

DEEP LEARNING MIT 6.S191

MAR 20 2021

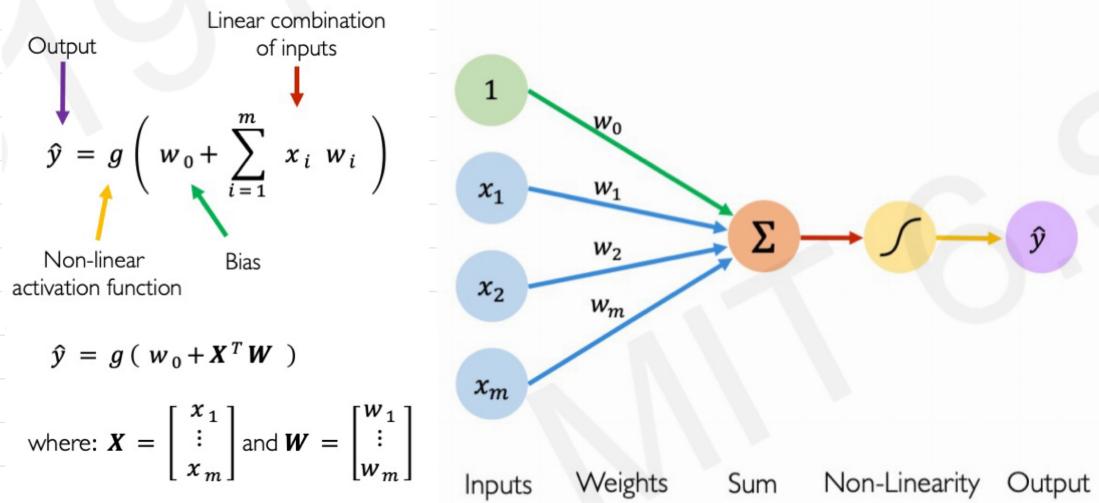


# L1-INTRO To DL

## BUILDING BLOCKS

### PERCEPTRON

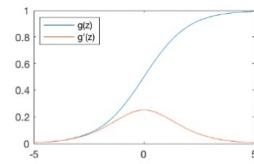
#### FORWARD PROPAG.



### ACTIVATION FUNCTION

- Funzione (non lineare) che mappa gli output in un dato range

Sigmoid Function

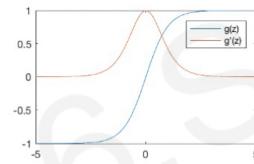


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

`tf.math.sigmoid(z)`

Hyperbolic Tangent

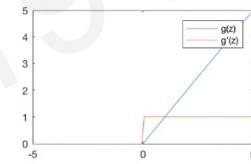


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



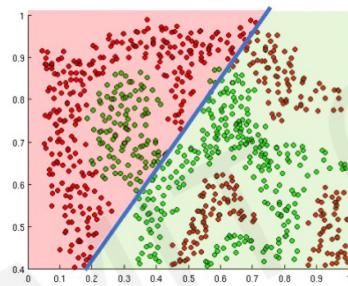
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

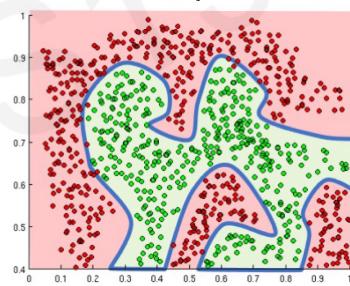
`tf.nn.relu(z)`

- La NON-LINEARITY ha lo scopo pratico di approssimare funzioni complesse

↳ Ad esempio per separare i punti colorati

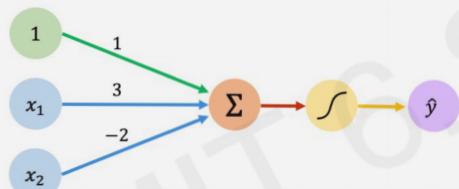


Linear activation functions produce linear decisions no matter the network size

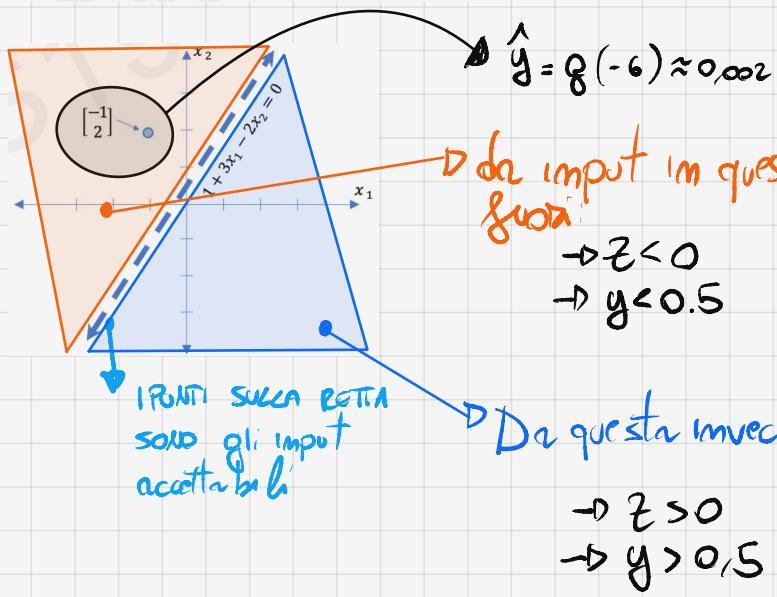


Non-linearities allow us to approximate arbitrarily complex functions

esempio



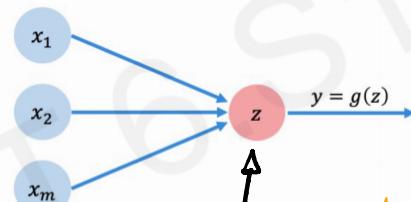
$$\hat{y} = g(1 + 3x_1 - 2x_2) \quad \text{2D}$$



From perceptron  
to Neural Net

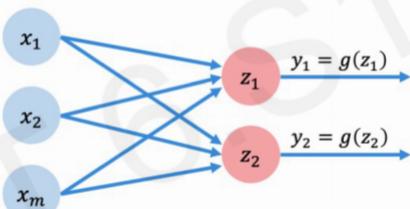
DENSE LAYER

• PERCEP. SEMPLIFICATE



Defn forward prop.

• MULTI-OUTPUT



→ Detto DENSE LAYER perché ogni  $z_i$  è compreso a ogni  $x_s$

In tensorflow

CLASS

```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

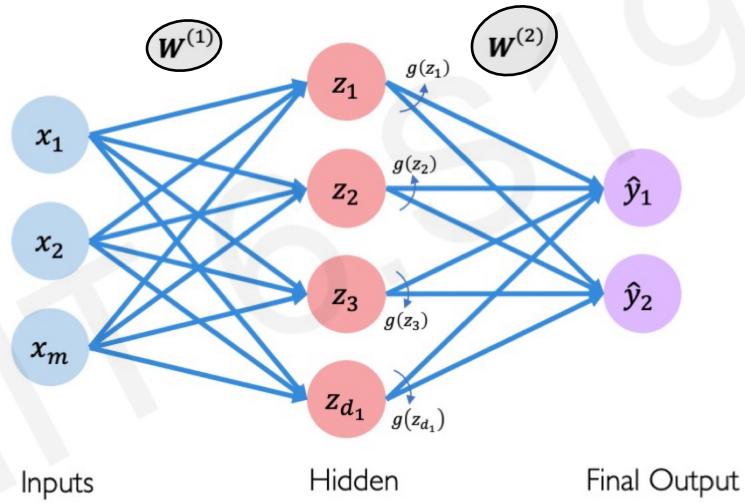
        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

        return output
```

INSTANCE

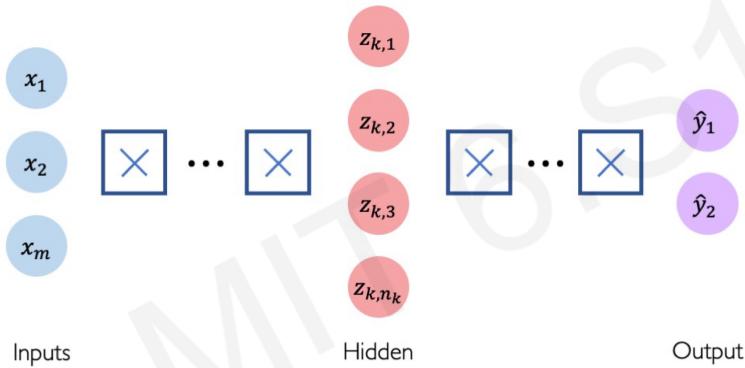
```
TF import tensorflow as tf
layer = tf.keras.layers.Dense(
    units=2)
```

## SINGLE LAYER NN



- Ogni layer ha una propria WEIGHT MATRIX

## DEEP NN (multi-layer NN)



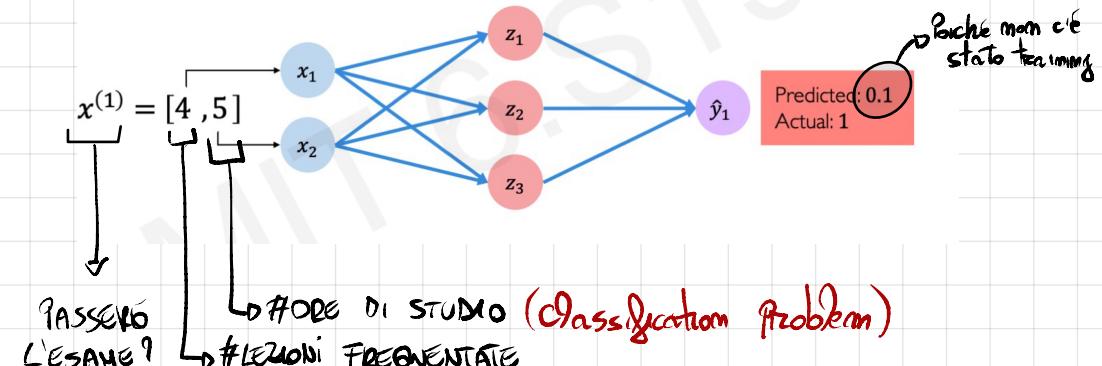
```

import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n1),
    tf.keras.layers.Dense(n2),
    ...
    tf.keras.layers.Dense(2)
])

```

## ESEMPIO

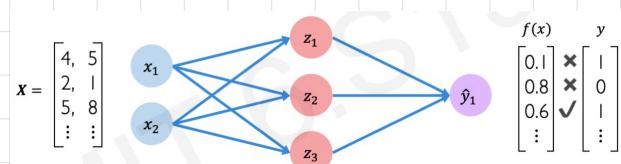


Come diciamo al modello quanto sbaglia?

