

Behind Neural Networks

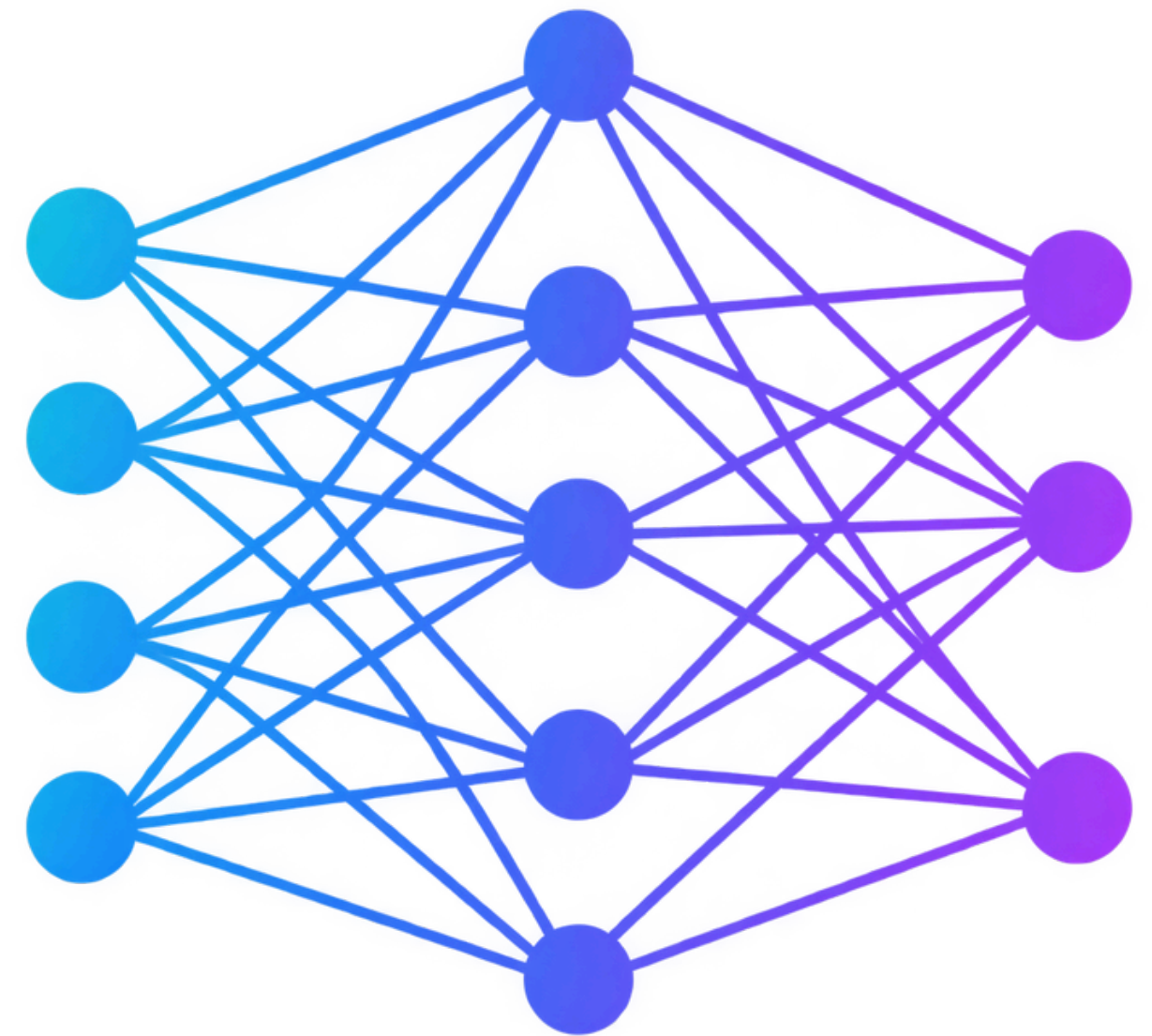
# INTRODUZIONE ALLE NEURAL NETWORKS

come funziona una rete neurale  
da un developer, per developers



**Filippo Zancanaro**

Software Engineer



**Marca User Group**

1 ottobre 2026

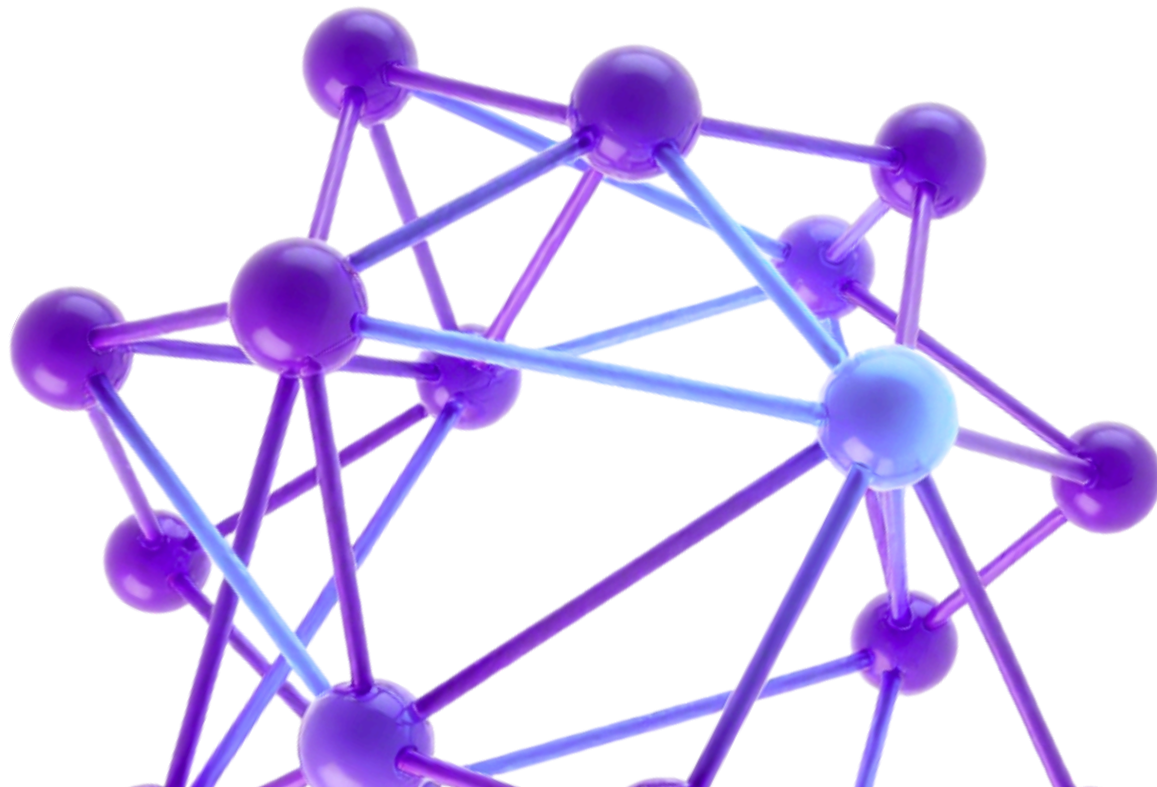
# ABOUT ME



Filippo Zancanaro

Software Engineer

*10+ anni di esperienza*



## Full Stack Developer

- Typescript + React & Angular
- C# + .NET
- Elixir & Phoenix
- Flutter



## Frontend, Backend,... Bartend?

- Bartender
- Shiba Inu #1 fan
- Gamer
- Gymrat

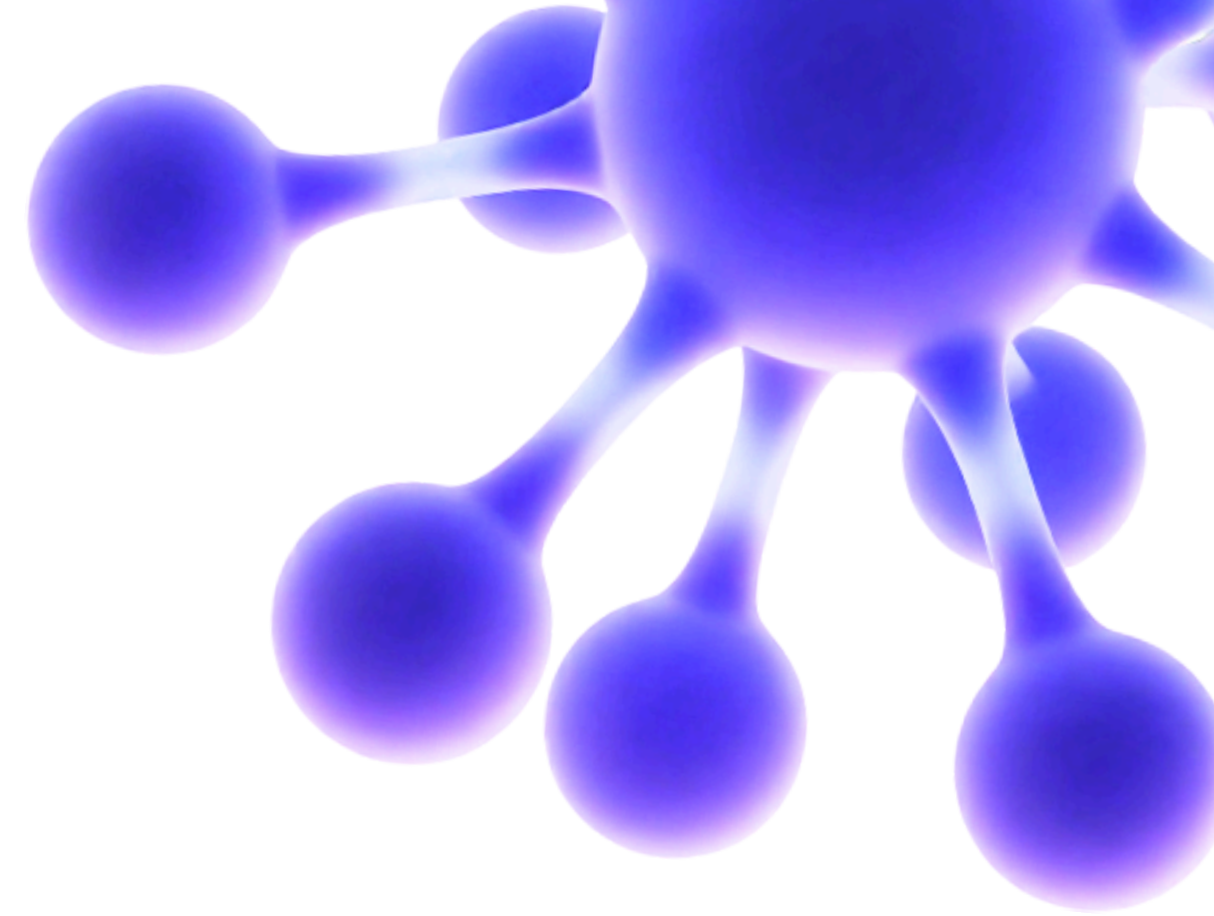


## Contatti

- [linkedin.com/in/filippoZancanaro](https://www.linkedin.com/in/filippoZancanaro)
- [zancanaro.filippo.97@gmail.com](mailto:zancanaro.filippo.97@gmail.com)
