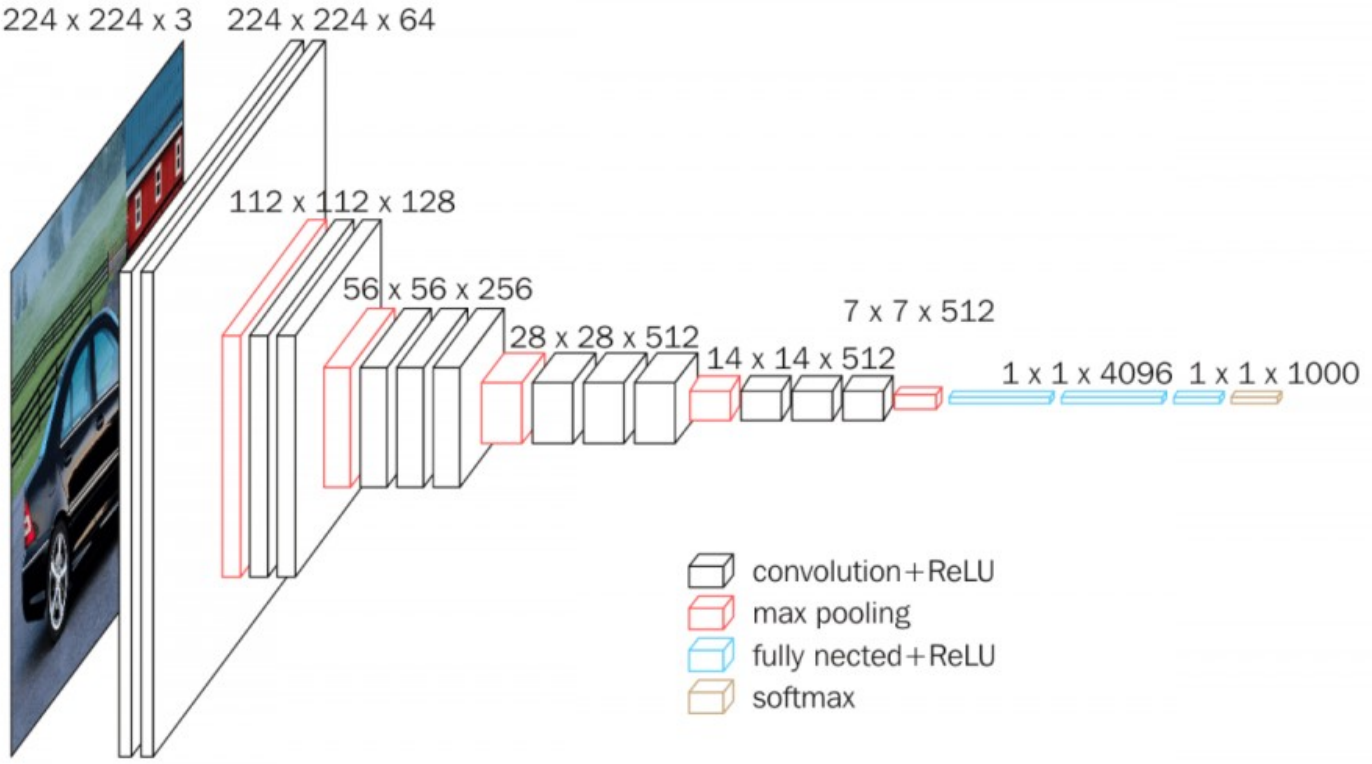
Conv 3: Texture

Conv 5: Object Parts

Fc8: Object Classes

VGG16

- 92.7% accuracy on ImageNet challenge 2014
- K. Simonyan and A. Zisserman, University of Oxford
- Impressive pooling gives very good results



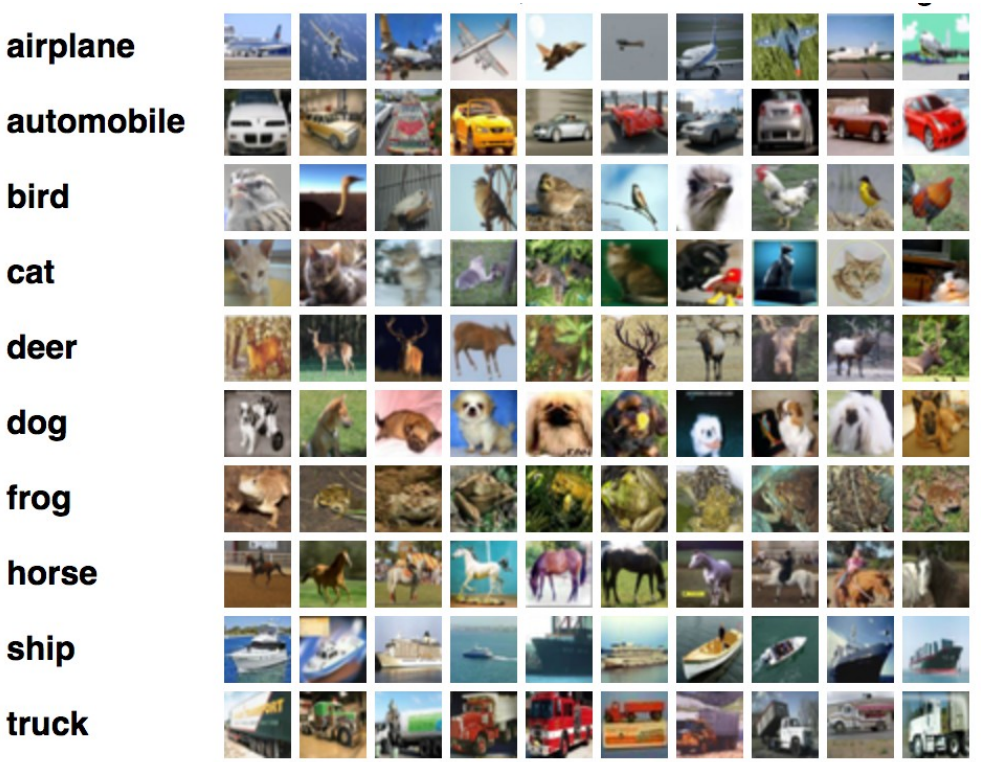
Pytorch implementation

- torch.nn includes all convolution and pooling functions
- torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
- torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

```
import torch.nn.functional as F
import torch.nn as nn

class convol(nn.Module):
        def __init__(self):
            super(convol,self).__init__()
            self.conv1=nn.Conv2d(3,6,5)#in_chans,out_chans,kern_size
            self.pool=nn.MaxPool2d(2,2)#kern_size,stride
            self.conv2=nn.Conv2d(6,16,5)
            self.fc1=nn.Linear(16*5*5,120)
            self.fc2=nn.Linear(120,84)
            self.fc3=nn.Linear(84,10)

        def forward(self,x):
            x=self.pool(F.relu(self.conv1(x))) #apply conv and pool
            x=self.pool(F.relu(self.conv2(x))) #apply conv and pool
            x=x.view(-1,16*5*5) #flatten the output
            x=F.relu(self.fc1(x))
            x=F.relu(self.fc2(x))
            x=self.fc3(x)                  #apply MLP
            return x
```

TorchVision : data

- Extra package including images datasets and generic transformations for vision
- Automatic download
- Very convenient




```
transform=transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_set=torchvision.datasets.CIFAR10(root="/tmp/Cifar10",
                  train=True,download=True,transform=transform)
train_loader=torch.utils.data.DataLoader(trainset,
                  batch_size=6,shuffle=True)
```

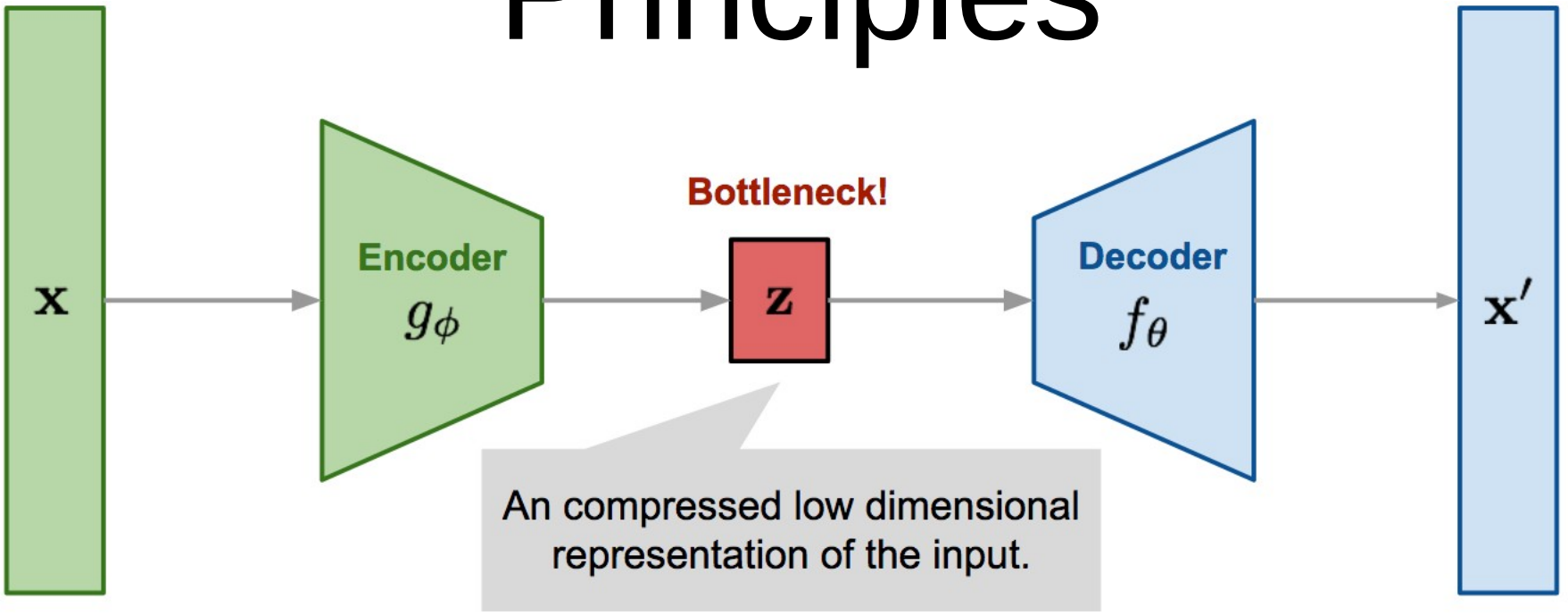
TorchVision : networks

- All famous network are already implemented and trained
- Allows Transfer Learning

```
import torchvision.models as models
resnet18=models.resnet18(pretrained=True)
alexnet=models.alexnet(pretrained=True)
squeezenet=models.squeezenet1_0(pretrained=True)
vgg16=models.vgg16(pretrained=True)
densenet=models.densenet161(pretrained=True)
inception=models.inception_v3(pretrained=True)
googlenet=models.googlenet(pretrained=True)
shufflenet=models.shufflenet_v2_x1_0(pretrained=True)
mobilenet=models.mobilenet_v2(pretrained=True)
resnext50_32x4d=models.resnext50_32x4d(pretrained=True)
wide_resnet50_2=models.wide_resnet50_2(pretrained=True)
mnasnet=models.mnasnet1_0(pretrained=True)
```

- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- **Auto-encoders**
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

Principles

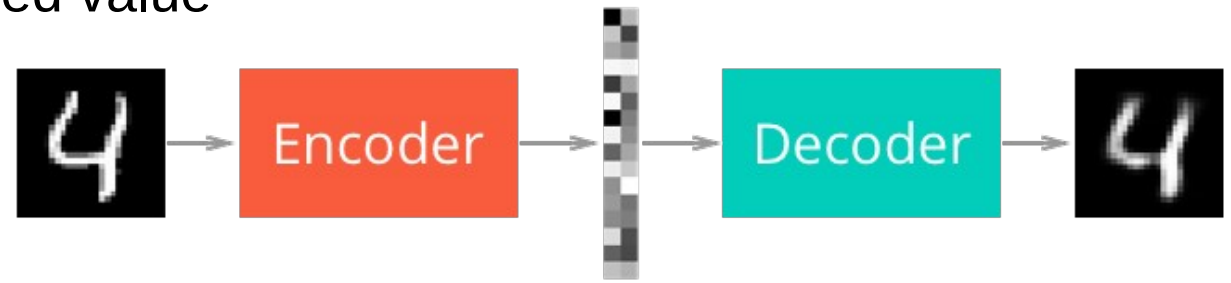


- Non-supervised learning : no label requested
- Encoder build a compressed representation of data : latent space representation
- Decoder reconstruct the data from latent space representation
- Loss function : difference between input and output

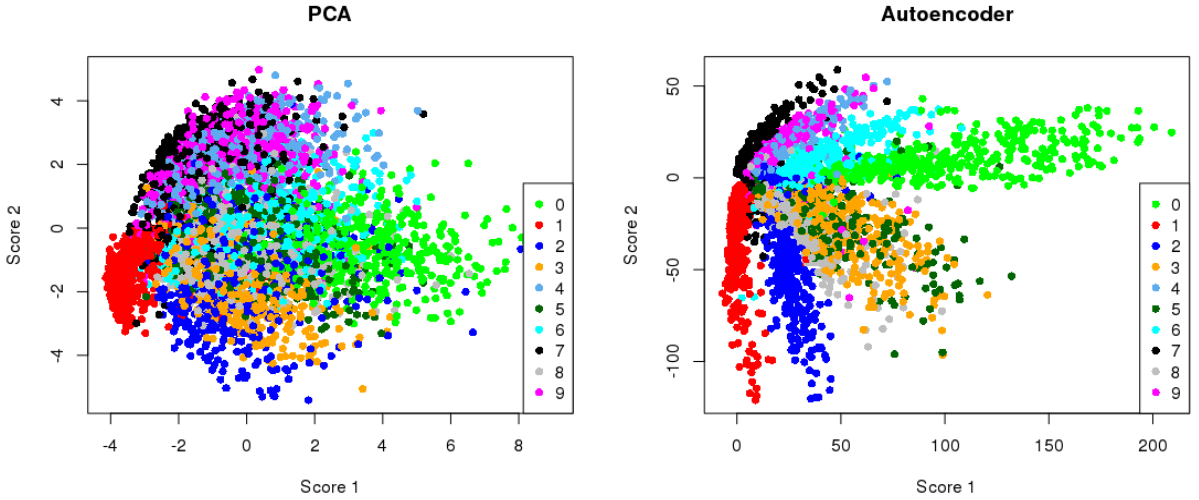
$$L(x) = (x - \hat{x})^2$$

Appli.1: Dimension reduction

- Lossy compression : latent space is smaller than input space → used as compressed value



- Dimensionality reduction of complex problems
  - Non linear principal component analysis (PCA) system
  - Ease clustering of events

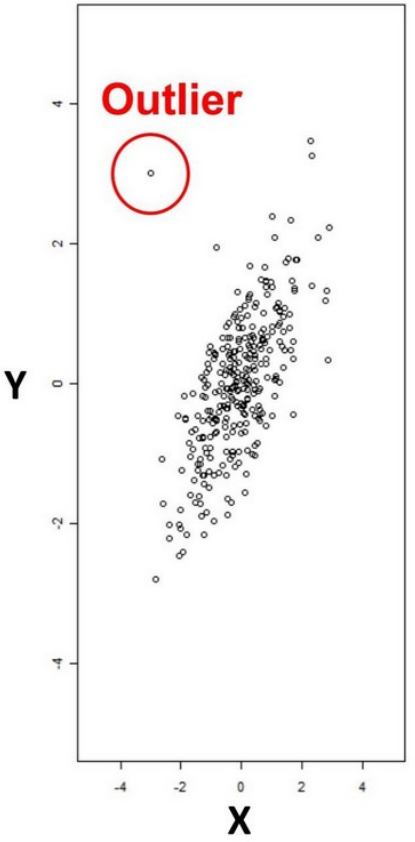


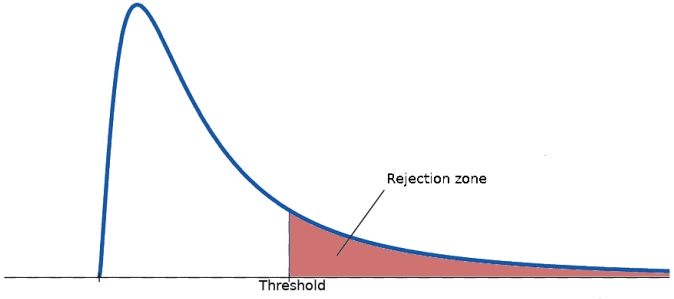
Appli.2 : Outlier Detection

- Anomaly detection
- Principle : outlier reconstruction is worse because they fit less in the generative model
- Implementation
  - Calculate the quality of reconstruction

$$L(x) = (x - \hat{x})^2$$

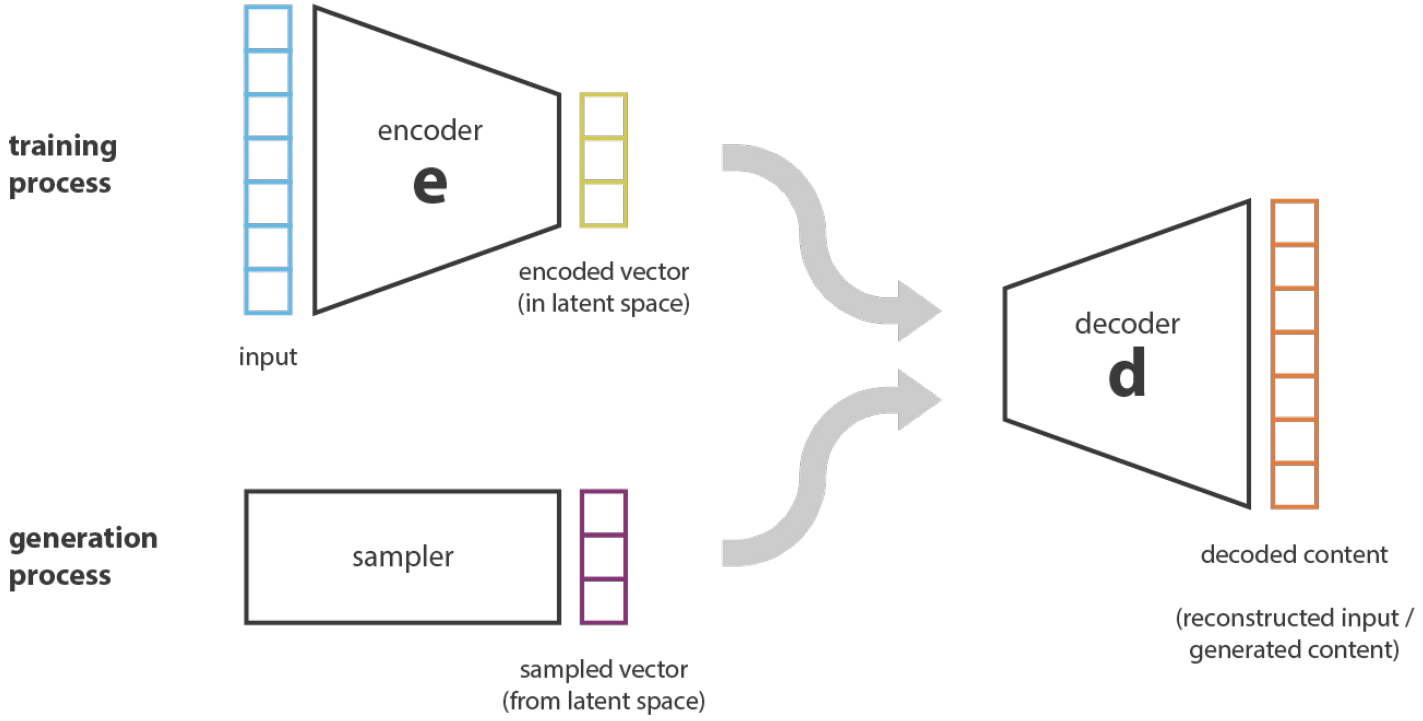
  - Compare the value to the statistic of training
  - Use a threshold to decide the outlier nature





Appli.3 : Fast Data Generation

- Generate data directly from latent space
- Generate vector of coordinate in latent space
  → apply the decoder
- Fastsim (GEANT 4)

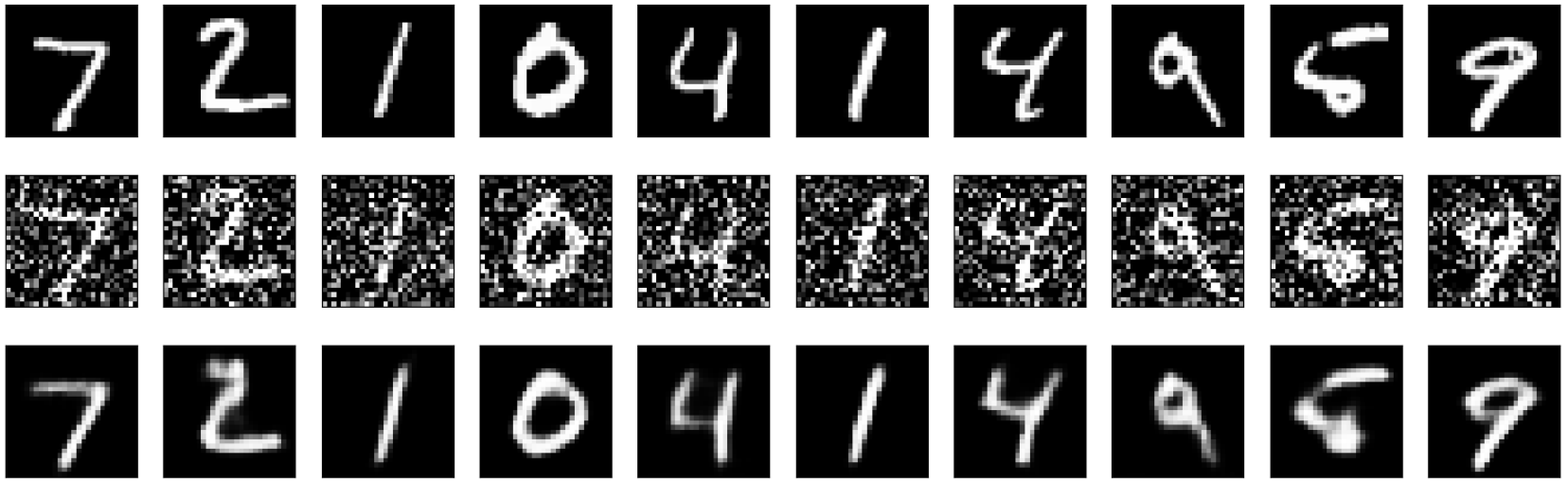


Appli. 4 : Denoising

- Noise the input $\tilde{x} = noise(x)$
- Calculate the decoded version $\hat{x}$
- Train on this decoded version and original input