LOSS

- $\mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$

Predicted                      Actual



L<sub>D</sub>

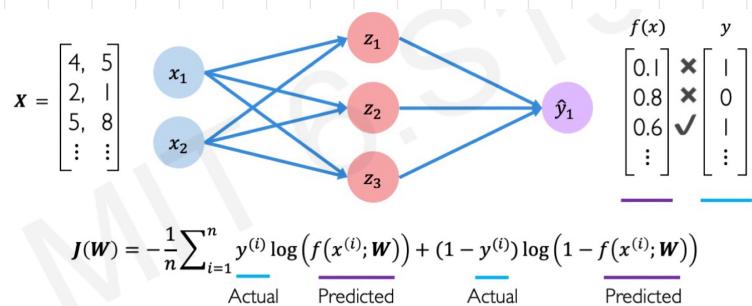
Also known as:  
• Objective function  
• Cost function  
• Empirical Risk

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

Predicted                      Actual

EMPIRICAL LOSS

## BINARY CROSS EMPTY LOSS

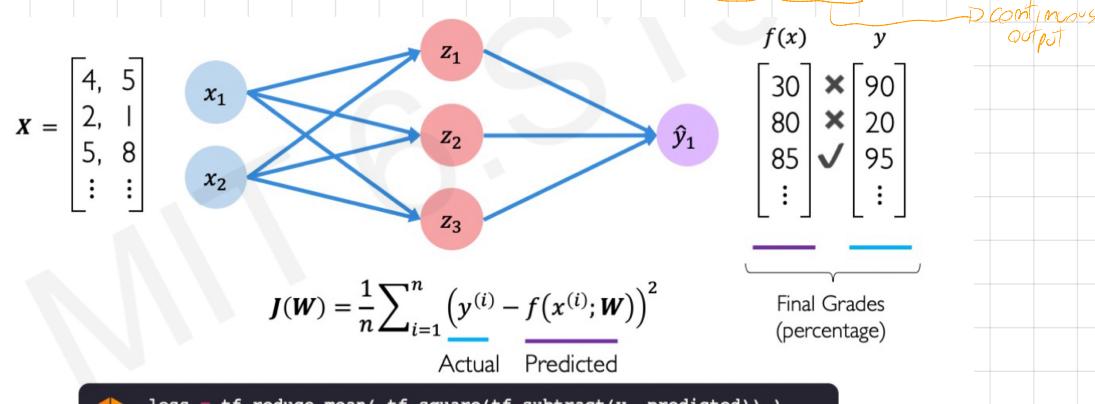


```
TensorFlow loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

→ confronta la distribuzione di probabilità (ATTESA vs OTTENUTA)

## MSE

- Se invece di un BINARY OUTPUT volessimo predire il VOTO, ci serve un'altra metrica per l'errore (regression problem)



```
TensorFlow loss = tf.reduce_mean(tf.square(tf.subtract(y, predicted)))  
loss = tf.keras.losses.MSE(y, predicted)
```

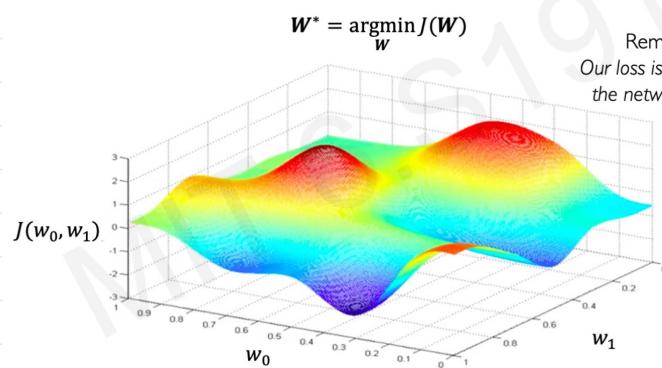
## TRAINING

### Loss Optimization

- Vogliamo i pesi che minimizzano la LOSS

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

- Con un problema a due pesi, possiamo plottarne  $\mathbf{W}^*$



### es. GRADIENTE

- Scegliamo un punto nel grafico, calcoliamo i GRAD. e ci spostiamo nella sua direzione fino al LOCAL OPT.

# GRADIENT DESCENT (ALGO)

NOTA se il learning rate è troppo piccolo restiamo bloccati in LOCAL OPT. se troppo grande si diffida la convergenza

## Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(W)}{\partial W}$
4. Update weights,  $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

LEARNING RATE  
(Quanto vogliamo imporre  
in diretta)

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

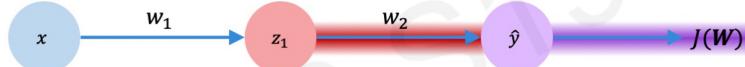
while True: # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
    gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

## • Come calcoliamo il gradiente?

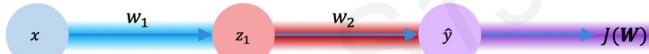
### BACKPROPAGATION

- Ci interessa quanto un **PICCOLO CAMBIAMENTO** in un pes  $w_i$  influenzi  $J(W)$
- Può essere espresso come rapporto tra derivate  $\frac{\partial J(W)}{\partial w_2}$  e quale applichiamo la **CHAIN RULE**



$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

## • Possiamo espanderci a $\frac{\partial J(W)}{\partial w_1}$



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

- La chain rule si applica a tutti i modi ricorsivamente in NN con più layer

### IN Practice

Come sceglio il LEARNING RATE?

ADAPTIVE LEARN. RATE

- Il GRADIENT DESCENT non funge con molti local opt.

- Il LEARNING RATE va settato per bene

↗ GRAD. DESC. ALGO

- Il LEARN. RATE non è fisso, ma è ottimizzato durante

- IL LEARNING PROCESS

Algorithm	TF Implementation
• SGD	tf.keras.optimizers.SGD
• Adam	tf.keras.optimizers.Adam
• Adadelta	tf.keras.optimizers.Adadelta
• Adagrad	tf.keras.optimizers.Adagrad
• RMSProp	tf.keras.optimizers.RMSProp

```

import tensorflow as tf
model = tf.keras.Sequential([...])
# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()
while True: # loop forever
    # forward pass through the network
    prediction = model(x)
    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)
    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```



Can replace with any TensorFlow optimizer

LA BACKPROP è  
possibile?

IDEA

- Impossibile com la BACKPROPAGATION è troppo da calcolare, soprattutto se lo facciamo per tutto il dataset.
- Lo computiamo per un sottoinsieme di punti

## STOCHASTIC GRADIENT DESCENT (SGD)

- Ci permette di aumentare la learning rate

### Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point  $i$
4. Compute gradient  $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

com un singolo punto è troppo noisy

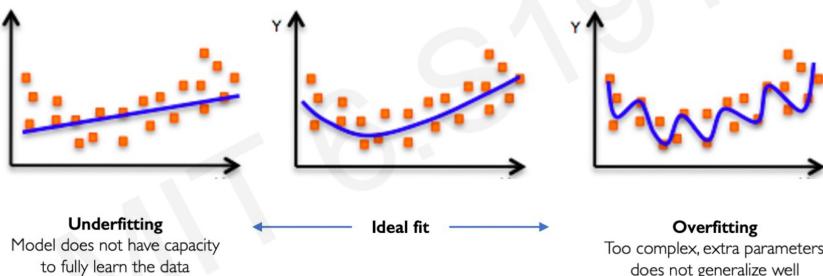
### Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of  $B$  data points
4. Compute gradient  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

così va meglio

## FITTING MODEL

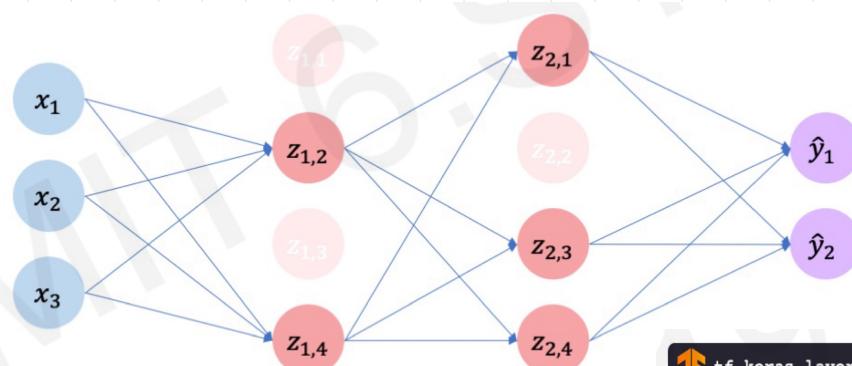
- Come rappresenta i dati il modello



- La linea blu è il modello usato per rappresentare i dati

# REGULARIZATION DROPOUT

- Tecnica che impedisce al modello di andare in **OVERTFITTING**
- Durante il training settiamo alcune activation a  $\emptyset$ .



Lo indotte aumenta la velocità di train, poiché deve lavorare su meno modi.