- [github.com/filippoZancanaro](https://github.com/filippoZancanaro)



# SAPPIAMO USARE I MODELLI, MA SAPPIAMO DAVVERO COME FUNZIONANO?



**01**

Come lavorano?

**02**

Come imparano?

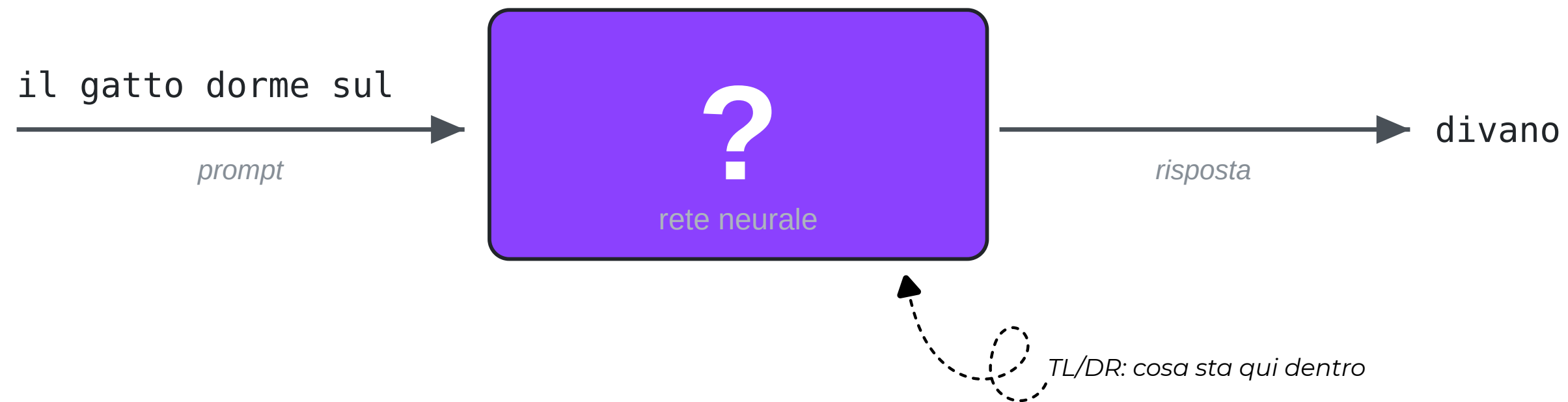
**03**

Perché a volte  
inventano?

**04**

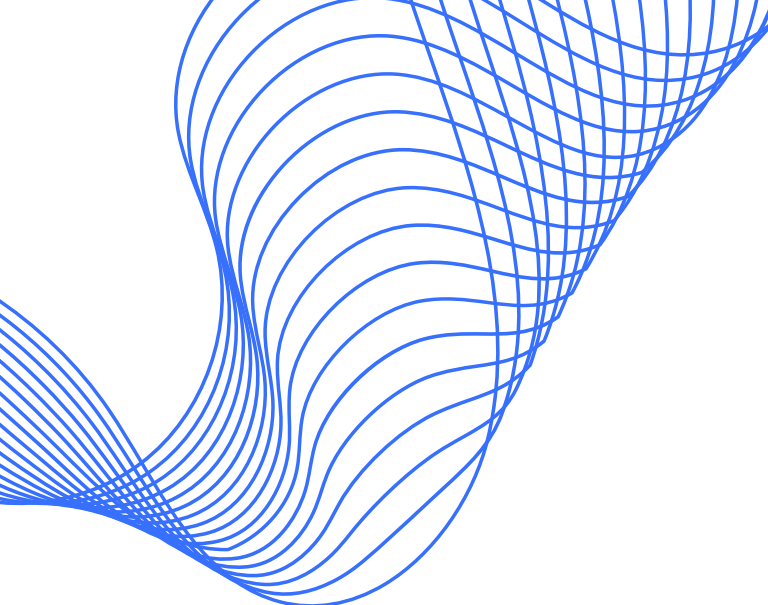
Come correggono gli  
errori?

# COSA VEDREMO OGGI



demistificheremo il funzionamento di una rete neurale

accompagnandoci poi con un semplice progetto di esempio:  
una mini app simil-T9 che, a fronte di una frase di input, suggerirà come output un  
completamento quanto più probabile possibile



# PREREQUISITI PER SEGUIRE IL TALK



## COSA CI SERVE

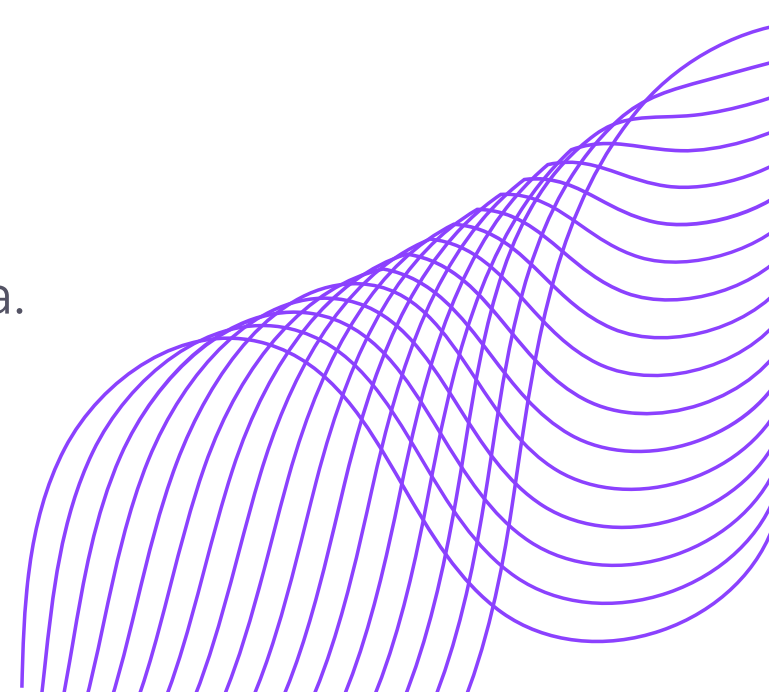
Sapere cos'è un array, cosa sono i cicli e cos'è una funzione.  
In pratica, le basi dello sviluppo software.



