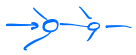
KERAS

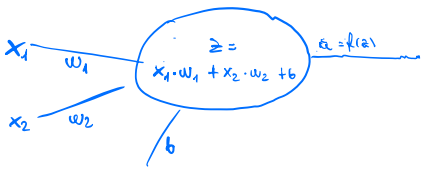
M1

FORWARD
BACK propagation

INPUT — HIDDEN LAYERS — OUTPUT



ACTIVATION FUNCTION



SIGMOID => $a = f(x) = \frac{1}{1+e^{-z}}$

M2

GRADIENT DESCENT
↳ find minimum
↳ select step learning rate

BACK PROPAGATION
↳ weight & biases OPTIMIZATION
PROPAGATE ERROR back
ERROR TO update w

ACTIVATIO
1. BINARY
2. LINEAR
3. SIGMOID → ⇒ vanishing ⇒ values 0 to 1
4. Hyperbolic Tangent → SCALED SIGMOID → ⇒ values −1 to 1 ⇒ vanishing on deep NN
5. ReLU → NO vanishing
6. Leaky ReLU
7. Softmax — output layer of classifier — probability OUTPUT [ ]

VANISHING
end ex ACTIVATION
fast on front
slow at end

COST FUNCTION
$S = \sum_{i=1}^{m} (z_i - wx_i - b^2)$
MINIMIZE w/gradient descend
updating $w = w - \eta \cdot \frac{\partial S}{\partial w}$

M3

libs
↳ TF (PRODUCTION) Google
↳ Keras HIGH LEVEL API on top of TF by GOOGLE → QUICK
↳ Pytorch GPU trained by TF ++ CUSTOMIZATION → DEEP

KERAS
EXL CONCRETS

```
import keras
from k import Sequential
from k import Dense

model = Sequential()
n_cols = concrete_data.shape[1]
```

LINEAR

OR → Complex data → Model Class functional API

```
L1 model.add(Dense(neurons, activation, input shape()))
:
Ln

model.compile(optimizer, loss)
```

mean squared REGRESSION

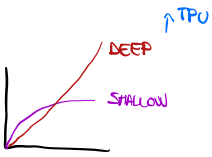
EX_2 CAR DATA not encoded

```
import keras as k
from k.models import Sequential
from k.layers import Dense
from k.utils import to_categorical

model = Sequential()
n_cols = cor_data.shape[1]
target = to_categorical(target)

model.add(Dense(neurons, activation, input_shape()))
model.add(Dense(5, "relu")
model.add(Dense(4, "softmax")   => 0 to 1 % output

model.compile(optimizer, loss, metrics=['accuracy'])
model.fit(predictors, target, epochs)
model.predict(test_data)   1 in 4
```

M4 Deep Learning

Shallow NN → INPUT VECTOR

Deep NN → large AMOUNT OF DATA



Convolutional NN (CNN)

↳ IMAGES AS INPUTS

IMAGE RECOGNITION, CV

CONV LAYERS (uses RELU)

↳ POOLING max for each section avg

```
model = Sequential
input_shape = (128,128,3)
model.add(Conv2D(#filters, kernel_block_size, strides, padding, activation)
        .(MaxPooling2D
        .(Conv2D(32
        .(MaxPooling2D
        .(Flatten
        .(Dense(100
        .(Dense
```

Recurrent NN (RNN)

input output before + new input

TEXT, GENOME, FINANCE

IMAGE GENERATION

Ex short term model

1	~~11~~
2	~~12~~
3	13
4	14
5	15
~~6~~	16
~~7~~	17
~~8~~	18
9	19
10	20

AUTO ENCODERS

data denoise
can't be reused
uses Backprop.
ends minimize

1. fixin imbalanced dataset
2. estimating missing values
3. automatic feature extraction

$a_{11}=e$ $\quad a_{21}$

$z_{11}=0.51$

$z_{22}=$

$a_{12}=e$ $\quad a_{22}$

$$\frac{\partial E}{w_5}=\frac{\partial E}{\partial a_{21}}$$

$$\left(-(t-a_{22})\right)\cdot\left(a_{22}(1-a_{22})\right)\cdot(a_{12})$$

PYTORCH BASICS

M1

TENSORS

DATA → TENSOR → NN inputs
vectors
array
matrix
w/ NumPy

torch.float32 .float
.float64 .double
.float16 .half
.int8
.uint8
.int16 .short
.int32 .int
.int64 .long
.bool

1D TENSORS

```
import torch
a = torch.tensor([0,1,2,3,4])
a.dtype   -> type of data
a.type    -> type of tensor
```

SPECIFIC TYPE tensor.FloatTensor
change TYPE a = a.type(torch.FloatTensor)

```
a.size
a.ndimension
```

convert 1D to 2D

[0 ... 6]
to [0 ... 6] a.view(6,1)

OPERATIONS

VECTOR ADDITIONS (same type) torch_1 + torch_2
VECTOR x SCALAR 3 * torch
PRODUCT 2 VECTORS torch_1 * torch_2
(dot product)
PRODOTTO VETTORIALE torch.dot(torch_1, torch_2)
ADD COSTANT torch_1 + #

APPLY FUCTIONS torch_1.mean()
.max()
UNIVERSAL FUNCTION .sin
.cos

tensor PYTORCH ⇄ array NUMPY

```
np_x = np.array([0 ... 5])
torch_x = torch.from_numpy(np_x)
back_to_np = torch_x.numpy()
```

torch.LINSPACE(start, end, steps

INDEXING

```
torch_x = torch.tensor([0 .. 5])
torch_x[0]=2   -> ([2 ... 5]
```

SLICING

↳ extract etc = torch_x[3:5]
↳ change torch_x[3:5] = torch.tensor([6,9,10])

2D TENSORS

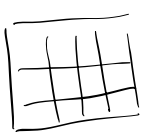

Matrix

Table & Images as grid


```
data=[[r1],[r2],[r3]]
tensor = torch.tensor(data)
  .shape()
  .size()
  .numel()
```

VECTOR
MATRIX MULT → mm(

DERIVATIVE as graph

↳ $y = x^2$ backward derivative of y

$z(x) = x^2 + 2x + 1$

$z(2) = 2^2 + 2(2) + 1 = 9$

$\frac{\partial z(x)}{\partial x} = 2x + 2$

$\frac{\partial z(2)}{\partial x} = 2(2) + 2 = 6$

```
x = torch.tensor(2, requires_grad=True)
z = x**2 + 2*x + 1
z.backward()
x.grad
```

data	2
grad_fn	None
grad	8
is_leaf	True
requires_grad	True

$\downarrow (\cdot)^2$

data	4
grad_fn	Pow Backward
grad	8
is_leaf	False
requires_grad	True

$f(u,v) = uv + u^2$

$\frac{\partial f(u,v)}{\partial u} = v + 2u$

$\frac{\partial f(u,v)}{\partial v} = u$

```
f(u=1, v=2) = uv + u^2
u = torch.tensor(1, requires_grad=True)
v = torch.tensor(2, requires_grad=True)
f = u * v + u**2
f.backward()
u.grad
v.grad
```

SIMPLE DATASET

```
class element_DS(dataset)
    def __init__(self, lenght, transform)

    def __getitem__(self, index)
        return

    def __len__(self)
        return self.len
```

METHODS REQUIRED for COMPATIBILITY
- __init__
- __getitem__
- __len__

APPLY multiples data Transform using Compose method

`transform.Compose([f1(), f2(), ..., fn()])`

DATASETS

```
from torch.utils.data import Dataset, Dataloader
```

ZALANDO FASHION

#	category	image
0	"t-shirt"	url_1

```
class Dataset(dataset)
    def __init__(self, csv_file, data_dir, transform=None):

    def __len__(self)
        return self.len

    def __getitem__(self, idx)
        load csv,
             image
             assign label
        return image, label
```

```
import torchvision.transforms as transforms
transforms.CenterCrop(20)
transforms.ToTensor()
croptensors_data =
    = transforms.Compose
        ([transforms.CenterCrop(20),
          transforms.ToTensor()]
dataset = Dataset(csv_file = file
                  data_dir = path
                  transform=)
```

USE KNOWN DATASETS

```
import torchvision.dataset as dsets

dataset = dsets.MNIST( root = './data', train = False, download = True, transform = transforms.ToTensor())
#convert image to tensor
```