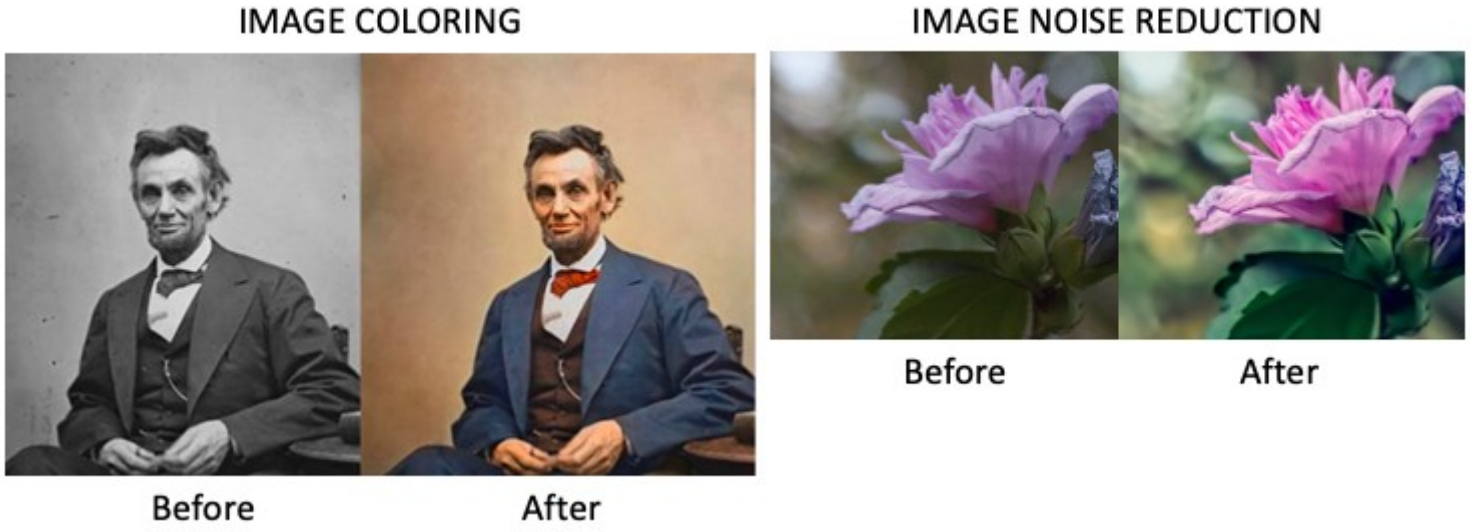
$$L(x, \hat{x})$$

- The AE learn how to remove the noise



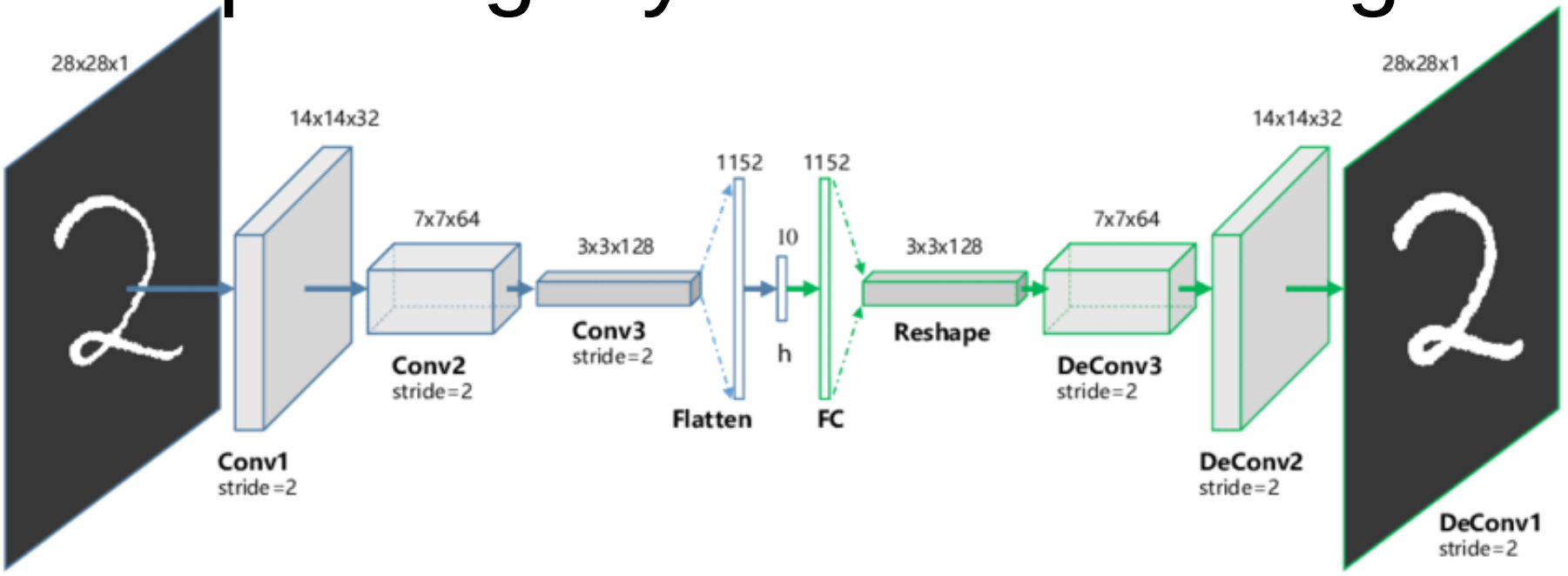
Appli. 5 : Coloring and color denoising

- Same idea as denoising
- The network learns how to generate the colors
- The colored output is trained vs the white and black'ified version

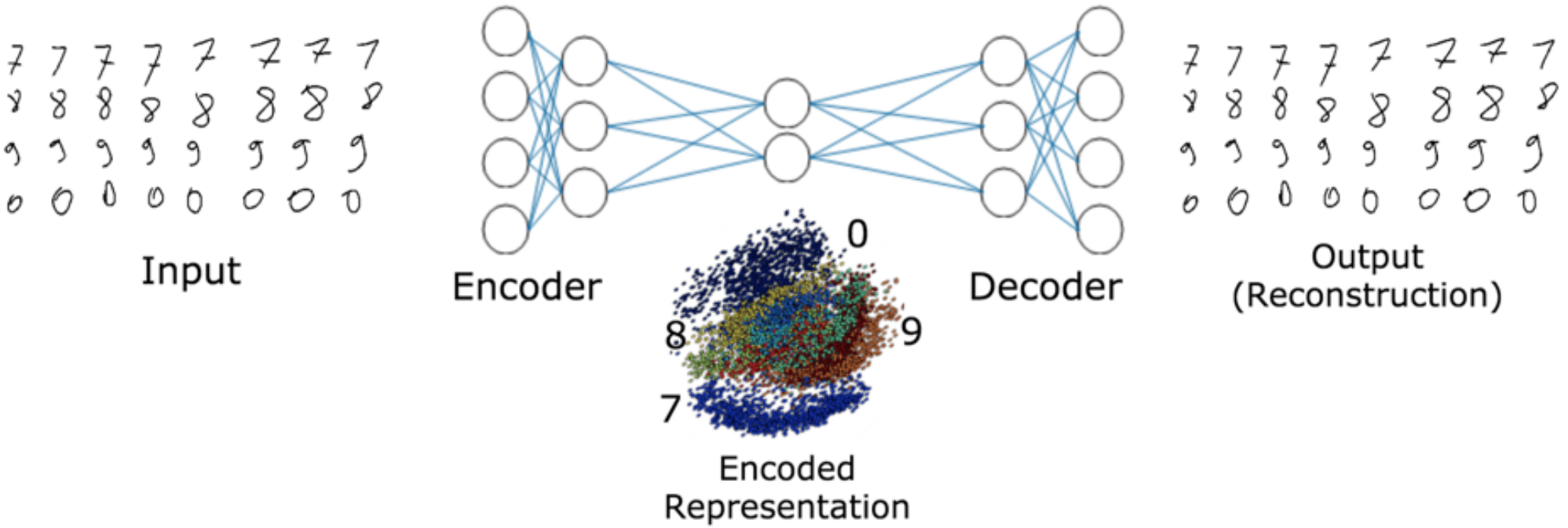


Convolutive AE

- Based on convolution and pooling layers for encoding
- Based on deconvolution and unpooling layers for decoding



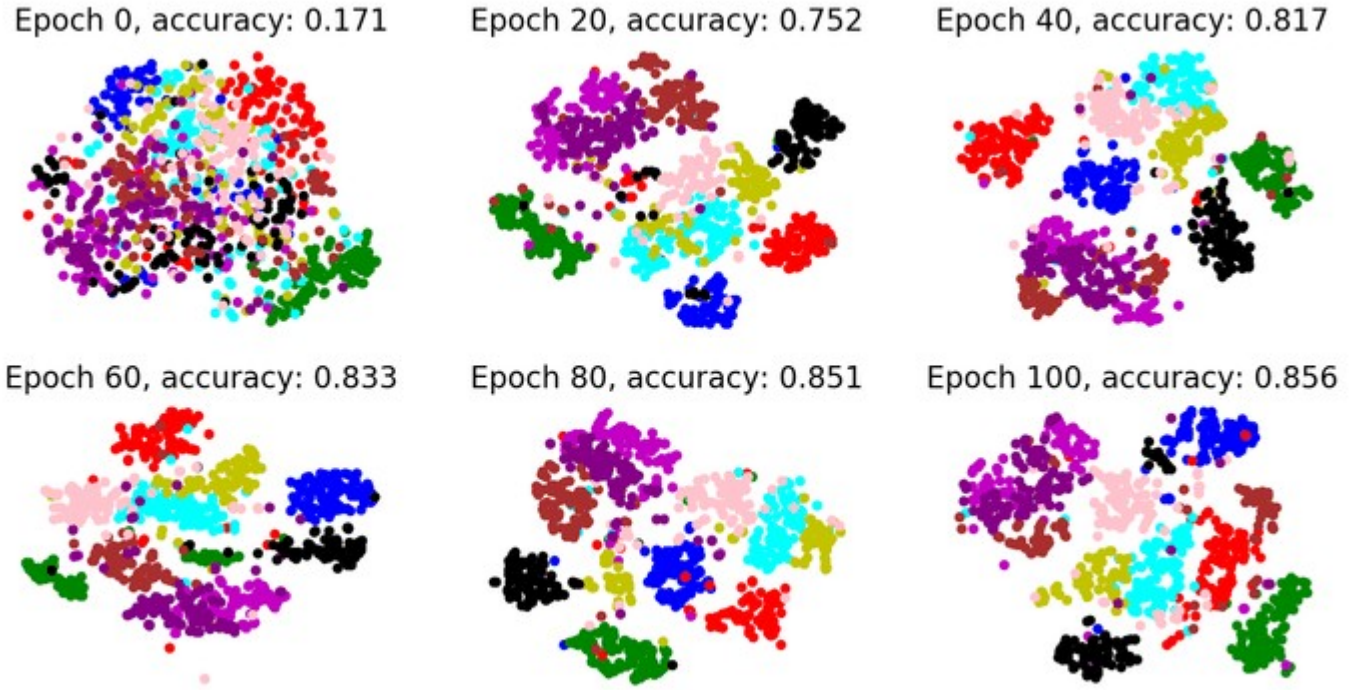
Latent space



- The latent space is an internal representation of the data
- Vector of real numbers
- Non linear version of PCA

Latent Space Evolution

- In a perfect AE, latent space representation evolves during training
  - Grouping identical outputs
  - Reducing the space between the clusters

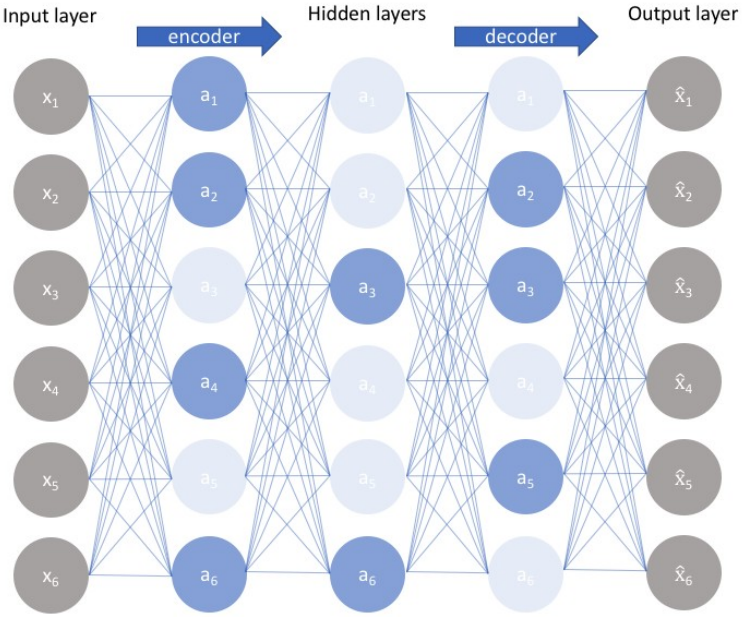


Overfitting problems

- Very simple loss function induces overfitting
- Autoencoder learns nothing

  → only some kind of identity
- Need regularization
  - Penalization of complexity

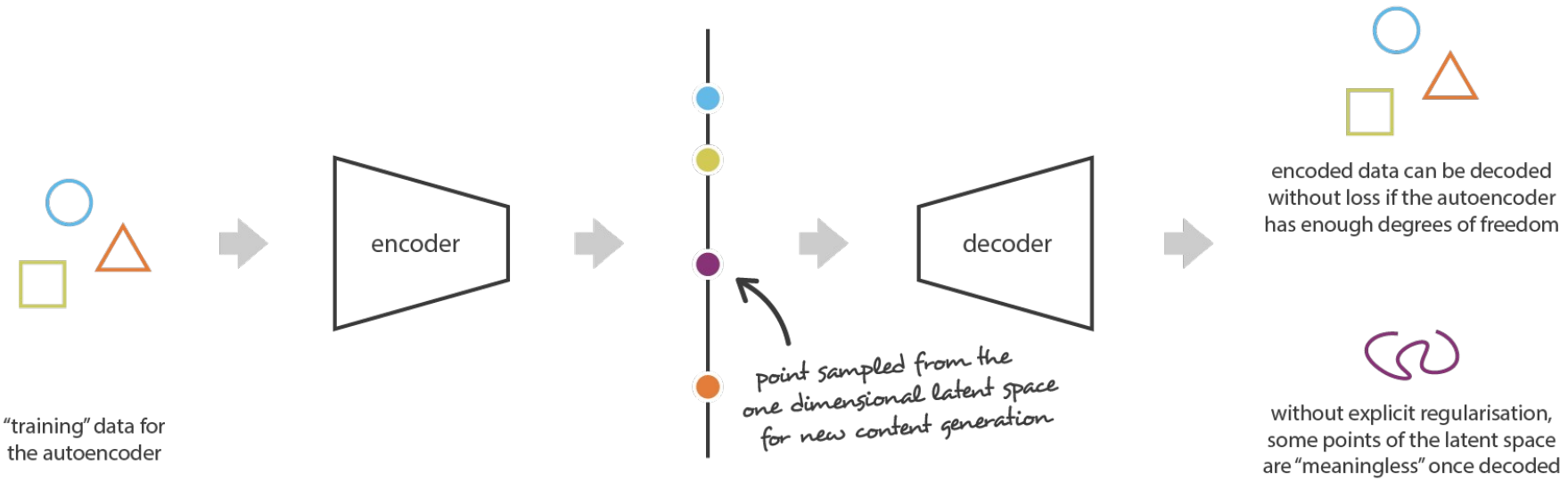
$$L_\theta(x) = (x - \hat{x})^2 + \lambda.P(\theta)$$



- Possibility to use L1 → sparse autoencoders
  - Less useful neuron weights are reduced to zero

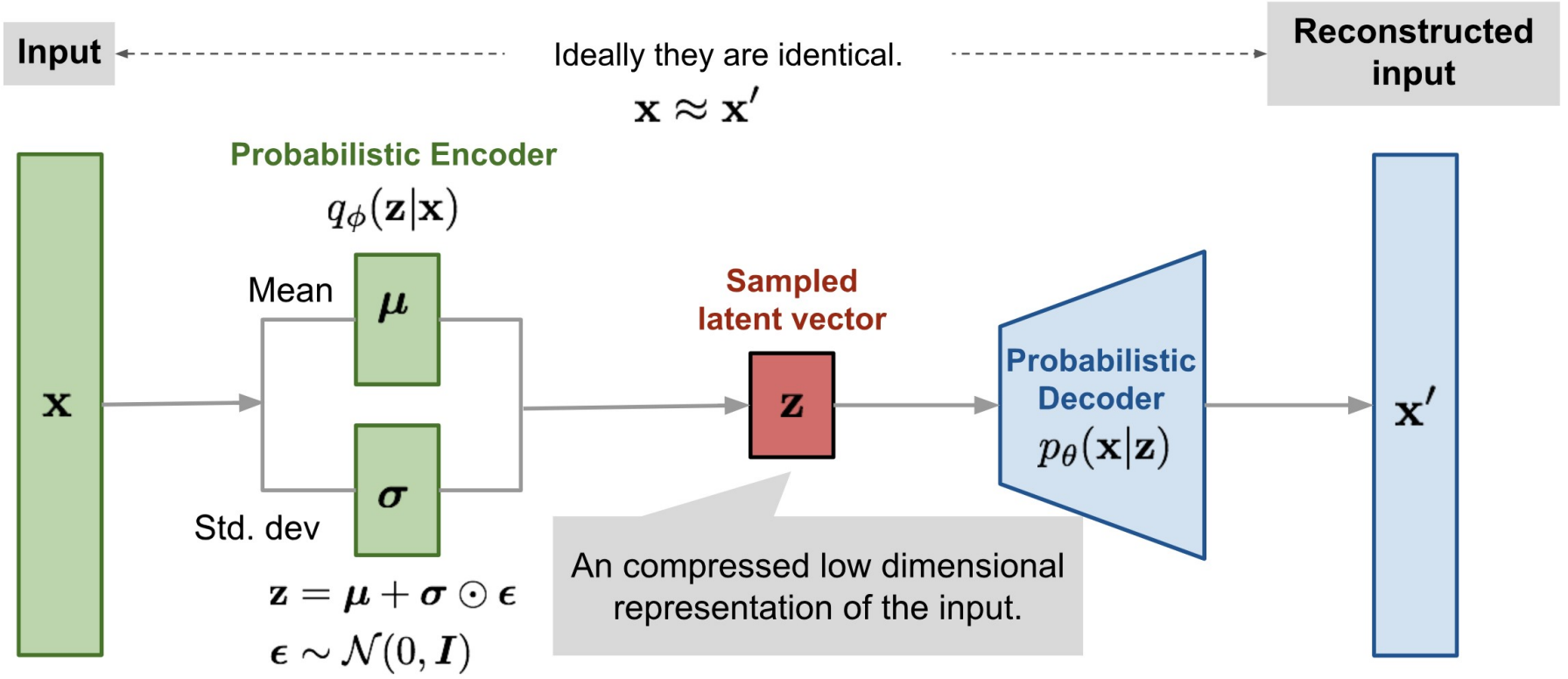
Completeness problems

- Loss function does not induce organization of latent space
- Latent space has to be compact for generative tasks



Variational autoencoders

- Regularized version of autoencoder
  - Avoid overfitting by regularization
  - Guarantee latent space completeness
- instead of encoding an input as a single point, we encode it as a distribution over the latent space



AE vs VAE

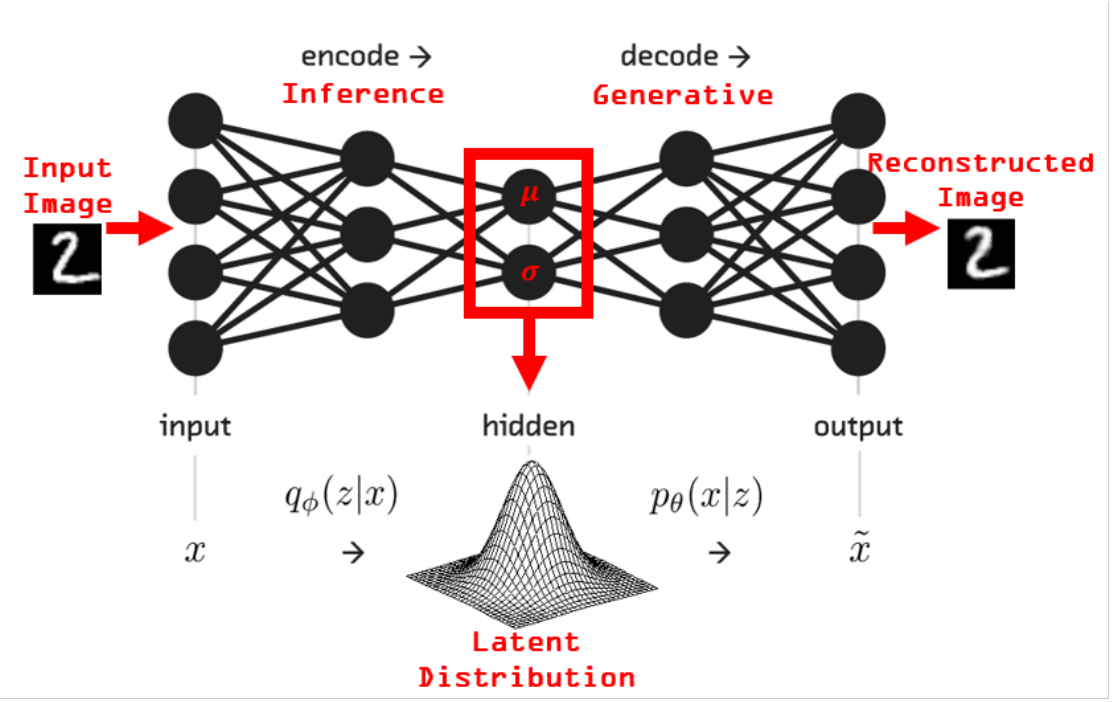
simple autoencoders

input **x** → encoding → latent representation **z = e(x)** → decoding → input reconstruction **d(z)**