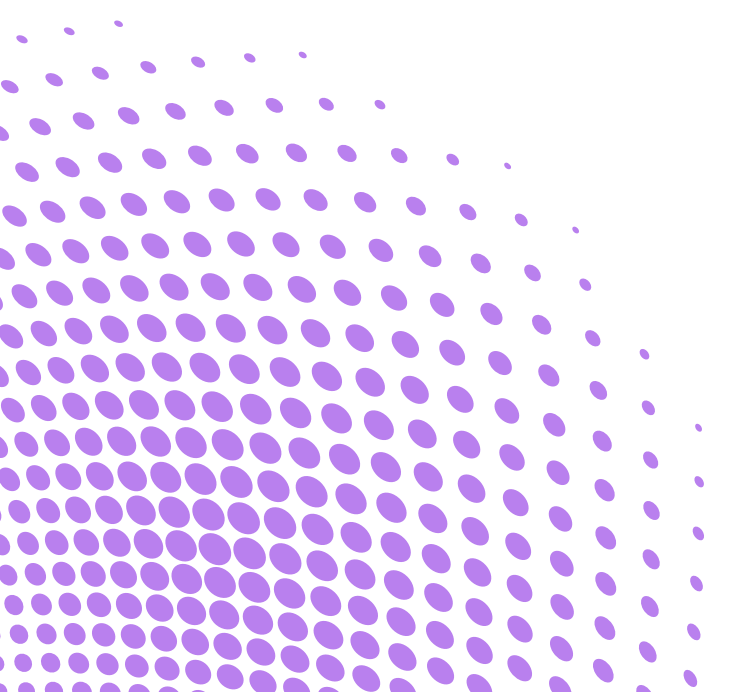
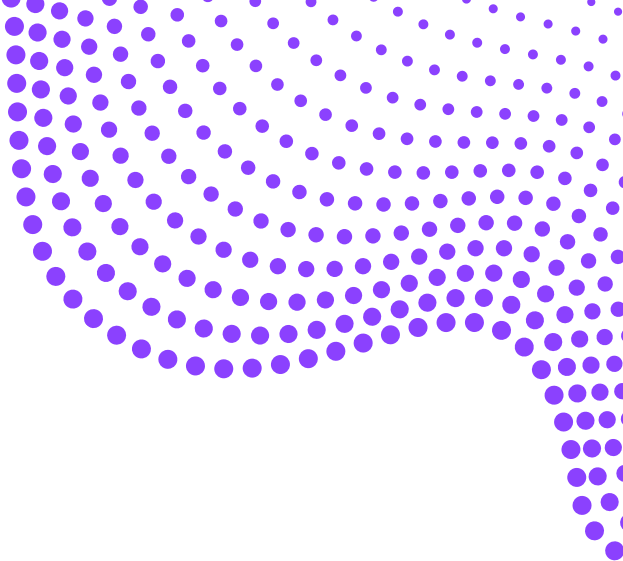
## COSA NON CI SERVE

Algebra lineare, conoscenze pregresse sul ML o un background matematico  
(e/o aver superato analisi 1 e 2 di ingegneria informatica, per fortuna).

Partiremo dalle basi e affronteremo ogni termine e concetto prima di vederlo all'opera.







# **UN PO' DI DEFINIZIONI**

# COS'È UN MODELLO?

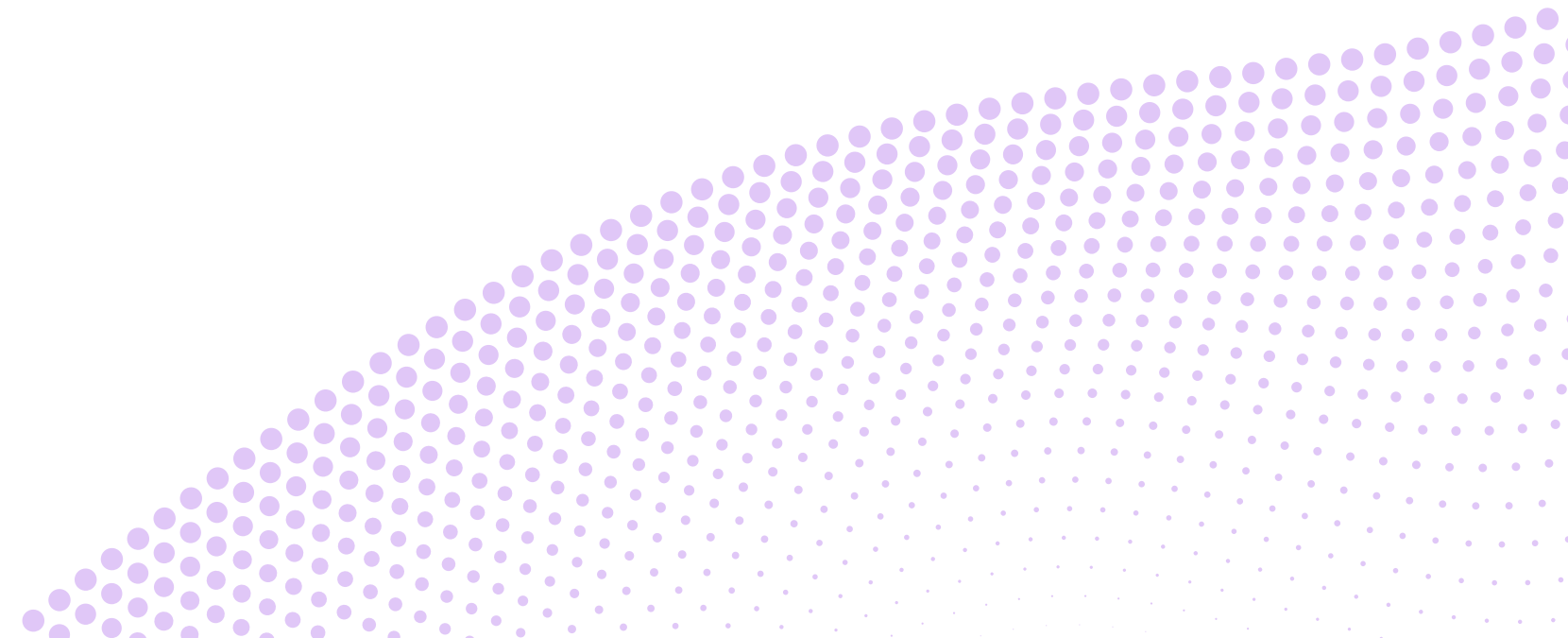


## MODELLO - DEFINIZIONE

Un modello è una **funzione parametrica**: **noi** possiamo **definirne l'architettura**, mentre i **valori** dei **parametri** vengono **appresi durante il training** tramite un algoritmo di ottimizzazione.

Durante il training, l'ottimizzatore modifica i parametri per ridurre la loss sul dataset di addestramento.

Una volta addestrato, il modello utilizza quei parametri durante l'inferenza per produrre le proprie previsioni.



# I PARAMETRI DI UN MODELLO



## COSA SONO

I **parametri** sono i **valori interni** del modello che vengono **appresi durante il training**.

Possono essere **singoli valori** oppure **tensori** contenenti migliaia o milioni di valori.

