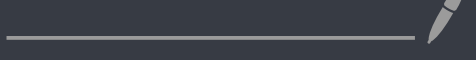
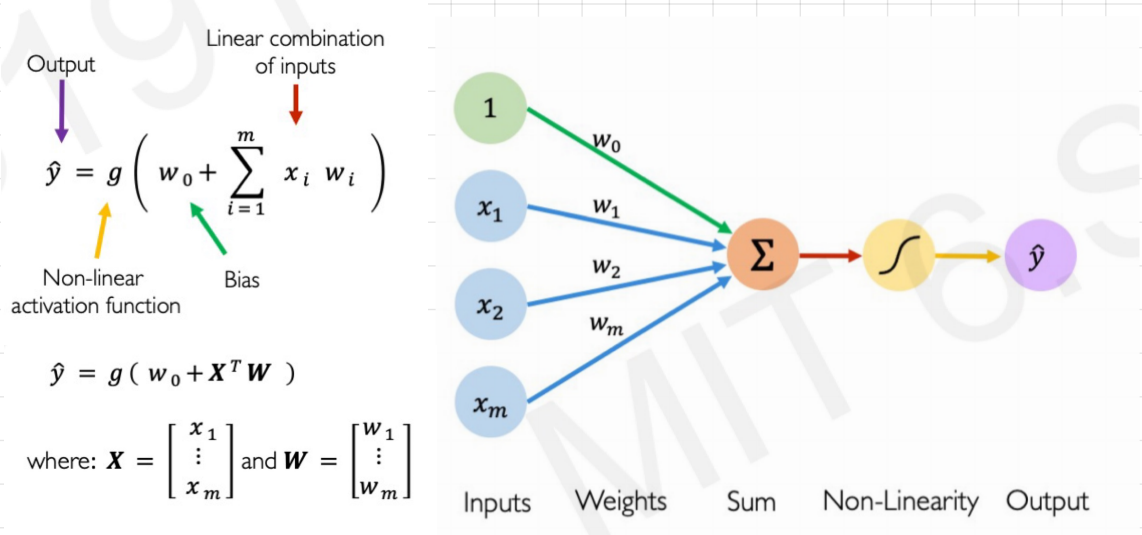


L1 - INTRO TO DL

BUILDING BLOCKS

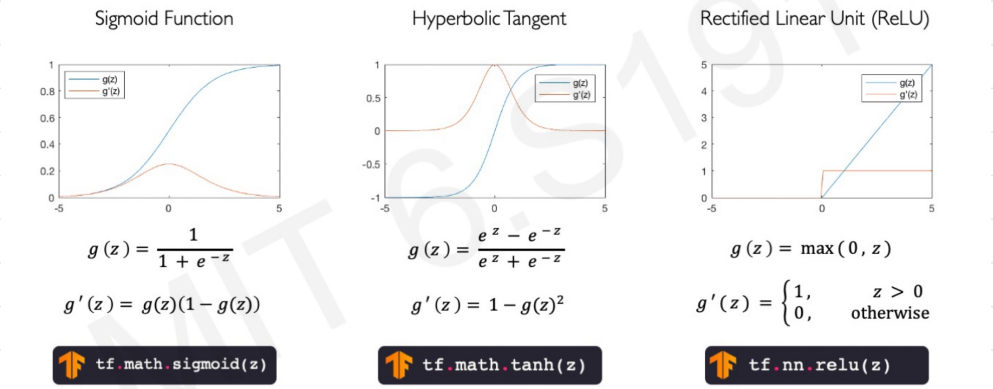
PERCEPTRON

FORWARD PROPAG.



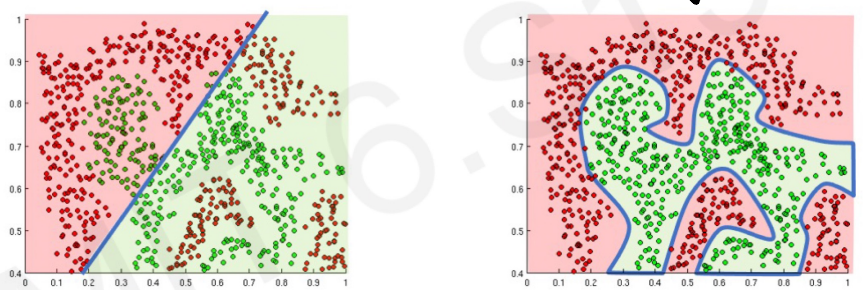
ACTIVATION FUNCTION

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



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

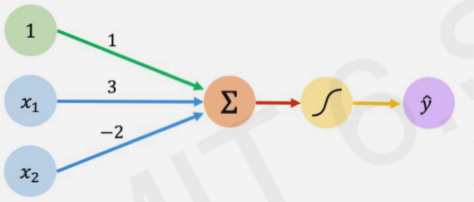
↳ Ad esempio per separare i punti colorati:



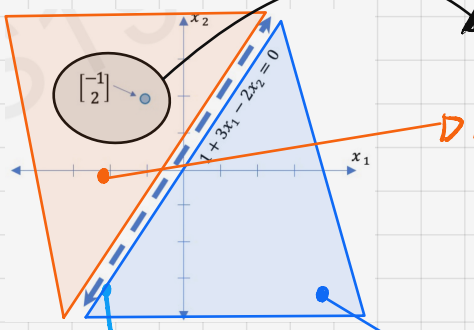
Linear activation functions produce linear decisions no matter the network size