P(z|x) is chosen to be a Gaussian to ease computations

variational autoencoders

input **x** → encoding → latent distribution **p(z|x)** → sampling → sampled latent representation **z ~ p(z|x)** → decoding → input reconstruction **d(z)**



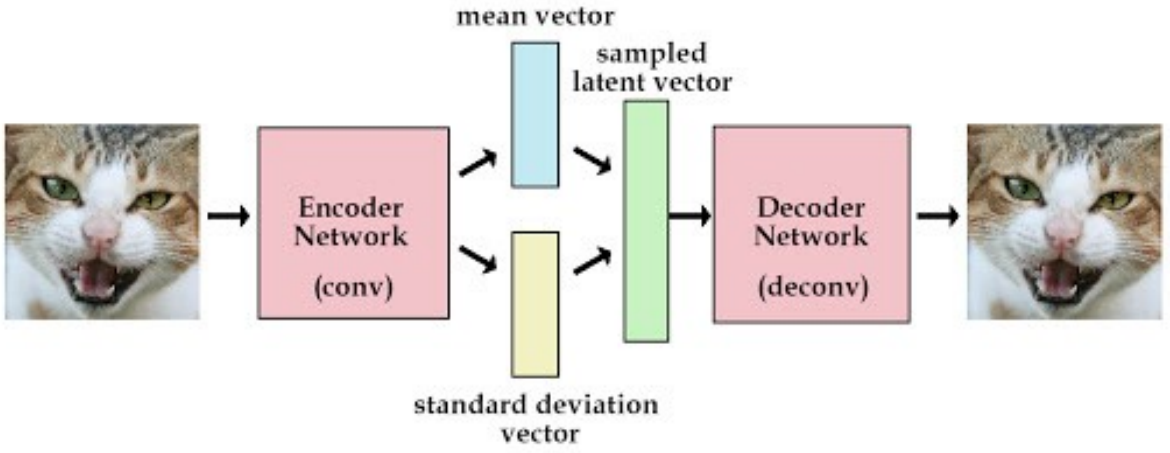
VAE training procedure

1)The input is encoded as a distribution over the latent space

2)A point from the latent space is sampled from that distribution

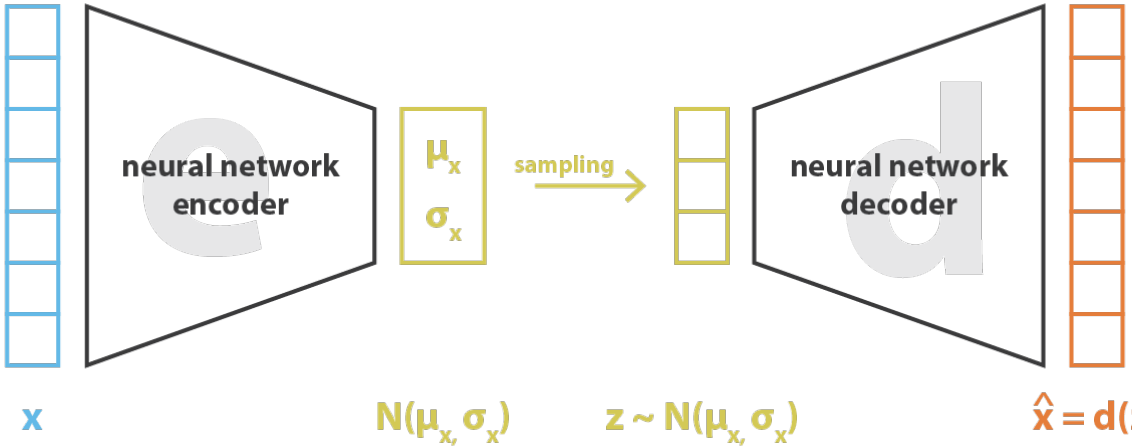
3)The sampled point is decoded and the reconstruction error can be computed

4)The reconstruction error is backpropagated through the network



VAE regularization

- VAE is regularized by a penalization of the model complexity
  - Kulback-Leibler divergence
  - Distance from the distribution to the centred and reduced normal distribution

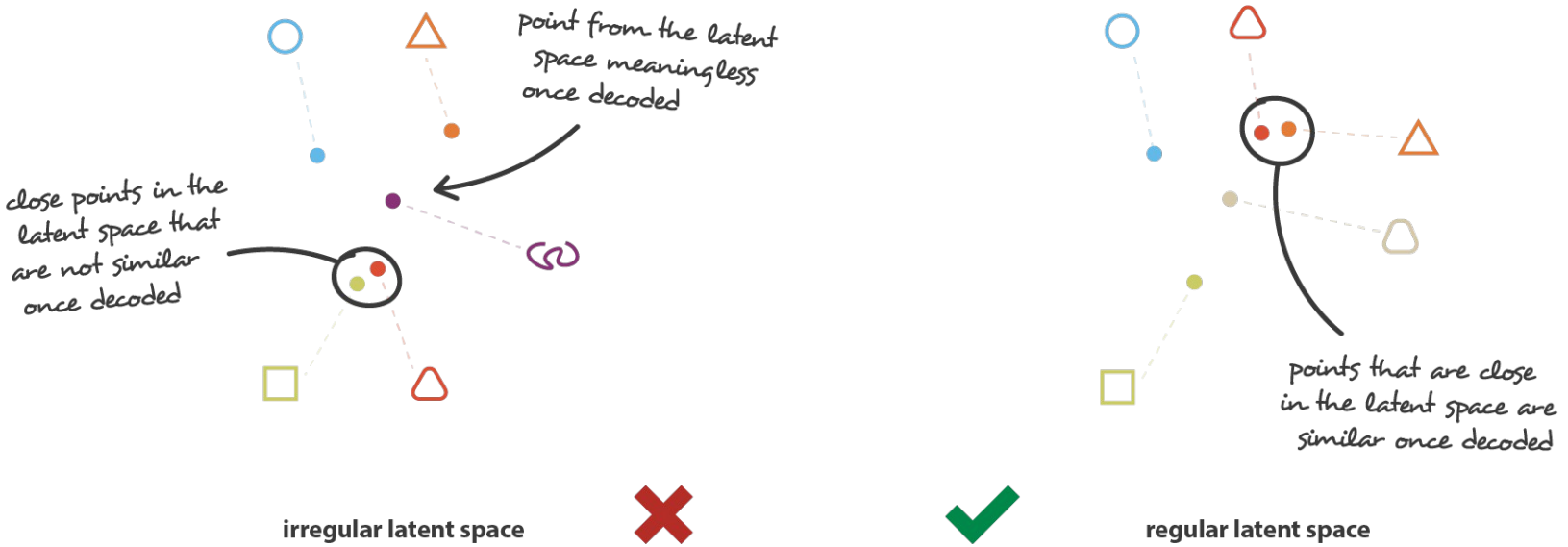


Could be multiplied by a constant to reduce the number of used latent dimensions : disantangled VAE

$$\text{loss} = \|x - \hat{x}\|^2 + KL[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + KL[N(\mu_x, \sigma_x), N(0, I)]$$

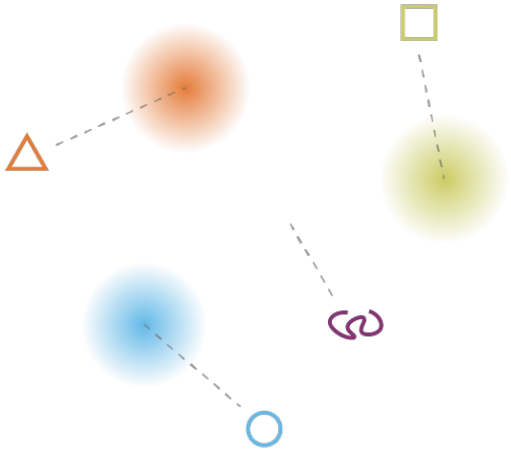
Obtained Continuity

- Regular latent space
- Two close point in latent space should not give two completely different decoded content

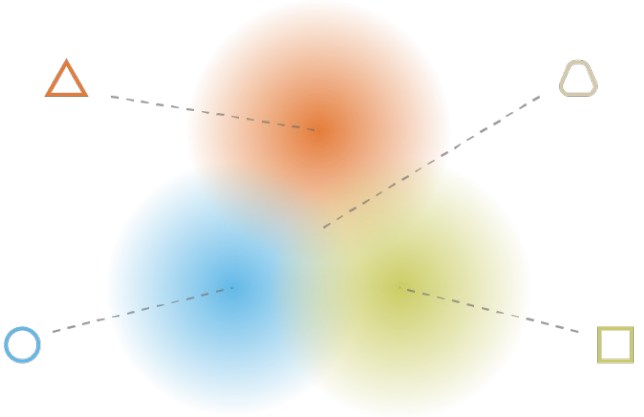


Obtained Completeness

- Any sampled point in the latent space should give a meaningful decoded content
- Tends to create a “gradient” over the information encoded in the latent space (half-half representation)



what can happen without regularisation



what we want to obtain with regularisation

Pytorch implementation

- Convolutional autoencoder
- Encoder is composed of three operations
  - Convolution
  - Flattening
  - 2 layers Perceptron
- Decoder
  - 2 layers perceptron
  - Unflattening
  - Deconvolution

Zeiner & al, Deconvolutional networks, 2021

from https://ai.plainenglish.io/denoising-autoencoder-in-pytorch-on-mnist-dataset-a76b8824e57e

full code and results on https://llrbox.in2p3.fr/owncloud/index.php/s/LWRBCaaaYT81PoV

Encoder implementation

- For future applications, the encoder and the decoder are implemented separately
- Both in the same file autoencoder.py

```
class Encoder(nn.Module):

    def __init__(self,encoded_space_dim,fc2_input_dim):
        super().__init__()
        self.encoder_cnn=nn.Sequential(
            nn.Conv2d(1,8,3,stride=2,padding=1), nn.ReLU(True),
            nn.Conv2d(8,16,3,stride=2,padding=1),
            nn.BatchNorm2d(16), nn.ReLU(True),
            nn.Conv2d(16,32,3,stride=2,padding=0), nn.ReLU(True)
        )
        self.flatten=nn.Flatten(start_dim=1)
        self.encoder_lin=nn.Sequential(
            nn.Linear(3 * 3 * 32,128), nn.ReLU(True),
            nn.Linear(128,encoded_space_dim)
        )
    def forward(self,x):
        x=self.encoder_cnn(x)
        x=self.flatten(x)
        x=self.encoder_lin(x)
        return x
```

Decoder implementation

```
class Decoder(nn.Module):
    def __init__(self,encoded_space_dim,fc2_input_dim):
        super().__init__()
        self.decoder_lin=nn.Sequential(
            nn.Linear(encoded_space_dim,128), nn.ReLU(True),
            nn.Linear(128,3 * 3 * 32), nn.ReLU(True)
        )
        self.unflatten=nn.Unflatten(dim=1
                                    ,unflattened_size=(32,3,3))
        self.decoder_conv=nn.Sequential(
            nn.ConvTranspose2d(32,16,3,stride=2,output_padding=0),
            nn.BatchNorm2d(16), nn.ReLU(True),
            nn.ConvTranspose2d(16,8,3,stride=2,
            padding=1,output_padding=1),
            nn.BatchNorm2d(8), nn.ReLU(True),
            nn.ConvTranspose2d(8,1,3,stride=2,padding=1
                               ,output_padding=1)
        )
    def forward(self,x):
        x=self.decoder_lin(x)
        x=self.unflatten(x)
        x=self.decoder_conv(x)
        x=torch.sigmoid(x)
        return x
```

Creating dataset

```
import torchvision as tv
from torch.utils.data import DataLoader,random_split
ddir='dataset'
train_dset=tv.datasets.MNIST(ddir,train=True,download=True)
test_dset =tv.datasets.MNIST(ddir,train=False,download=True)
transform=tv.transforms.Compose([
tv.transforms.ToTensor(),
])
train_dset.transform=transform
test_dset.transform=transform
m=len(train_dset)
train_data,test_data=random_split(train_dset,
                                  [int(m-m*0.1),int(m*0.1)])
torch.save(train_dset,"train_dataset.pth")
torch.save(test_dset,"test_dataset.pth")

bs=256 #batch size
train_loader=DataLoader(train_dset,batch_size=bs)
test_loader=DataLoader(test_dset,batch_size=bs)
torch.save(train_loader,"train_loader.pth")
torch.save(test_loader,"test_loader.pth")
```

Instanciate

```
loss_fn=torch.nn.MSELoss()

torch.manual_seed(0)

#size of latent space
d=4

#define the network
encoder=Encoder(encoded_space_dim=d,fc2_input_dim=128)
decoder=Decoder(encoded_space_dim=d,fc2_input_dim=128)

#define the optimizer
lr= 0.001 #learning rate

params_to_optimize=[
    {'params': encoder.parameters()},
    {'params': decoder.parameters()}
]

optim=torch.optim.Adam(params_to_optimize,
                       lr=lr,weight_decay=1e-05)
```

Create noise

- The noise will be added to the inputs to make autoencoder to learn the denoising

```
def add_noise(inputs,noise_factor=0.3):
     noisy=inputs+torch.randn_like(inputs)*noise_factor
     noisy=torch.clip(noisy,0.,1.)
     return noisy
```

Train function

- Train a full epoch

```
def train_epoch_den(encoder,decoder,device,dataloader,
                    loss_fn,optimizer,noise_factor=0.3):
    encoder.train()
    decoder.train()
    train_loss=[]

    for image_batch,_ in dataloader: # "_" ignore labels
        image_noisy=add_noise(image_batch,noise_factor)
        encoded_data=encoder(image_noisy)
        decoded_data=decoder(encoded_data)
        loss=loss_fn(decoded_data,image_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.detach().cpu().numpy())

    return np.mean(train_loss)
```

Test function

- Test on the test dataloader

```
def test_epoch_den(encoder,decoder,device,
                   dataloader,loss_fn,noise_factor=0.3):
    encoder.eval()
    decoder.eval()
    with torch.no_grad(): # No need to track the gradients
        conc_out=[]
        conc_label=[]
        for image_batch,_ in dataloader:
            image_noisy=add_noise(image_batch,noise_factor)
            encoded_data=encoder(image_noisy)
            decoded_data=decoder(encoded_data)
            conc_out.append(decoded_data)
            conc_label.append(image_batch)
        conc_out=torch.cat(conc_out)
        conc_label=torch.cat(conc_label)
        test_loss=loss_fn(conc_out,conc_label)
    return test_loss.data
```

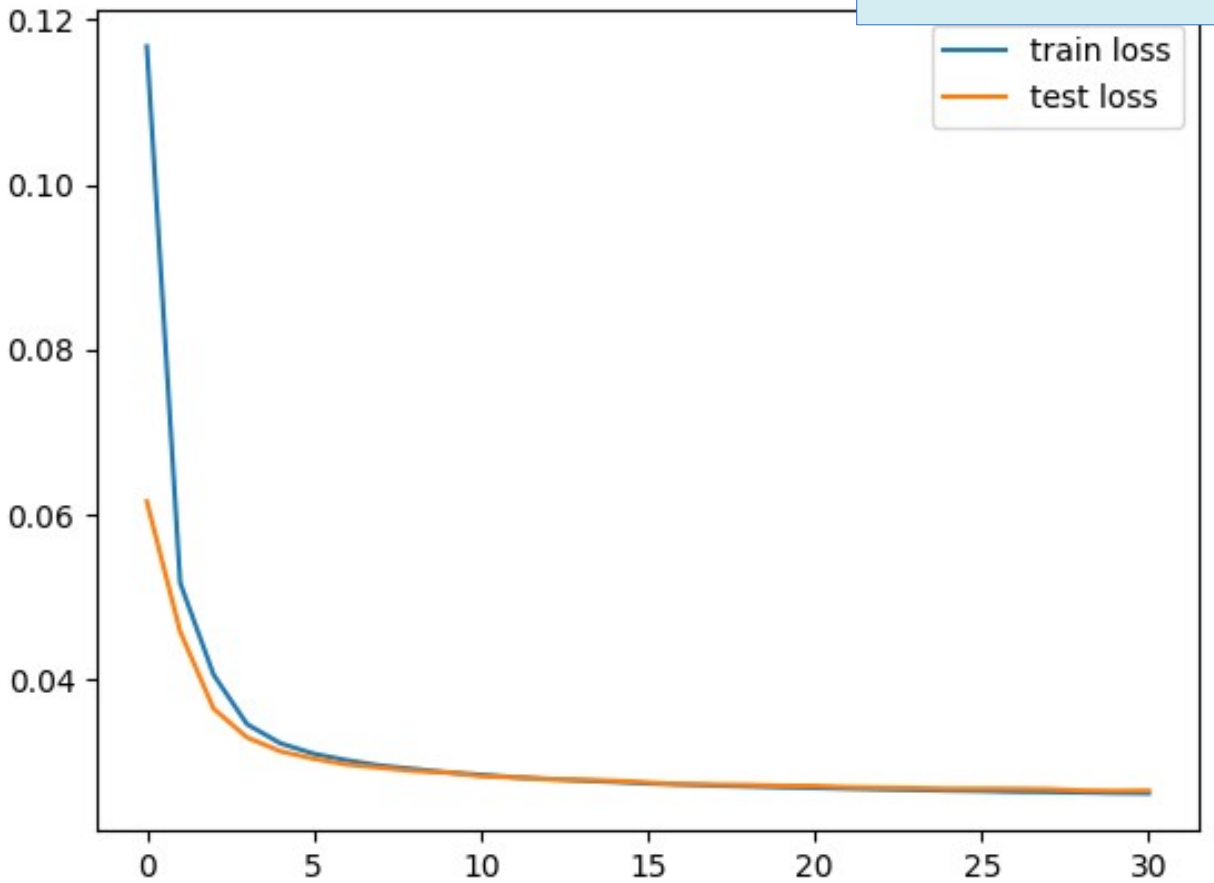
Training

```
noise_factor=0.3
nb_epochs=31
history_da={'train_loss':[],'test_loss':[]}

for epoch in range(nb_epochs):

    train_loss=train_epoch_den(encoder=encoder,
        decoder=decoder,device=device,
        dataloader=train_loader,
        loss_fn=loss_fn,optimizer=optim
        ,noise_factor=noise_factor)

    test_loss=test_epoch_den(encoder=encoder,
        decoder=decoder,device=device,
        dataloader=test_loader,
        loss_fn=loss_fn,noise_factor=noise_factor)

    history_da['train_loss'].append(train_loss)
    history_da['test_loss'].append(test_loss)

torch.save(encoder,"encoder.pth")
torch.save(decoder,"decoder.pth")
```

Plot Loss

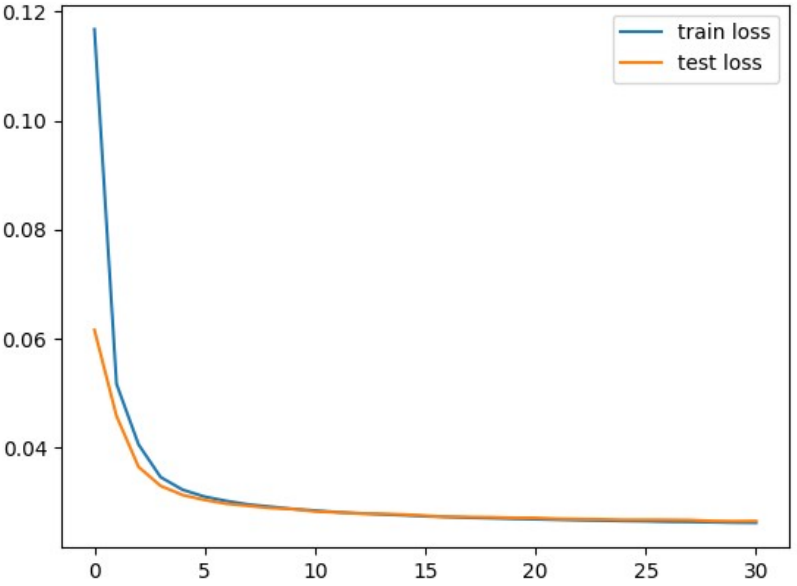
- We can see 4 things on this curve

```
plt.plot(list(range(nb_epochs))
    ,history_da['train_loss']
    ,label="train loss")
plt.plot(list(range(nb_epochs))
    ,history_da['test_loss']
    ,label="test loss")
plt.legend()
plt.show()
```