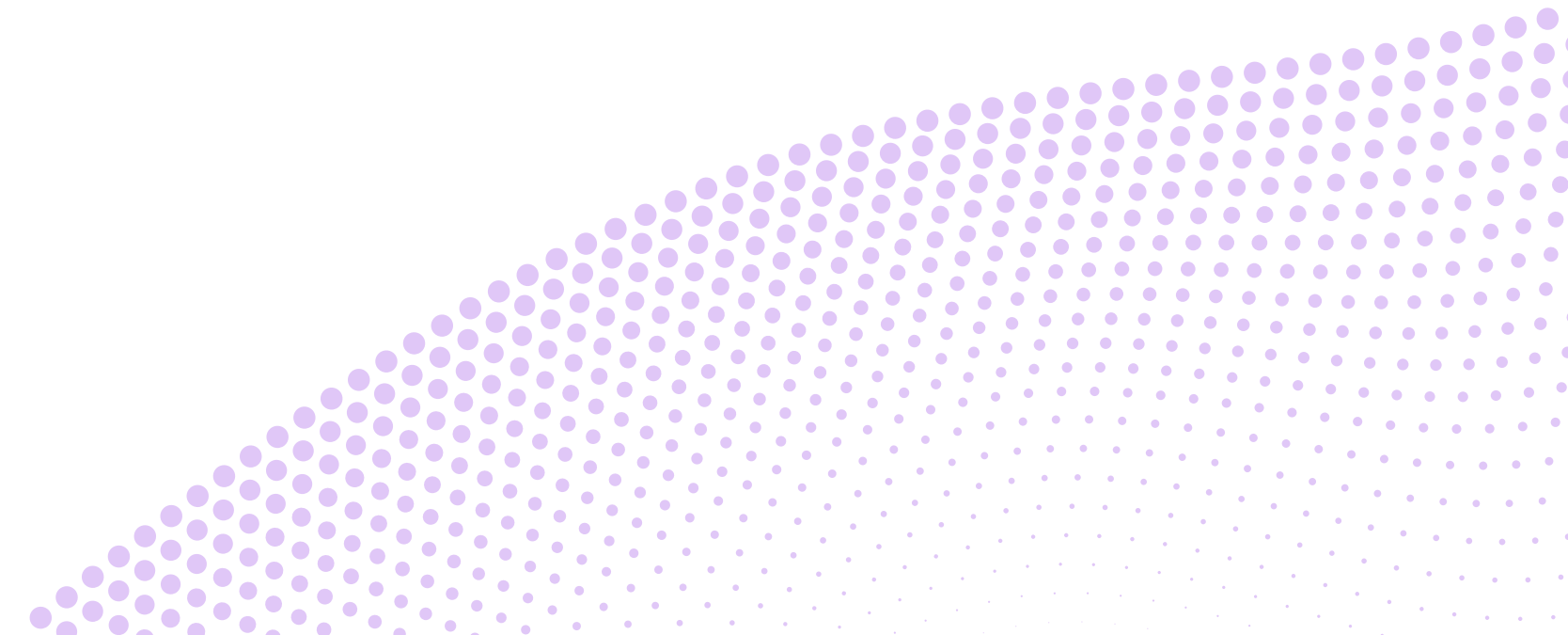
Durante il training, l'ottimizzatore modifica questi valori per ridurre la loss.



## CURIOSITA': QUANTO OCCUPANO

**Ogni valore** occupa tipicamente **2 o 4 byte**, a seconda della precisione utilizzata.

*Per esempio, 7 miliardi di valori a 2 byte  $\approx$  14 GB.*



# COS'È UN TENSORE?



## TENSORE

Un **tensore** è un **insieme multidimensionale di valori numerici**.

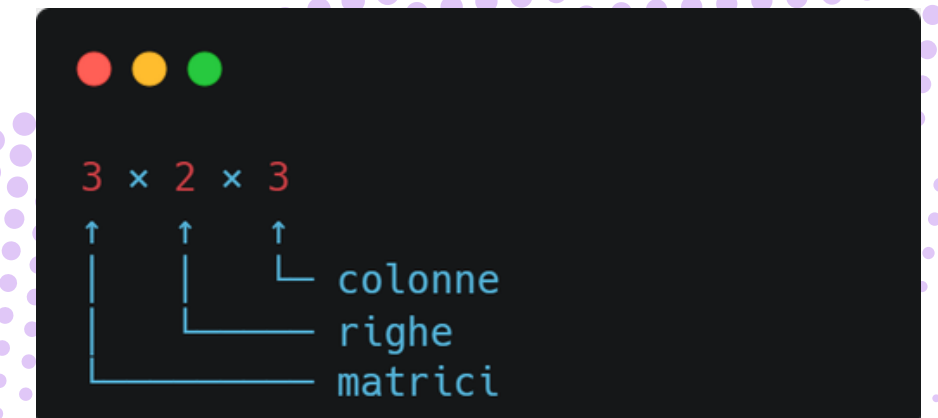
Per fare un paragone, una matrice potrebbe essere vista come un tensore a 2 dimensioni, mentre un tensore può avere 2, 3, 4 o più dimensioni.



## RAPPRESENTAZIONE SEMPLIFICATA

Per semplicità, possiamo immaginarlo come **più matrici sovrapposte**

```
[  
  [  
    [1, 2, 3],  
    [4, 5, 6]  
  ],  
  
  [  
    [7, 8, 9],  
    [10, 11, 12]  
  ],  
  
  [  
    [13, 14, 15],  
    [16, 17, 18]  
  ]  
]
```



# LE 2 FASI DI ESECUZIONE: TRAINING ED INFERENZA



## TRAINING

Il **training** è la fase in cui il **modello** utilizza molti esempi per **modificare gradualmente i propri parametri** e **migliorare le proprie predizioni**.

È generalmente la fase più costosa computazionalmente e può richiedere molto tempo.

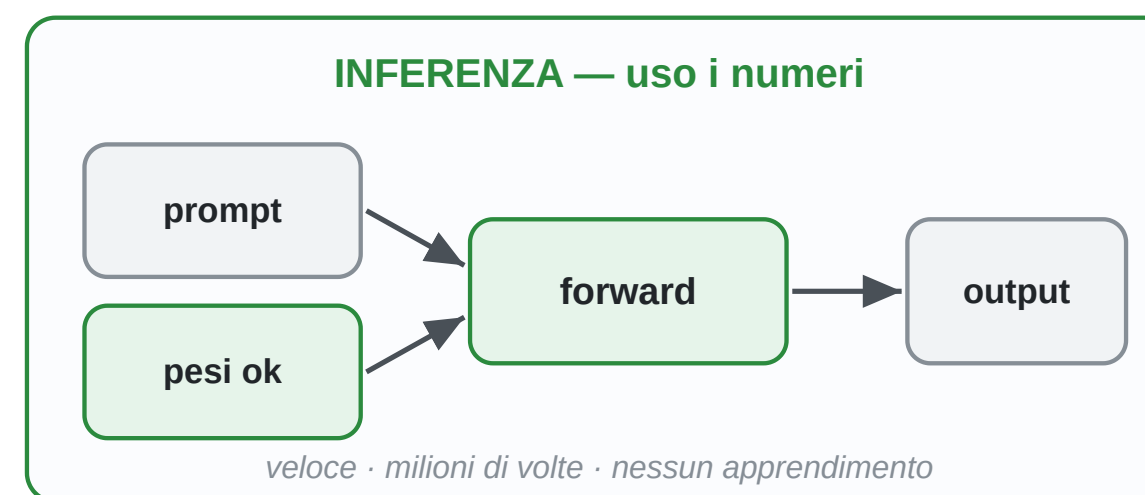
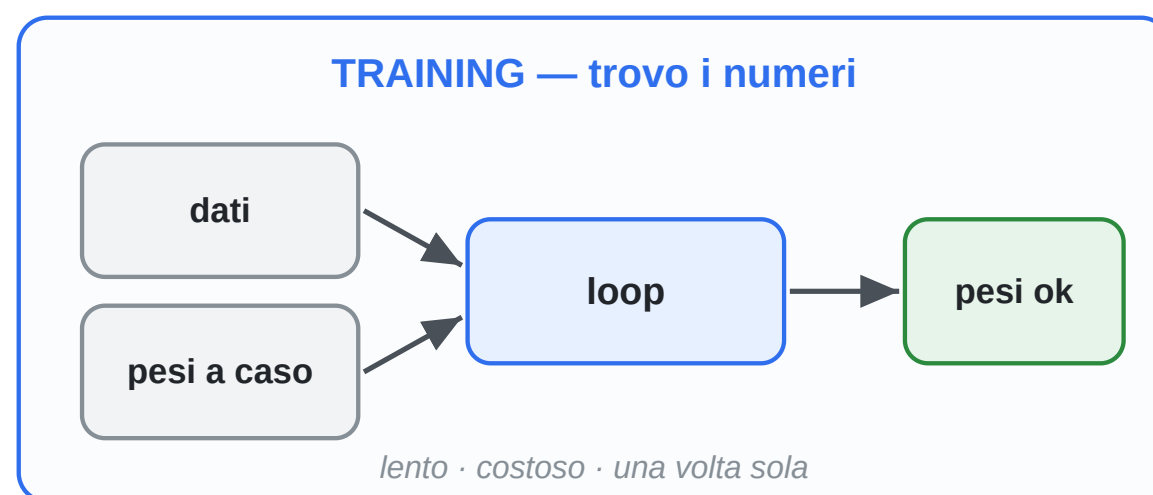
Una volta addestrato, il modello può essere utilizzato per l'inferenza; il training può comunque essere ripetuto o proseguito tramite retraining o fine-tuning.