# EARLY STOPPING

- Smette il training quando risulta troppo loss

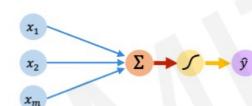
Stop training before we have a chance to overfit



# COSA ABBIAMO FATTO?

## The Perceptron

- Structural building blocks
- Nonlinear activation functions



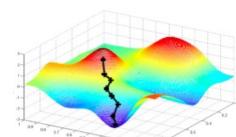
## Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



## Training in Practice

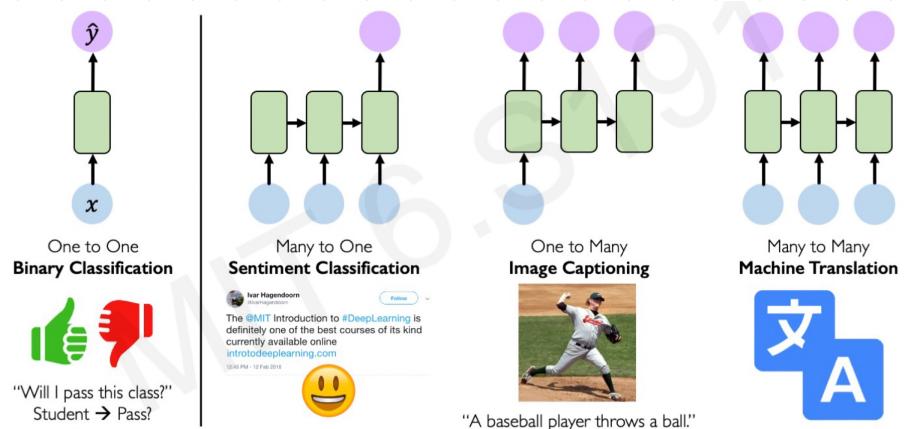
- Adaptive learning
- Batching
- Regularization



# DEEP SEQUENCE MODELLING

- Lavoreremo su dati sequenziali nel tempo

Sequence modeling  
Applications



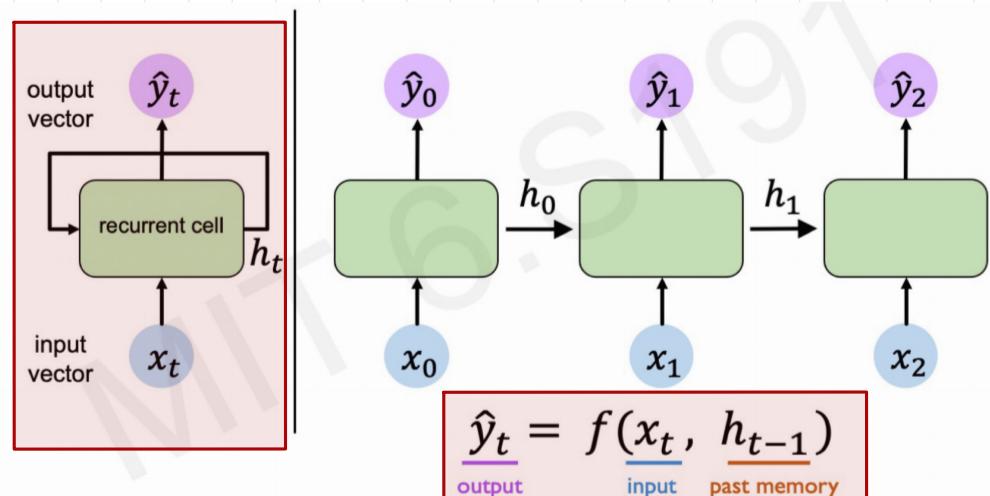
## NEURONS WITH RECURRENCE

IDEA:

- i dati sequenziali sono **TIME STEPS** individuali, ci serve un modello che sia in grado di gestirli individualmente, ma anche prendendo in considerazione anche gli stati passati

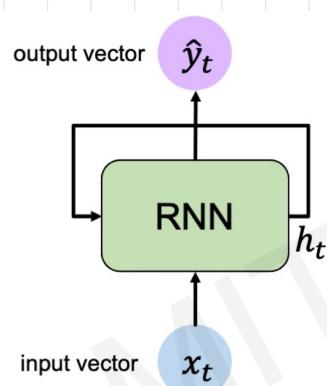
NEURONI CON  
RICORRENZA

$h_t$  sta per HIDDEN STATE



## RECURRENT NEURAL NETWORKS (RNN)

RECURRENCE  
RELATION



- Ad ogni time step calcoliamo l'hidden state  $h_t$  associato

$$h_t = f_w(x_t, h_{t-1})$$

cell state      function with weights  $w$   
input            old state

```

my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"

```

# Passi del processo

NOTA: a somo 3  
WEIGHT MATRICES

(1) INPUT VECTOR  $x_t$

(2) UPDATE di  $h_t$

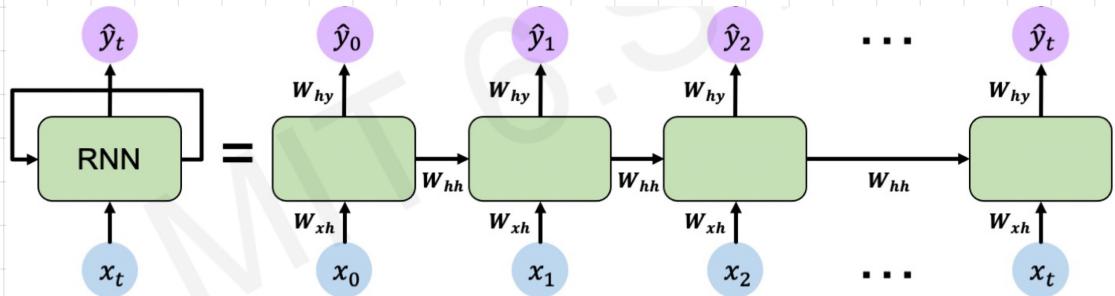
- moltiplichiamo  $h_{t-1}$  e  $x_t$  per le rispettive WEIGHT MATRICES

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

(3) CALCOLIAMO  $\hat{y}_t$  moltiplicando  
il hidden state per un'altra  
WEIGHT MATRIX

$$\hat{y}_t = W_{hy}^T h_t$$

... graficamente



OBSs

- Ad ogni output è associata una Loss, i cui insieme ci permette di allenare il modello.
- È importante da notare che le weight matrices sono sempre le stesse a ogni step.

oooPython

```
class MyRNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        super(MyRNNCell, self).__init__()

        # Initialize weight matrices
        self.W_xh = self.add_weight([rnn_units, input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim, rnn_units])

        # Initialize hidden state to zeros
        self.h = tf.zeros([rnn_units, 1])

    def call(self, x):
        # Update the hidden state
        self.h = tf.math.tanh( self.W_hh * self.h + self.W_xh * x )

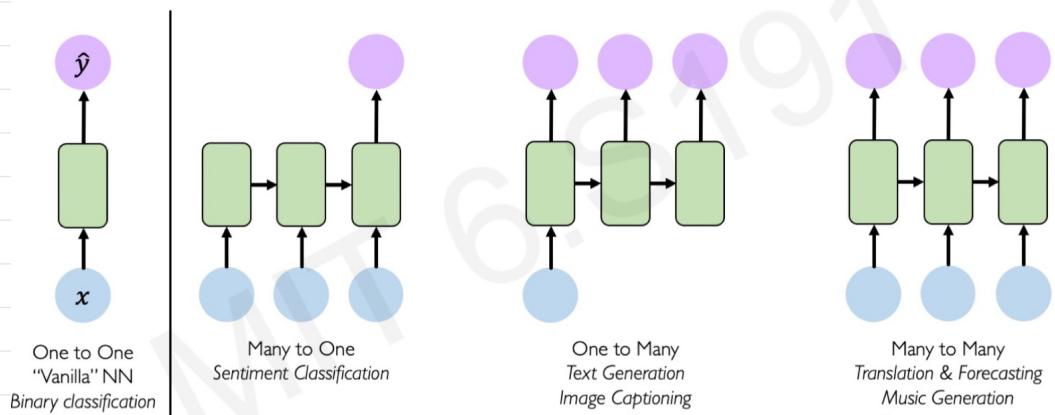
        # Compute the output
        output = self.W_hy * self.h

        # Return the current output and hidden state
        return output, self.h
```

- Ovviamente ci pensa tensorflow, non dobbiamo nulla implementarlo

```
tf.keras.layers.SimpleRNN(rnn_units)
```

# RNNs per Sequential modeling



## ENCODING ISSUE

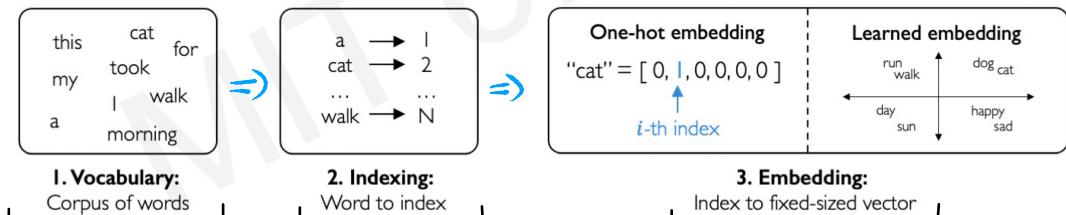
✗ "deep" ✗ "learning"

Neural networks cannot interpret words

✓  $\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \end{bmatrix} \rightarrow \text{Learned embedding} \rightarrow \begin{bmatrix} 0.9 \\ 0.2 \\ 0.4 \end{bmatrix}$

Neural networks require numerical inputs

SOL



## DESIGN CRITERIA

• 1) Modello deve avere l'abilità di:

(1) Gestire seq. di **LUNGHEZZA VARIABILE**

(2) Tracciare **LONG-TERM DEPENDENCIES**

"France is where I grew up, but I now live in Boston. I speak fluent \_\_\_\_."

(3) Mantenere informazioni sull'**ORDINE** → delle sequenze

(4) CONDIVIDERE PARAMETRI tra le sequenze

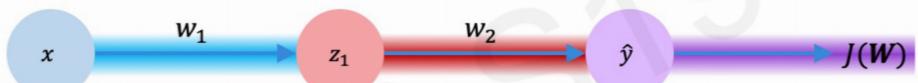
The food was great  
vs.  
We visited a restaurant for lunch  
vs.  
We were hungry but cleaned the house before eating

The food was good, not bad at all.  
vs.  
The food was bad, not good at all.  
No food icon

# BACKPROPAGATION THROUGH TIME (BPTT)

RIC

- Abbiamo visto che in BACKPROPAGATION consiste nel calcolare i GRADIENTI DELLA LOSS TOTAL ( $\partial J(W)$ ) rispetto ogni parametro ( $\frac{\partial J(W)}{\partial w_1}$ ), e poi modificare i parametri per diminuire la loss.