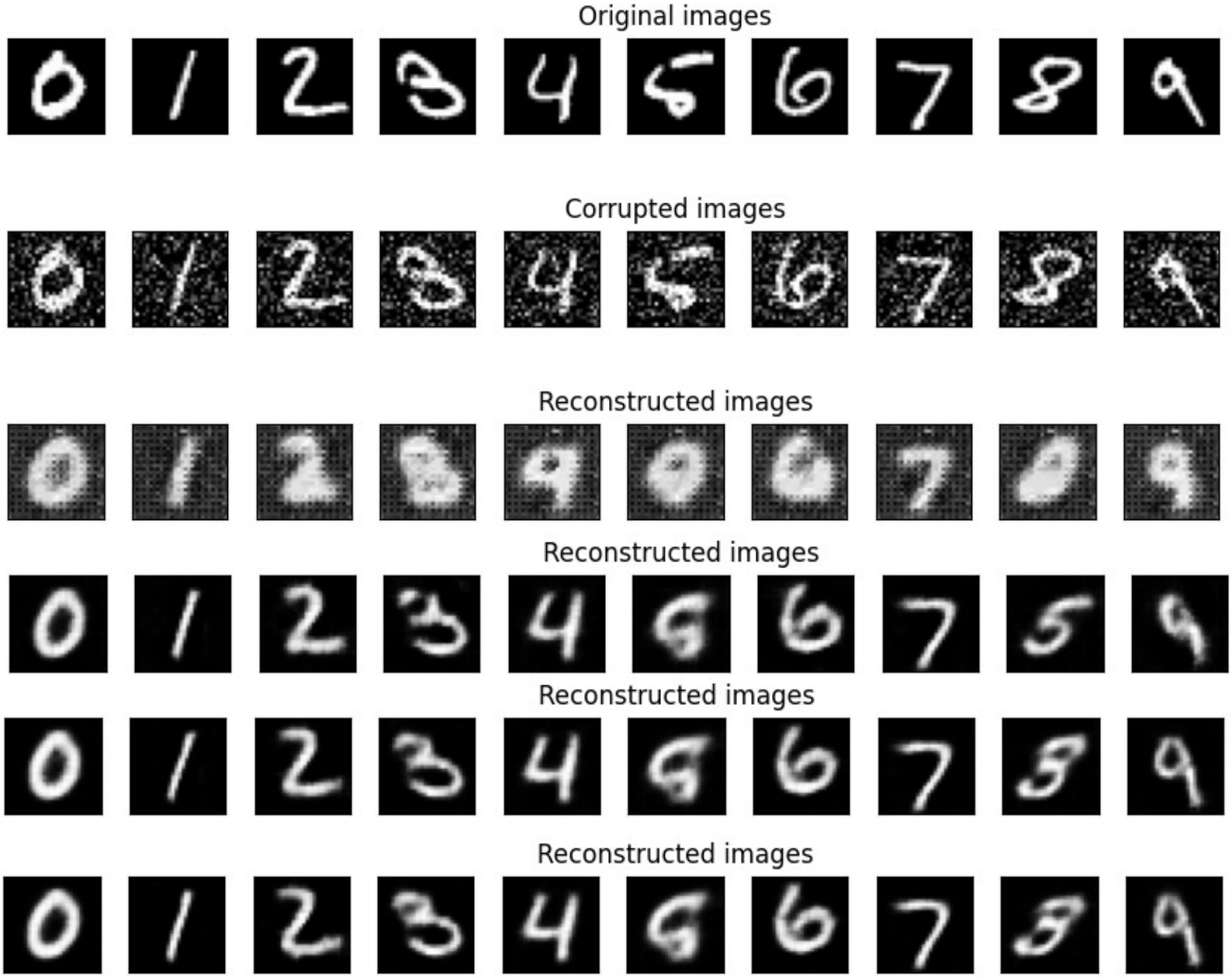


Loss curve

- The training is going well (exponential shape)
- MSE is not bad 2 %
- The training is not completely finished (slope not null at end), needs more epochs
- The network generalizes well (test curve=train curve at the end) – no overfitting



Results



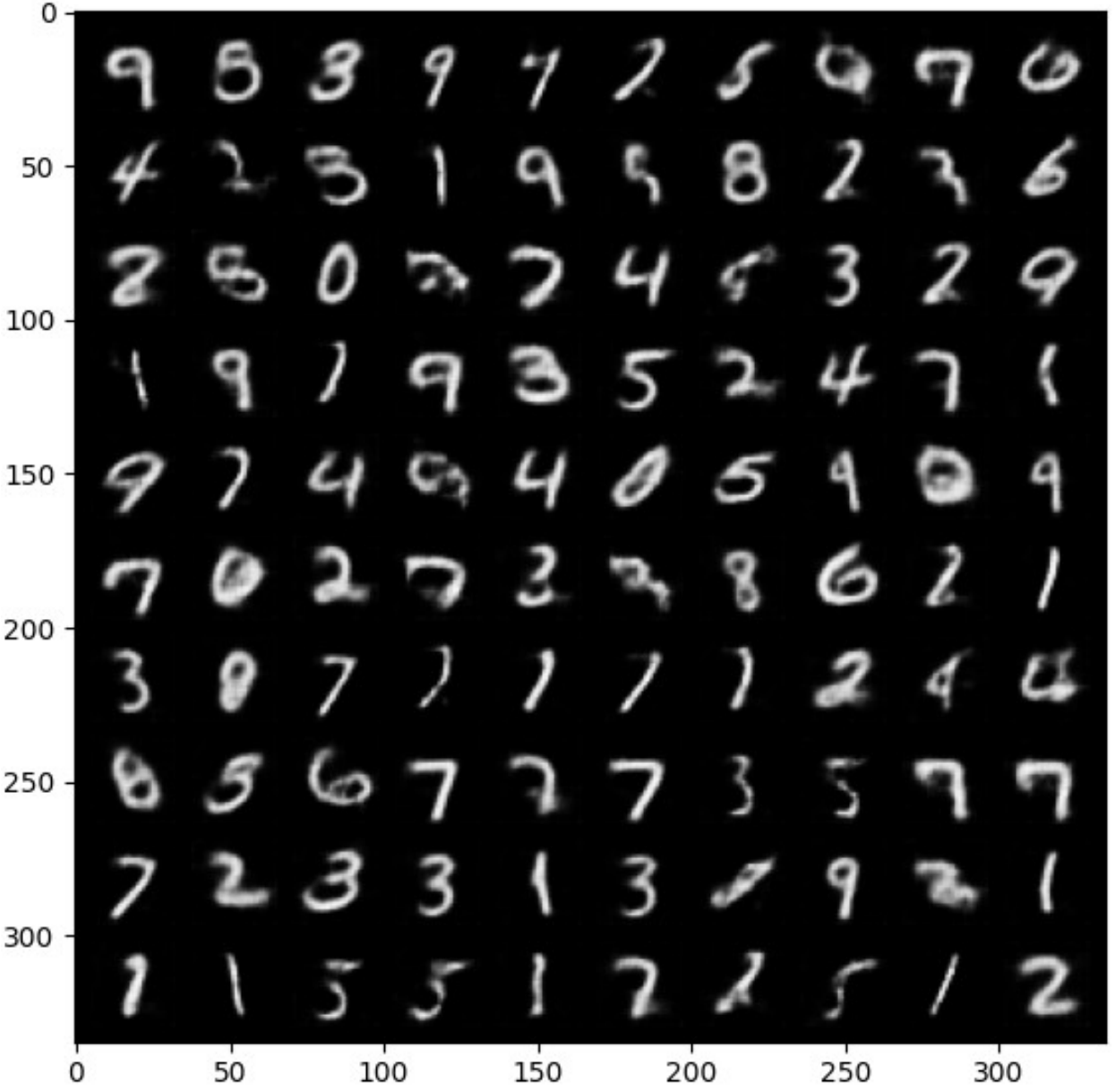
epoch 1

epoch 11

epoch 21

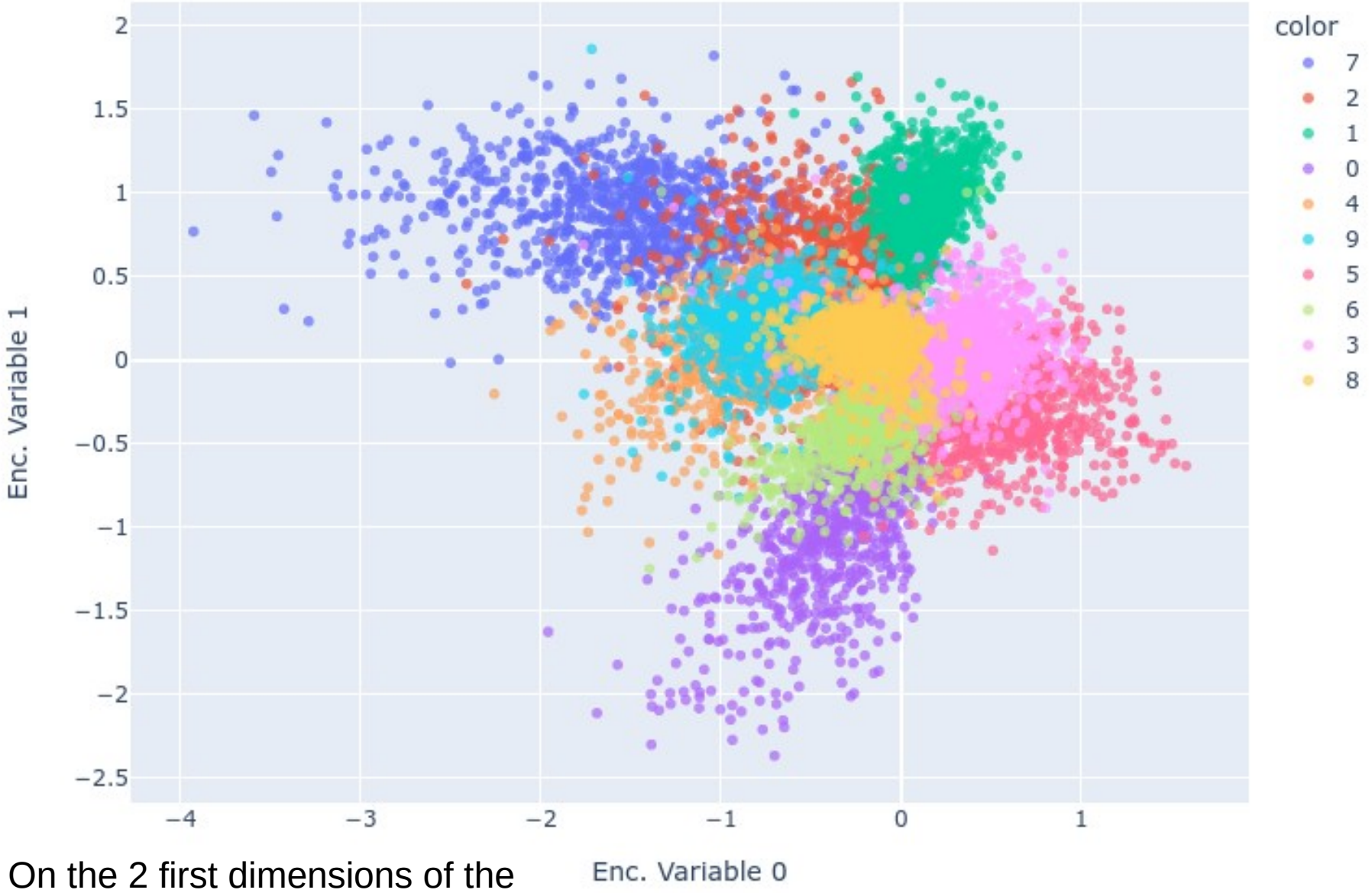
epoch 31

Generation from latent space



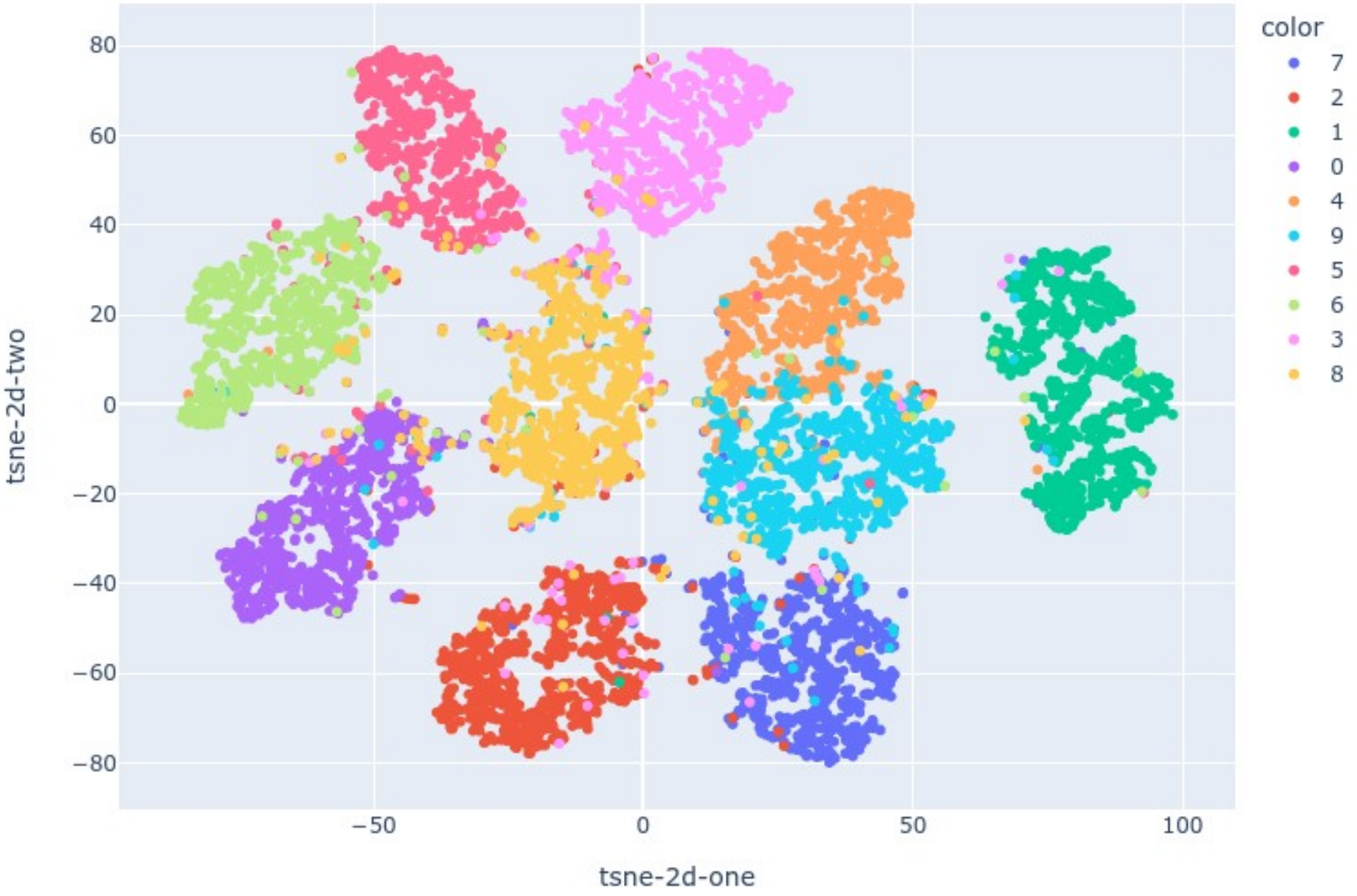
Latent space is not compact, thus, unknown digits

Latent space vizualization



On the 2 first dimensions of the latent space

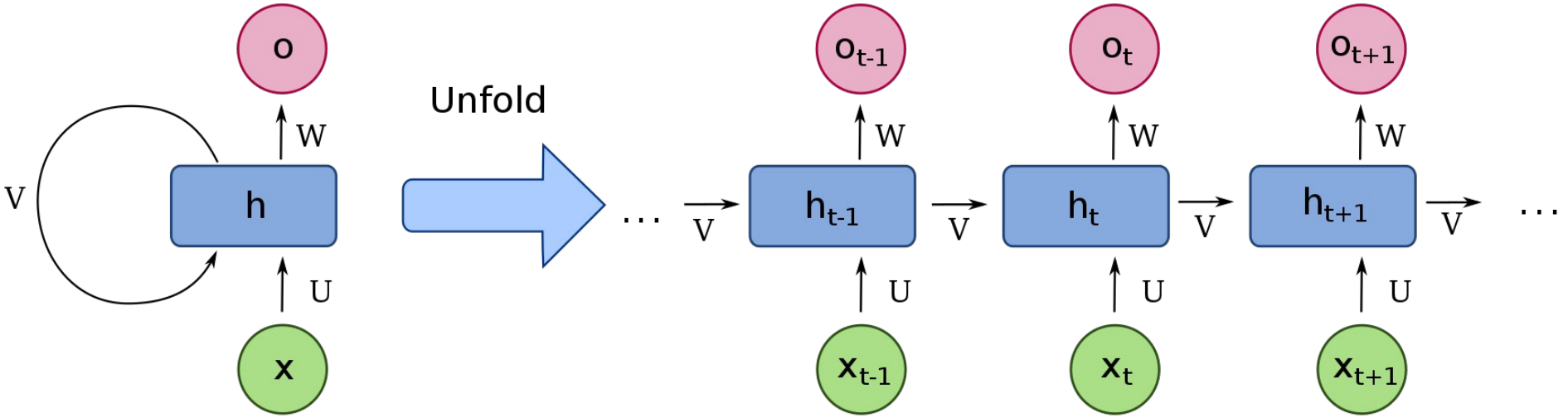
Vizualization with TSNE



- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- **Recurent networks**
- Adversarial networks
- Point cloud neural network for particle physics
- FPGA implementation principles

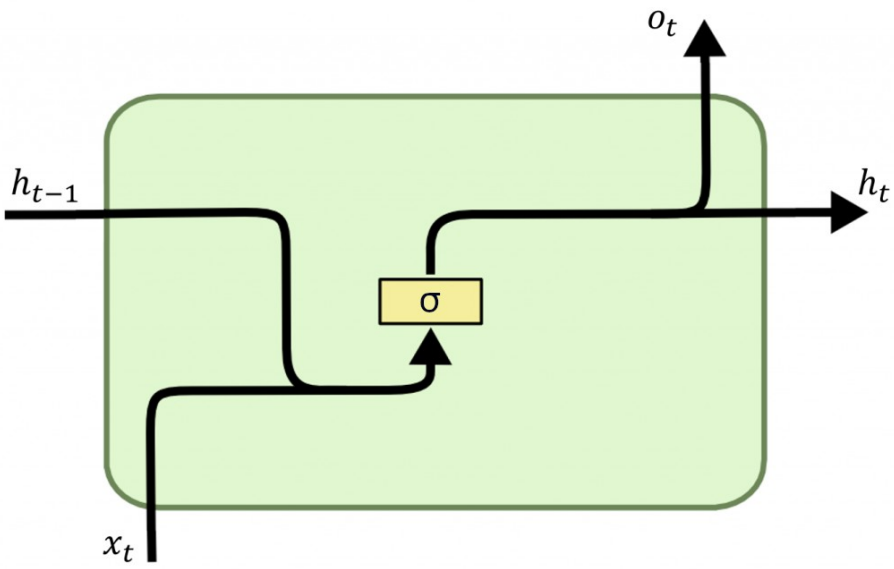
Recurent networks

- Loop network with context save
- Adapted to sequence input
  - variable length data
  - temporal series
  - langage recognition
  - Automatic translation
  - Shape recognition



Vanilla RNN

- Basic version of RNN
- Every value is indexed by the time t
- From a sequence of input vectors (x_t)
- At each iteration
  - produce a context vector (h_t) compact value describing the history
  - produce an output vector (o_t)



$$h_t = \sigma(W_h x_t + U_h h_{t-1} + b_h)$$

$$o_t = \sigma(W_y h_t + b_y)$$

Gradient vanishing

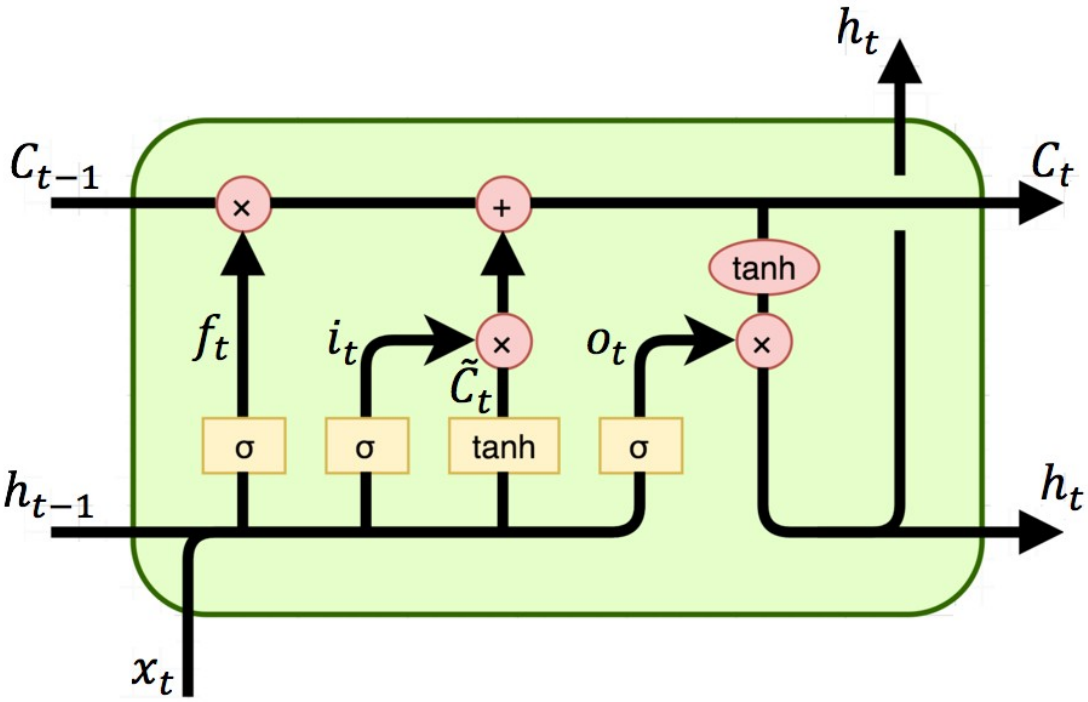
- On vanilla RNN, gradient decreases exponentially over time
- Prevent from modifying weights over past events
- When T increases

$$\frac{\partial(E)}{\partial W} = \sum_{t=1}^{T} \frac{\partial(E_t)}{\partial W} \to 0 \qquad W \leftarrow W - \alpha \frac{\partial E}{\partial W} \approx W$$