## INFERENZA

Durante l'**inferenza** il **modello** utilizza i **parametri già appresi** per produrre una predizione a partire da un nuovo input.

**Non modifica i propri parametri** e, rispetto al training, richiede generalmente meno calcoli per singola predizione.





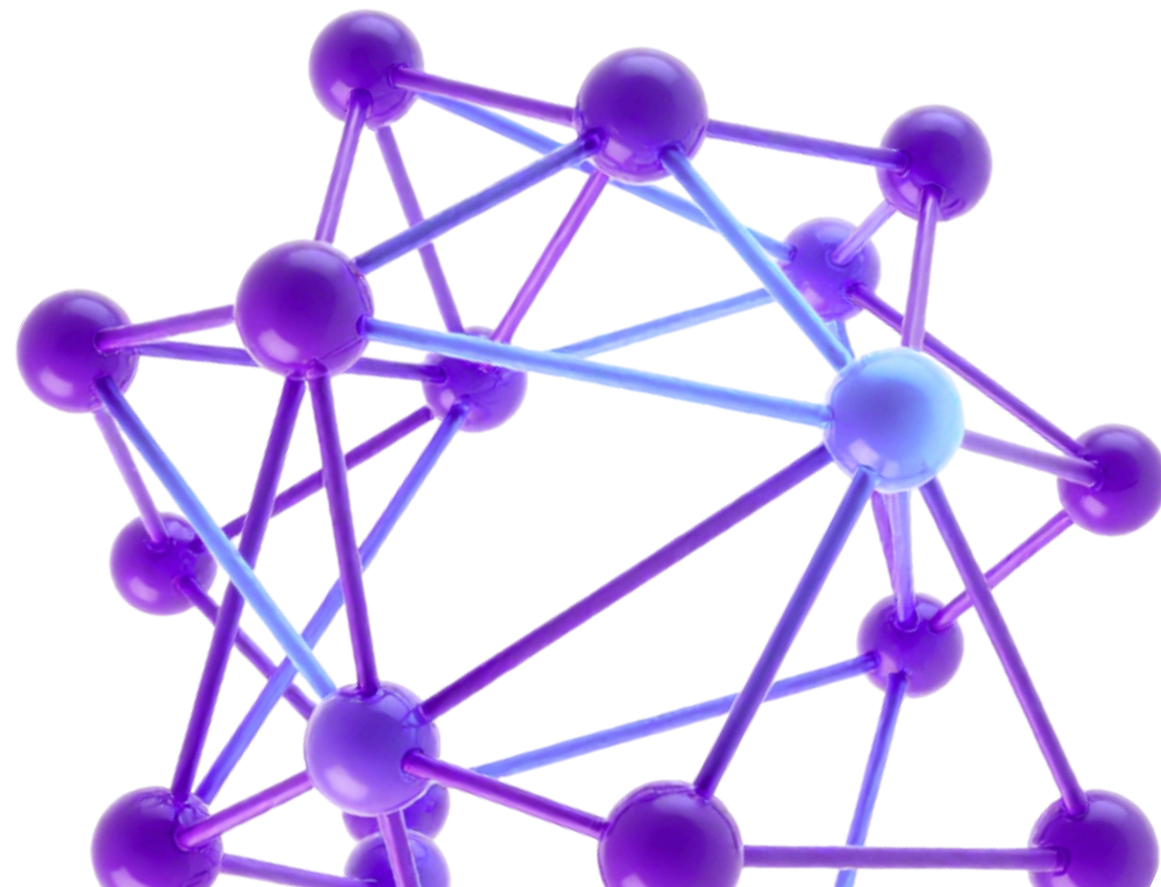
# IL NEURONE

Un neurone (virtuale) è un singolo tassello del puzzle virtuale che compone il layer nascosto di una rete neurale.

Fa sostanzialmente 3 cose:



```
output = f(x1·w1 + x2·w2 + x3·w3 + b)
```



## 1) Moltiplica

ogni input viene moltiplicato per il suo peso



## 2) Somma (e aggiusta il bias)

il risultato di ogni moltiplicazione viene sommato.

Questa somma viene poi sommata ad un valore di regolazione chiamato “bias” (che vedremo a breve)



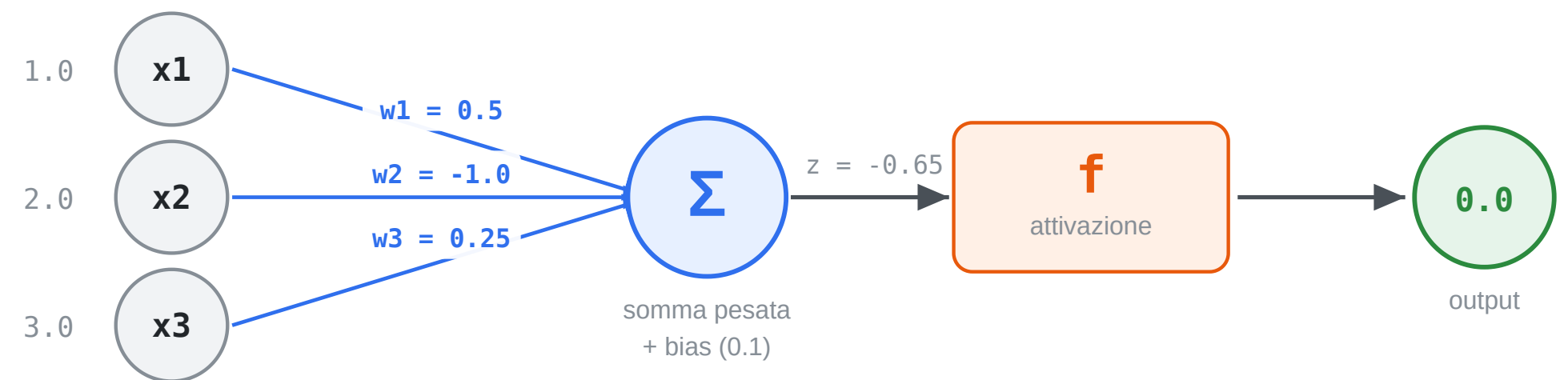
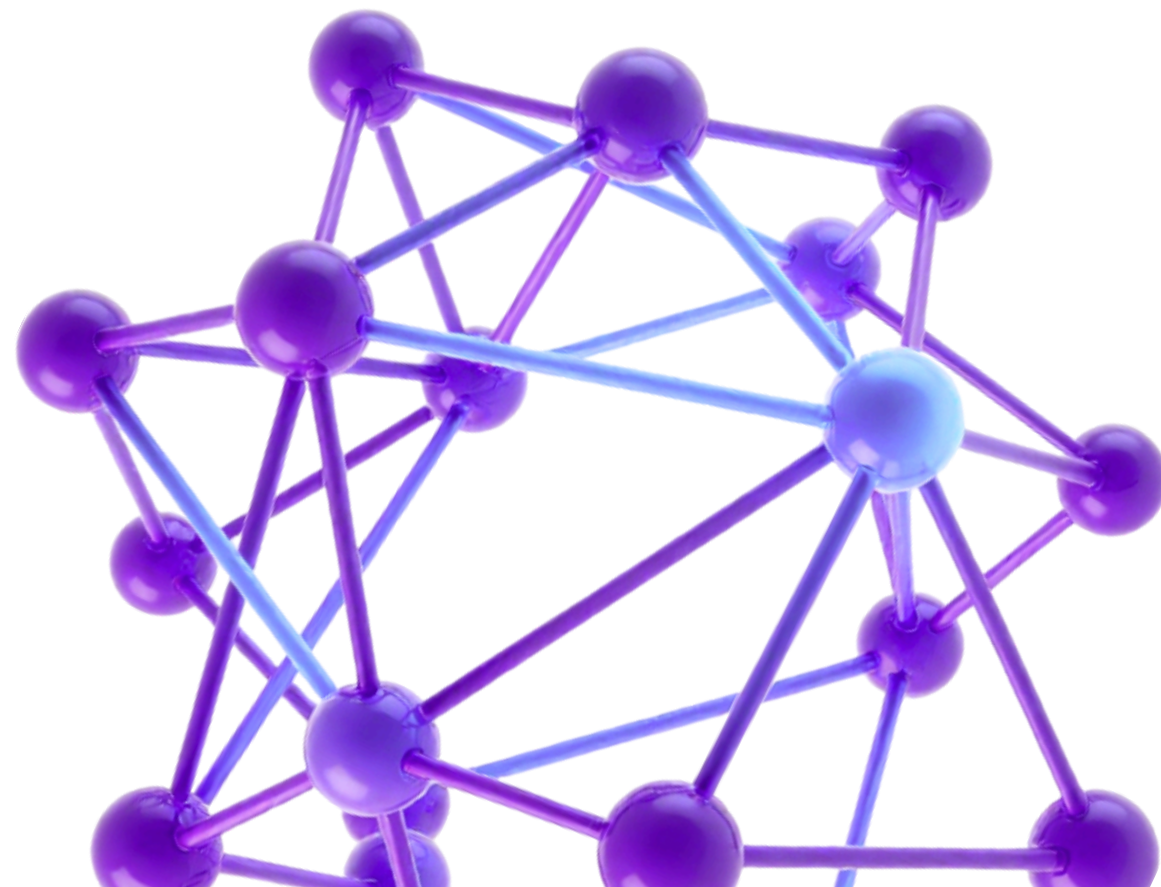
## 3) Schiaccia