Non-linearities allow us to approximate arbitrarily complex functions

esempio



$\hat{y} = g(1 + 3x_1 - 2x_2)$ ← 2D



$\hat{y} = g(-6) \approx 0{,}002$

→ da input in questa zona viene fuori:
→ $z < 0$
→ $y < 0.5$

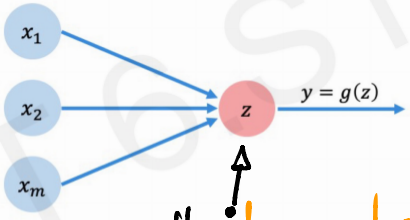
I PUNTI SULLA RETTA SONO gli input accettabili

→ Da questa invece
→ $z > 0$
→ $y > 0{,}5$

Zoom perceptron to NEURAL NET

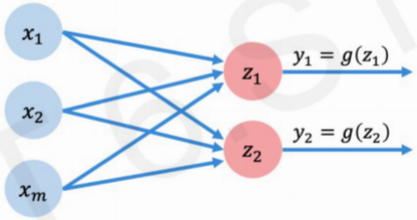
DENSE LAYER

• PERCEP. SEMPLIFICATO



Detta forward prop.

• MULTI-OUTPUT



↳ Detto DENSE LAYER poiché ogni z_i è connesso a ogni x_j

In tensorflow

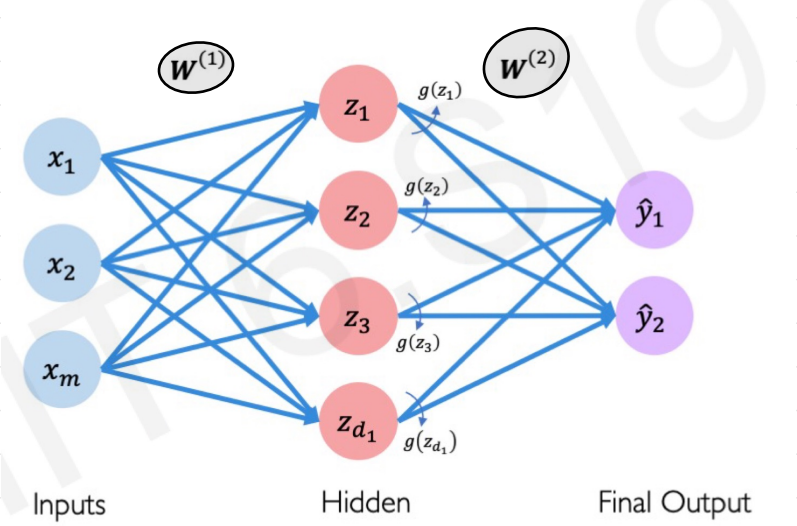
CLASS

```
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

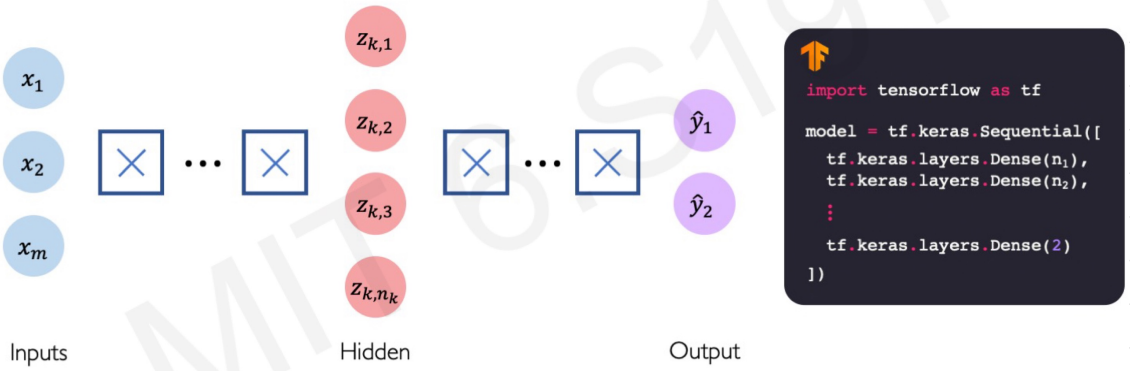
```
import tensorflow as tf