LSTM

Sepp Hochreiter & Jürgen Schmidhuber, 1997

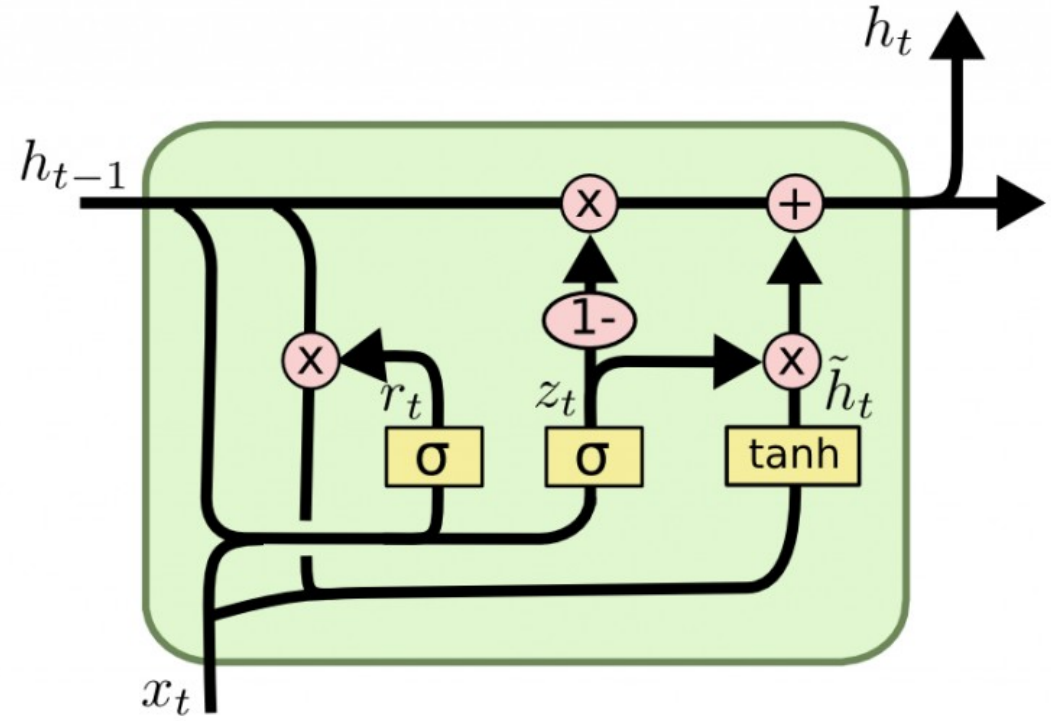
- Solve the gradient vanishing by a more complicated structure
- The context is composed of two parts
  - h_t the hidden state as in vanilla
  - C_t is the context with constant gain (long term memory up to 1000 cells)



$$
\begin{aligned}
F_t &= \sigma(W_F x_t + U_F h_{t-1} + b_F) \\
I_t &= \sigma(W_I x_t + U_I h_{t-1} + b_I) \\
O_t &= \sigma(W_O x_t + U_O h_{t-1} + b_O) \\
c_t &= F_t \circ c_{t-1} + I_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= O_t \circ \tanh(c_t) \\
o_t &= f(W_o h_t + b_o)
\end{aligned}
$$

GRU

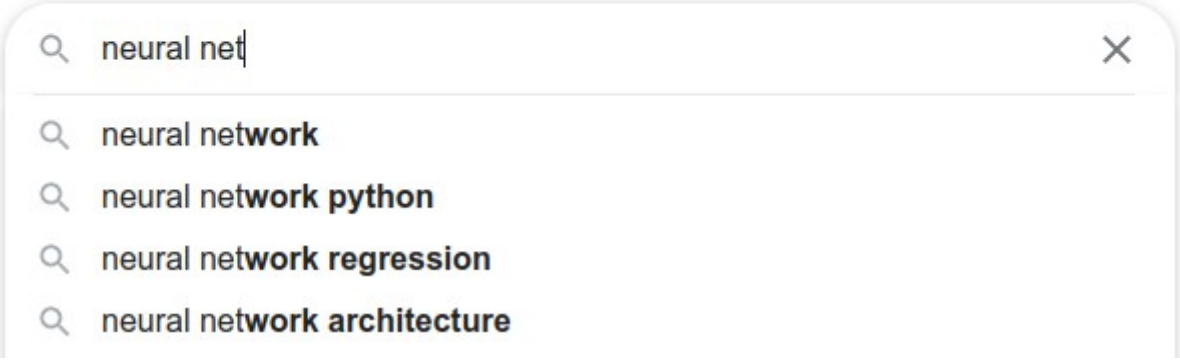
- Alternative to LSTM with less parameters
- only one hidden state mixed with long term memory

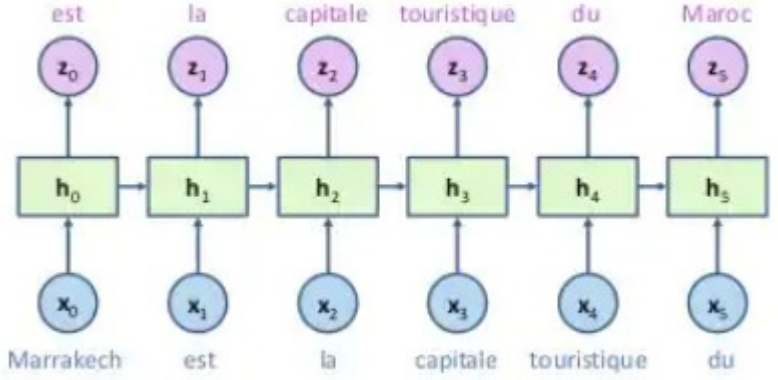


$$Z_t = \sigma(W_Z x_t + U_Z h_{t-1} + b_Z)$$
$$R_t = \sigma(W_R x_t + U_R h_{t-1} + b_R)$$
$$h_t = Z_t \circ h_{t-1} + (1 - Z_t) \circ \tanh(W_h x_t + U_h(R_t \circ h_{t-1}) + b_h)$$

Appli.1 sequence prediction

- Predict the sequel of a sequence
- learn on a sequence database
- Very short context
- Search suggestion, inputs are letters
- Text generation, inputs are words
- Output : probability matrix over a dictionnary

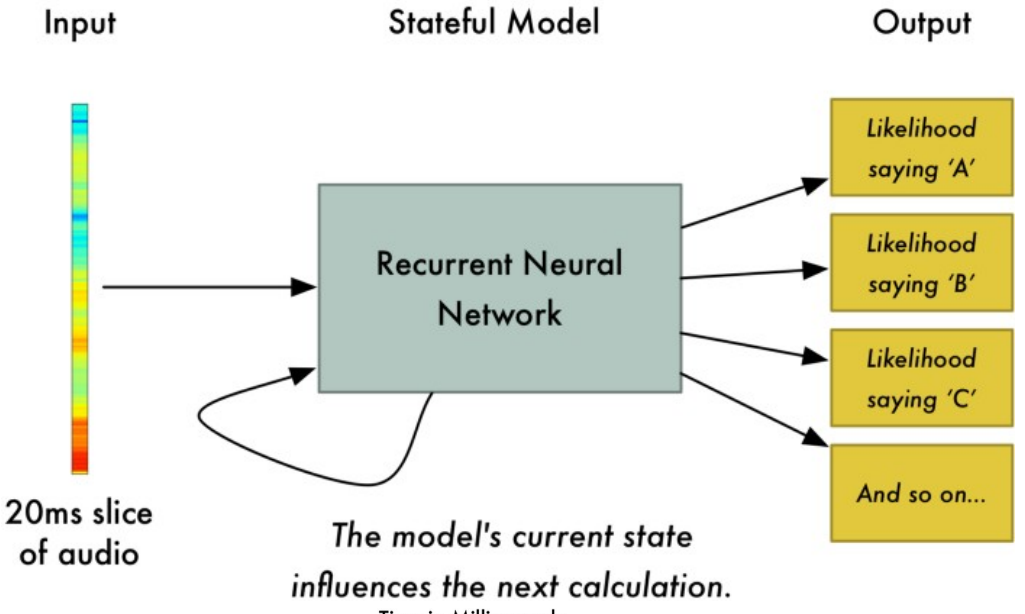


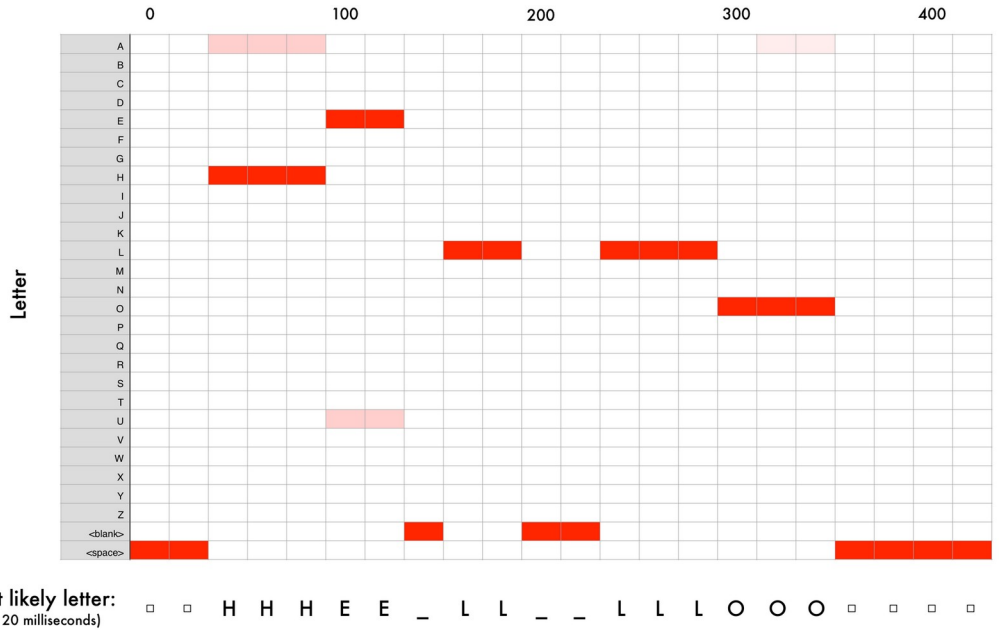


$$L(\theta) = -\sum_{t=1}^{T} logP(y_t|x_t)$$

Appli. 2 Speech recognition

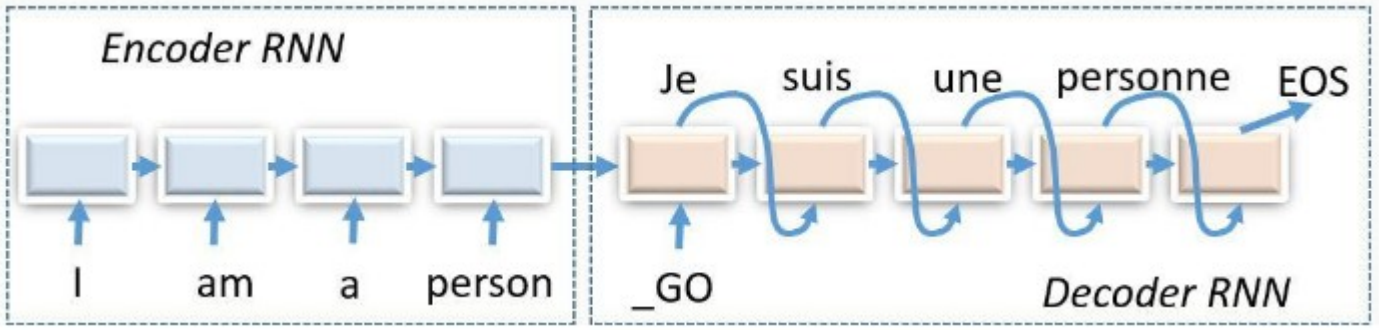
- Sound is sampled and normalized
- Samples are injected in a RNN
- Sequence of heard sound are produced
- Second level RNN to build correct words





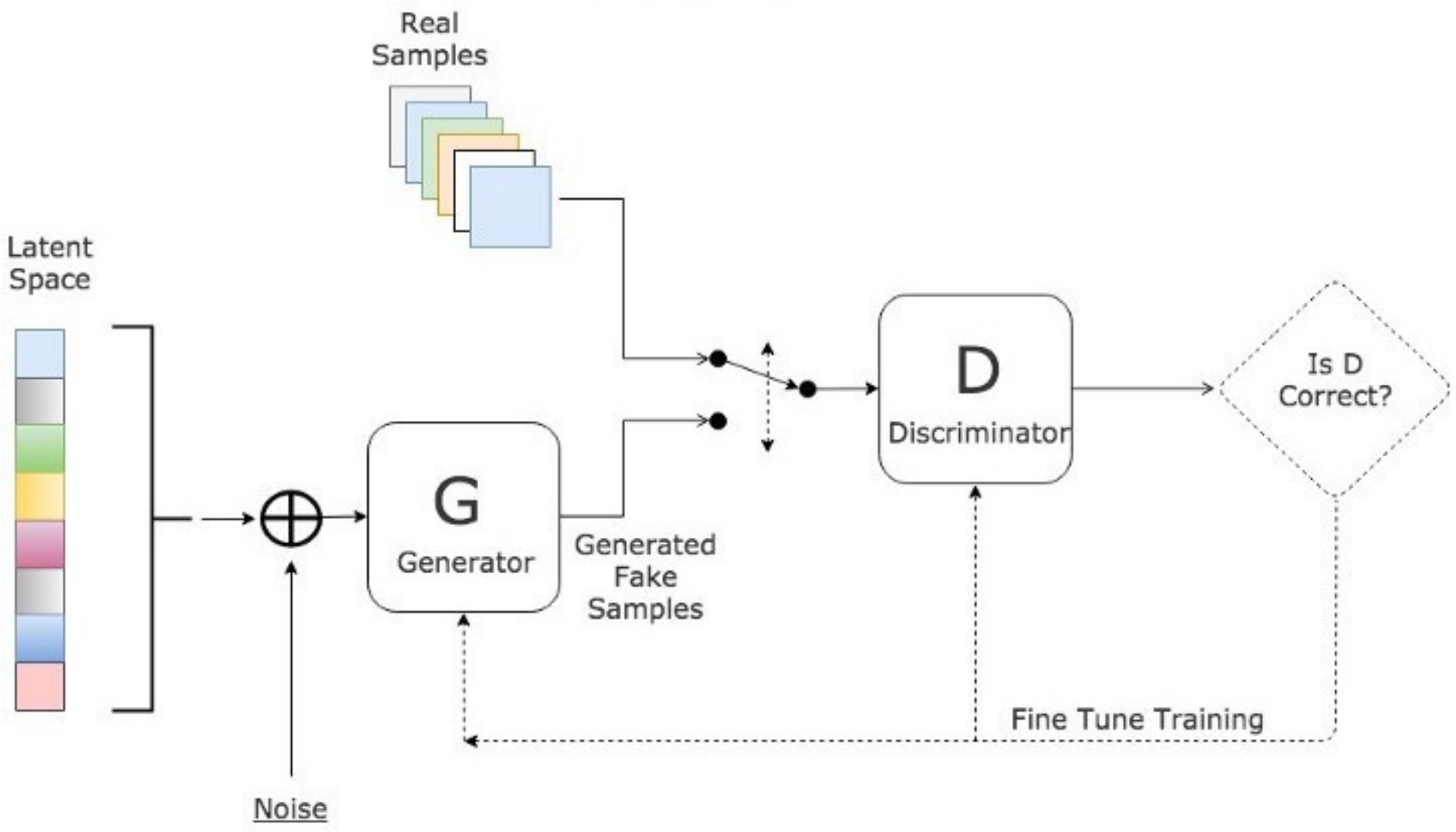
Appli.3 Translation

- Based on two RNN
  - One for encoding the source language structure
  - One for decoding the target language structure
  - Avoid litteral translation by handling the specific structures of the two languages

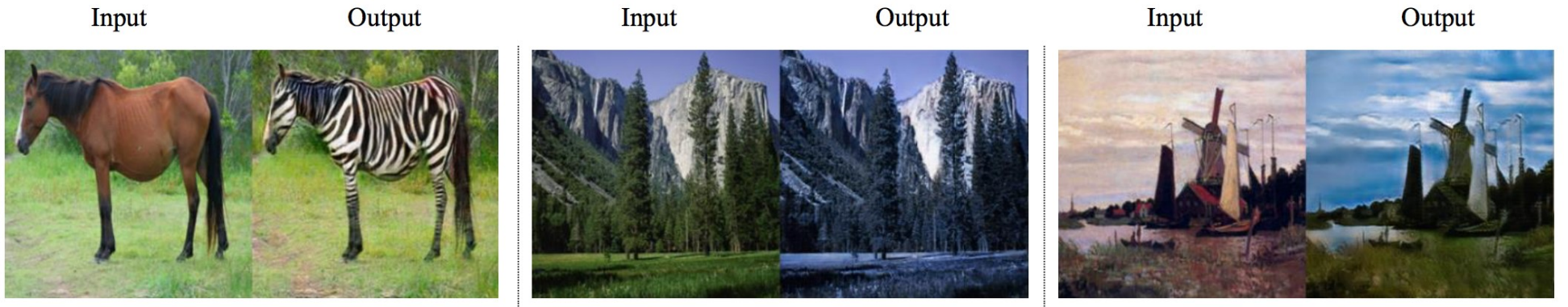


- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- Recurent networks
- **Adversarial networks**
- Point cloud neural network for particle physics
- FPGA implementation principles

Generative adversarial networks (GAN)

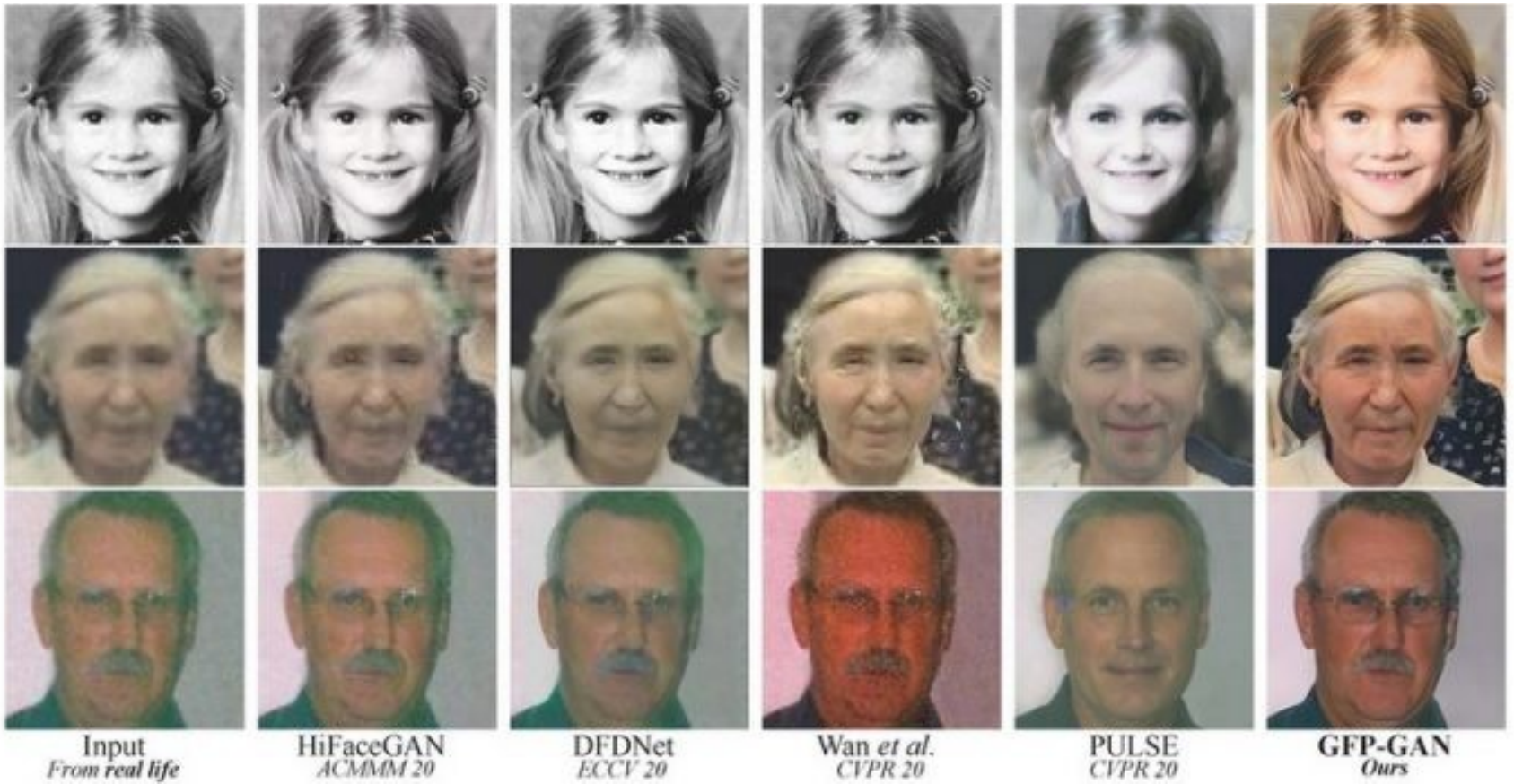


GAN Results


Input
Output
Input
Output
Input
Output
Horse → Zebra
Summer → Winter
Monet → Photo
Zebra → Horse
Winter → Summer
Photo → Monet