12

LINEAR REGRESSION 1D

feature x
target y

$y = b + wx$ (b: BIAS, w: WEIGHT)

DATA → TRAIN → FORWARD

CLASS LINEAR

```
from torch.nn import Linear
model = Linear(in_features = 1, out_features = 1)
y = model(x)
```

sum to # parameters

CUSTOM MODULE


```
import torch.nn as nn
class LR(nn.Module)
    def __init__(self, in_size, out_size):
        super(LR, self).__init__()
        self.linear = nn.Linear(in_size, out_size)

    def forward(self, x)
        out = self.linear(x)
        return out
```

```
model = LR(1, 1)
# no need to load all data

model.state_dict()
# python dict  keys & values
```

TRAIN

$D = \{(x_1, y_1) \dots (x_m, y_m)\}$

always error → Gaussian (deviation from the line)
NOISE ASSUMPTION

↳ MINIMIZZO la FUNZIONE
mean squared error

LOSS distance from module estimate

loss

$y_1 \rightarrow \text{loss}(y_1, \hat{y}) \leftarrow \hat{y} = wx_1$

$\text{loss}(w) = (y_1 - \hat{y}_1)^2 = (y_1 - wx_1)^2$ ($\hat{y}_1$: estimate)

0 → GOOD
HIGH → BAD

MINIMIZE THE ERROR → slope
provides a number that helps estimate how good the prediction is.

? how to find best w?
↳ minimum

GRADIENT DESCENT (or BATCH G.D.)

↳ method find min of a function

considering loss function
Δlearning rate is the step
if: too big we can miss the minimum
if: too low → small

COST FUNCTION

↳ can be iterated

$$\frac{dl(w)}{dw} = -\frac{2}{N}\sum_{m=1}^{N}(y_m - wx_m)x_m$$

check if derivate is + or -
during training

all samples are called Batch

WE CAN STOP TUNING GRADIENT DESCENT

initial value is random → then tune

LINEAR REGRESSION Pytorch

x is torch → numpy → visualization w/ matplotlib

```
w = torch.tensor(-10.0, requires_grad=True)
x = torch.arange(start, stop, range).view(-1,1)
f = -3*x
```

forward → return w*x ≈ $\hat{y} = wx$

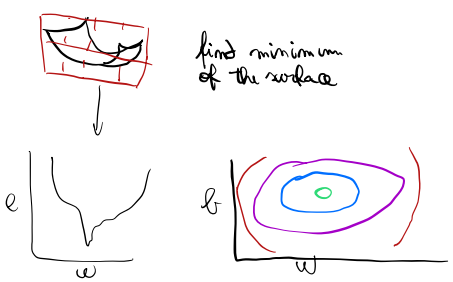
criterion torch.mean((yhat - y)**2) ≈ $\frac{1}{N}\sum_{n=1}^{N}(y_n - wx_n)^2$

```
COST=[]
going trough data = EPOCH
for epoch
  yhat = forward(x)
  loss = criterion(yhat, y)
  loss.backward
  w.data = w.data - η (STEP) * w.grad.data
  w.grad.data.zero_()
  COST.append(cost.item())      # displayable
```

GRADIENT points in the direction of the next iteration

TRAIN SLOPE & BIAS

$\ell(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - wx_n - b)^2$ → pytorch




```
epoch
yhat = forward
loss = criterion(yhat, y)
loss.backward()

w.data = w.data - η * w.grad.data
w.grad.data.zero_()

b.data = b.data - η * b.grad.data
b.grad.data.zero_()
```

OPTIMIZATION → Stochastic Gradient Descent

minimize 1 approximation at time

```
η =#
LOSS=[]
COST=[]
for epoch in range
   total=0
   for x,y in zip(x,y)
      yhat = forward(x)
      loss = criterion(yhat,y)
      loss.backward()
      w.data = w.data - η * w.grad.data
      b.data = w.data - η * b.grad.data
      w.grad.data.zero_()
      b.grad.data.zero_()
      LOSS.append(loss.item())
      total+= loss.item()
COST.append(total)
```

OPTIMIZATION → mini-Batch Descent

process larger data, we use few samples at time

? = # iterations = $\frac{\text{training size}}{\text{batch size}}$

OPTIMIZATION in PyTorch

DATASET

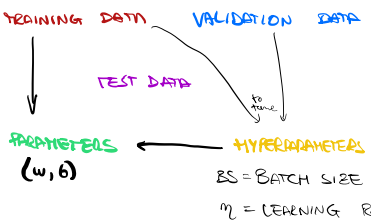
CUSTOM MODULE

COST MODULE →

```
model = LR(1,1)
from torch import nn, optim
optimizer = optim.SGD(model.parameters(), lr = η)
```

```
for epoch in range(100)
    for x,y in trainloader
        yhat = model(x)
        loss = criterion(yhat, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        w.data = w.data - η * w.grad.data
        b.data = w.data - η * b.grad.data
```

TRAINING, VALIDATIO & TEST SPLIT



we test multi models
↳ Then check where cost is less on choose that model

SAVING MODEL USING CHECKPOINTS

```
class Dataset
    train_data = Data()
    val_data = Data(train = False)
```

→ IBM_013

EARLY STOPPING

We save a model, based on cost results

```
DATASET
model
    TRAINING
    loss_val = criterion(model(val_data.x), val_data.y).item()
    if loss_val < min_loss
        value = epoch
        min_loss = loss_val
        torch.save(model.state_dict(), 'best_model.pt')
        # we save if value is less
```

MODEL + CHECKPOINT

```
checkpoint["epoch"]
"    ["model_state_dict"] = model.state_dict()
"    ["optimizer_state_dict"] = optimizer."()
"    ["loss"] = loss
torch.save(checkpoint, checkpoint_path)
```

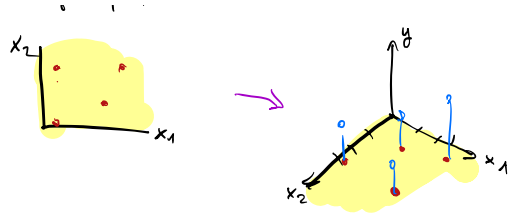
```
checkpoint = torch.load(checkpoint_path)
```

ML

MULTIPLE LINEAR REGRESSION (MLR)

$\hat{y} = b_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$

b_0 = bias
w_1 = weight of x_1



$X \cdot w \rightarrow$ DOT PRODUCT

2D TENSORS / MATRIX

$$\begin{bmatrix} x \end{bmatrix} \cdot \begin{bmatrix} w \end{bmatrix} + b$$

use CLASS LINEAR

```
from torch.nn import Linear
torch.manual_seed(1)
model = Linear(in_features=2, out_features=1)
list(model.parameters())
```

CUSTOM MODEL

```
import torch.nn as nn
class LR(nn.Module):
    def __init__(self, input_size, output_size):
        super(LR, self).__init__()   # create module without need to init
        self.linear = nn.Linear(input_size, output_size)
    def forward(self, x):
        out = self.linear(x)
        return(out)
```

MULTIPLE LINEAR REGRESSION

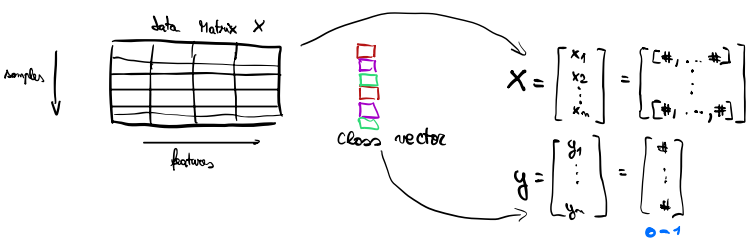
$$\hat{y}_1 = b_{01} + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{1d}x_d$$
$$\hat{y}_2 = b_{02} + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{2d}x_d$$

$$[\hat{y}_1 \ \hat{y}_2] = [x_1 \cdots x_d] \begin{bmatrix} w_{11} & w_{21} \\ \vdots & \vdots \\ w_{d1} & w_{d2} \end{bmatrix} + [b_{01} \ b_{02}]$$