layer = tf.keras.layers.Dense(
    units=2)
```

SINGLE LAYER NN



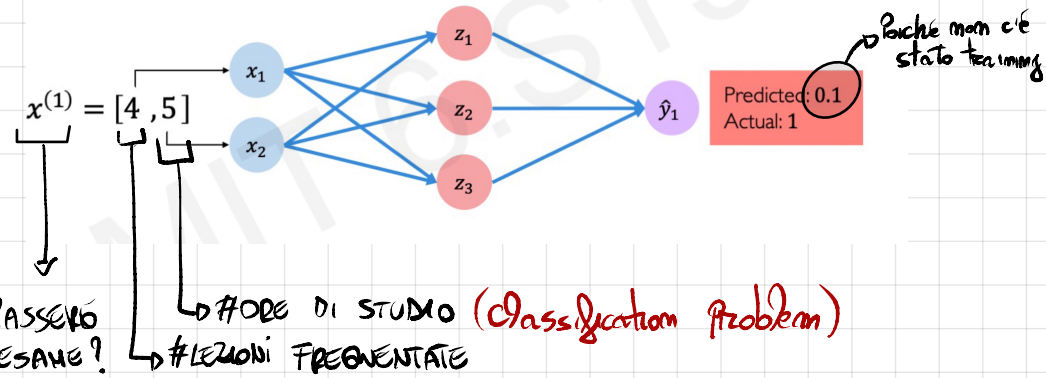
- Ogni layer ha una propria **WHEIGHT MATRIX**

DEEP NN
(multi-layer NN)



---

esempio

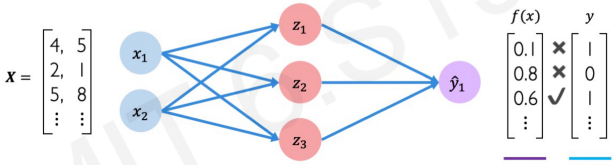


Come diciamo al modello quanto sbaglia?

LOSS

- $\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$
  Predicted — Actual

EMPIRICAL LOSS

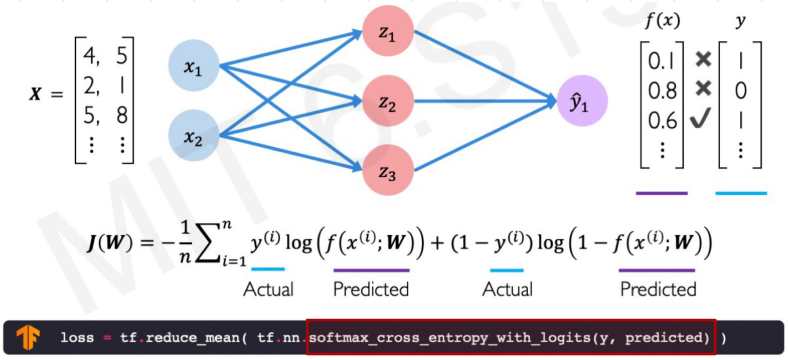
- 

↳ Also known as:
- Objective function
- Cost function
- Empirical Risk

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

Predicted — Actual

BINARY CROSS ENTROPY LOSS



$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\left(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}}\right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\left(1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}}\right)$$

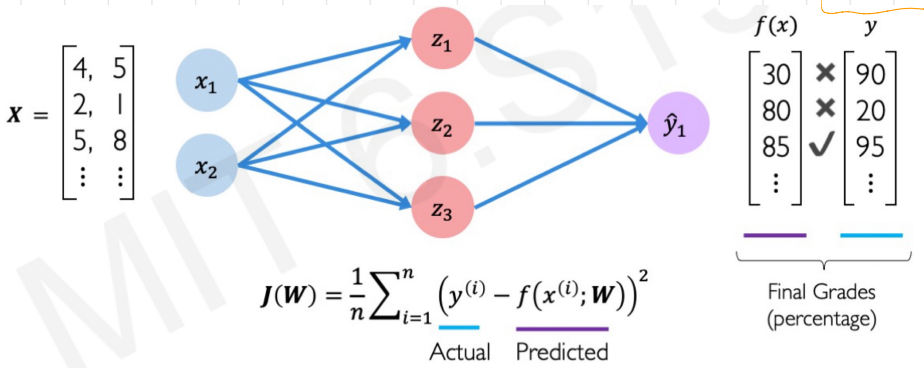
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

→ confronta la distribuzione di probabilità (ATTESA vs OTTENUTA)

MSE

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

→ continuous output



$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}}\right)^2$$

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

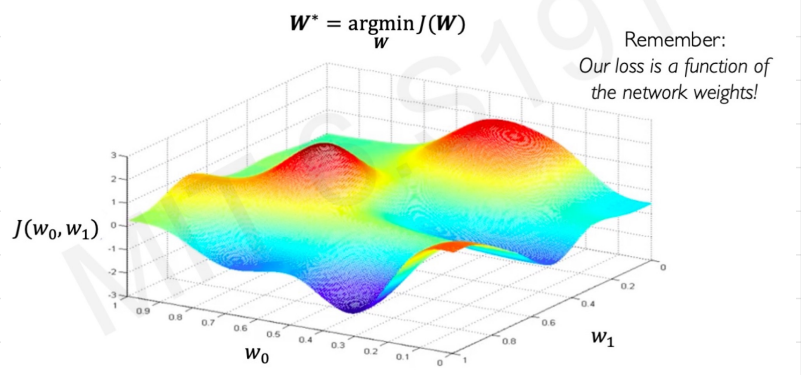
TRAINING

LOSS OPTIMIZATION

- Vogliamo i pesi che minimizzano la LOSS

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

- Con un problema a due pesi possiamo plottare W^*



es. GRADIENTE

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

GRADIENT DESCENT (ALGO)

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 imparare in fretta)

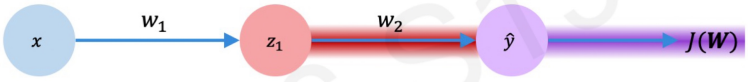
```
import tensorflow as tf