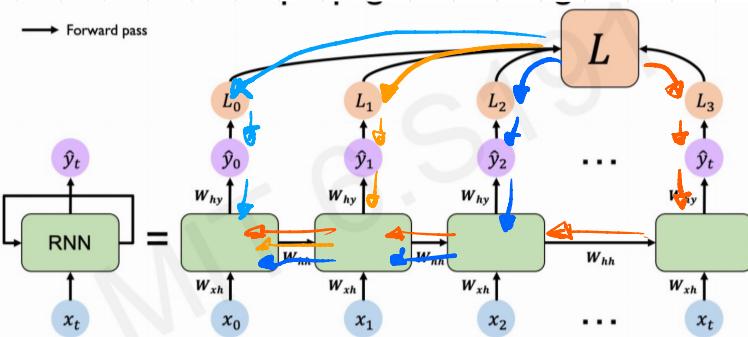


$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

NON GRADIENT FLOW

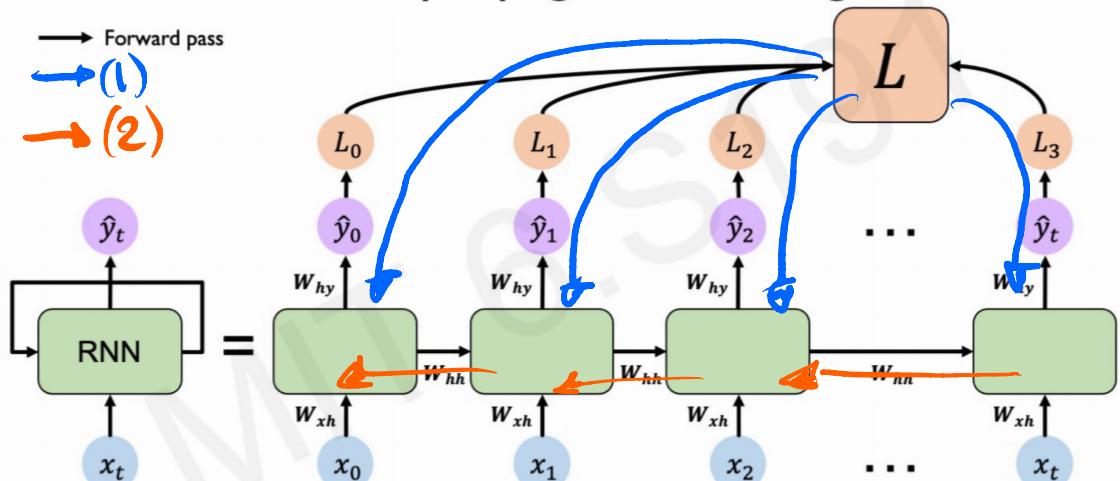
NO !

- Dala total loss invece di computare la Backprop. per ogni TIME STEP



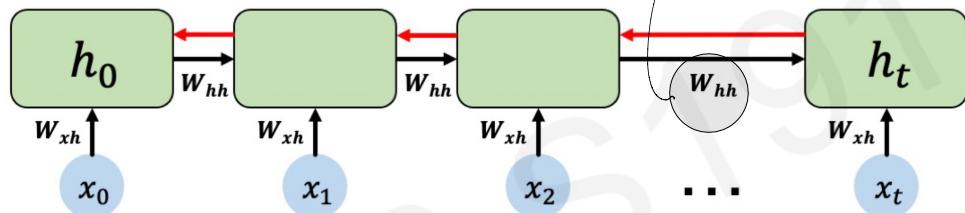
⇒ conviene farze un solo viaggio, dal TIME STEP più recente.  
In questo modo l'errore ce lo potremo in una sola direzione.

SI !



- Calcolare il GRADIENTE rispetto  $h_0$ , comporre:

- Molti moltiplicazioni per  $W_{hh}$
- Computare molti gradienti



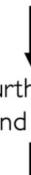
## Exploding Gradient

## Vanishing Gradient

- Se abbiamo molti valori  $> 1$  in questo modo, i gradienti crescono ed overze moltiplicazioni per il GRADIENTE
- se abbiamo invece valori  $< 1$  i GRADIENTI diminuiscono troppo

### Why are vanishing gradients a problem?

Multiply many small numbers together

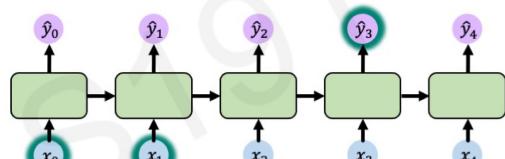


Errors due to further back time steps have smaller and smaller gradients

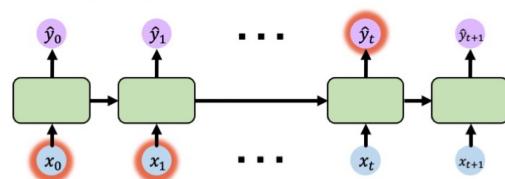


Bias parameters to capture short-term dependencies

"The clouds are in the \_\_\_"



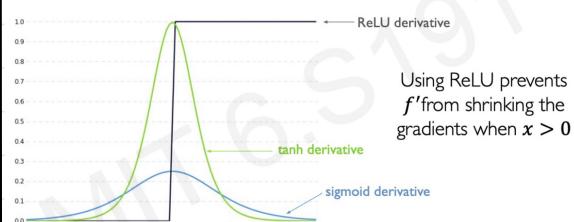
"I grew up in France, ... and I speak fluent \_\_\_"



## SOL

- Ce sono vari trick per gestire i VANISHING GRADIENTS

### Trick #1: Activation Functions



### Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Initialize **biases** to zero

This helps prevent the weights from shrinking to zero.

### Solution #3: Gated Cells

Idea: use a more complex recurrent unit with gates to control what information is passed through

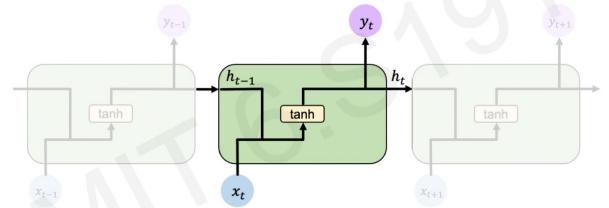
gated cell

LSTM, GRU, etc.

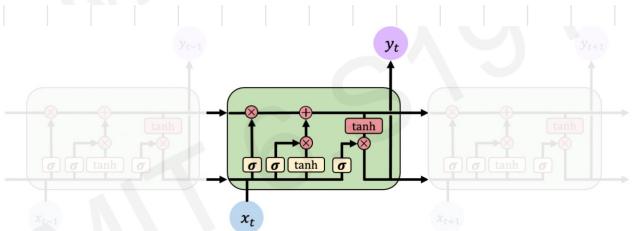
Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.

# LONG SHORT TERM MEMORY

- STANDARD RNN



- LSTM RNN



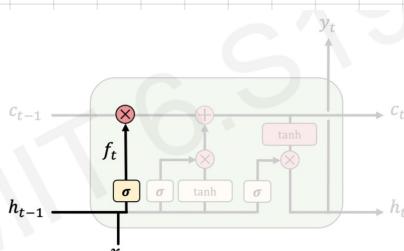
`tf.keras.layers.LSTM(num_units)`

- I GATES decidono quale informazione far passare o meno
- Lo fanno tramite, ad esempio, **SIGMOIDI** (come nelle imgs)

CELL STATE  
(TIPPI)  $\rightarrow$

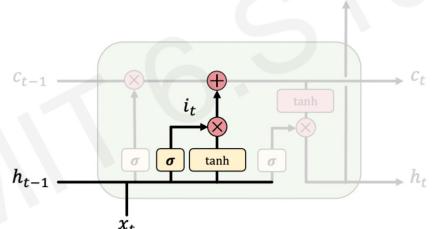
- Il parametro **C** è una variabile contenente informazioni passate
- Si divide in 4 fasi

(1)

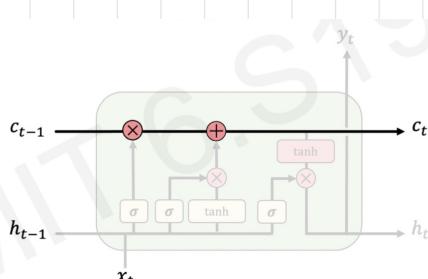


$\rightarrow$  **FORGET**: butta parti irrilevanti dello stato precedente

(2) **STORE**: salva le info irrilevanti sullo Stato prossimo

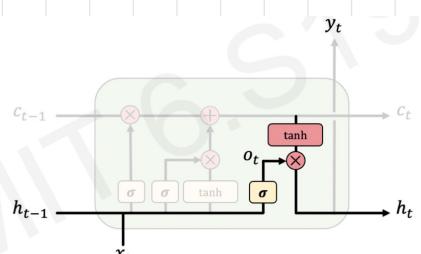


(3)

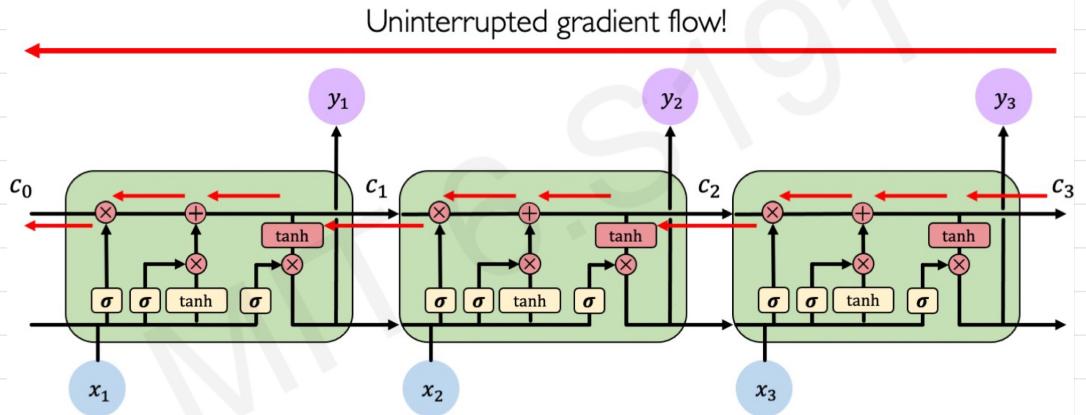


$\rightarrow$  **UPDATE**: aggiorna il valore di C<sub>t</sub>

(4) **OUTPUT**: controlla q'info  $\rightarrow$  invia alla stato successivo



• i9 GRADIENTE E ora calcolato attraverso i9 **CELL GATES** e  
elimina i9 **VANISHING GRADIENT PROBLEM**



## Concetti di LSTM

1. Maintain a **separate cell state** from what is outputted
2. Use **gates** to control the **flow of information**
  - **Forget** gate gets rid of irrelevant information
  - **Store** relevant information from current input
  - Selectively **update** cell state
  - **Output** gate returns a filtered version of the cell state
3. Backpropagation through time with **uninterrupted gradient flow**

Cosa abbiamo fatto?