GFP-GAN

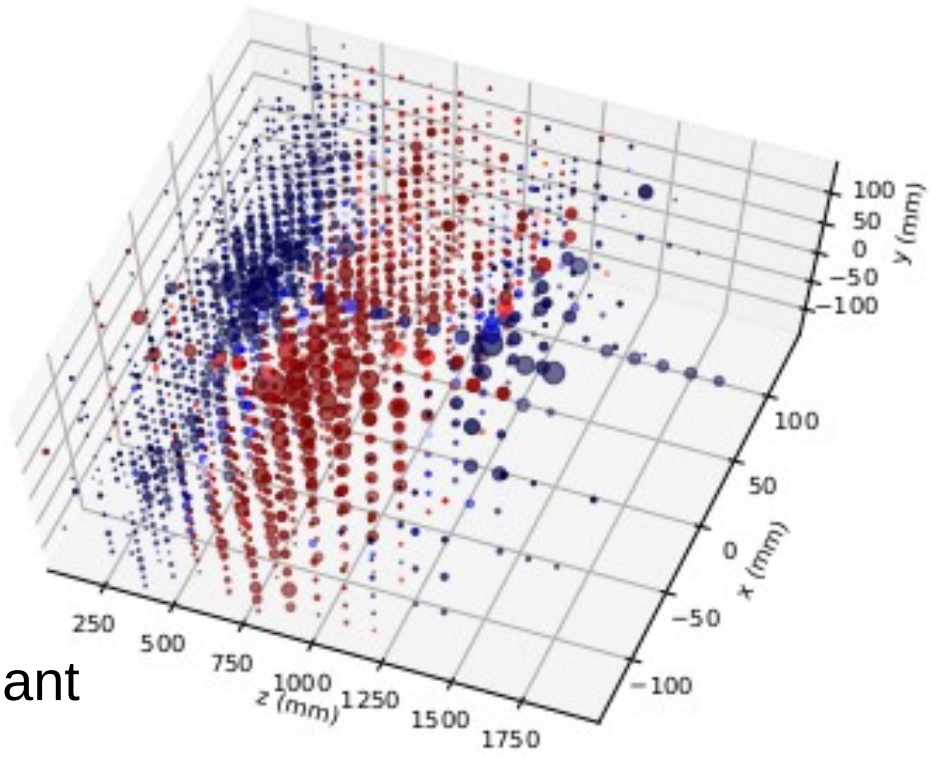
- Mix between an auto-encoder and a GAN : image regeneration



- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- Recurent networks
- Adversarial networks
- **Point cloud neural network for particle physics**
- FPGA implementation principles

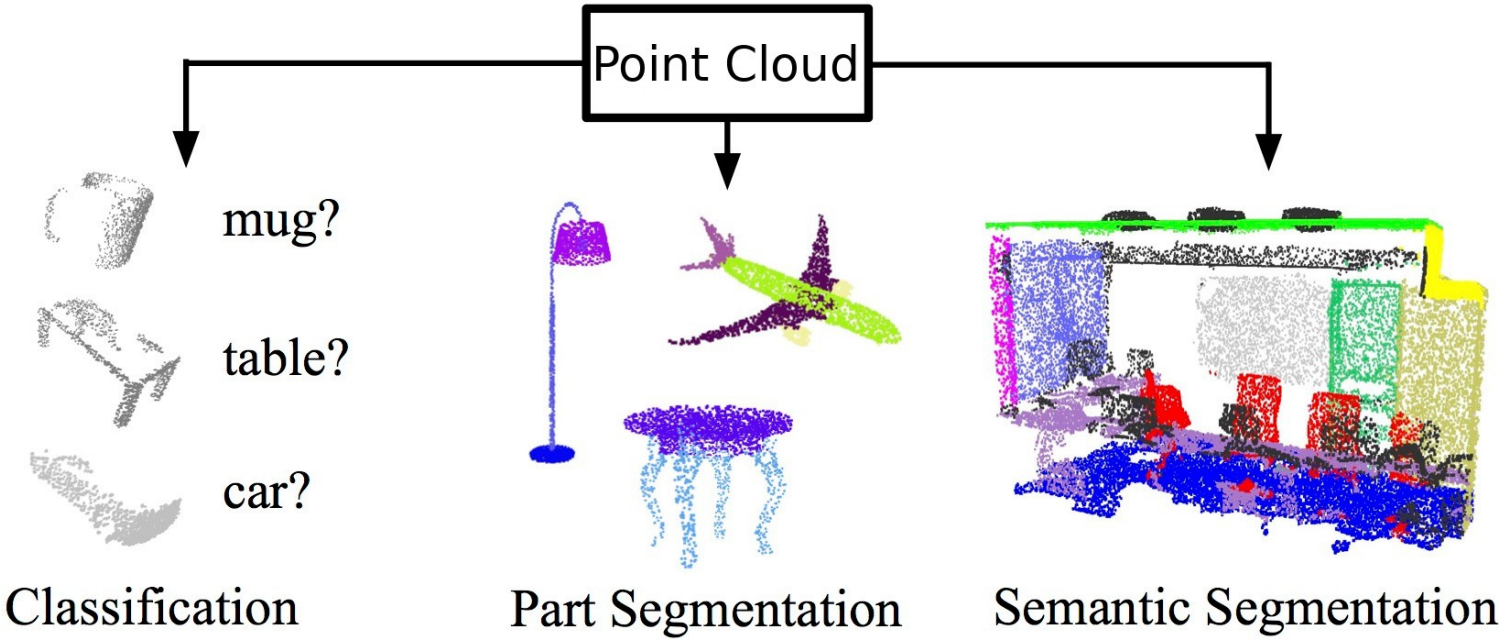
Input Data from HEP Experiments

- Point cloud data
- Points x_i in R^k ($k \geq 3$)
  - Euclidian 3D coordinates (barycenter of sensors)
  - (k-3) features (timestamp, energy...)
- Labeled when simulated (GEANT4)
- 4 main properties
  - unordered : need for a permutation invariant operator
  - Interaction among points : the metric distance defines meaningful neigbourings
  - Invariance under transformation : rotation and translation should not modify the result
  - Sparsity



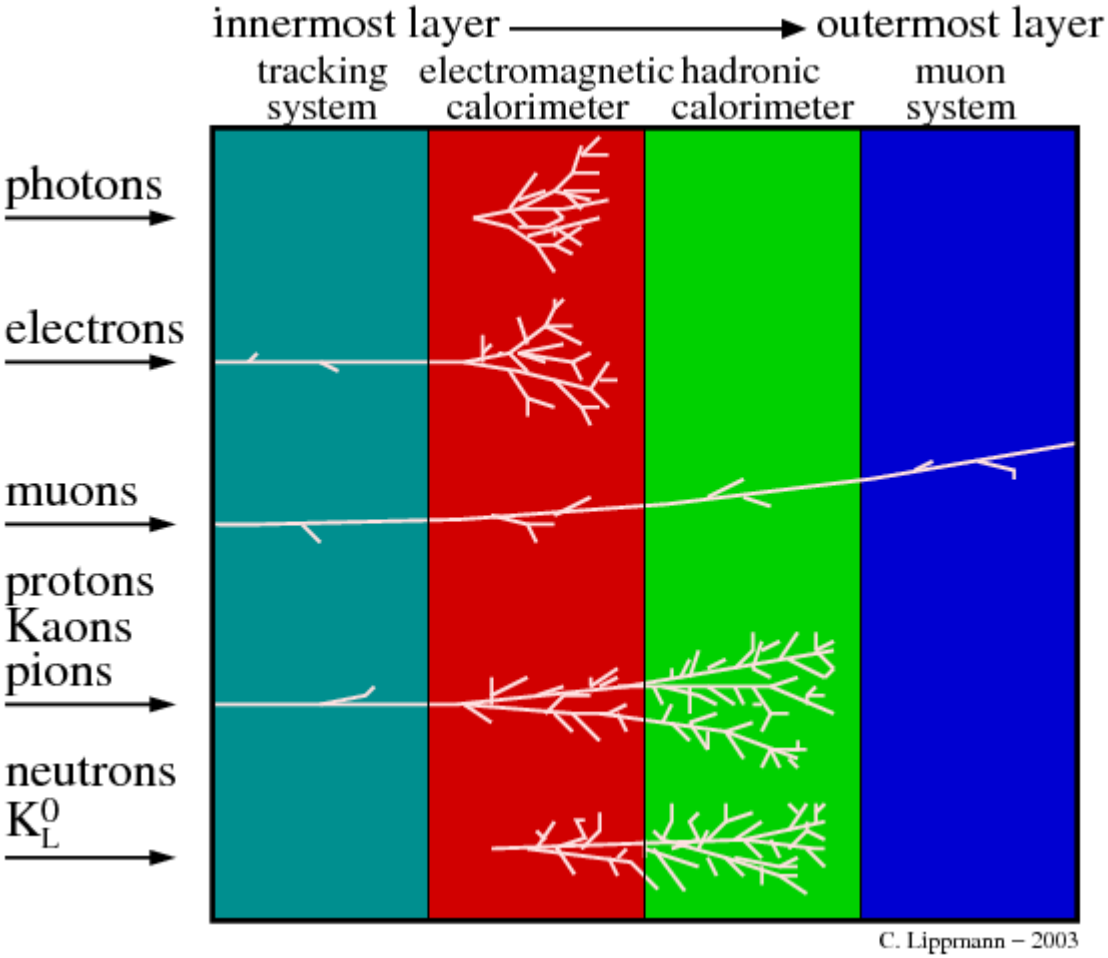
Point Cloud Problematics

- Three problematics can be adressed
  - Classification
  - Part segmentation
  - Semantic segmentation



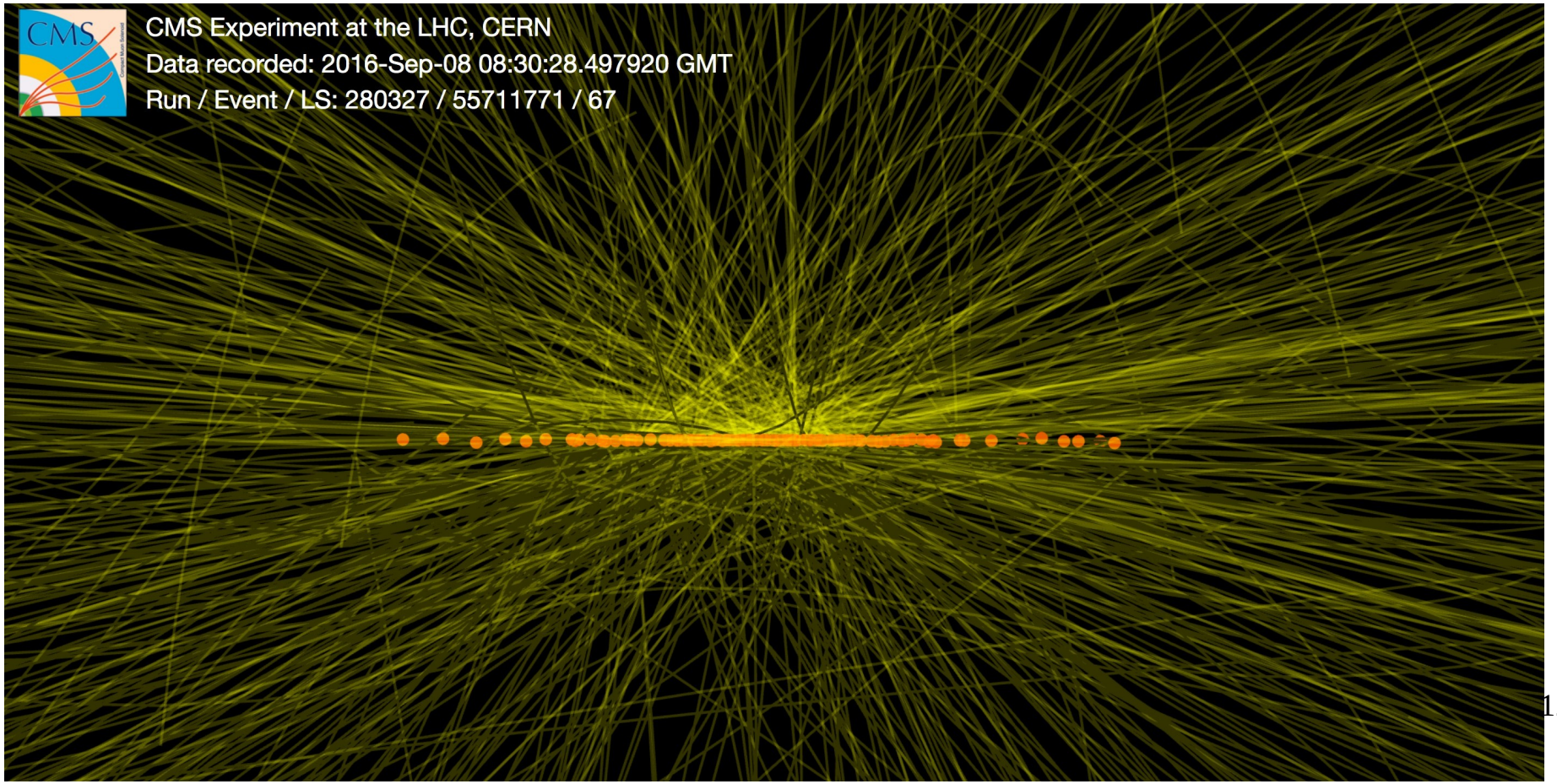
Classification : Particle Id

- 3D Shape of shower → particle id
- Regression of energy and impact position



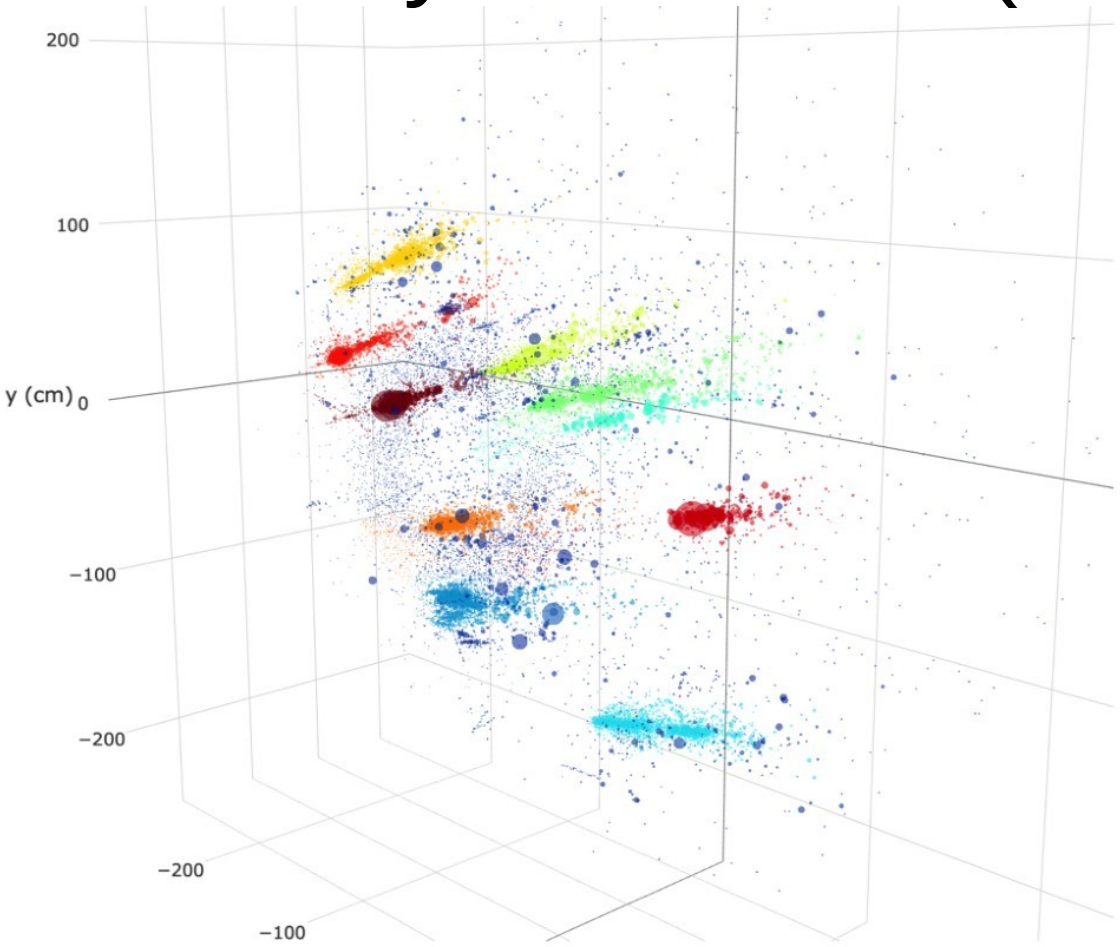
Part Segmentation: shower disentanglement

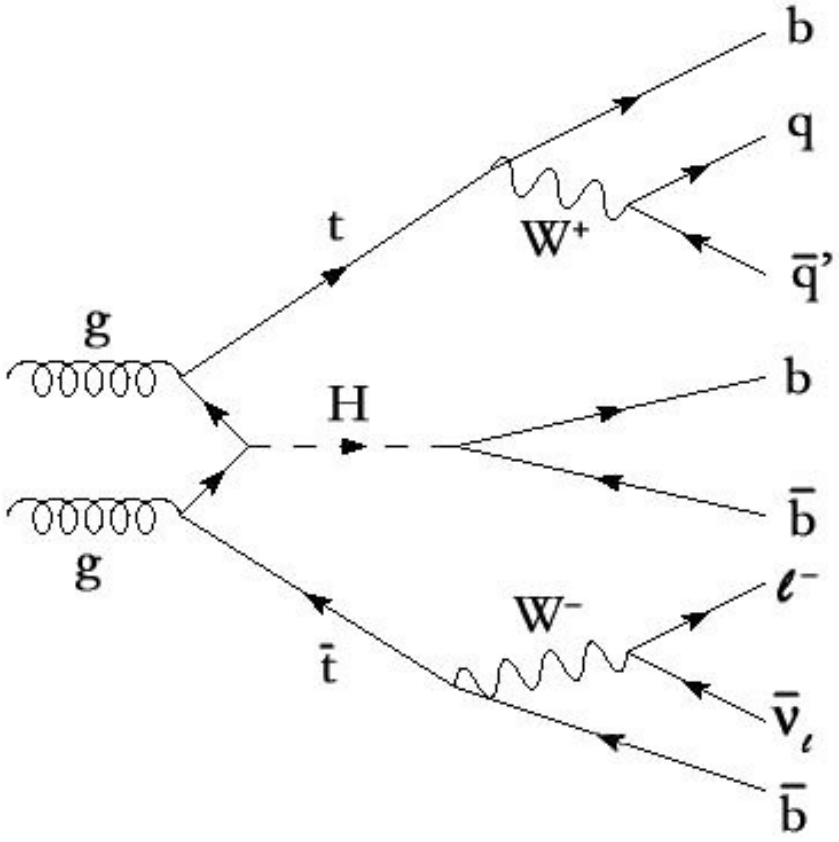
- High luminosity → High pileup (up to 200 in future)



Semantic Segmentation: secondary particle Id

- Identify long-lived intermediate particles
- Identify Jet Nature (Hadronization)

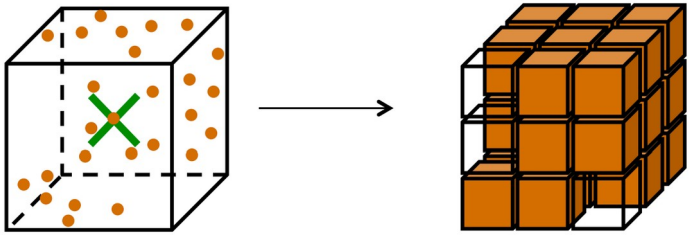




Point Cloud Neural Networks

Three main techniques

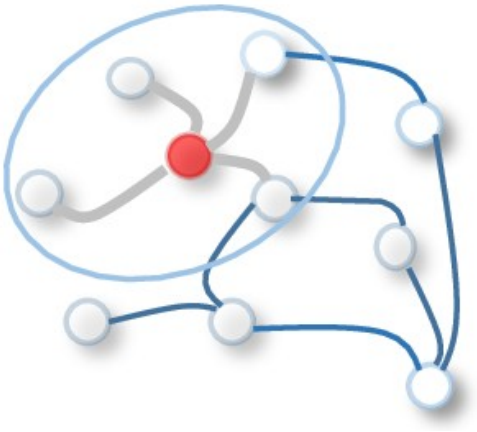
- Voxelization and 3D Convolution (2015-2016)



- Symmetric pooling (2017-2018)

$$f(x_1, \ldots, x_n) \approx g(h(x_1), \ldots, h(x_n))$$

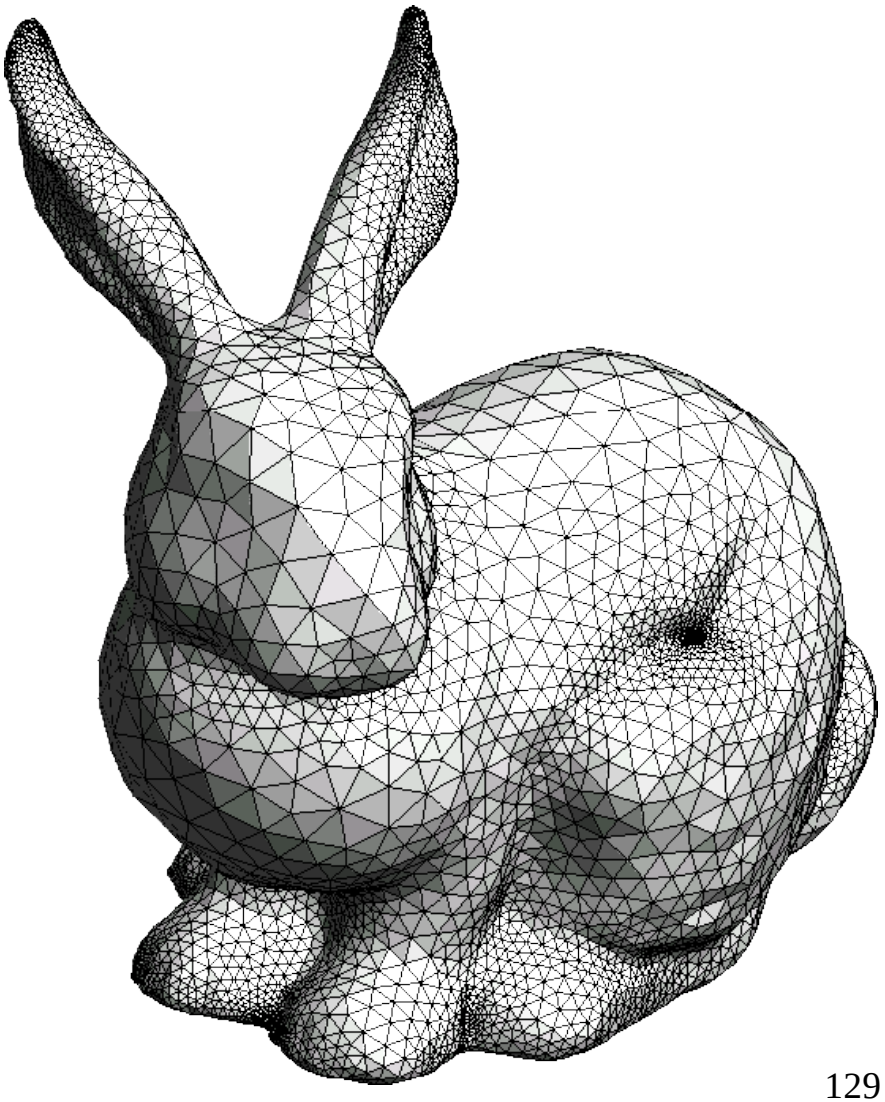
- Graph Convolution (2017-2020)