il risultato della somma viene condensato con una “funzione di attivazione”, che decide se e con quale valore quel neurone deve attivarsi o meno

# IL NEURONE

```
def neurone(inputs, pesi, bias):  
    # 1. moltiplica e somma  
    z = sum(x * w for x, w in zip(inputs, pesi)) + bias  
    # 2. "schiaccia" (attivazione)  
    return max(0.0, z) # ReLU: negativo -> 0, positivo -> sé stesso  
print(neurone([1.0, 2.0, 3.0], [0.5, -1.0, 0.25], bias=0.1))  
# -> 0.0 perché 0.5 - 2.0 + 0.75 + 0.1 = -0.65, e ReLU lo azzerava
```

Un neurone: moltiplica, somma, schiaccia

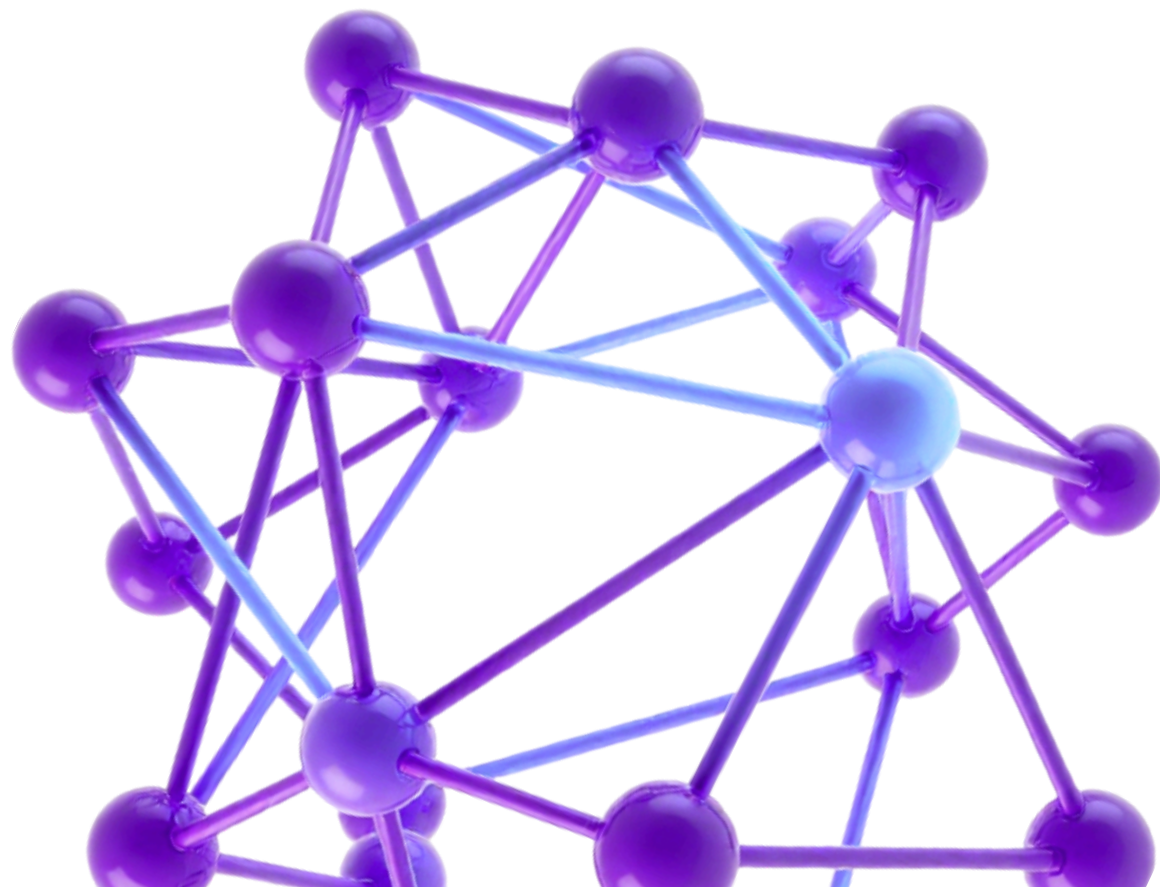


$output = f(x1 \cdot w1 + x2 \cdot w2 + x3 \cdot w3 + b)$  — con ReLU, un totale negativo diventa 0

# I PESI & IL BIAS



```
z = (input · pesi) + bias
```



## PESI

Il **peso** è un **moltiplicatore** per ogni input, che **decide quanto quell'input pesa/contribuisce nel risultato del neurone**. Sono la maggior parte dei parametri del modello.



## BIAS

Il **bias** è un **valore che viene sommato alla somma pesata** degli input.

Permette al neurone di **spostare il punto** in cui la **funzione di attivazione inizia a produrre determinati valori**.

È un **parametro**, proprio come i pesi, e viene **appreso durante il training**.