## Deep Learning for Sequence Modeling: Summary

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence relation**
3. Training RNNs with **backpropagation through time**
4. Gated cells like **LSTMs** let us model **long-term dependencies**
5. Models for **music generation**, classification, machine translation, and more

# DEEP COMPUTER VISION



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	6	124	131	111	120	204	166	16	56	180
194	68	157	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	148	191	163	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	17	228	43	95	234
190	216	115	149	236	187	85	150	75	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	195	238	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	101	206	138	243	236
195	206	123	207	177	121	123	200	175	18	96	218

What the computer sees

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	163	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	17	228	43	95	234
190	216	116	149	236	187	85	150	75	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	195	238	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	101	206	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

An image is just a matrix of numbers [0,255].  
i.e., 1080x1080x3 for an RGB image

- Quali sono le task? - REGRESSION , CLASSIFICATION

## CLASSIFICATION

- Si basa sull'identificazione delle caratteristiche chiave



Nose,  
Eyes,  
Mouth



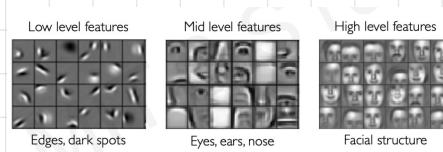
Wheels,  
License Plate,  
Headlights



Door,  
Windows,  
Steps

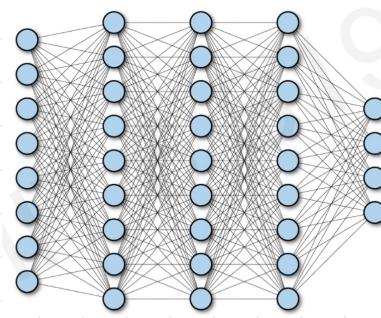
- È impossibile estrarre a mano le caratteristiche da classificare, ci sono troppe variazioni

- SIAMO IN GRADO DI ESTRARRE UNA GERARCHIA DI FEATURES DI RETTAMENTE DAI DATI?



## LEARNING VISUAL FEATURE

- Abbriamo già visto come lavorare con FEED FORWARD, DENSE LAYER, RNN, ma qui c'è senso un'architettura diversa.



• Com'è un DENSE LAYER che prende in input un'IMMAGINE 2D, e fa perdere le informazioni spaziali (Invece di rimanere di gruppi di pixel ad esempio)

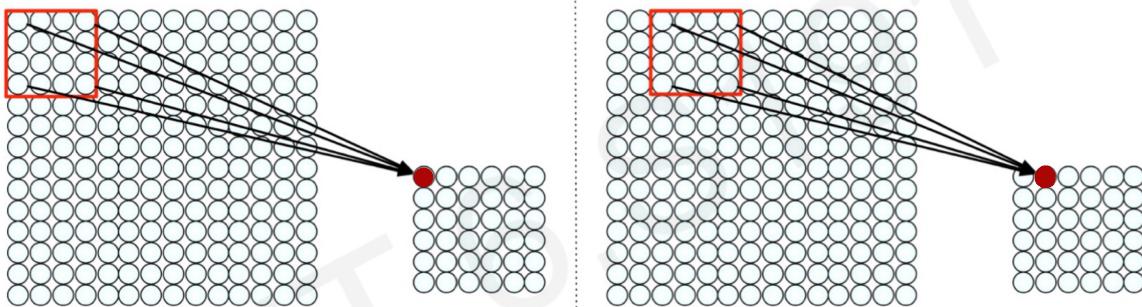


Questa operazione  
è una **CONVOLUTION**



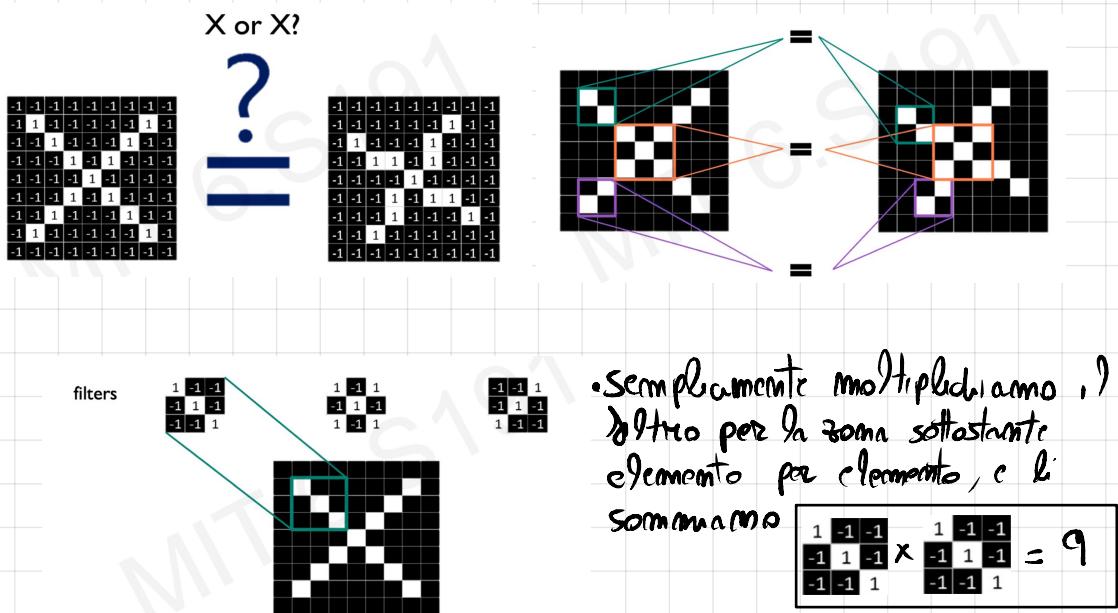
"Sfocare per far  
risaltare caratteristiche  
rilevanti"

- Utilizziamo un **SPATIAL STRUCTURE** e tramite un **MASCHERA** commettiamo un insieme di pixel nella stessa zona a un incisore in output, poi facciamo **SUDARE** la maschera



COME ASSEGNAANO I PESI ALLA MASCHERA?

ATTIVAZIONE DI  
UN FILTRO



→ con lo slide del filtro, la matrice che esce dalle sue attivazioni è chiamata **FEATURE MAP**



• Semplicemente moltiplichiamo i valori per la zona sottostante elemento per elemento, e li sommiamo

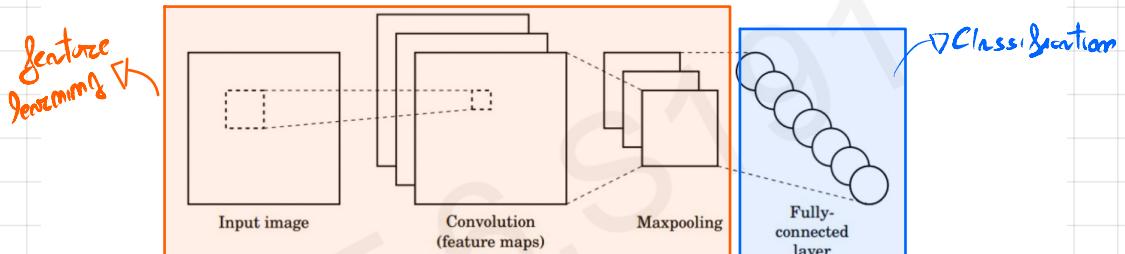
$$\begin{matrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{matrix} \times \begin{matrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{matrix} = 9$$

• A seconda del filtro possiamo estrarre le caratteristiche che ci interessano

# CONVENTIONAL NEURAL NETWORK (CNN)

## CNN Per la CLASSIFICAZIONE

- Vogliamo estrarre le caratteristiche da un'immagine per poi mapparle su una **CLASSIFICATION TASK**
- Le operazioni di base sono:

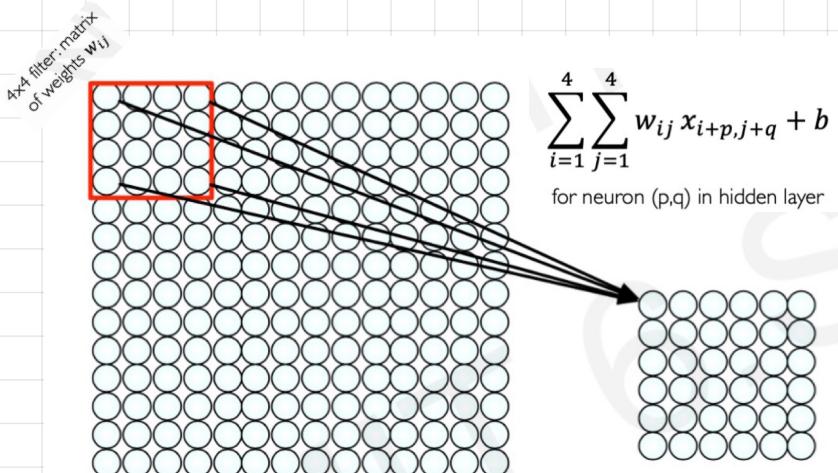


1. **Convolution:** Apply filters to generate **feature maps**.
2. **Non-linearity:** Often ReLU.
3. **Pooling:** Downsampling operation on each feature map.

tf.keras.layers.Conv2D  
 tf.keras.activations.\*  
 tf.keras.layers.MaxPool2D

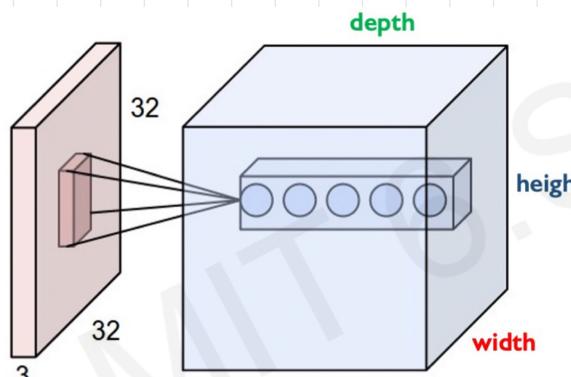
Lo si permette di ridurre la size delle feature map

## CONVENTIONAL LAYER



CHE POSSIAMO IN UN SINGOLO CONVENTIONAL LAYER ARETTE PIÙ FILTRI?