Precision

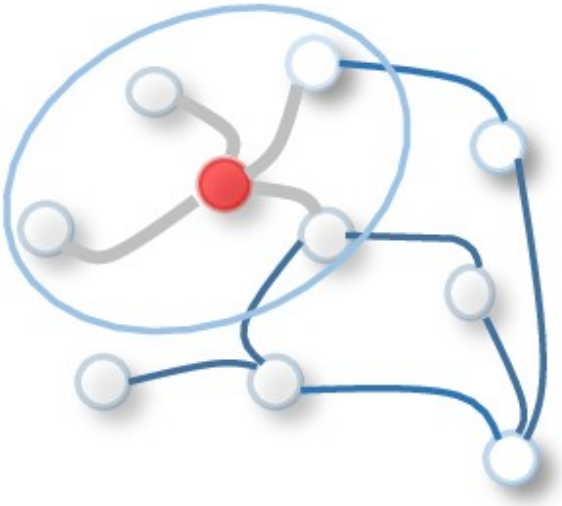
Idea of Graph convolution

- Build a graph structure with the point cloud
- Capture the locality in the graph adjacency
- Apply known techniques of graph convolution
- Use the result of convolution for classification and segmentation



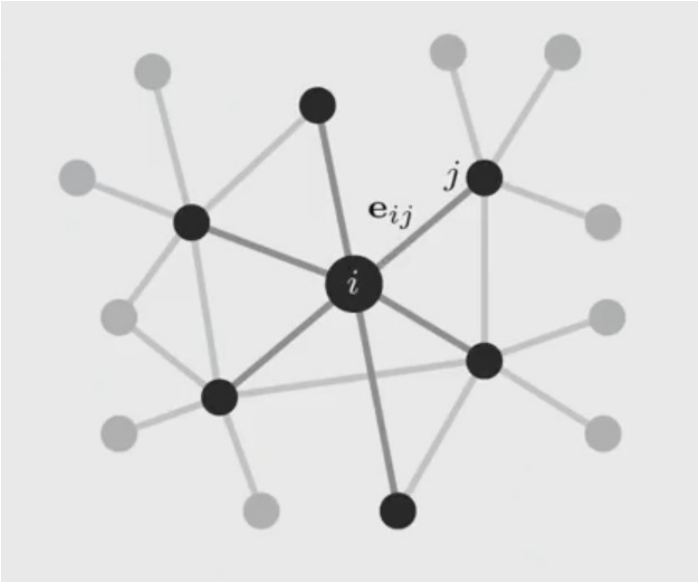
Neural Message Passing Network

- Generic recipe for spatial graph convolution
- Convolves the central node x_i with its neighbors x_j in N(v)



$$x_i^k = \gamma(x_i^{k-1}, \square_{j \in N(i)} \phi_\theta(x_i^{k-1}, x_j^{k-1}, e_{i,j}))$$

- □ is a symmetric normalized operator like mean or max
- Nice complexity O(m)



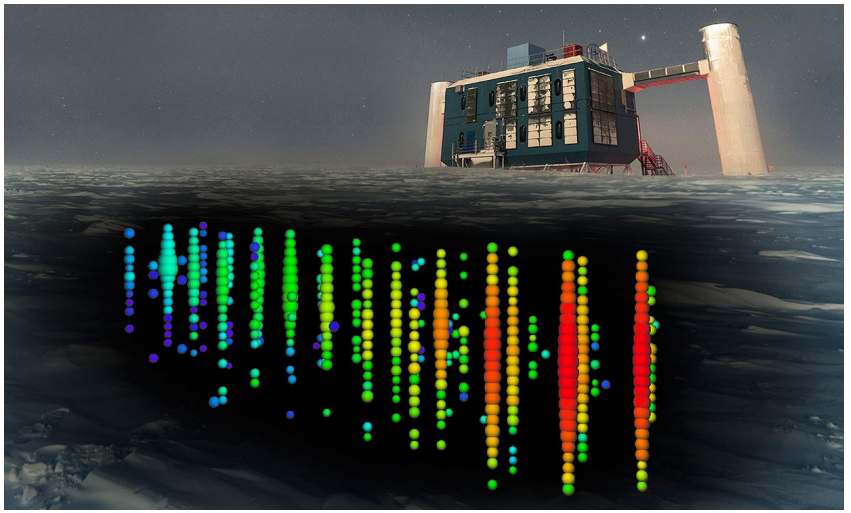
Gilmer & al, Neural message passing for quantum chemistry, 2017

MoNet & Icecube

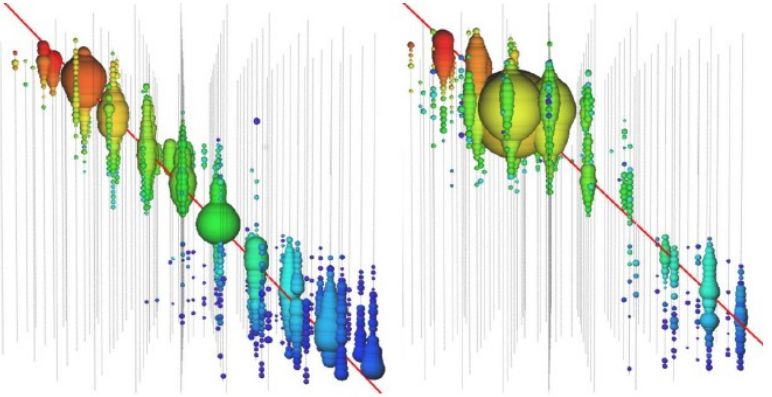
Monti & al, Geometric deep learning on graphs and manifolds using mixture model CNNs, 2017

- Fixed graph
- Pseudo-coordinates u + gaussian kernel g_{wn}
- MPNN Convolution

$$x'_{im} = \sum_{\forall j,(i,j)\in E} ReLU(\theta_m.(x_j \odot g_{wn}(u(x_i, x_j))))$$



- Used as classifier for Icecube neutrinos detector
- Stochasticity evaluation



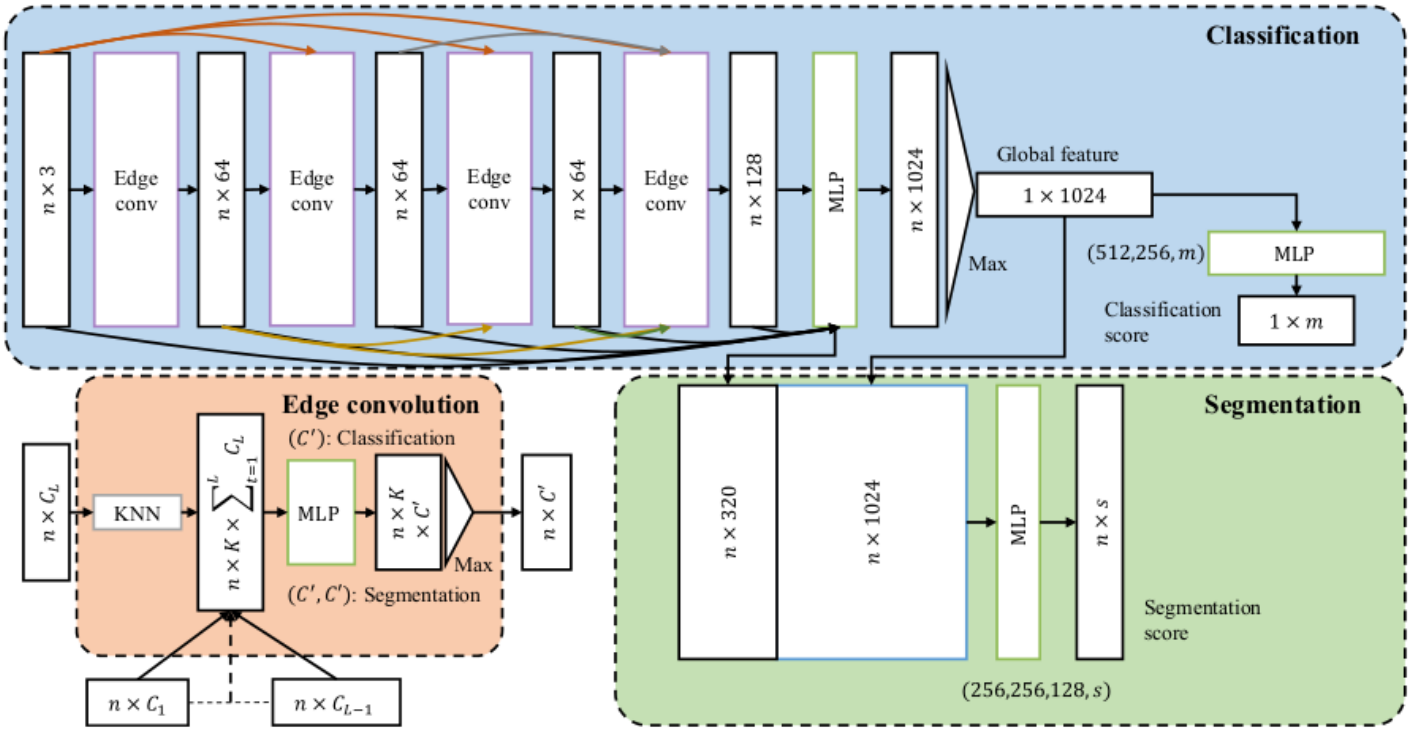
Choma & al, Graph neural networks for IceCube signal classification,2018

Dynamic Graph Convolution

- Wang & al, Dynamic Graph CNN for Learning on Point Clouds, 2019
- Zhang & al, Linked DGCNN, 2019

- Dynamically build graph (K-Nearest Neighbors) and apply graph convolution
- Gives Permutation invariance and partial translation invariance
- Very computational resource demanding (KNN)

Method	#Parameters (M)	Forward time (ms)	OA (%)
PointNet [18]	3.48	**0.8**	89.2
PointNet++ [20]	1.48	1.4	90.7
DGCNN [21]	1.84	3.1	92.2
LGDCNN	**1.08**	2.8	**92.9**

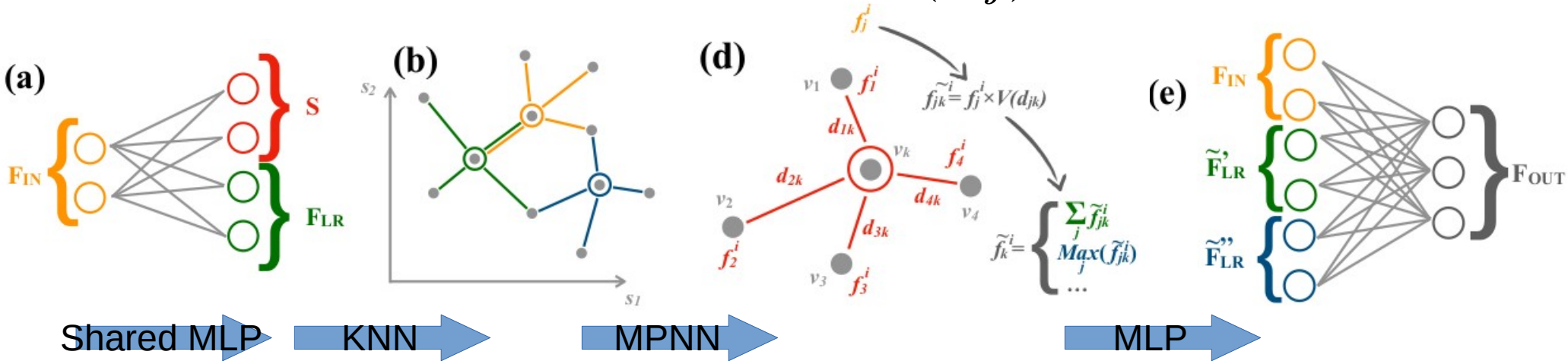


HGCal Application

Qasim, Kieseler & al, Learning representations of irregular particle-detector geometry with distance-weighted graph networks, 2019

- Performed at CERN, dedicated to HGCal style application
- Mix between PointNet and DGCNN → try to reduce computational complexity
- Adaptation to HEP problematics
- Tested on GEANT4 simulations (square sensors, charged pions between 10 and 100 GeV, 16Mevents)

$$V(d_{ij}) = e^{-|d_{ij}|}$$

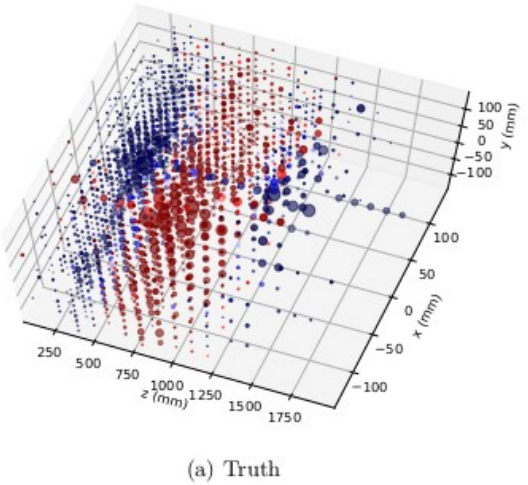


DGCNN Particle Segmentation

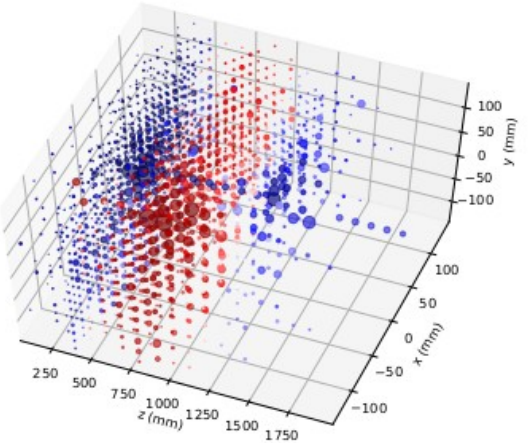
- Prediction of the energy fraction of each sensor belonging to each shower (for 2 shower only)
- Define a loss function for hit segmentation

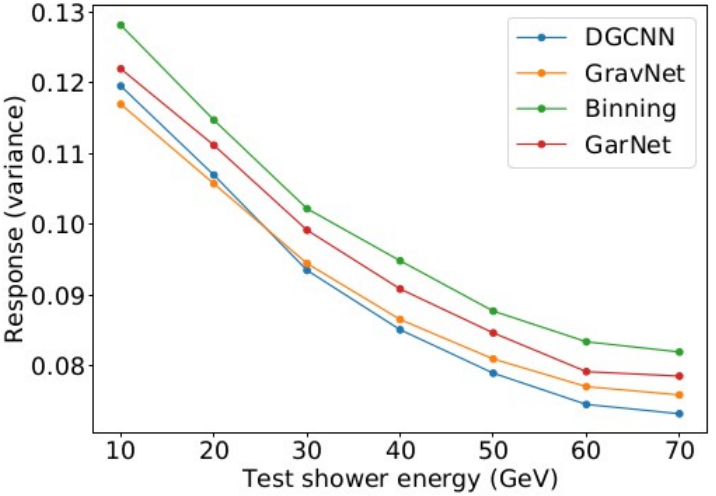
$$L = \sum_k \frac{\sum_i \sqrt{E_i t_{ik}} (p_{ik} - t_{ik})^2}{\sum_i \sqrt{E_i t_{ik}}}$$

t_{ik} is the true energy fraction in sensor i for shower k, p_{ik} is the predicted energy fraction, E_i is the total energy deposited in the sensor i



(a) Truth

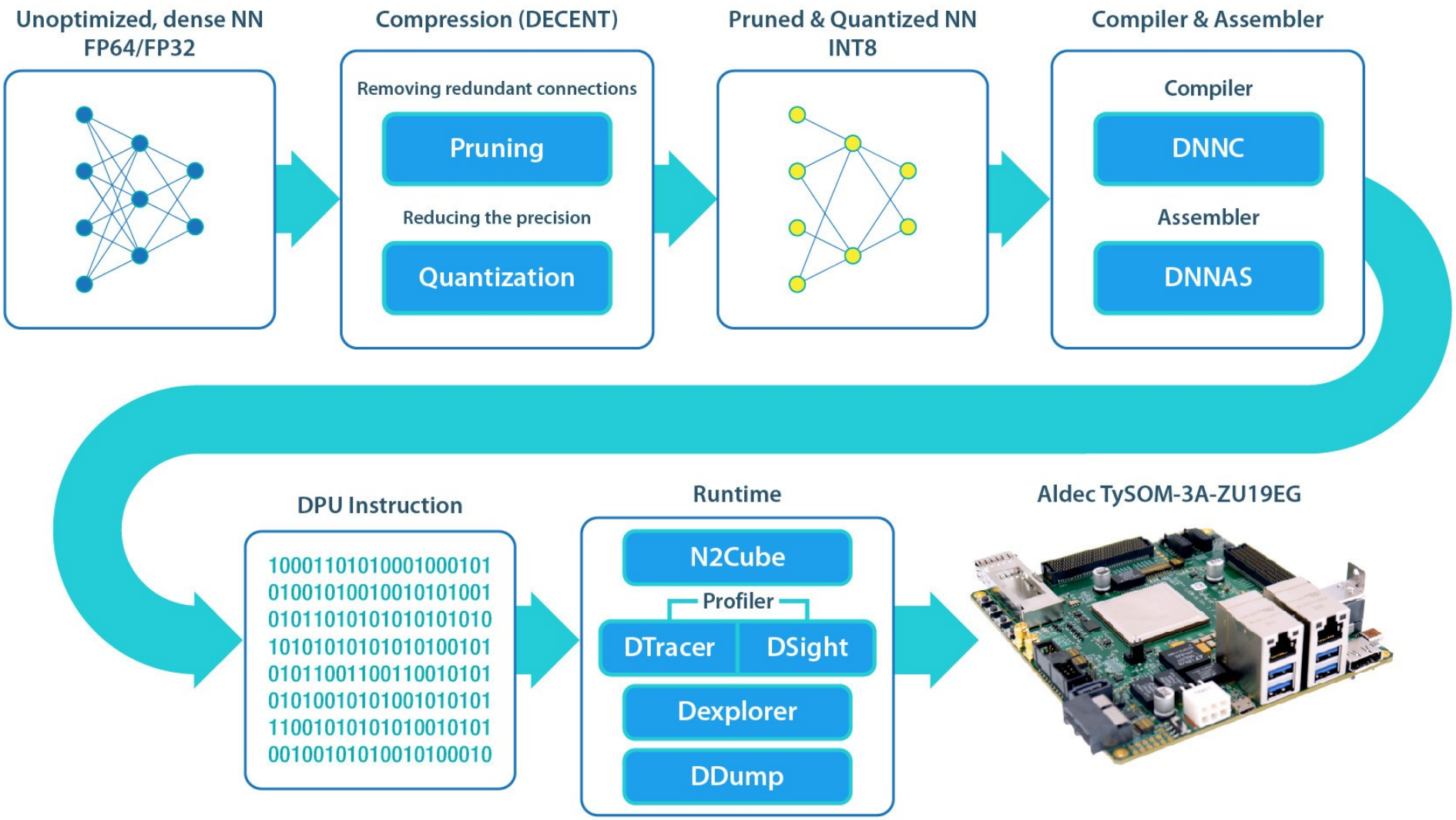




(d) Variance

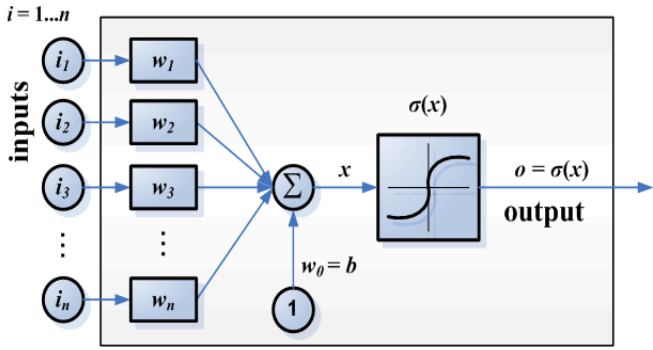
- Problematics
- Neuron, perceptron and back-propagation
- PyTorch Hands on
- Convolutional networks
- Auto-encoders
- Recurent networks
- Adversarial networks
- Point cloud neural network for particle physics
- **FPGA implementation principles**