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
```

NOTA se il learning rate è troppo piccolo restiamo bloccati in LOCAL OPT. se troppo grande è difficile convergere

- Come computiamo il gradiente?

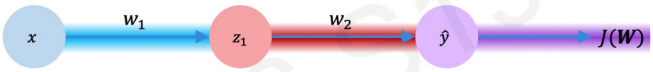
BACKPROPAGATION

- ci interessa quanto un PICCOLO CAMBIAMENTO in un peso w_i influenzi $J(W)$
- Può esser espresso come rapporto tra derivate $\frac{\partial J(W)}{\partial w_2}$ al 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 nodi ricorsivamente in NN con più layer

---

IN PRATICA

- il GRADIENT DESCENT non funge con molti local opts.
- il LEARNING RATE va settato per bene
- il LEARN. RATE non è fissato, ma è ottimizzato durante il LEARNING PROCESS.

Come scelgo il LEARNING RATE? ADAPTIVE LEARN. RATE

(GRAD. DESC. ACCOS)

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

```python
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)))
```

LA BACKPROP è fattibile?

- Inoltre con la BACKPROPAGATION è tosto da calcolare soprattutto se la calcoliamo per tutto il dataset.

IDEA

- Lo computiamo per un sottoinsieme di punti

STOCHASTIC GRADIENT DESCENT (ALGO)

- Ci permette di aumentare il learning rate

Algorithm

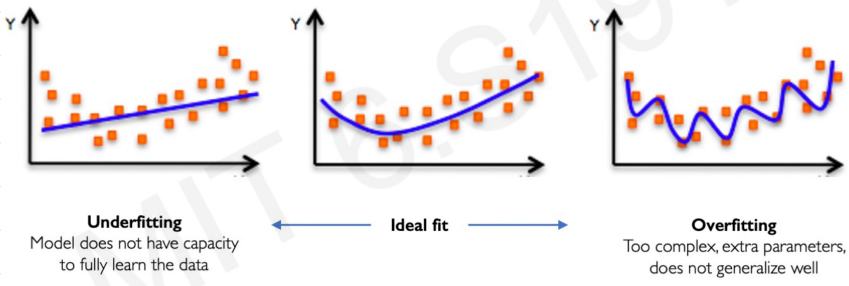
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i → con un singolo punto è troppo noisy
4. Compute gradient $\frac{\partial J_i(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights

Algorithm

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

FITTING MODEL

- Come rappresenta i dati il modello

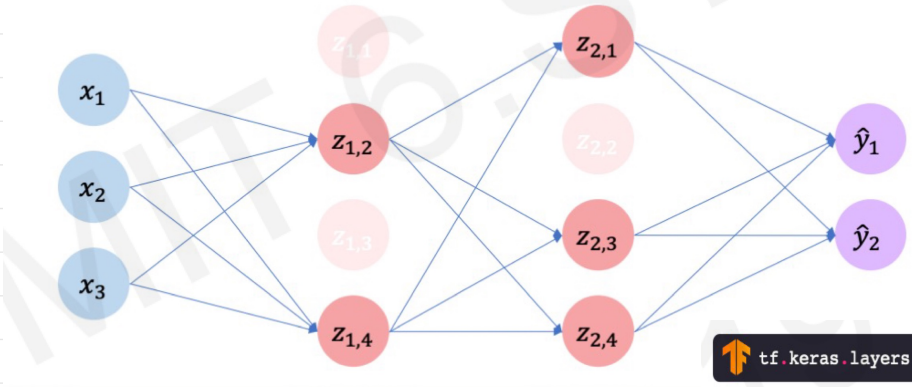


- La linea blue è il modello usato per rappresentare i dati

REGULARIZATION

DROPOUT

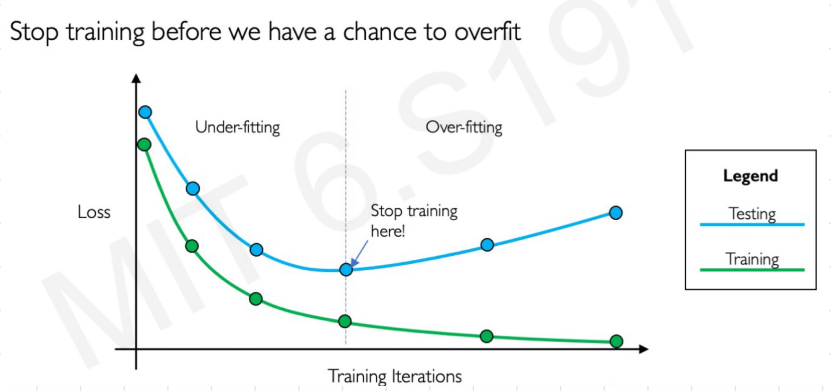
- Tecnica che impedisce al modello di andare in OVERFITTING
- Durante il training settiamo alcune activations a ϕ.



↳ Inoltre aumenta la velocità di train, poiché deve lavorare su meno nodi

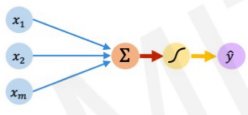
EARLY STOPPING

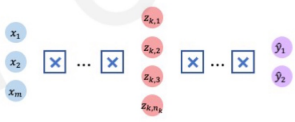
- Smette il training quando riduce troppo loss

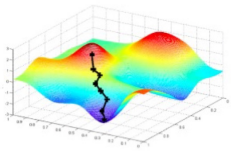


COSA ABBIAMO FATTO?

The Perceptron	Neural Networks	Training in Practice
• Structural building blocks • Nonlinear activation functions	• Stacking Perceptrons to form neural networks • Optimization through backpropagation	• Adaptive learning • Batching • Regularization





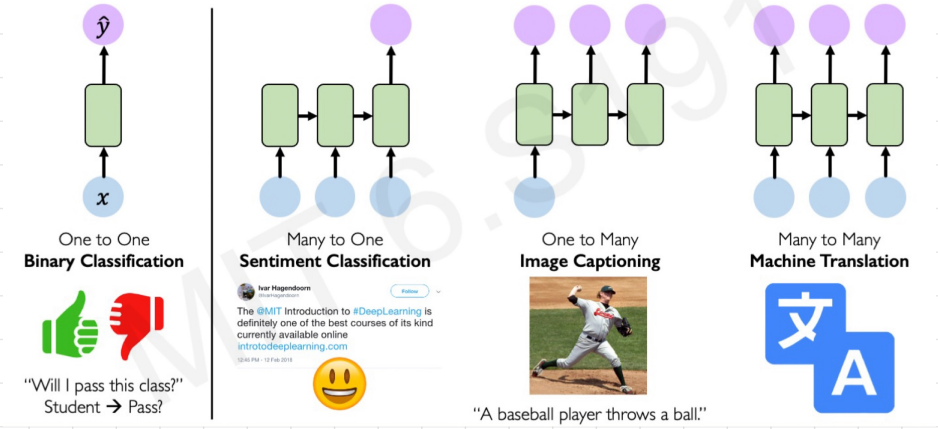


L2

DEEP SEQUENCE MODELING

- Lavoreremo su dati sequenziali nel tempo

Sequence modeling Applications



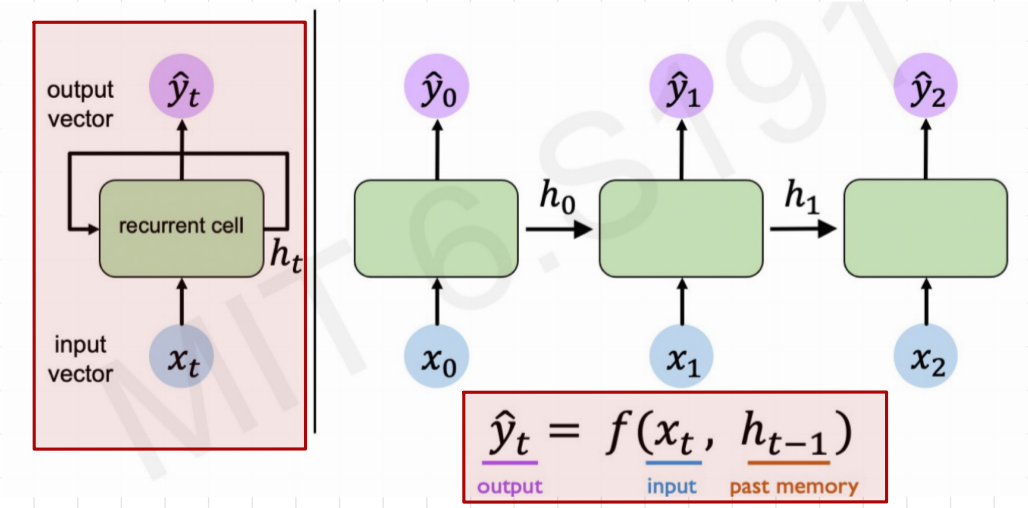
NEURONS WITH RECURRENCE

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

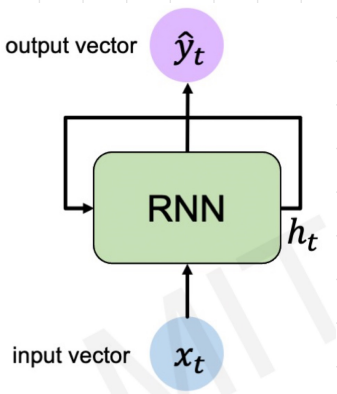
NEURONI CON RICORRENZA

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!"
```