DOT PRODUCT

M5

LINEAR CLASSIFIERS



if a dataset is separable by a line it's LINEAR SEPARABLE

LINEAR CLASSIFIER
↳ threshold functions

SIGMOID FUNCTION

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

LOGISTIC REGRESSION

$z = wx + b \rightarrow \sigma(z) \rightarrow \hat{y}$

MODEL — SIGMOID — PREDICTION

LOGISTIC FUNCTION

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

z = torch.arange(-100, 100, 0.1).view(-1, 1)   -> z [-100 ... 100]

sig = nn.Sigmoid()                              -> σ(z) = [0 ... 1]

yhat = sig(z)
```

build LOGISTIC REGRESSION

```
model = nn.Sequential(
            nn.Linear(1, 1),
            nn.Sigmoid())

yhat = model(x)
```

CUSTOM MODEL

```
class logistic_regression(nn.Module):
    def __init__(self, in_size, 1):
        super(logistic_reg...,self).__init__()
        self.linear = nn.Linear(in_size,1)
    def forward(self, x):
        x = torch.sigmoid(self.linear(x))
        return x
```

MAKING PREDICTIONS

torch.tensor([ ][ ])

yhat = model(x) ← predict

_019_

BERNOULLI DISTRIBUTION

θ = Bernoulli parameter

COIN FLIP

HEAD HEAD TAILS

$\theta \times \theta \times (1-\theta)$

$0.2 \times 0.2 \times 0.8) = 0.096$

	HEAD m=1	TAIL m=2	HEAD m=3	TAIL m=4	LIKELIHOOD
$\theta=0.5$	0.5	0.5	0.5	0.5	0.0625
$\theta=0.2$	0.2	0.8	0.2	0.8	0.0256

We can estimate the actual parameter by considering parameter values that maximise our likelihood

BERNOULLI DISTRIBUTION

$\theta = \text{argmax}\,(P(Y|\theta))$

each probability is a function

↳ $\ln(P(Y|\theta))$ EASY TO DEAL WITH

$l(\theta) = \ln(P(y|\theta)) = \sum_{m=1}^{N} y_m \ln(\theta) + (1-y_m)\ln(1-\theta)$

CROSS ENTROPY LOSS

loss function $l(\theta) = \frac{1}{N}\sum_{m=1}^{N}(y_m - \sigma(x_m+b))^2$ CE_{loss} no horizontal problem ∪

we SIGMOID

we TRESHOLD $l(\theta) = \sum_{m=1}^{N}(y_m - THR(x_m+b))^2$! horizontal line

MSE

MAY NOT REACH MINIMUM

MAXIMUM LIKELIHOOD

↳ obtain a parameter that maximise function

CROSS ENTROPY (COST)

$R(\theta) = -\frac{1}{N}\sum_{m=1}^{N} y_m \ln(\sigma(wx_m+b)) + (1-y_m)\ln(1-\sigma(wx_m+b))$

↑ pytorch implementation

```
def criterion(yhat, y):
    out = -1 * torch.mean(y * torch.log(yhat) + (1-y) * torch.log(1-yhat))
    return out
```

MEAN SQUARE ERROR

criterion = nn.MSELoss()

criterion = nn.BCELoss() for CROSS ENTROPY

LOGISTIC REGRESSION in pytorch

```
for epoch in range(100)
  for (x,y) in trainloader
    yhat = model(x)
    loss = criterion(yhat,y
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

DEEP LEARNING PYTORCH

M1

SOFTMAX input classification

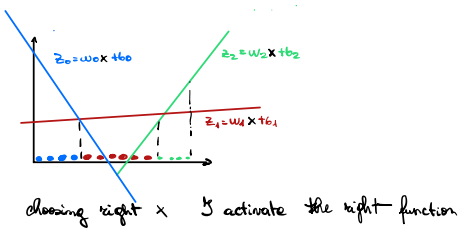
```
import torch.nn as nn
class Softmax(nn.Module):
    def __init__(self, in_size, out_size):
        super(Softmax, self).__init__()
        self.linear = nn.Linear(in_size, out_size)

    def forward(self, x):
        out = self.linear(x)
        return out
```

```
from torch.nn import Linear
torch.manual_seed(1)

model = Softmax(2,3)
x = torch.tensor([1,0],[2,0])
z = model(x)
yhat = z.max(1)
```

MULTIDIM INPUT

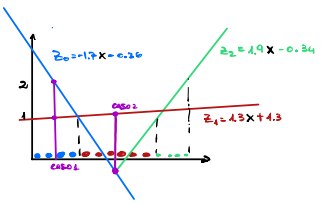
```
X = torch.tensor([1.0,1.0],[1.0,2.0],[1.0,3.0])
z = model(x)
yhat = z.max(1)
```

SOFTMAX USING LINES TO CLASSIFY DATA



choosing right x & activate the right function

IN USE ⇒



case 1

i	z_0	z_1	z_2
	0	1	2
	2.33	1.15	-3.3

$\hat{y} = \text{argmax}_i \{z_i\}$

$\hat{y} = 0$

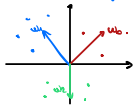
case 2

i	z_0	z_1	z_2
	0	1	2
	-1.8	1.1	-1.8

$\hat{y} = \text{argmax}_i \{z_i\}$

$\hat{y} = 1$

MNIST 2D SOFTMAX



i	w	$z_i = wx$
1	$\left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	
2	$\left[\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	
3	$[0, -1]$	

extract i by getting higher value

CROSSENTROPY LOSS

QUIZ

given an array of tensors

yhat = z.max(1)

↳ extract for each array index of max value

Softmax(input_features, #_of_classes)

SOFTMAX PyTorch

transforms.ToTensor()

↳ CONVERTS THE IMAGE TO A TENSOR

x.view(-1, 28*28)

↳ IMAGE ARE SQUARE & WE HAVE TO CONVERT THEM TO VECTOR

image is 10 x 10, converting to vector

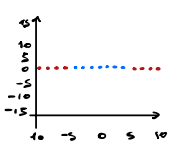
how many elements does it have?

↳ 100

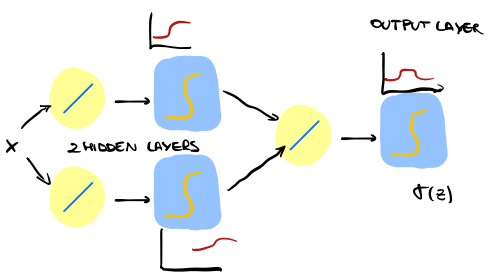
NEURAL NETWORK

is a function that can be used to approximate most functions

BUILD nn WITH LINEAR CLASSIFIER



NN ⇒



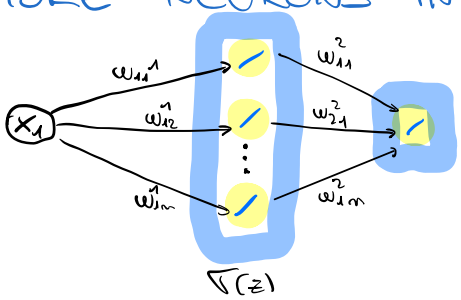
PYTORCH IMPLEMENTATION

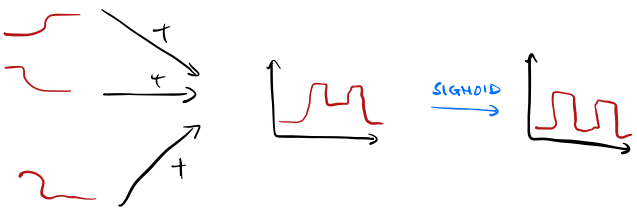
```
import torch
import torch.nn as nn
from torch import sigmoid
```

```
class Net(nn.Module)
  def __init__(self, D_in, H, D_out)
    super(Net, self).__init__()
    self.linear1 = nn.Linear(D_in, H)
    self.linear2 = nn.Linear(H, D_out)
  def forward(self, x):
    x = sigmoid(self.linear1(x))
    x = sigmoid(self.linear2(x))
    return x
```

```
model = Net(1,2,1)
x = torch.tensor([0.0])
yhat = model(x)
```

MORE NEURONS IN HIDDEN LAYER





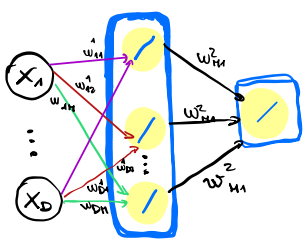
↳ CODE

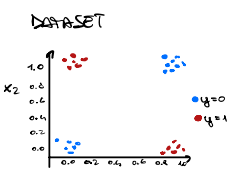
```
model = Net(1,6,1)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
train(data_set, criterion, train_loader, epochs=1000)
```

→

```
model = torch.nn.Sequential(
    torch.nn.Linear(1, 7)
    torch.nn.Sigmoid()
    torch.nn.Linear(7,1)
    torch.nn.Sigmoid()
)
```

NEURAL NETWORKS WITH MULTIPLE DIMENSION INPUT





HOW TO SEPARATE CLASSES?