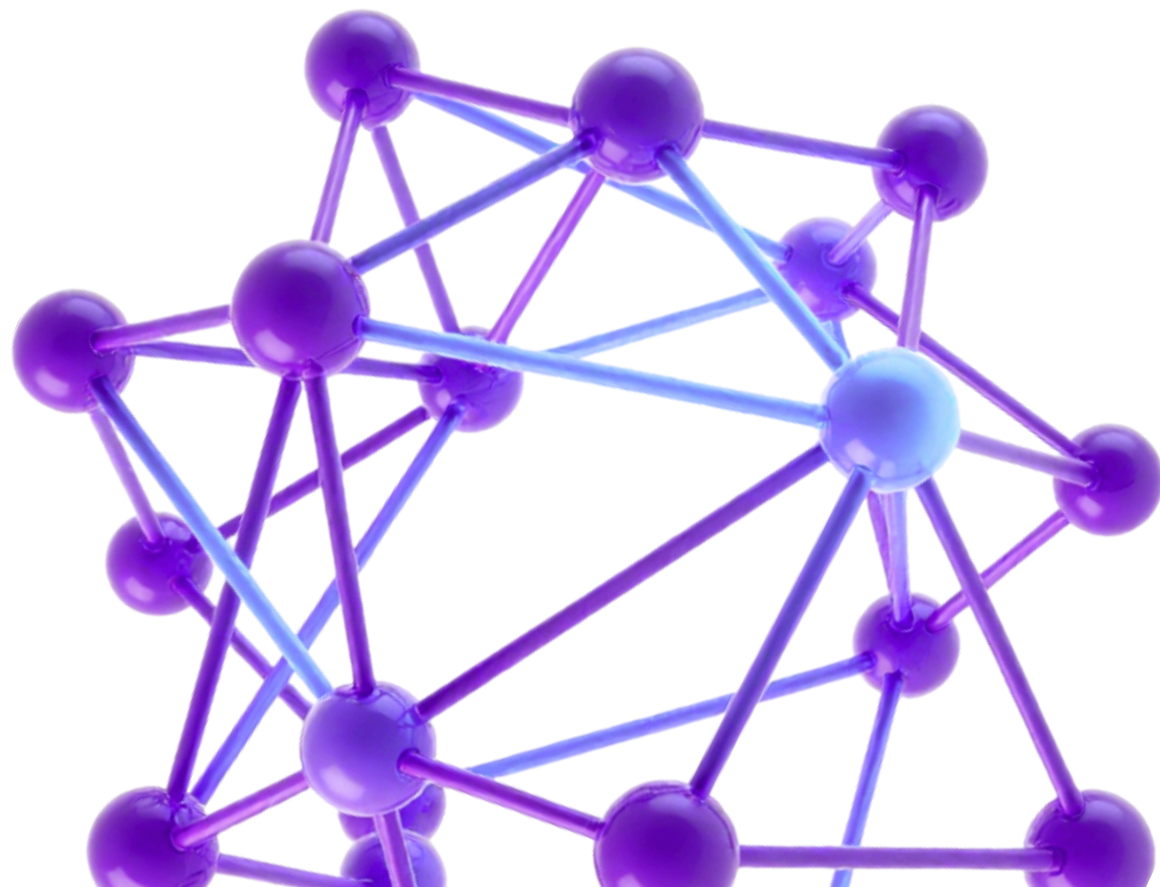
# LE FUNZIONI DI ATTIVAZIONE



## COSA SONO

Una funzione di attivazione è una **funzione non lineare applicata all'uscita di un neurone, prima di passare al layer successivo.**

In parole povere, la funzione di attivazione **permette alla rete neurale di imparare relazioni non lineari** e di svolgere **valutazioni complesse su più step** di valutazione.



## PERCHE' SERVONO

Cento layer senza attivazione hanno esattamente la stessa capacità espressiva di un solo layer.

Senza le funzioni di attivazione, tutti i layer collasserebbero in uno solo diventando poco più di una grande calcolatrice lineare, non potendo più riprodurre o gestire scenari complessi e valutazioni “su più step”.



# LE FUNZIONI DI ATTIVAZIONE QUALI SONO?

## Funzioni di attivazione: la funzione e la sua derivata

la derivata (tratteggiata) è ciò che decide se il gradiente riesce a passare: dove è vicina a zero, quel neurone smette di imparare





# ALCUNE FUNZIONI DI ATTIVAZIONE



## ReLU

Azzera i valori negativi e lascia invariati quelli positivi.  
È semplice e molto veloce, ed è stata a lungo una scelta comune nelle reti neurali.



## SIGMOID

Schiaccia qualsiasi valore nell'intervallo  $[0, 1]$ .  
È comunemente utilizzata, ad esempio, nell'output di classificatori binari.



## GELU

Una funzione di attivazione più morbida di ReLU, che riduce gradualmente l'input in base al suo valore.  
È utilizzata in molte architetture Transformer (una rete neurale che usa l'attention per capire quali parti dell'input sono rilevanti tra loro, anche quando sono molto distanti)



## Leaky ReLU

Come ReLU, ma mantiene una piccola pendenza anche per valori negativi.  
Riduce il problema dei neuroni che possono rimanere bloccati nella regione negativa.



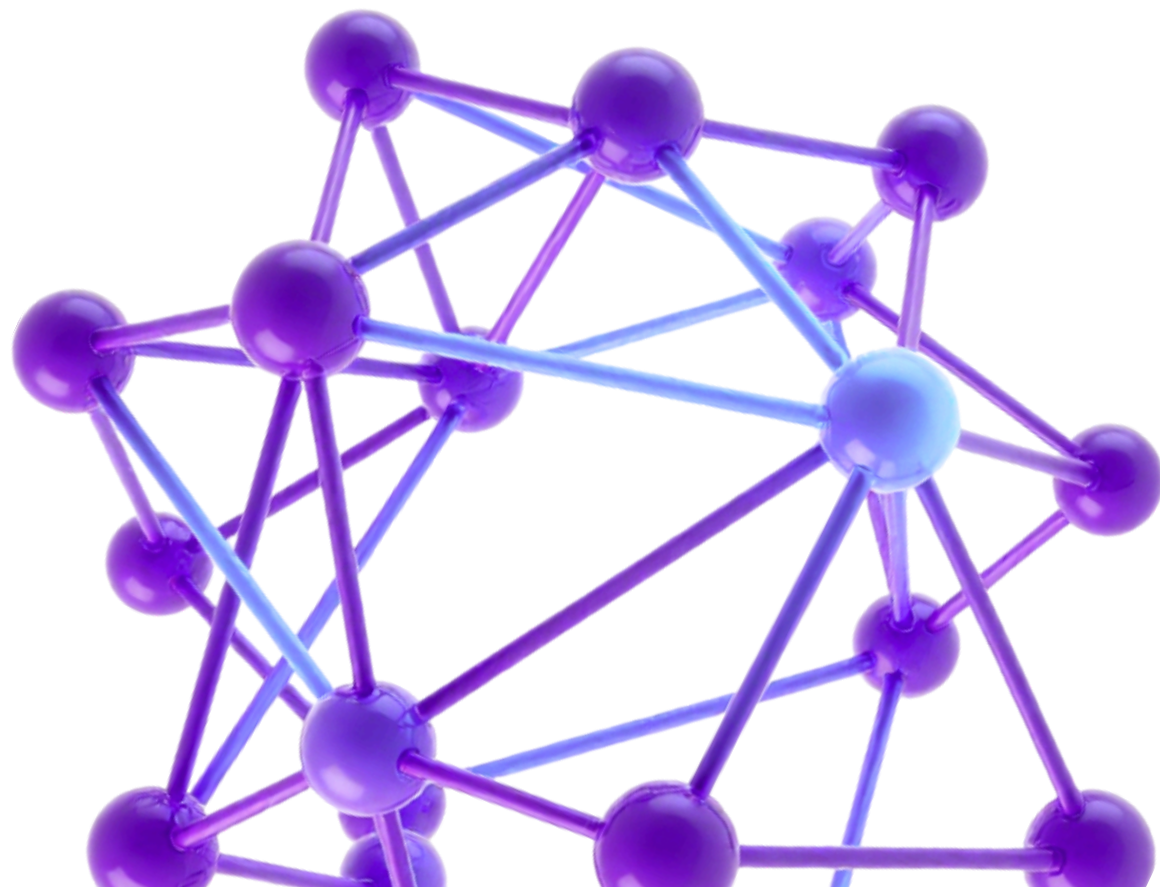
```
import torch
import torch.nn.functional as F

# valori iniziali
x = torch.tensor([-2.0, -0.5, 0.0, 0.5, 2.0])

# funzioni di attivazione + output commentato affianco:
print(F.relu(x)) # tensor([0.0000, 0.0000, 0.0000, 0.5000, 2.0000])
print(torch.sigmoid(x)) # tensor([0.1192, 0.3775, 0.5000, 0.6225, 0.8808])
```

# I DATI

3 rapide definizioni dei 3 principali macro-gruppi di dato che potremmo incontrare



## Training set

I dati usati per addestrare il modello.  
Il modello li utilizza per imparare e aggiornare i propri pesi.



## Validation set

I dati usati per verificare come sta andando durante lo sviluppo.  
Servono per confrontare configurazioni diverse e individuare problemi come l'overfitting o l'underfitting



## Test set

I dati tenuti separati fino alla valutazione finale.  
Servono per verificare quanto bene il modello funziona su dati che non ha mai visto.