fasi del processo

(1) INPUT VECTOR x_t

(2) UPDATE di h_t

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

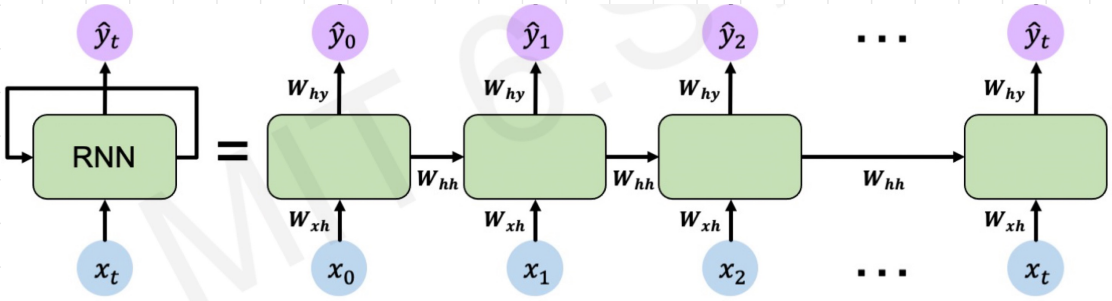
$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

NOTA: ci sono 3 WEIGHT MATRICES

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

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

... graficamente



OBSs

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

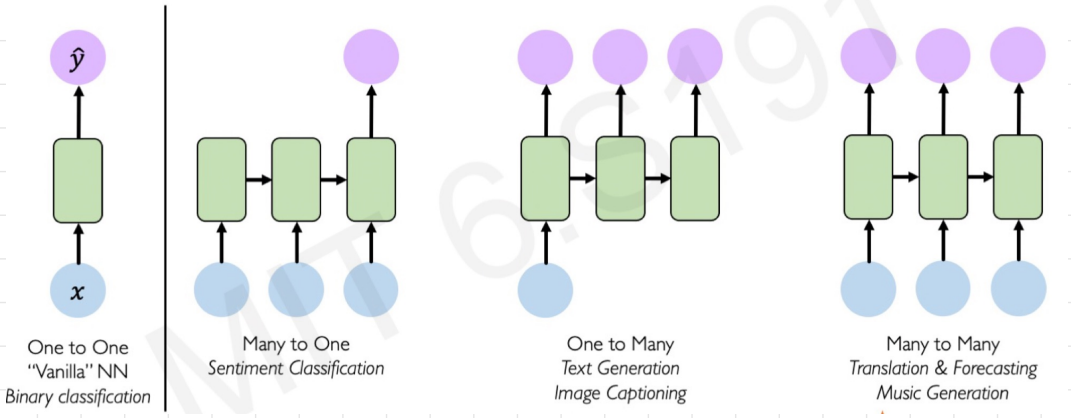
...python

```python
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, noon dobbiamo mica implementarlo

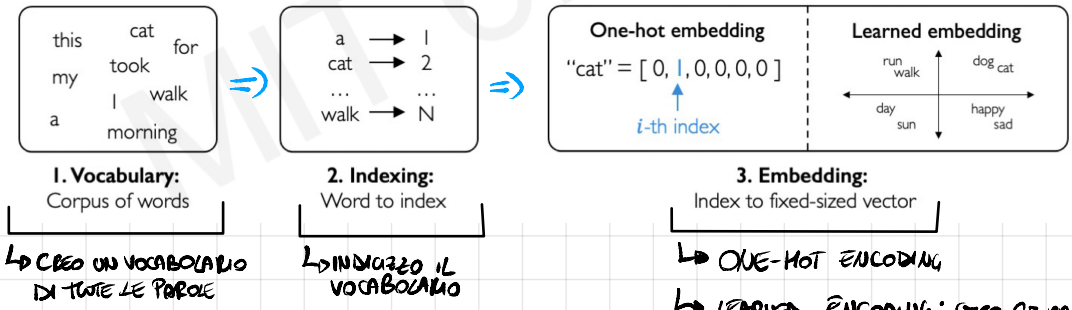
```python
tf.keras.layers.SimpleRNN(rnn_units)
```

RNNs per Sequential modeling



ENCODING ISSUE

SOL



↳ CREO UN VOCABOLARIO DI TUTTE LE PAROLE

↳ INDIRIZZO IL VOCABOLARIO

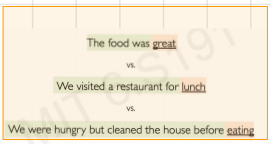
↳ ONE-HOT ENCODING

↳ LEARNED ENCODING: Creo gruppi di parole semanticamente simili

DESIGN CRITERIA

- Il Modello deve avere l'abilità di:

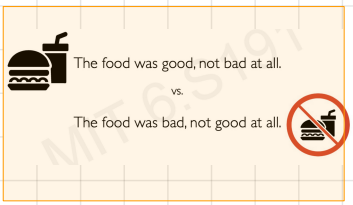
(1) Gestire seq. di LUNGHEZZA VARIABILE →



(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

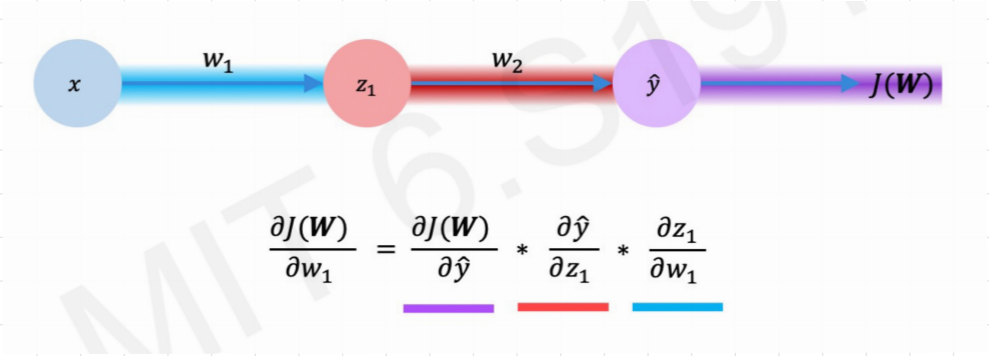


(4) CONDIVIDERE PARAMETRI tra le sequenze

BACKPROPAGATION THROUGH TIME (BPTT)

RIC

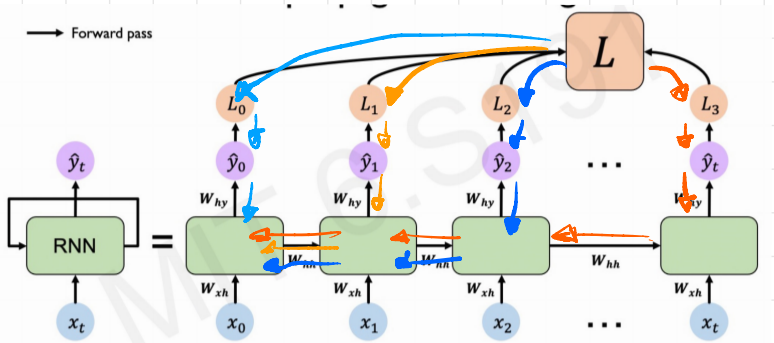
- Abbiamo visto che la **BACKPROPAGATION** consiste nel calcolare il GRADIENTE DELLA LOSS TOTALE ($\frac{\partial J(W)}{\partial w_1}$) rispetto ogni parametro ($\frac{\partial J(W)}{\partial w_1}$), e poi modificare i parametri per diminuire la loss.



RNN GRADIENT FLOW

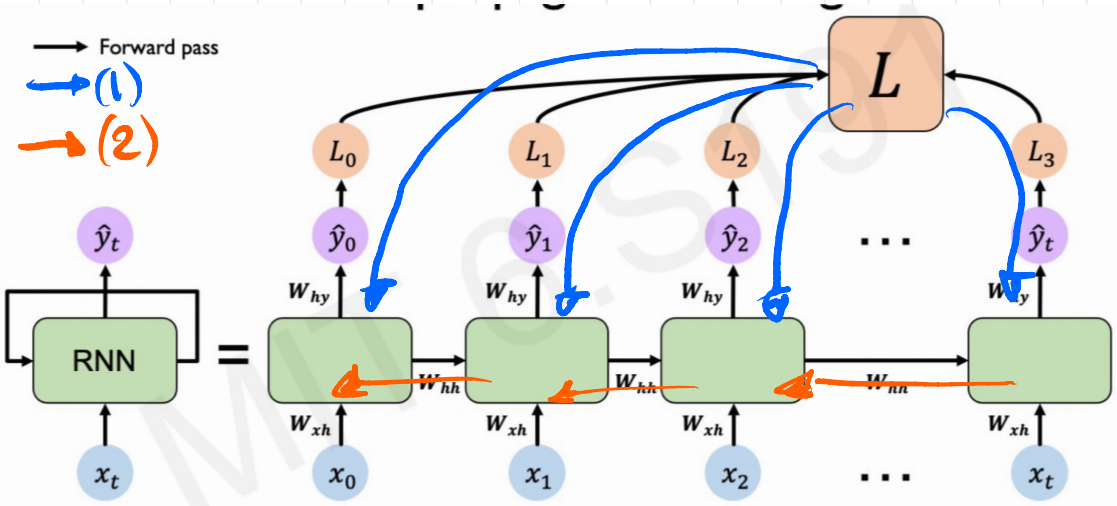