NON LINEAR FUNCTION

↳ CODE

```
import torch
import torch.nn as nn
from torch import sigmoid
from torch.utils.data import Dataset, DataLoader
```

```
class XOR_Data(Dataset):

    # Constructor
    def __init__(self, N_s=100):
        self.x = torch.zeros((N_s, 2))
        self.y = torch.zeros((N_s, 1))
        for i in range(N_s // 4):
            self.x[i, :] = torch.Tensor([0.0, 0.0])
            self.y[i, 0] = torch.Tensor([0.0])

            self.x[i + N_s // 4, :] = torch.Tensor([0.0, 1.0])
            self.y[i + N_s // 4, 0] = torch.Tensor([1.0])

            self.x[i + N_s // 2, :] = torch.Tensor([1.0, 0.0])
            self.y[i + N_s // 2, 0] = torch.Tensor([1.0])

            self.x[i + 3 * N_s // 4, :] = torch.Tensor([1.0, 1.0])
            self.y[i + 3 * N_s // 4, 0] = torch.Tensor([0.0])

            self.x = self.x + 0.01 * torch.randn((N_s, 2))
        self.len = N_s

    # Getter
    def __getitem__(self, index):
        return self.x[index],self.y[index]

    # Get Length
    def __len__(self):
        return self.len
```

```
def train(data_set, model, criterion, train_loader, optimizer, epochs)
  COST = []
  ACC = []
  for epoch in range(epochs):
    total = 0
    for x, y in train_loader:
      optimizer.zero_grad()
      yhat = model(x)
      loss = criterion(yhat, y)
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

```
#model Creation

class Net(nn.Module):
    def __init__(self, D_in, H, D_out)
        super(Net, self).__init__()
        #hidden layer
        self.linear1 = nn.Linear(D_in, H)
        #output layer
        self.linear2 = nn.Linear(H, D_out)

    def forward(self, x):
        x = torch.sigmoid(self.linear1(x))
        x = torch.sigmoid(self.linear2(x))
        return x
```

```
    #cumulative loss
    total += loss.item()
  ACC.append(accuracy(model,data_set))
  COST.append(total)
return COST
```

TRAINING PROCESS

```
criterion = nn.BCELoss()
data_set = XOR_DATA()
train_loader = DataLoader(dataset=data_set, batch_size=1)
```

```
model = Net(2,4,1)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
train(data_set, criterion, train_loader, optimizer, epochs=500)
```

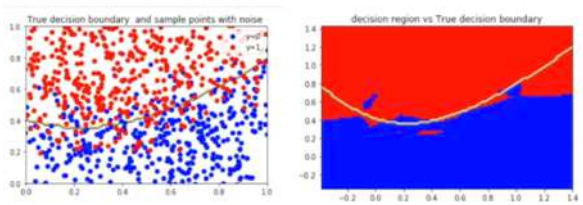
OVERFITTING

model too complex for the data
→ TOO MANY NEURONS IN THE HIDDEN LAYER

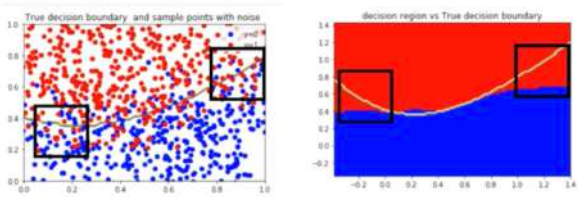
UNDERFITTING

model cannot capture the complexity of the data
→ TOO FEW NEURONS

Example of Overfitting



Example of Underfitting

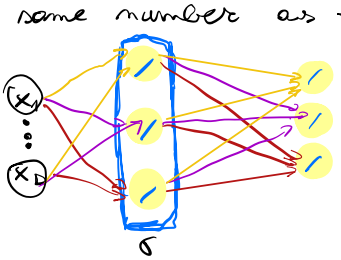


↳ SOLUTION

- Use VALIDATION DATA TO DETERMINE OPTIMUM # of NEURONS
- GET MORE DATA
- regularization: DROPOUT

MULTI CLASS NEURAL NETWORK

same number as # of classes



3 classes
⇐ 3 OUTPUT NEURONS

in PyTorch

```
class Net(nn.Module):
    # Constructor
    def __init__(self,D_in,H,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)

    # Prediction
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x))
        x=self.linear2(x)
    return x
```

With Sequential

```
model = torch.nn.Sequential(
torch.nn.Linear(input_dim,hidden_dim),
 torch.nn.Sigmoid(),
torch.nn.Linear(hidden_dim,output_dim))
```

Quiz

1. correct cost function for Multi class NN? → nn.CrossEntropyLoss()
2. What cost function can be used for a 2 class problem? → nn.BCELoss()
3. The prediction step in a multi-class network utilizes the same procedure as softmax function? → True

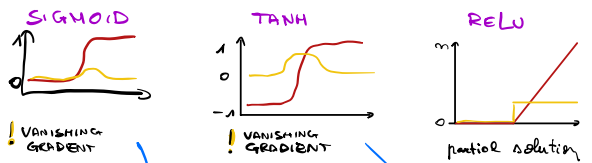
BACKPROPAGATION

reduces computation

USES THE DERIVATIVE OF THE 1st PARAMETER IN THE OUTPUT LAYER TO HELP CALCULATE THE PARAMETER OF THE NEXT LAYER.

! VANISHING GRADIENT → SOLUTION change activation function or optimization methods 2.6

ACTIVATION FUNCTIONS



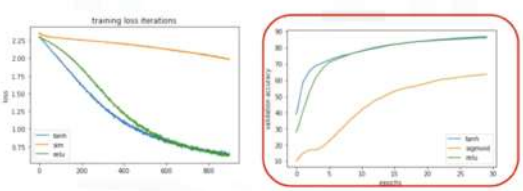

```
class Net(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x)))
        x=self.linear2(x)
    return x
```

```
class Net_Tanh(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(Net_Tanh,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.tanh (self.linear1(x)))
        x=self.linear2(x)
    return x
```

```
class NetRelu(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(NetRelu,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.relu(self.linear1(x)))
        x=self.linear2(x)
    return x
```

```
model_Tanh=nn.Sequential( torch.nn.Linear(input_dim, hidden_dim),
nn.Tanh(),nn.Linear(hidden_dim, output_dim),)

modelRelu=torch.nn.Sequential( torch.nn.Linear(input_dim,hidden_dim),
nn.ReLU(),nn.Linear(hidden_dim, output_dim),)
```

Results MNIST

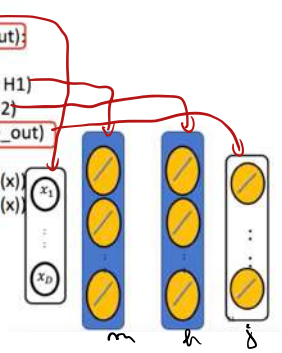


M3

DEEP NEURAL NETWORKS

multi hidden layers

```
class Net(nn.Module):
    def __init__(self,D_in,H1,H2,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in, H1)
        self.linear2=nn.Linear(H1, H2)
        self.linear3=nn.Linear(H2, D_out)
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x))
        x=torch.sigmoid(self.linear2(x))
        x=self.linear3(x)
        return x
```



model = Net (D_in = D, H1 = m , H2 = h , D_OUT = j)

AUTOMATE with nn.ModuleList()

↳ CODE

Layers = [2, 3, 4, 3]

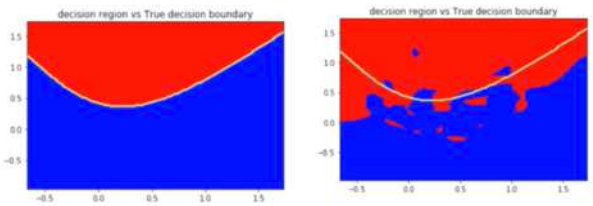
(2: feature size; 3: # neurons 1st layer; 4: # of neurons 2nd layer; 3: # of classes for output)

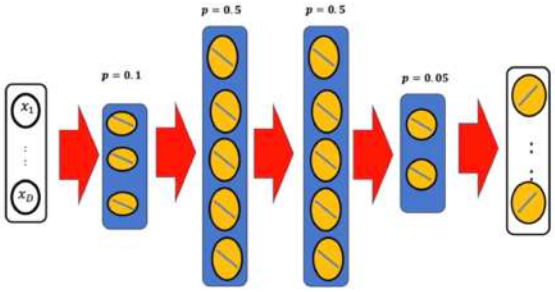
With Sequential