- Possiamo estrarre più caratteristiche da una stessa zona, applicando più filtri. Per ciò avremo un output con una terza dimensione: **DEPTH = #FILTRI**

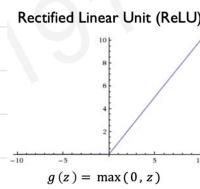


tf.keras.layers.Conv2D( filters=d, kernel\_size=(h,w), strides=s )

## ACTIVATION (RELU)

- Dobbiamo ricordarci l'activation function sulle feature map in output.

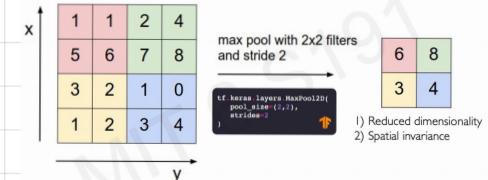
Inoltre si usano la funzione **NON LINEARE RELU** che porta tutti i valori negativi a 0.



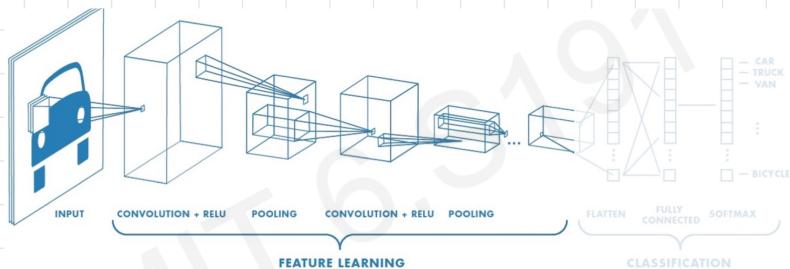
`tf.keras.layers.ReLU`

## POOLING

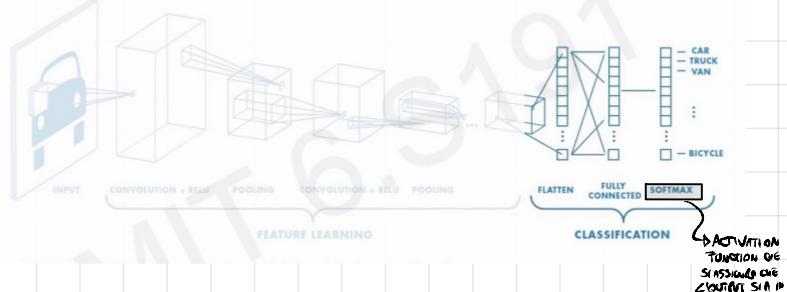
- Reduce la dimensione delle **FEAT MAP** preservando **SPATIAL INVARIANCE**



- Le operazioni descritte servono in sostanza per estrarre le feature **AD ALTO NIVELLO** e semplificare l'immagine al punto che può essere data in input a un **DENSE LAYER** senza i problemi descritti prima (**SPATIAL INVARIANCE**)



- Il **DENSE LAYER** a questo punto utilizza le feature per classificare l'immagine e il suo **OUTPUT** è in **PROB. DI APPARTENENZA A UNA DATA CLASSE**.



... in Tensorflow

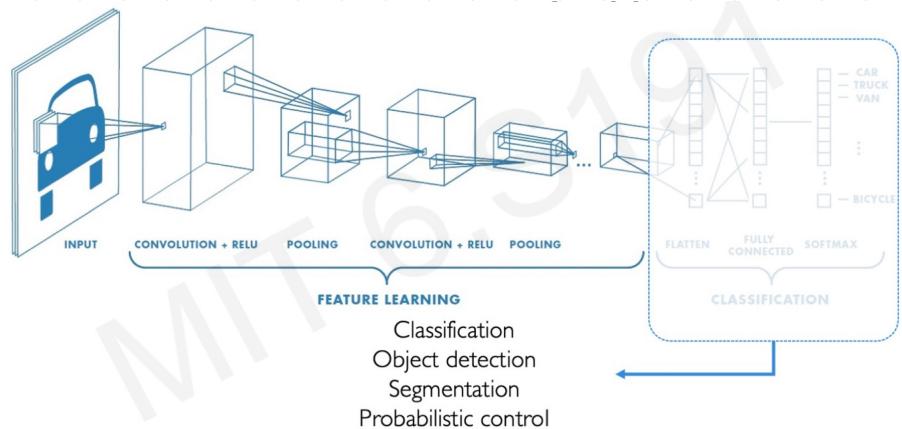
```
import tensorflow as tf

def generate_model():
    model = tf.keras.Sequential([
        # first convolutional layer
        tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),
        # aumentato le caratteristiche da estrarre
        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

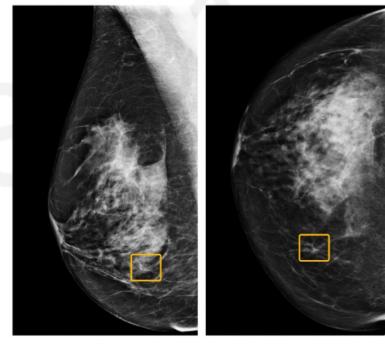
        # fully connected classifier
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax') # 10 outputs
    ])
    return model
```

# ALTRÉ APPLICAZIONI

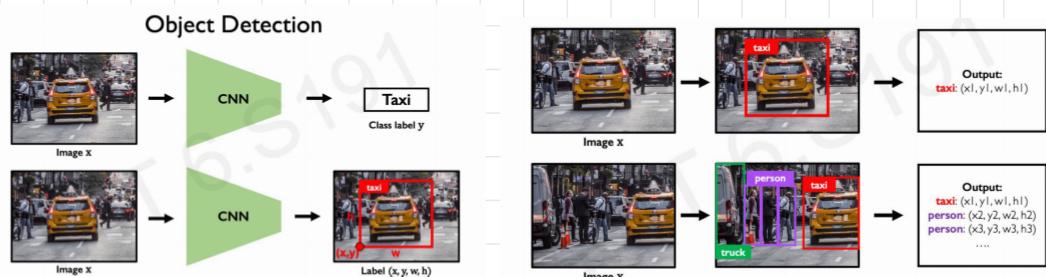
- I modelli definiti sopra non è limitato alla classificazione, ma a un'infinità di applicazioni



Esempio



OBJ DETECTION



Io a sì sono troppo dettagli!

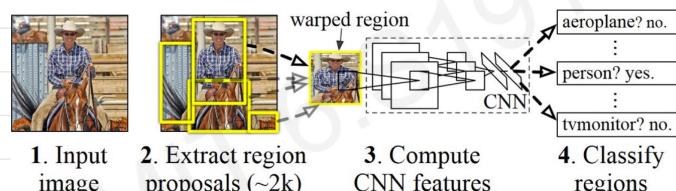
- Un approccio è di **SPARIRE BOX** sull'immagine e passare quelle sottoimmagini alla CNN.

Lo spartire spartire box a caso (o calcolarli tutti) è una **John**, ci serve qualche **HEURISTIC**

## Object Detection with R-CNNs

①

R-CNN algorithm: Find regions that we think have objects. Use CNN to classify.



**Problems:** 1) Slow! Many regions; time intensive inference.

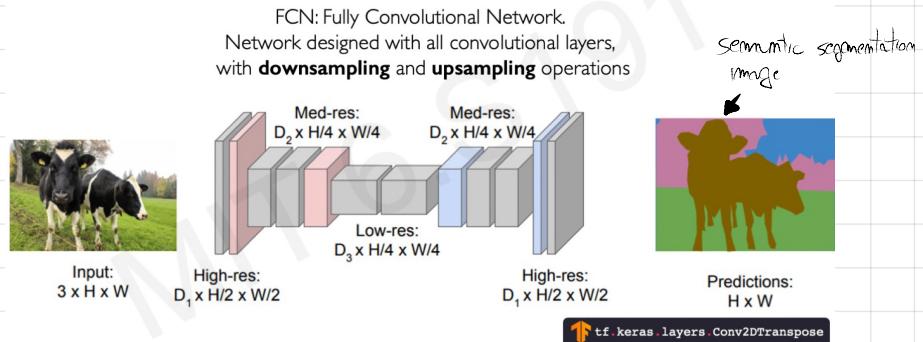
2) Brittle! Manually defined region proposals.

## ② Faster R-CNN Learns Region Proposals

VARIANTE del 'R-CNN': Al posto dell'algoritmo di prima inseriamo una parte del nostro modello TRAINED per riconoscere regioni interessanti della immagine

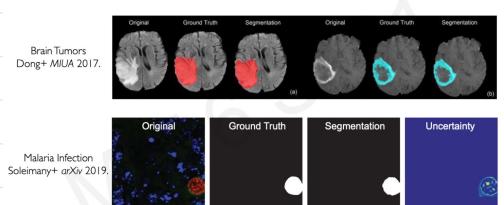
③

## Semantic Segmentation: Fully Convolutional Networks

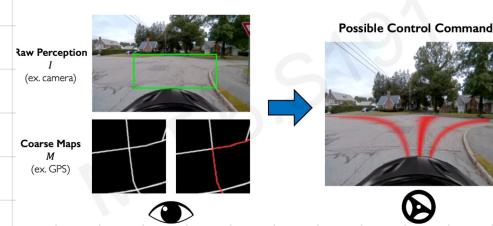


Aspettivamente classifica l'appartenenza di ogni pixel a una data regione, ad esempio i pixel delle mucche hanno tutta la stessa classe (colore)

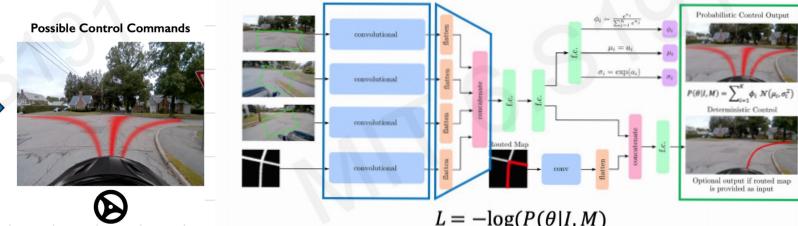
### Semantic Segmentation: Biomedical Image Analysis



### Continuous Control: Navigation from Vision



Entire model is trained end-to-end **without any human labelling or annotations**



ooo SOMMARIO

## Deep Learning for Computer Vision: Summary

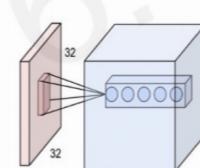
### Foundations

- Why computer vision?
- Representing images
- Convolutions for feature extraction



### CNNs

- CNN architecture
- Application to classification
- ImageNet



### Applications

- Segmentation, image captioning, control
- Security, medicine, robotics



# DEEP GENERATIVE MODELS

- Fino ad ora abbiamo visto solo il **SUPERVISED LEARNING** in cui forniamo al modello **DATA** che **LABELS** associate.
- Nel' **UNSUPERVISED** abbiamo solo **DATA**.