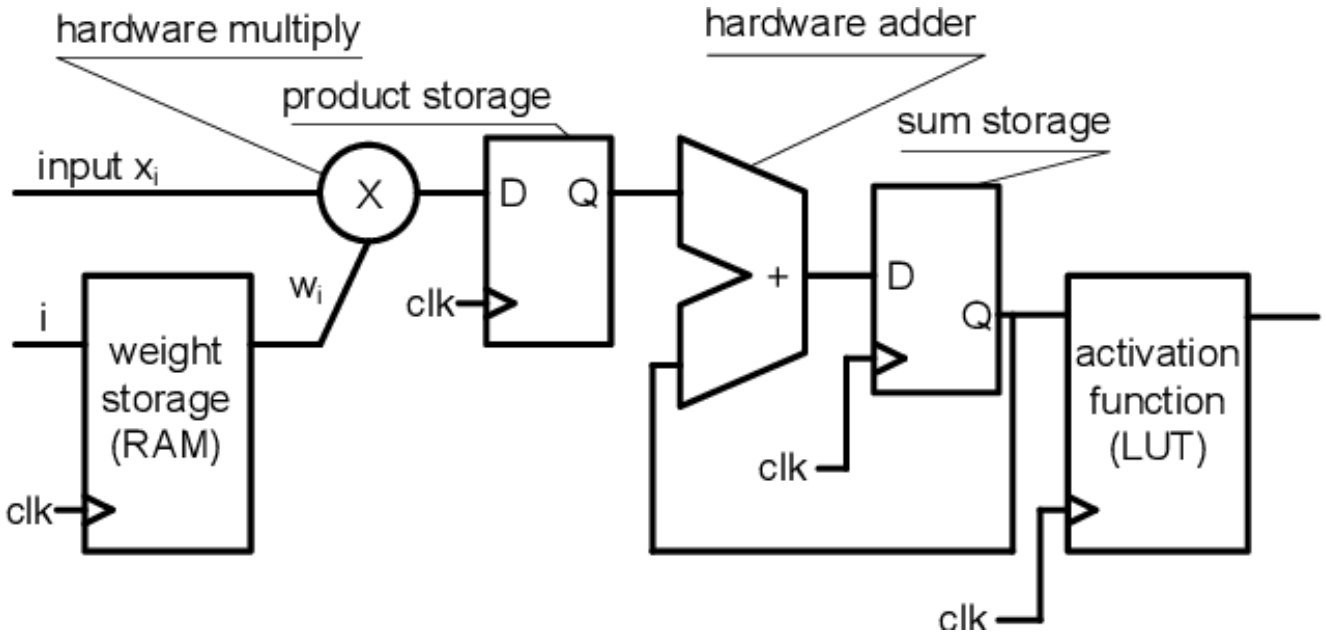
FPGA implementation principles



Neuron in FPGA

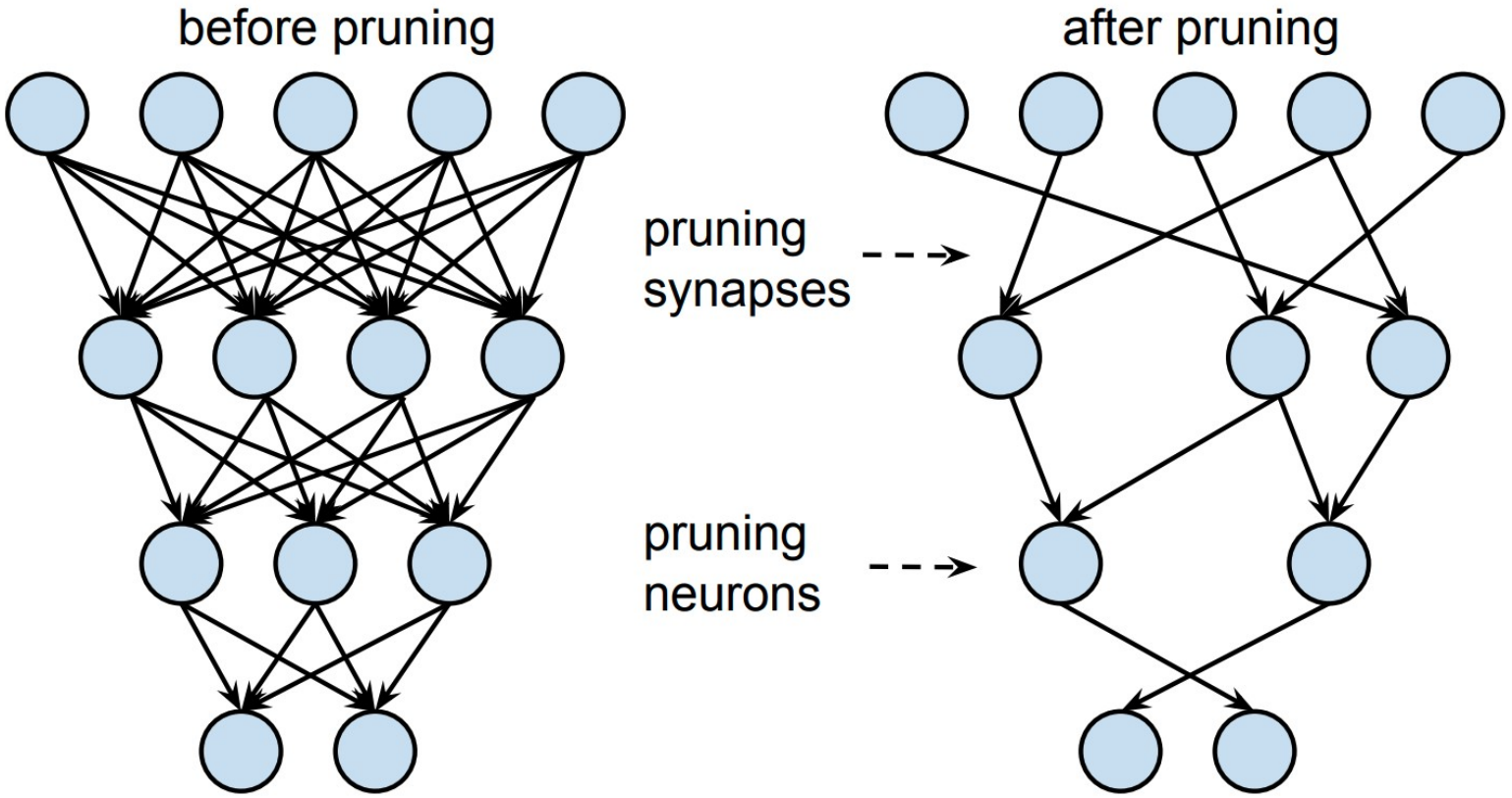
- Only the forward part is implemented
- Sum and multiplication are done in DSP
- Non linearity are implemented in LUT
- Intermediate results are stored in block RAM
- Need plenty of resources
- Need to shrink network before implementation





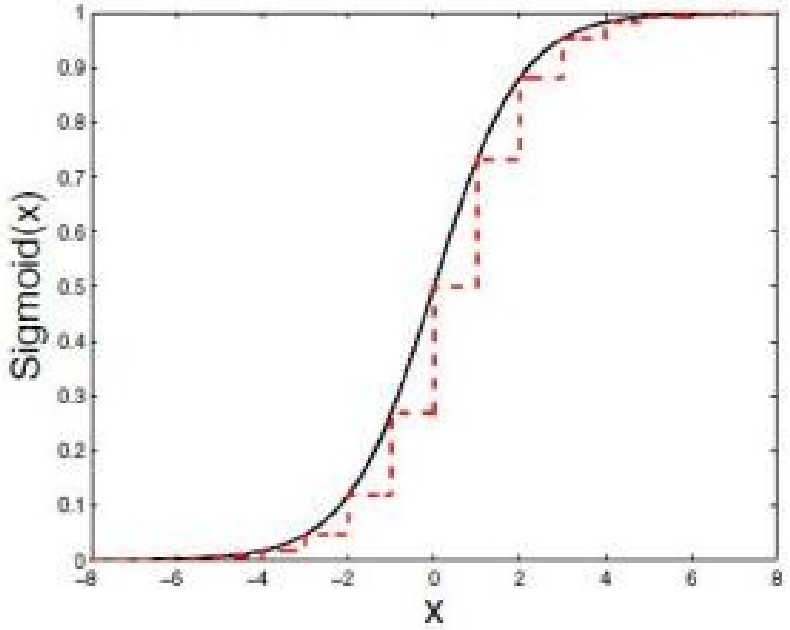
Pruning

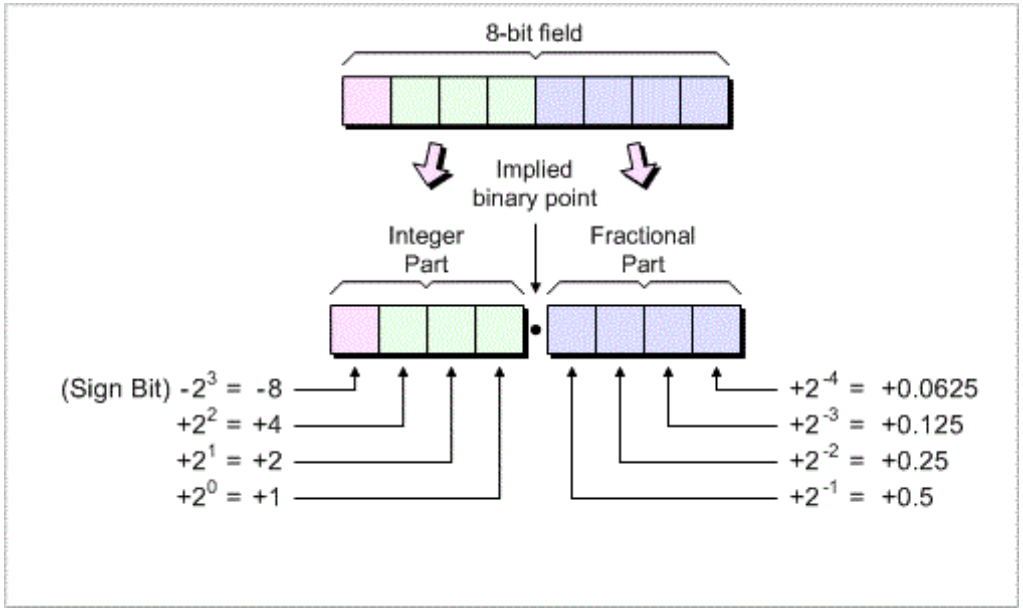
- Reducing network density for implementability
- Aggregation of similar neurons (in terms of weights)
- Removal of almost zero weight connections



Quantization

- Move from floating point operations to fixed point in DSP
- Reduce the range and precision of available weights
- Reduce the precision of non-linearity by LUT
- Need to re-train the network after quantization
- Extrem forms : binary of ternary networks

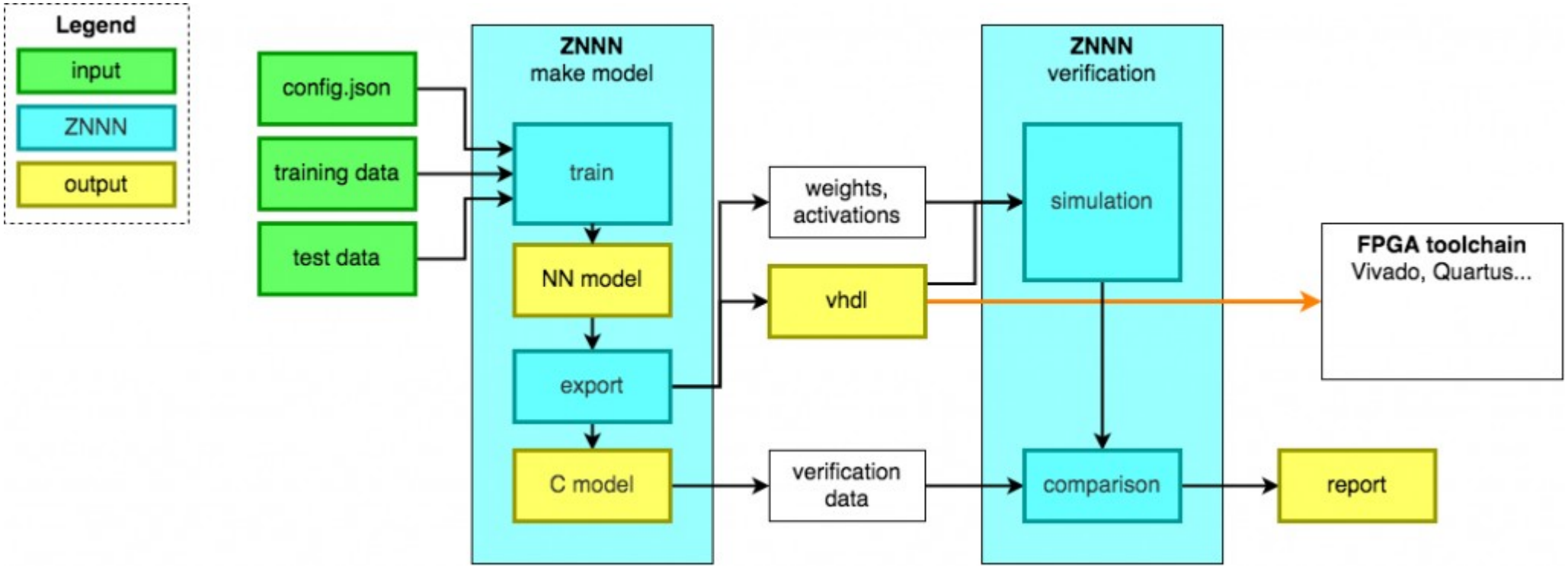




Compilation

- Two main ways
  - Native neuron VHDL implementation
    → VHDL synthesis
  - intermediate high level implementation
    → High level synthesis

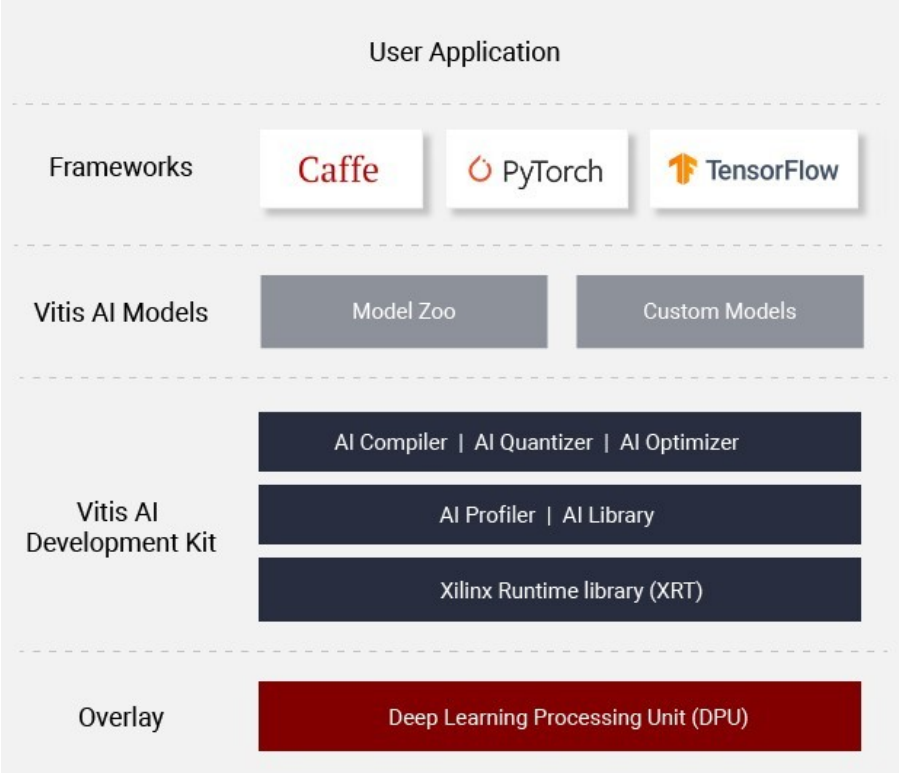
An example of VHDL framework ZNNN



Xilinx Vitis AI

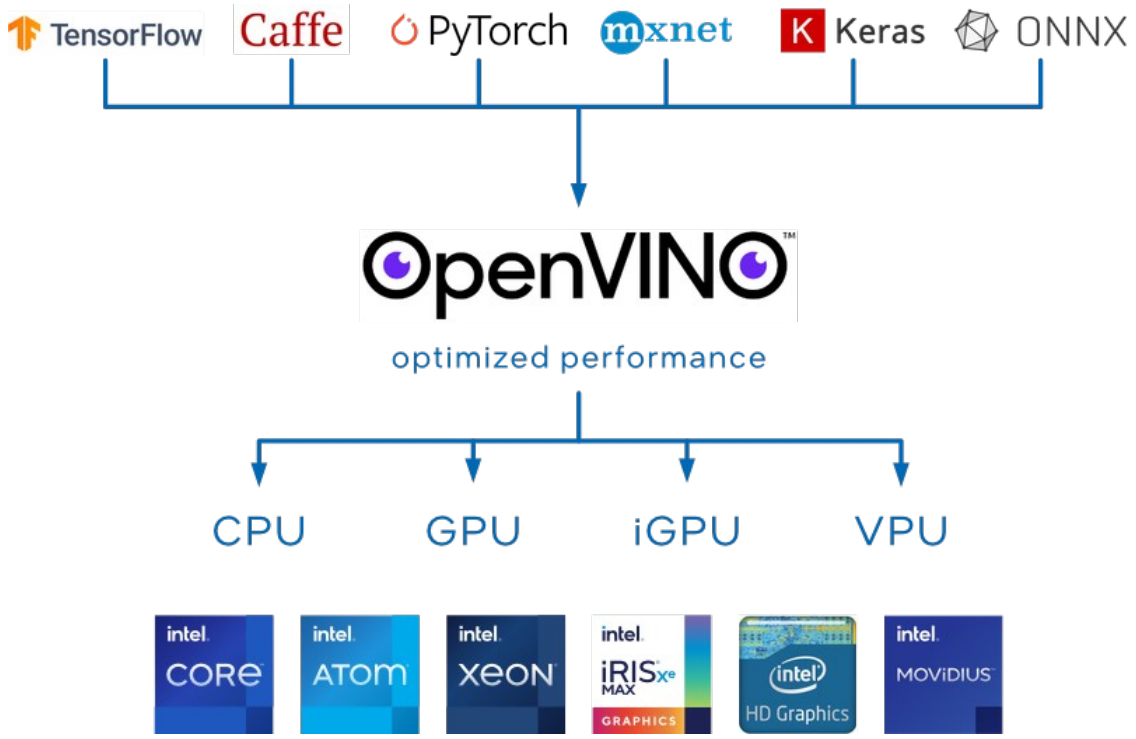
Full framework for neural network implementation on Xilinx hardware (Alveo)

- Importation of networks from PyTorch, Tensorflow or Caffe
- AI Optimizer for pruning and fine tuning
- AI Quantizer converting 32 bits FP to INT8
- AI Compiler (produces DPU)
- AI Profiler full performance analysis



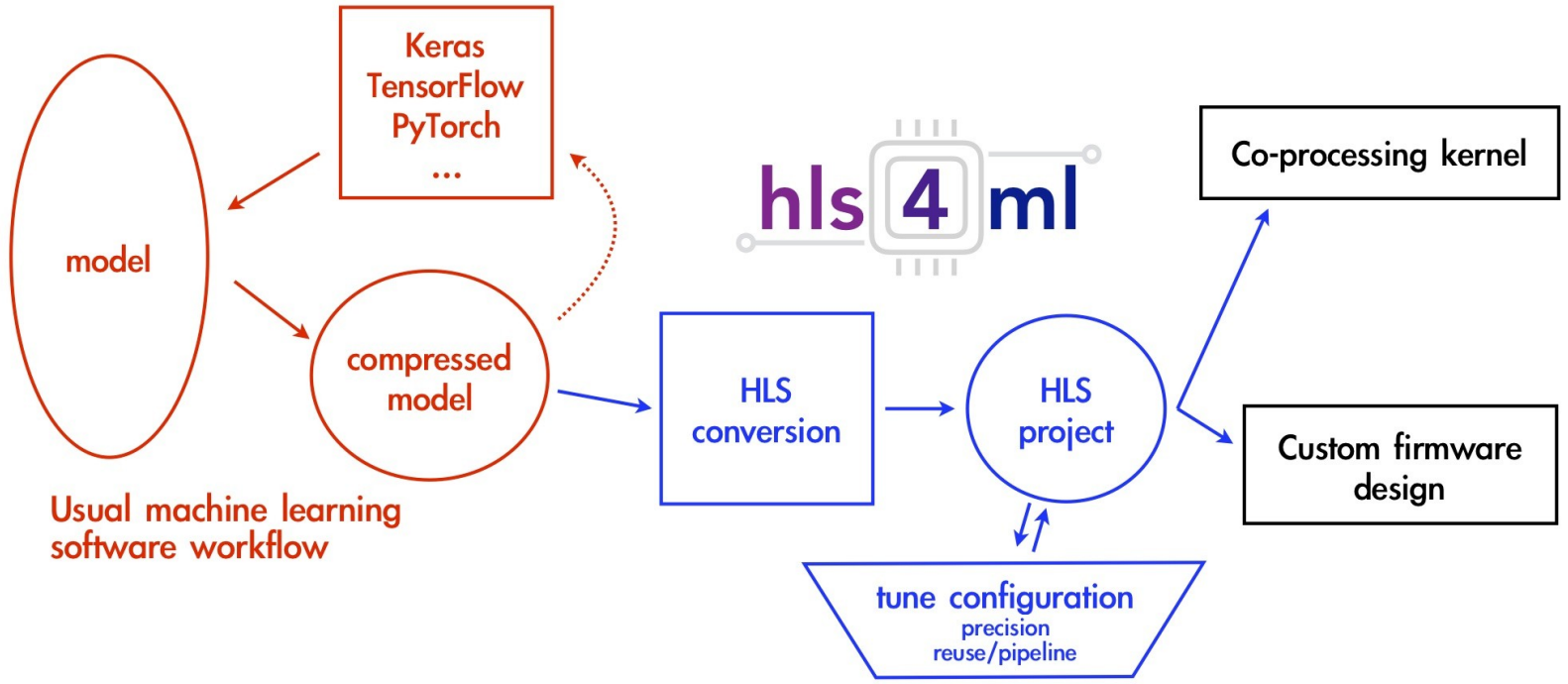
Intel OpenVINO

- Neural network converter to multiple architecture
- FPGA Plugin for Arria 10 GX FPGA



HLS4ML

- High level synthesis framework
- Developed by a community leaded by CERN and Fermilab
- https://arxiv.org/abs/1804.06913



Thank you for your attention