```
model = torch.nn.Sequential(
torch.nn.Linear(input_dim,hidden_dim1),  torch.nn.Sigmoid(),
torch.nn.Linear(hidden_dim1, hidden_dim2), torch.nn.Sigmoid(),
torch.nn.Linear(hidden_dim2,output_dim1) )
```

DROPOUT

$r \sim Ber(1-p)$



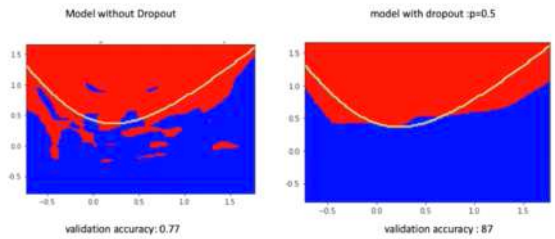


p is an Hyperparameter

less neurons lower the P value

higher P = 1 removes all neurons

EVALUATION




```
class Net(nn.Module):
    def __init__(self,in_size,n_hidden,out_size,p=0):
        super(Net,self).__init__()
        self.drop=nn.Dropout(p=p)
        self.linear1=nn.Linear(in_size,n_hidden)
        self.linear2=nn.Linear(n_hidden,n_hidden)
        self.linear3=nn.Linear(n_hidden,out_size)
    def forward(self,x):
        x=torch.relu(self.linear1(x))
        x=self.drop(x)
        x=torch.relu(self.linear2(x))
        x=self.drop(x)
        x=self.linear3(x)
        return x
```

With Sequential

```
model= torch.nn.Sequential(torch.nn.Linear(1,
10),
 torch.nn.Dropout(0.5),
torch.nn.ReLU(),
torch.nn.Linear(10, 12),
torch.nn.Dropout(0.5),
torch.nn.ReLU(),
torch.nn.Linear(12, 1),
```

```
model_drop = Net(2, 300, 2, p=0.5)
model_drop.train()
optimizer = torch.optim.ADAM(model_drop.parameters(),lr=0.01)
```

! model.eval() NOT USES DROPOUT

QUIZ

1. In what situation would's you use dropout for classification? if training accuracy much longer test accuracy
2. What is the purpose of using dropout? Reduce the impact of noise or overfitting
3. Prediction w model w dropout → model.eval() yhat = model(x)
4. 40% of activations shut off → nn.Dropout(0.4)

NEURAL NETWORK INITIALIZATION WEIGHTS

What happen when the weights in the same layer have the same values

WHEN WE DON'T INITIALIZE THE WEIGHTS PYTORCH RANDOMIZE FOR US

HOW TO FIX THE PROBLEM OF VANISHING GRADIENTS DUE TO A LARGE NUMBER OF NEURONS
→ Scale the width of the distribution by the inverse of the number of neurons. $\frac{1}{\#\text{ OF NEURONS}}$

DIFFERENT INITIALIZATION in PyTorch using Default Method

DEFAULT in PyTorch $\pm \frac{1}{\sqrt{L_{in}}}$ using `nn.Linear(input_size, output_size)`

XAVIER method → $\frac{\sqrt{6}}{\pm\sqrt{L_{out}+L_{in}}}$ → `nn.Linear(input_size, output_size)`
`torch.nn.init.xavier_uniform(linear.weight)`

HE method → `linear = nn.Linear(input_size, output_size)`
`torch.nn.init.kaiming_uniform_(linear.weight, nonlinearity='relu')`

QUIZ

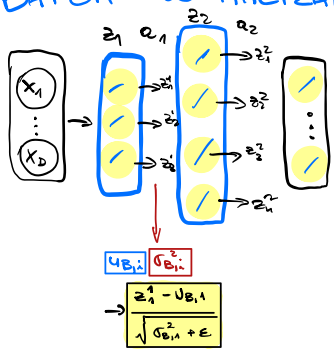
4. Why do we randomly initialize our parameters, instead of setting all the parameters to the same value?
Each neuron will have the same output and such have the same gradient update

GRADIENT DESCENT with MOMENTUM

High velocity to go out of flat zone
↳ allow sometimes to go to the local minima faster

CODE: `torch.optim.SDG(model.parameters(), lr=0.01, momentum = *)`

BATCH NORMALIZATION



↳ It normalizes the distribution of inputs of each layer making the network less dependent on specific neuron weights, It operates on each batch of data. IMPROVE SPEED, PERFORMANCE & STABILITY OF NN

```
class NetBatchNorm(nn.Module):
  def __init__(self,in_size,n_hidden1,n_hidden2,out_size):

    super(NetBatchNorm,self).__init__()

    self.linear1=nn.Linear(in_size,n_hidden1)
    self.linear2=nn.Linear(n_hidden1,n_hidden2)
    self.linear3=nn.Linear(n_hidden2,out_size)

    self.bn1 = nn.BatchNorm1d(n_hidden1)
    self.bn2 = nn.BatchNorm1d(n_hidden2)

def forward(self,x):
    x=F.sigmoid(self.bn1( self.linear1(x)))
    x=F.sigmoid(self.bn2( self.linear2(x)))
    x=self.linear3(x)
    return x
```

M4

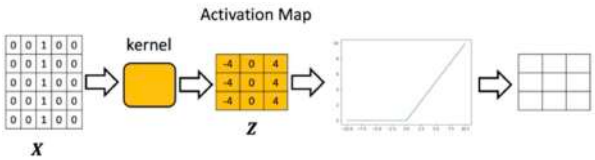
CNN (CONVOLUTIONAL NEURAL NETWORK)

Looking at relative positions of the pixels rather than absolute positions.
Using the image and something called, the KERNEL, we are going to perform a convolution
The result is called an activation map.

↳ CODE

`conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, stride=2)`
(H-W)/ stride +1
`image = torch.zeros(1,1,5,5)` (images in our tensor, # of channels)
`image[0,0,:,2] = 1`
`z = conv(image)`

ACTIVATION FUNCTIONS & MAX POOLING → will reduce the size of the activation maps reducing the number of parameters



MULTIPLE INPUT & OUTPUT CHANNELS

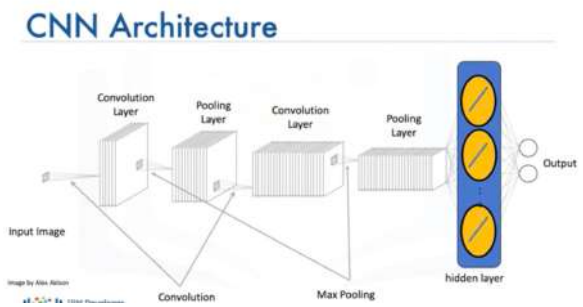
CASE: MULTIPLE OUTPUT

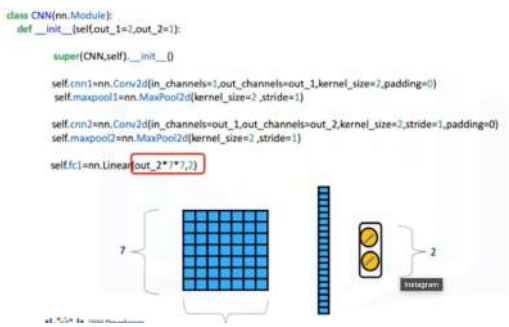
```
conv1 = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=3)
image1 = torch.zeros(1,1,5,5)
image1[0,0,:,2] = 1
z = conv1(image1)
```

mini BATCH x Channels

Kernels = in * out _channels

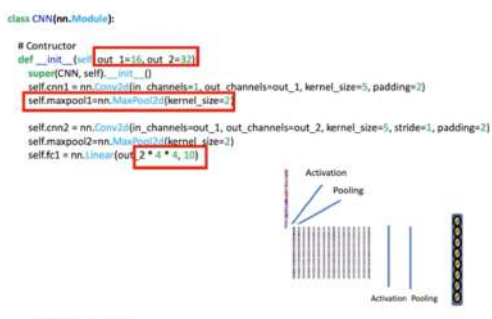
CONVOLUTIONAL NEURAL NETWORK



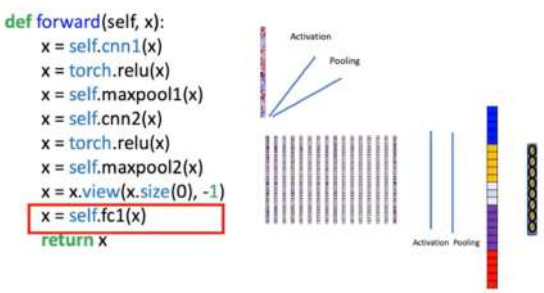