- Dalla total loss invece di computare la Backprop. per ogni TIME STEP

NO!

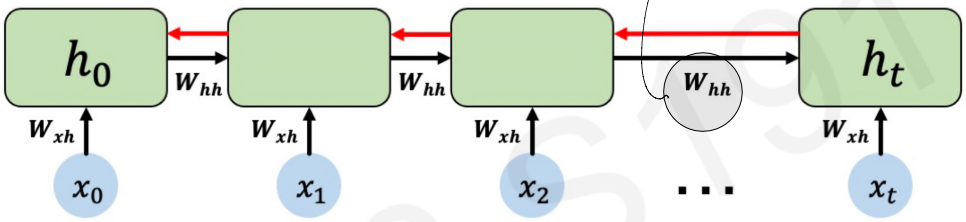


⇒ conviene fare un solo viaggio, dal TIME STEP più recente. In questo modo l'errore ce lo portiamo in una sola direzione.

SI!



- Calcolare il GRADIENTE rispetto h_0, comporta:
  - Molte moltiplicazioni per W_{hh}
  - Computare molti gradienti



EXPLODING GRADIENT

- Se abbiamo molti valori >1 in queste moltiplicazioni iniziamo ad avere numeri enormi per il GRADIENTE

VANISHING GRADIENT

- se abbiamo invece valori <1 i GRADIENTI diventano troppo piccoli

Why are vanishing gradients a problem?

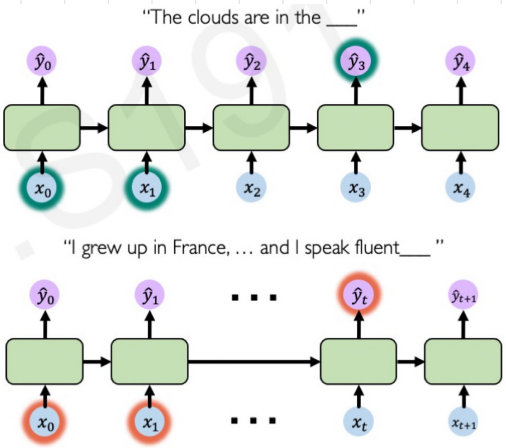
Multiply many **small numbers** together

↓

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

↓

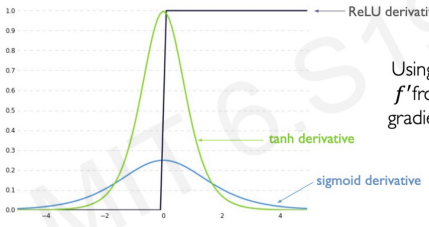
Bias parameters to capture short-term dependencies



SOL

- Ci sono vari trick per gestire i VANISHING GRADIENTS

Trick #1: Activation Functions



Using ReLU prevents f' from shrinking the gradients when $x > 0$

Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

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

This helps prevent the weights from shrinking to zero.

Solution #3: Gated Cells

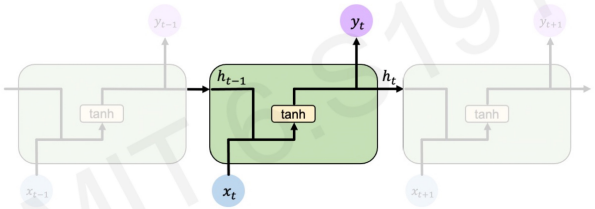
Idea: use a more **complex recurrent unit with gates** to control what information is passed through

gated cell
LSTM, GRU, etc.

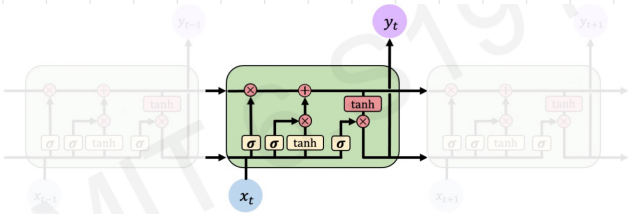
Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.

LONG SHORT TERM MEMORY

- STANDARD RNN



- LSTM RNN



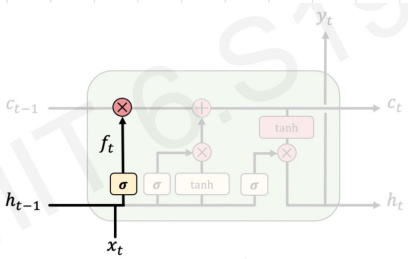