## Supervised Learning

**Data:**  $(x, y)$   
 $x$  is data,  $y$  is label

**Goal:** Learn function to map  
 $x \rightarrow y$

**Examples:** Classification,  
regression, object detection,  
semantic segmentation, etc.

## Unsupervised Learning

**Data:**  $x$   
 $x$  is data, no labels!

**Goal:** Learn the hidden or  
underlying structure of the data

**Examples:** Clustering, feature or  
dimensionality reduction, etc.

# GENERATIVE MODELING

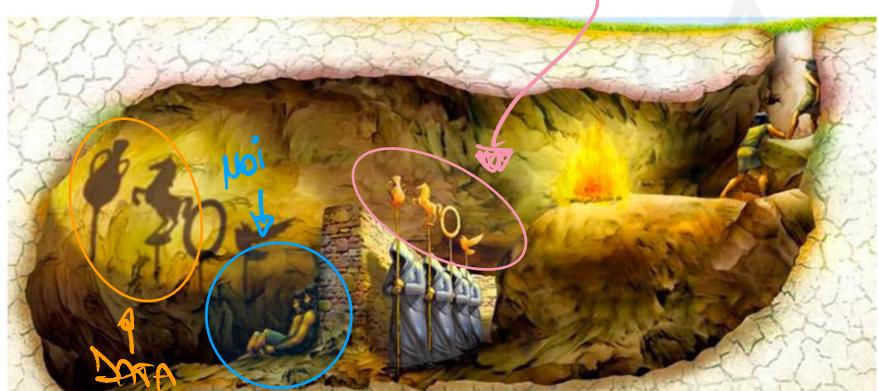
## OBBIETTIVO

- Dato un training set in cui Prob. N.S. dei sample  $x$ :  $P_{\text{Data}}(x)$ ; vogliamo un modello che Istruisce  $P_{\text{Model}}(x) \sim P_{\text{Data}}(x)$ .

COME GENERIAMO NUOVI SAMPLE?  $\rightarrow$  grazie a  $P_{\text{Model}}(x)$

- In figura che un UNSUPERVISED LEARNING trova fuori correlazioni tra features che mancano ci immaginiamo

- CAZZO È UNA LATENT VARIABLE?



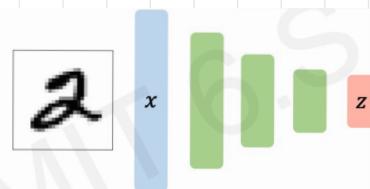
- POSSIAMO CAPIRE Q'UNcosa DELLE LATENT VARS A PARTIRE DATI DATI CHE OSSERVANO?

$\Rightarrow$  Sono in pratica **HIDDEN FEATURES**, esiste un modello che le scopre chiamato **AUTOENCODER**

# AUTOENCODER (AUTOMATICALLY ENCODING)

- È in grado di mappare il sample  $x$  in un vettore di HIDDEN FEATURES (variables)  $z$ , per questo è detto **ENCODER**

- $z$  ha una dimensione minore di  $x$ , quindi ci permette di avere POCHE MA BUONE feature

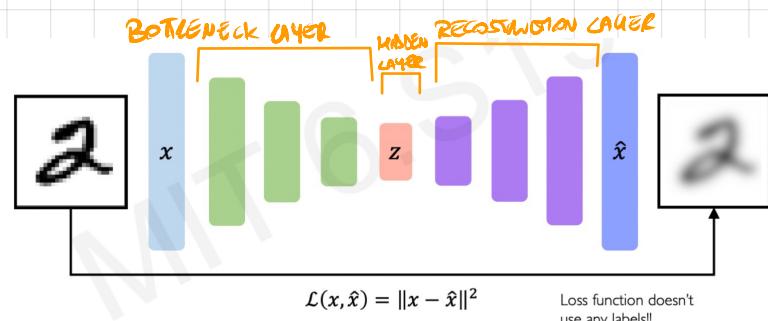


COME LO TRAINIANO?

Ci insegniamo, in un giro del modello chiamato **DECODER**, a trovare  $x$  da  $z$ , o altranno  $\hat{x}$

## DECODER

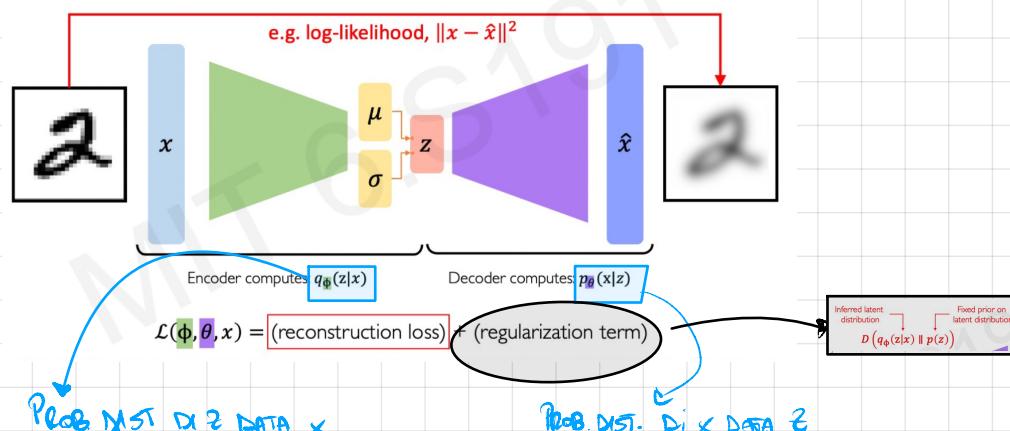
- Per allearo il **DECODER** sfruttiamo le differenze tra  $x$  e  $\hat{x}$



## ISSUE

- Più compressiamo, quindi più la DIMENSIONALITÀ di  $z$  è piccola, più c'è un BOTTLENECK per la ricostruzione

## VARIATIONAL AUTOENCODER



- Nell' **AUTOENCODER** tradizionale  $z$  è deterministico, dito stesso  $x$  e stesi pesi esce sempre  $z$

- Il **VAE** introduce la **probabilità**, vogliono che anche immagini  $\hat{x}$  simili a  $x$  siano processate bene

↳ invece di  $z$  troviamo una **MEDIA** e una **STD.**, ovvero i parametri che ci danno un Prob. DIST. associati a  $z$ , cioè la rappresentazione probabilistica

## REGULARIZATION

- 19 REGULARIZATION TERM ci permette di formare il modello in base alle ipotesi su  $z$ , così cercherà di avvicinarsi alle priorità da noi fatte.

$$D(q_{\phi}(z|x) \parallel p(z))$$

Inferred latent distribution      ↓      Fixed prior on latent distribution

- una scelta classica è una GAUSSIANA NORMALE, in questo quando il modello cerca di generare  $z$  intorno al centro del LATENT SPACE.  
Inoltre facendo il modello quando questo cerca di memorizzare i dati (conservando punti in REGIONI SPECIFICHE)

$$p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

$$D(q_{\phi}(z|x) \parallel p(z))$$

$$= -\frac{1}{2} \sum_{j=0}^{k-1} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$$

KL-divergence  
between the two  
distributions

(ci permette di calcolare la  
divergenza delle distribuzioni  
ottenute dalla Gaussian)

- OBIETTIVO**
- PROPRIETÀ che la REGULARIZZAZIONE ci aiuta a raggiungere

1. Continuity: points that are close in latent space  $\rightarrow$  similar content after decoding
2. Completeness: sampling from latent space  $\rightarrow$  "meaningful" content after decoding

## ISSUE

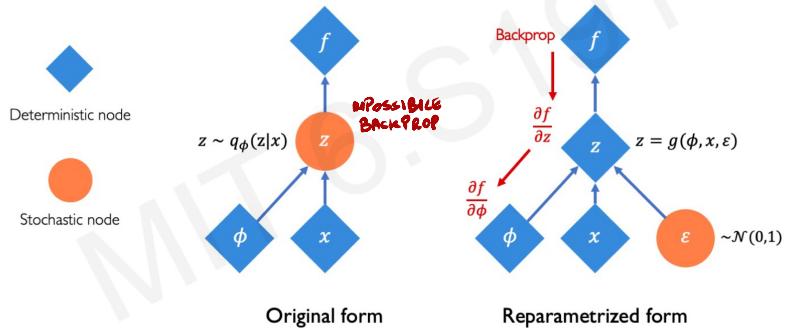
- Più regolarizziamo più è possibile che la ricostruzione sia poco imprecisa.
- Inoltre, per il CAMPO STOCHASTICO non possiamo fare BACKPROPAGATION

## IDEA

- Considera  $z$  come somma di un vettore  $\mu$  e uno scalare da una costante random  $\varepsilon \sim \mathcal{N}(0,1)$  (NOISE FACTOR)