```
class CNN(nn.Module):
  def __init__(self,out_1=2,out_2=1):
      super(CNN,self).__init__()
      self.cnn1=nn.Conv2d(in_channels=1,out_channels=out_1,kernel_size=2,padding=0)
      self.maxpool1=nn.MaxPool2d(kernel_size=2 ,stride=1)
      self.cnn2=nn.Conv2d(in_channels=out_1,out_channels=out_2,kernel_size=2,stride=1,padding=0)
      self.maxpool2=nn.MaxPool2d(kernel_size=2 ,stride=1)
      self.fc1=nn.Linear(out_2*7*7,2)
```


→ FORWARD METHOD



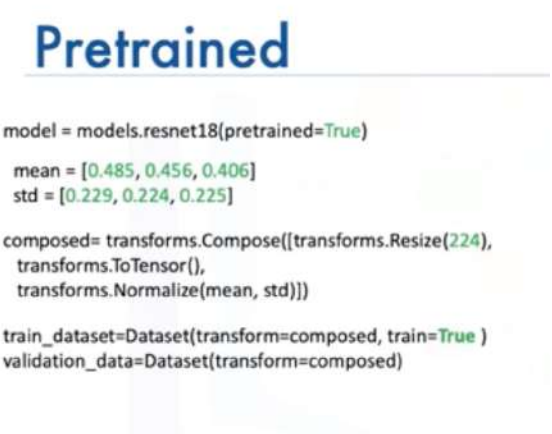
```
class CNN(nn.Module):
  # Contructor
  def __init__(self, out_1=16, out_2=32):
    super(CNN, self).__init__()
    self.cnn1 = nn.Conv2d(in_channels=1, out_channels=out_1, kernel_size=5, padding=2)
    self.maxpool1=nn.MaxPool2d(kernel_size=2)
    self.cnn2 = nn.Conv2d(in_channels=out_1, out_channels=out_2, kernel_size=5, stride=1, padding=2)
    self.maxpool2=nn.MaxPool2d(kernel_size=2)
    self.fc1 = nn.Linear(out_2 * 4 * 4, 10)
```

```
def forward(self, x):
    x = self.cnn1(x)
    x = torch.relu(x)
    x = self.maxpool1(x)
    x = self.cnn2(x)
    x = torch.relu(x)
    x = self.maxpool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    return x
```


TORCH VISION MODELS

ResNet18 use pretrained models



```
Pretrained

model = models.resnet18(pretrained=True)

mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]

composed= transforms.Compose([transforms.Resize(224),
  transforms.ToTensor(),
  transforms.Normalize(mean, std)])

train_dataset=Dataset(transform=composed, train=True )
validation_data=Dataset(transform=composed)
```

PYTORCH

GPU

```
GPUs and Tensors

torch.tensor([1, 2, 3, 4])

torch.tensor([1, 2, 3, 4]).to(device)

tensor([1, 2, 3, 4], device='cuda:0')
```

```
Creating the CNN

model = CNN()
model.to(device)

Training

for epoch in range(num_epochs):
    for features, labels in train_loader:
        features, labels = features.to(device), labels.to(device)
        optimizer.zero_grad()
        predictions = model(features)
        loss = criterion(predictions, labels)
        loss.backward()
        optimizer.step()
```

45

AUTOENCODERS & MATRICES

MATRIX TRANSFORMATION

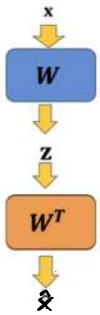
encoder & decoder are Matrix

$x \rightarrow W \rightarrow z \rightarrow W^{-1} \rightarrow x$

encoder code decoder

```
class AutoEncoder(nn.Module):
  def __init__(self, input_dim=2, encoding_dim=2):
    super(AutoEncoder, self).__init__()
    self.encoder = nn.Linear(input_dim,encoding_dim,bias=False)
    self.decoder = nn.Linear(encoding_dim,input_dim,bias=False)
  def forward(self, x):
    x=self.encoder(x)
    x=self.decoder(x)
    return x
```


```
auto_encoder_2Dcode=AutoEncoder(2,1)

W=torch.tensor([[1/2**(0.5),1/2**(0.5)]])
auto_encoder_2Dcode.state_dict()['encoder.weight'].data[:,:]=W
auto_encoder_2Dcode.state_dict()['decoder.weight'].data[:,:]=torch.transpose(W,0,1)

z=auto_encoder_1Dcode.encoder(torch.tensor([[1.0,1.0]]))
z:tensor([[0.3563]], grad_fn=<MmBackward>)

x_hat=auto_encoder_1Dcode.decoder(z)
x_hat:tensor([[-0.1929, -0.1561]], grad_fn=<MmBackward>)

x_hat=auto_encoder_1Dcode(x)
x_hat:tensor([0.1929, 0.1561], grad_fn=<SqueezeBackward3>)
```



PCA Principal components analysis

$$X = \begin{bmatrix} [x_1] \\ \vdots \\ [x_m] \end{bmatrix}$$

$X = X - X.mean(dim=0)$

$X = (X - \mu) D^{-1}$

CODE: (Before training network)