```
tf.keras.layers.LSTM(num_units)
```

- I GATES decidono quale informazione far passare o meno
- Lo fanno tramite, ad esempio, SIGMOIDI (come nelle imgs)
- Il parametro C_t è una variabile contenente informazioni passate

CELL STATE (TIPO h_t) →

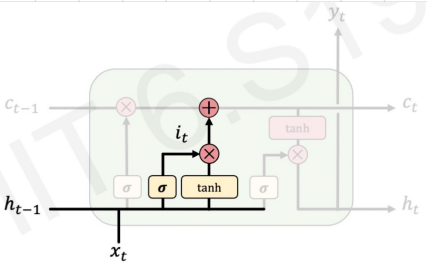
- Si divide in 4 fasi

(1)

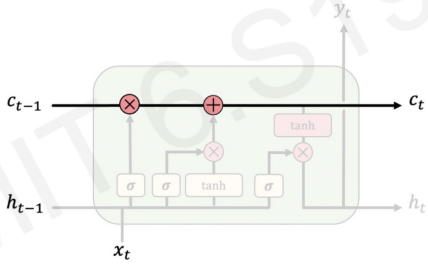


→ FORGET: butta parti irrilevanti dello stato precedente

(2) STORE: salva le info rilevanti sullo stato prima ←

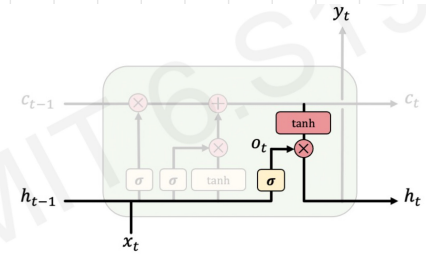


(3)

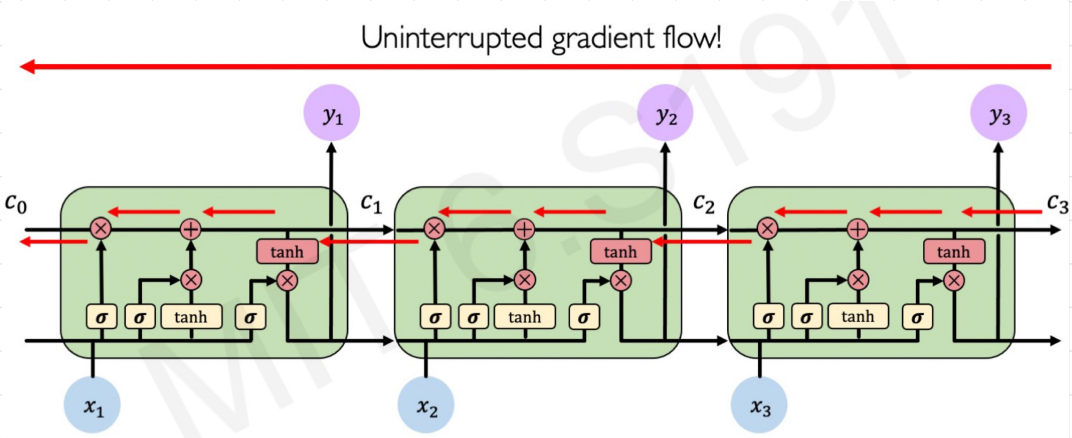


→ UPDATE: aggiorna il valore di C_t

(4) OUTPUT: controlla l'info mandata allo stato successivo ←



- Il GRADIENTE è ora calcolato attraverso i CELL GATES e elimina il 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