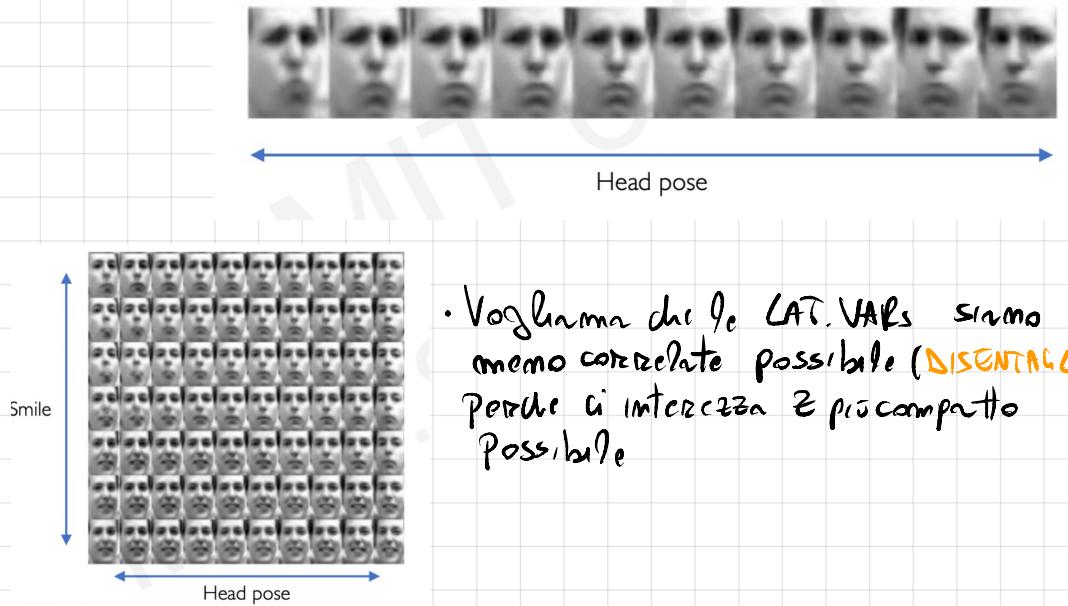
$$z \sim \mathcal{N}(\mu, \sigma^2) \Rightarrow z = \mu + \sigma \odot \varepsilon$$

Come si fa?



## DISENTANGLEMENT

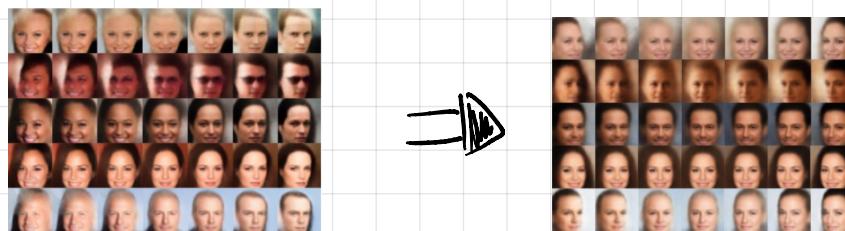
- Se perturbammo una LAT. VAR. fissando le altre



- Vogliamo che le LAT. VARS siano meno correlate possibile (**DISENTANGLER**) perché ci interessa è più compatto possibile

- Come si raggiunge?
- Introduciamo  $\beta$

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction term}} - \underbrace{\beta D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z}))}_{\text{Regularization term}}$$



• Con  $\beta=1$  se cambia la posizione della testa cambia anche il sorriso

• Con  $\beta=250$  il sorriso resta circa costante

## SOMMARIO

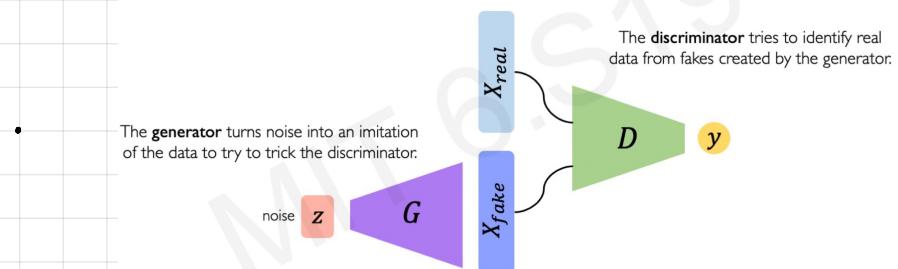
1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation
5. Generating new examples

Possiamo usare le latent distributions per creare buoni dataset?

# GENERATIVE ADVERSARIAL NETWORKS (GANs)

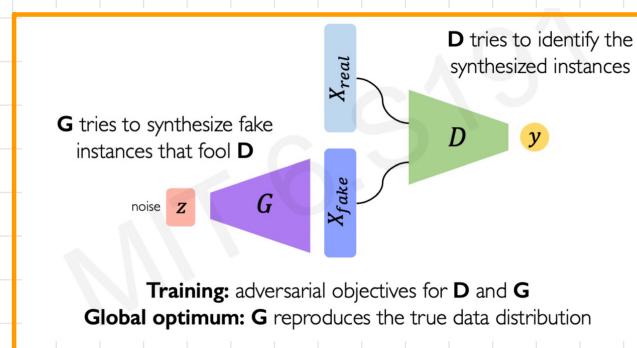
## IDEA

- Invece di modellare la densità, cerchiamo solo di generare nuove istanze
- Non possiamo partire direttamente da distribuzioni complesse
- L'idea è di avere due NETWORK che competono
  - (1) **GENERATOR** genera sample fake e l'ammach, vuole massimizzare  $\text{Prob. che l'altro li scambia per veri}$
  - (2) **DISCRIMINATOR** guarda ciò che gli viene mandato, lo confronta con i dati veri e dà un output in probabilità che per lui, sia vero.



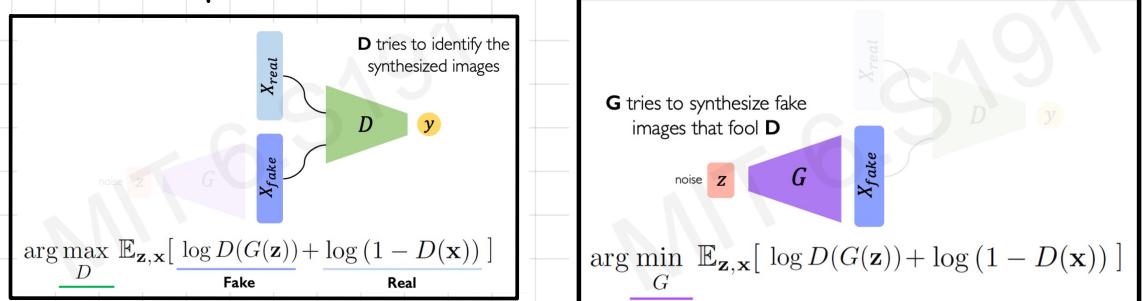
## TRAINING

- L'ottimo è che il **GENERATOR** reproduca perfettamente i **DATI veri**

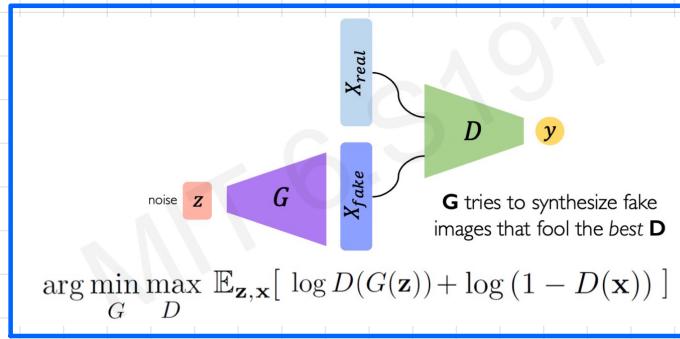


## LOSS FUNCT

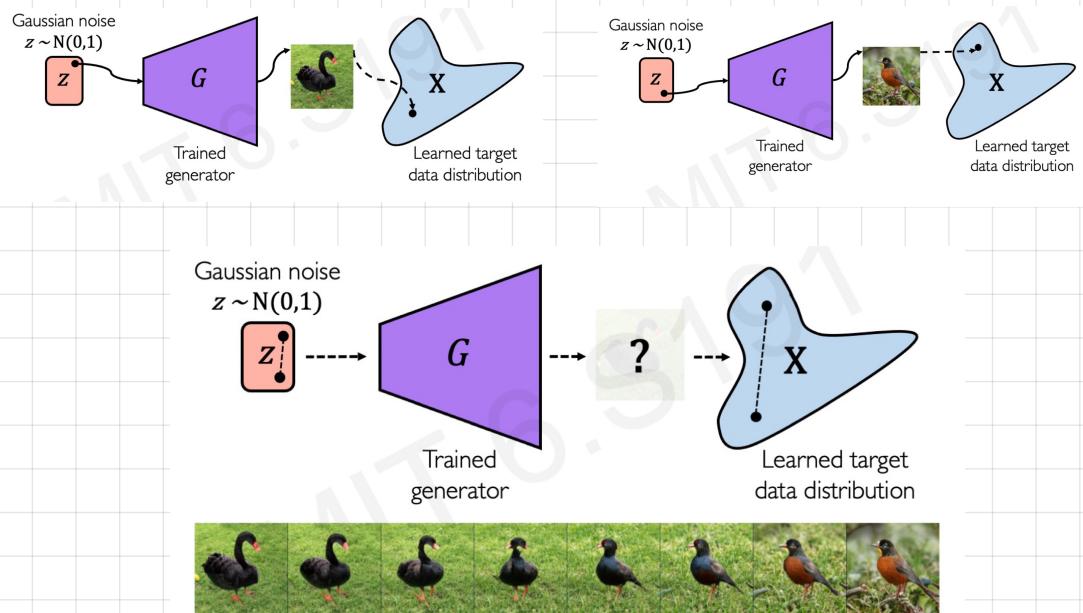
- IL **GENERATOR** vuole massimizzare  $\mathbb{E}_{z,x} [\log D(G(z)) + \log (1 - D(x)) ]$
- IL **DISCRIMINATOR** invece minimizza
  - $G(z) \rightarrow$  istanza generata
  - $D(G(z)) \rightarrow$  Probabilità che  $G(z)$  sia vero
  - $D(x) \rightarrow$  probabilità che  $x$ , vera, sia falso



⇒ Perché la Loss Function è espressibile come



• Possiamo generare nuovi DATA?

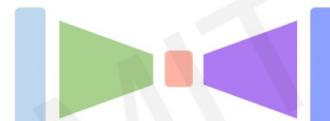


SUMMARY

## Deep Generative Modeling: Summary

### Autoencoders and Variational Autoencoders (VAEs)

Learn lower-dimensional **latent space** and **sample** to generate input reconstructions



### Generative Adversarial Networks (GANs)

Competing **generator** and **discriminator** networks