```
standard = preprocessing.StandardScaler
x_train = standard.fit_transform(x_train)
x_test = standard.transform(x_test)
```

PCA to AUTOENCODERS

input (D) code (E) output

SHALLOW AUTOENCODERS

DEEP AUTO ENCODERS

more than 1 layer of activation function

→ ANOMALY DETECTION

TENSORFOW

11

Data flow graph

NODES = Mathematical Operations

EDGES = Multi-Dimensional Array (tensors)

build -> execute

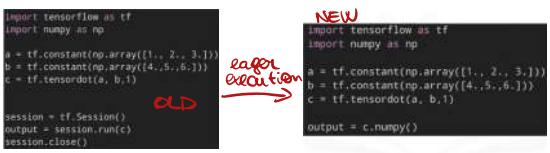
Tensor multidimensional array

placeholders created before usage

TF 2.x & EAGER EXECUTION

Keras is high level API

Performance OPT -> GPU

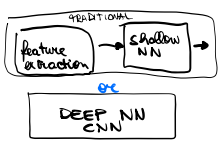


DEEP LEARNING

DEEP NEURAL NETWORKS

CNN -> Image & Learn

└ OBJECT DETECTION



RNN -> Recurrent Neural Network

-> sentiment
-> next word suggestion
-> translation
-> speech to text

AUTOENCODER

-> UNSUPERVISED
dimensionality reduction
feature extraction
image recognition

RBM Restricted Boltzmann Machines

-> RECONSTRUCT DATA
-> feature extraction / learning
-> dimensionality reduction
-> pattern recognition
-> recommender systems
-> handling missing values
-> topic modelling

DBN Deep Belief Network

-> BACKPROP. PROBLEM
-> CLASSIFICATION very ACCURATE SMALL DATASET

12

CNN

Key features

- Detect & classify
- Independence from POSE, SCALE, ILLUMINATION, CONFORMATION & CLUTTER

SEQUENCE OF STEPS

input image -> find primitive features -> detect object parts -> predict element

CNN FOR CLASSIFICATION

feature extraction, feature selection

IMG -> AUTOMATIC feature extractor feature selector -> dog? cat?

SHALLOW NN -> multiple CNN

CONVOLUTIONAL POOLING LAYER FULLY CONNECTED LAYER

CNN ARCHITECTURE

[1] CONVOLUTIONAL LAYER

↳ detect pattern & features from img Ex: EDGES

MATH → convolution W/Kernel

USING EXISTING FILTERS

(edges, curves)

tf.nn.relu(neuron)

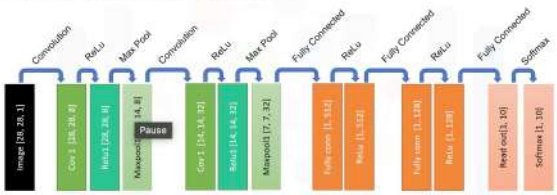
CODE

INPUT IMG as MATRIX [WxH] → Kernels #K → neurons [WxH]xK → ACTIVATION ReLU → MAX POOLING simplifies the input → FLATTEN → FULLY CONNECTED → READ OUT → SOFTMAX → $p_0 = 0.001$ ⋮ $p_9 = 0.99$

CONVOLUTION

STRIDES sliding parameter → [WxH] x K

CNN architecture

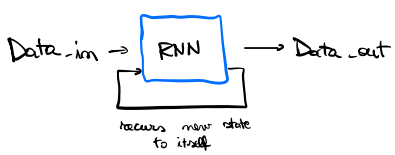


M3

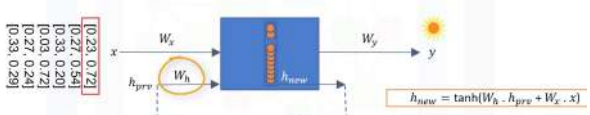
THE SEQUENTIAL PROBLEM

→ not handled well by traditional NN

- We want to have some MEMORY



The Recurrent Model

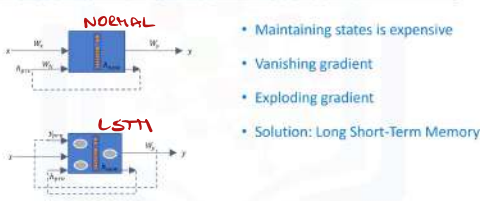


ISSUES

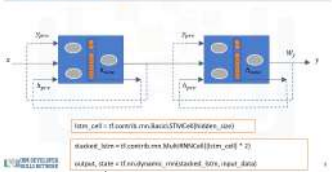
- Must remember all states at any given time
  - computationally expensive
  - only store states within a time window
- Sensitive to changes in their parameters
- Vanishing gradient (to 0)
- Exploding Gradient (exp to ∞)

LSTM (Long Short Term Model)

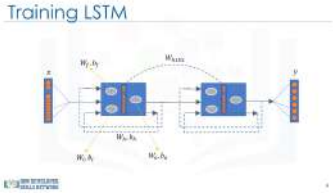
Recurrent Network Problems



Stacked LSTM

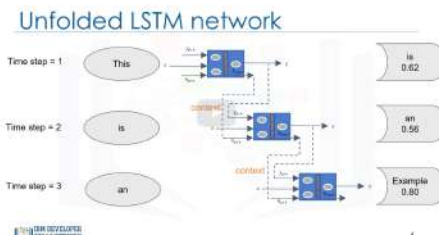


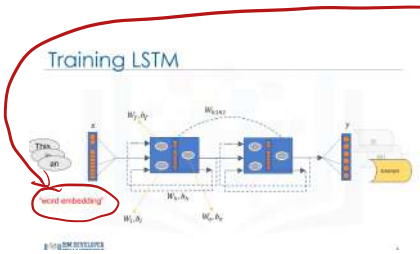
Training LSTM

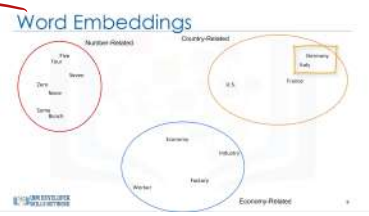


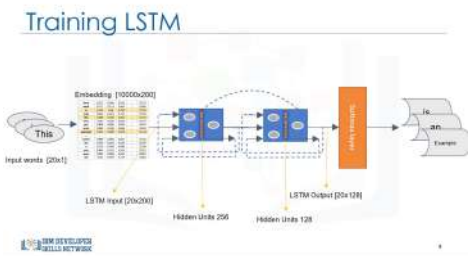
↳ CODE lstm_cell = tf.contrib.rnn.BasicLSTMCell(hidden_size)

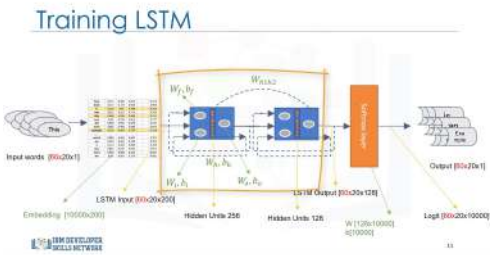
APPLYING RNN TO LANGUAGE MODELS





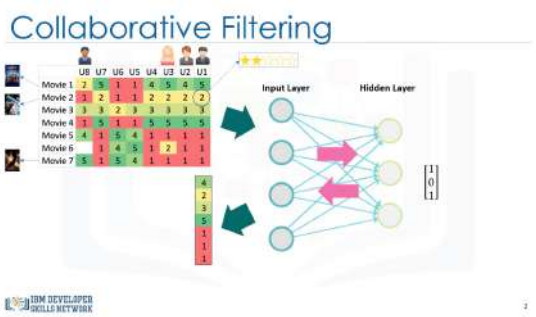


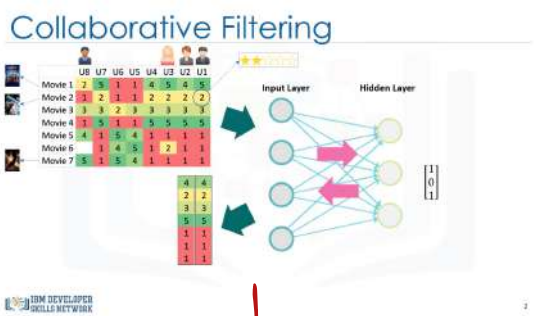


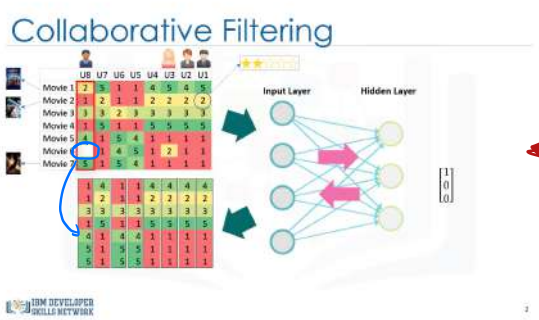


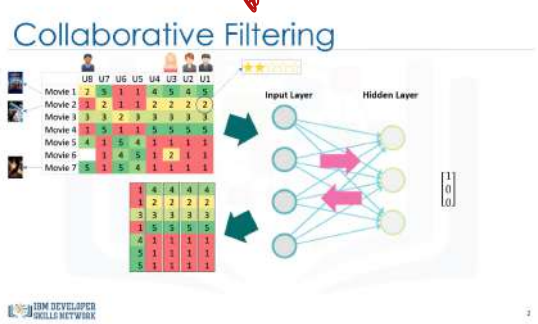
M4

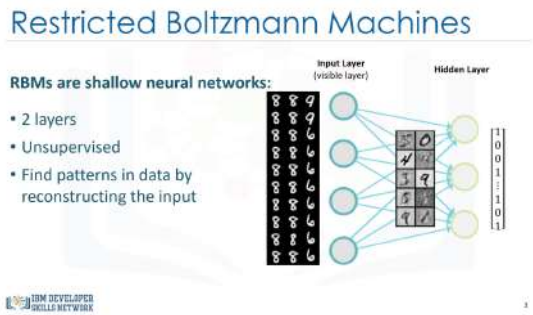
RBM



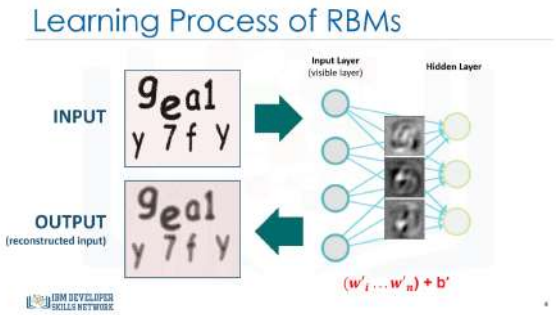






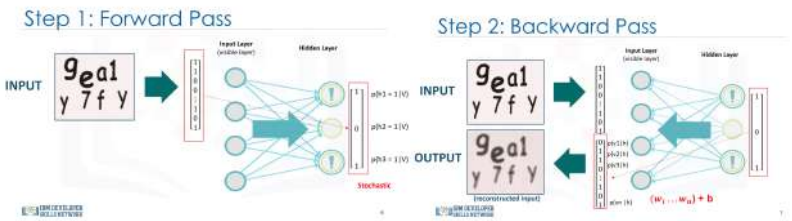


USAGE:
- dimensionality reduction
- Feature extraction
- Collaborative filtering
- main block of DBN



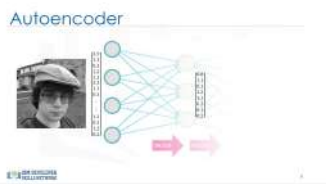
ADVANTAGES of RBM

- good at handling UNLABELED data
- RBM extract important features from input
- RBM are more efficient at dimensionality reduction than PCA.



MS

AUTOENCODERS



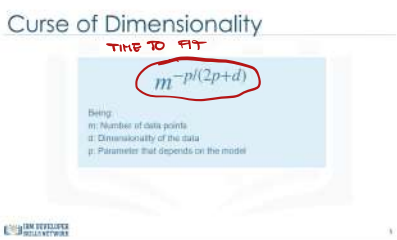
→ detect KEY FEATURES
→ data compression
→ learning generative models of data
→ Dimensionality reduction

HIGH DIMENSION #
time to fit increase exp
→ SPARSITY
→ OVERLAP

also use
→ PCA dimensionality reduction

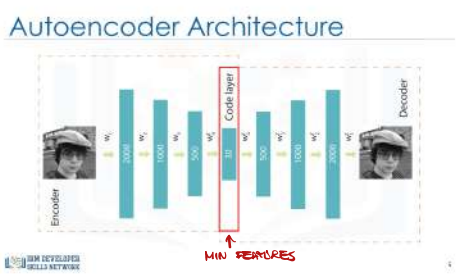
PCA << AUTOENCODER

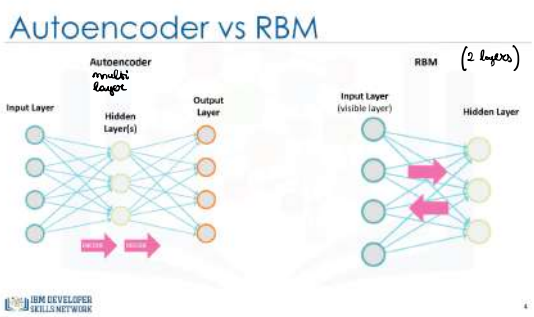
AUTOENCODER GIVES SEPARABILITY
→ clustering algos available

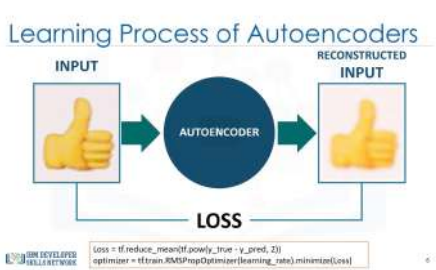


AUTOENCODER STRUCTURE

The main goal is to take unlabeled inputs encode them, and then try to reconstruct them afterwards based on the most valuable features identified in the data.

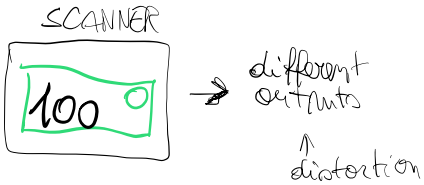


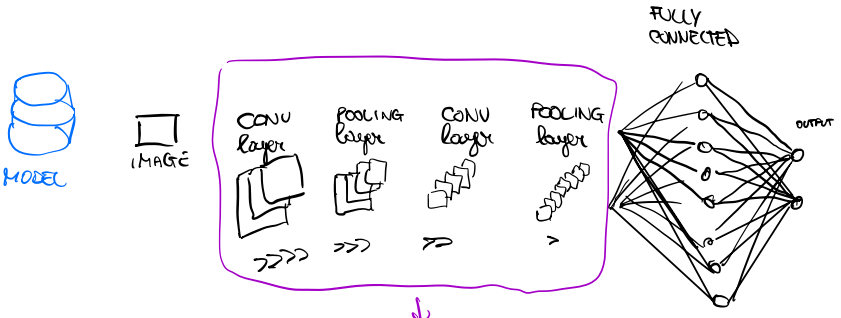




CAPSTONE - ATM

denomination	y
5	0
10	1
20	2
50	3
100	4
200	5
500	6





↓
I can use a PRETRAINED MODEL

1. Loading Data

resources/data / training_data_pytorch 0-69
/ validation_data_pytorch
/ test_data_pytorch

```
import matplotlib.pyplot as plt
from PIL import Image

train_dir = "/resources/data/training_data_pytorch"
name = "0.jpeg"

input = train_dir + name
img = Image.open(input)
plt.imshow(img)
plt.show()
```

2. Image Preprocessing

Given CSV file

*	TYPE SOURCE	PATH DENOMINATION	y CLASS
0	5	0.jpeg	0
1	5	1.jpeg	0
2	10	2.jpeg	1
⋮	10	3.jpeg	1
n	500	n.jpeg	5

```
class Dataset(Dataset):
    def __init__(self, csv_file, data_dir, transform=None):
        Load
        • img file name
        • class of each sample
        • directories of image
    def __getitem__(self, idx):
        Load image files
        Output image file
        Load Class Name
```

__INIT__

```
csv_file = "name_of_csv.csv"
data_name = pd.read_csv(csv_file)
self.data_name

data_dir = "/ker/data/train.../val.../test"
self.data_dir
```

__GET_ITEM__

```
file name: train_data_name.iloc[0,2]
y: data_name.iloc[0,3]

img_name = data_dir + data_name.iloc[0,2]
image = Image.open(img_name)
y = data_name.iloc[0,2]
if self.transform:
    image = self.transform(image)
return image, y
```

3 TRAINING WITH PyTORCH

We use ResNET 18

last layer & OUTPUT layer

↳ CODE

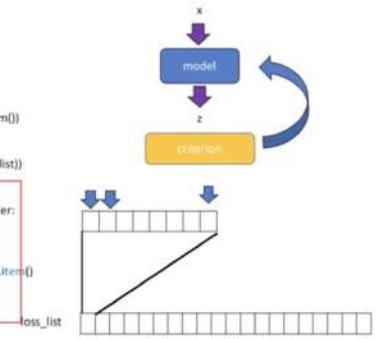
```
model = models.resnet18(pretrained=True)
mean = [#, ... #]
std = [#, ..., #]

transform_stuff = transforms.Compose(
                                    transforms.Resize(224),
                                    transforms.ToTensor
                                    transforms.Normalize(mean, std))
train_dataset = Dataset(transform=composed, csv_file, data_dir=train...)
validation_data = Dataset(transform=composed, csv_file, data_dir=validat....)
```

```
for param in model.parameters()
    param.requires_grad = False

model.fc = nn.Linear(521, 7)
                     NEURONS  CLASSES
```

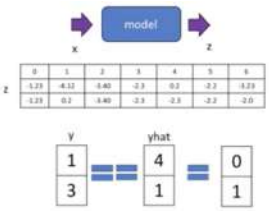
```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=15)
validation_loader = torch.utils.data.DATALOADER(dataset=train_dataset, batch_size=15)
shuffle = True
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam([param.. for param in model.param() if param, lr=0.003)
N_EPOCHS=100
loss_list=[]
accuracy_list=[]
correct=0
n_test=len(validation_dataset)
```

```
for epoch in range(n_epochs):
    loss_sublist = []
    for x, y in train_loader:
        model.train()
        optimizer.zero_grad()
        z = model(x)
        loss = criterion(z, y)
        loss_sublist.append(loss.data.item())
        loss.backward()
        optimizer.step()
    loss_list.append(np.mean(loss_sublist))
```

```
correct = 0
for x_test, y_test in validation_loader:
    model.eval()
    z = model(x_test)
    _, yhat = torch.max(z.data, 1)
    correct += (yhat == y_test).sum().item()
accuracy = correct / n_test
accuracy_list.append(accuracy)
```




```
correct = 0
for x_test, y_test in validation_loader:

    model.eval()
    z = model(x_test)

    _, yhat = torch.max(z.data, 1)

    correct += (yhat == y_test).sum().item()

accuracy = correct / n_test
accuracy_list.append(accuracy)
```



Manuel Acosta

13.7.2023