# DA UN NEURONE AD UN LAYER

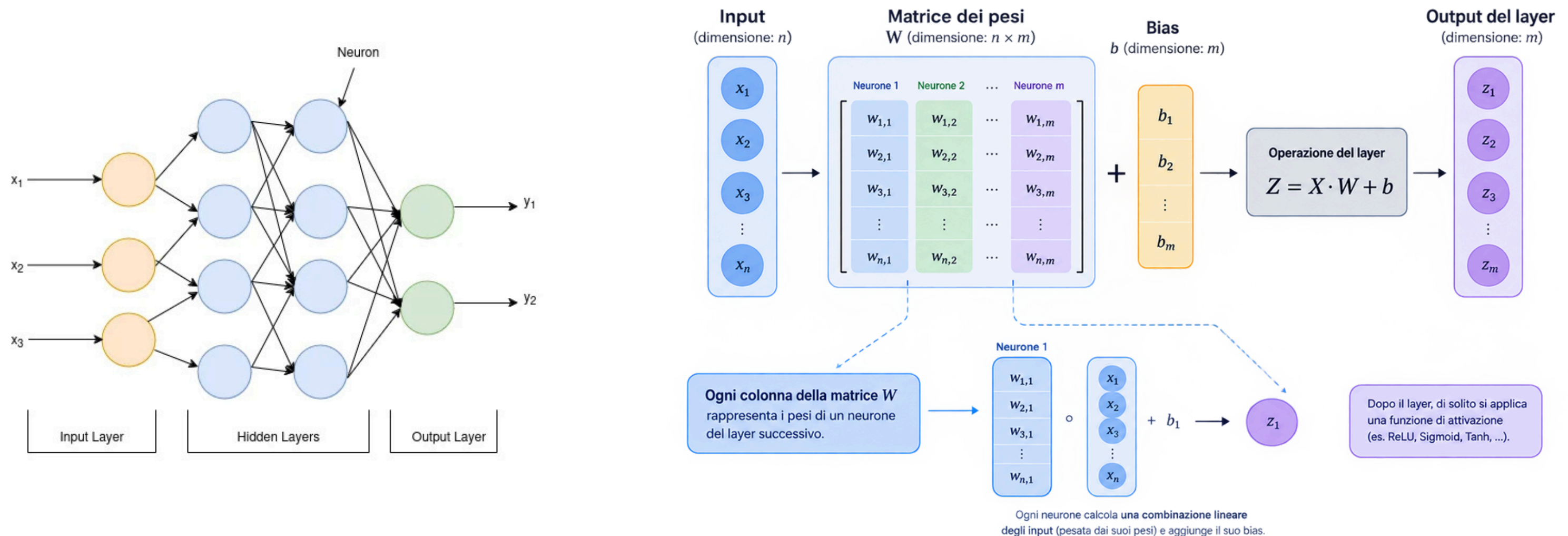


## UN LAYER = TANTI NEURONI IN PARALLELO

Ogni neurone ha i propri pesi.

Mettendo tutti questi pesi in tabella otteniamo una matrice: ogni colonna rappresenta un neurone del layer.

Un layer è semplicemente un insieme di neuroni che ricevono lo stesso input e producono insieme il prossimo livello di valori.



# DA UN NEURONE AD UN LAYER



## COME DEFINISCO QUANTI E QUALI LAYER CI SONO

La scelta di quanti layer ci sono e da quanti neuroni devono essere composti è, di solito, in mano a chi progetta l'architettura della rete neurale.



## CHI DECIDE COSA - l'architettura la decide il progettista

Decide, ad esempio:

- quanti layer ci sono
- quanti neuroni/nodi ha ogni layer
- come sono collegati
- quale funzione di attivazione usare
- che tipo di layer usare (Linear, Convolutional, Attention, ecc.)

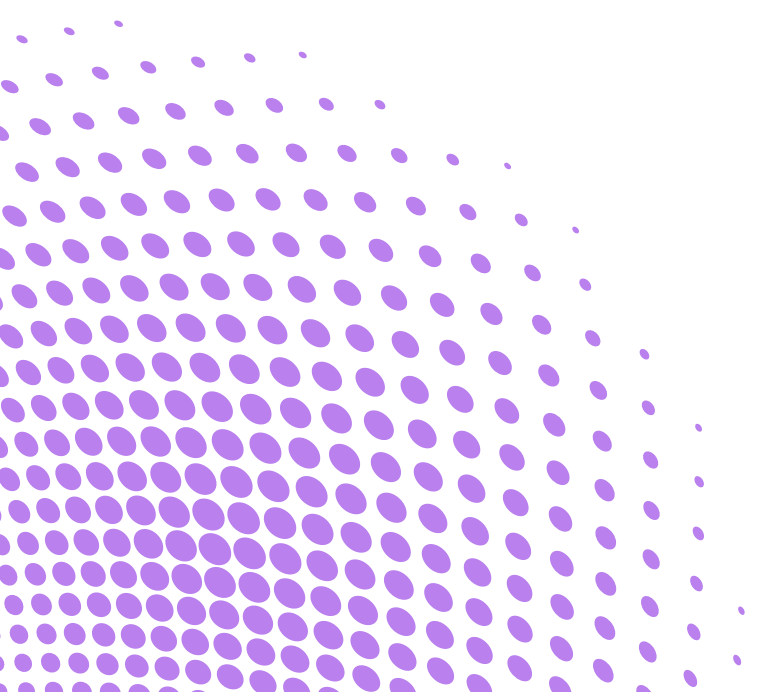
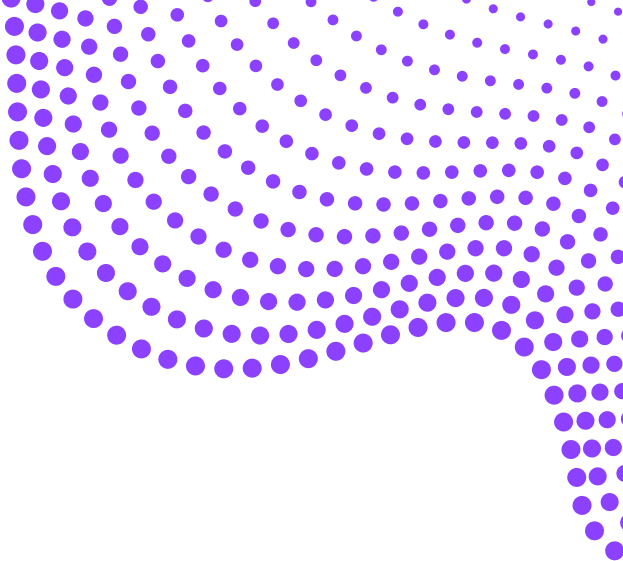
```
model = nn.Sequential(  
    nn.Linear(784, 128), # 784 input -> 128 neuroni  
    nn.ReLU(),          # funzione di attivazione applicata ai 128 output  
  
    nn.Linear(128, 64), # 128 input -> 64 neuroni  
    nn.ReLU(),          # nuova attivazione sui 64 output  
  
    nn.Linear(64, 10)   # 64 input -> 10 neuroni/output finali  
)
```



## CHI DECIDE COSA - i valori dei neuroni li impara il modello

Una volta stabilita l'architettura, inizialmente i pesi e bias sono generalmente inizializzati automaticamente ed aggiornati poi durante il training.





**MA IN TUTTO QUESTO,  
COME IMPARA?**

# IMPARARE DAGLI ERRORI



## ALL'INIZIO I PESI SONO CASUALI

Il modello appena creato dovrà addestrarsi per riuscire a valorizzare in modo corretto e sensato i suoi pesi.

Questa fase, come accennato in precedenza, si chiama **training**



## LA LOSS NEI MODELLI LINGUISTICI

Quando il modello deve prevedere il prossimo token, produce un punteggio/probabilità per ogni token del vocabolario.

Con una funzione chiamata cross-entropy, la loss prende la probabilità che il modello ha assegnato al token corretto e ne calcola il logaritmo negativo:  $-\log(p)$ , dove "p" è la probabilità che il modello ha assegnato al token corretto. Se  $p = 1$  la loss è 0; più p è bassa, più la loss cresce.

Esempio di loss: errore quadratico medio

```
def mse_loss(previsioni, veri):  
    return sum((p - v) ** 2 for p, v in zip(previsioni, veri)) / len(veri)  
  
print(mse_loss([3.0, 5.0], [3.0, 5.0])) # 0.0 -> perfetto  
print(mse_loss([2.0, 8.0], [3.0, 5.0])) # 5.0 -> male
```



## SERVE UN MODO PER MISURARE L'ERRORE

E, come accennato in precedenza, è qui che entra in gioco la funzione di perdita, la "loss".

Loss alta = maggiore errore, Loss bassa = minore errore.

Lo scopo del training è ridurre il valore.

### Loss

La loss è una funzione che prende le previsioni del modello e i valori attesi, e restituisce un singolo numero che misura quanto le due cose divergono. Zero (o il minimo) significa previsioni perfette; più è alta, più il modello sbaglia.

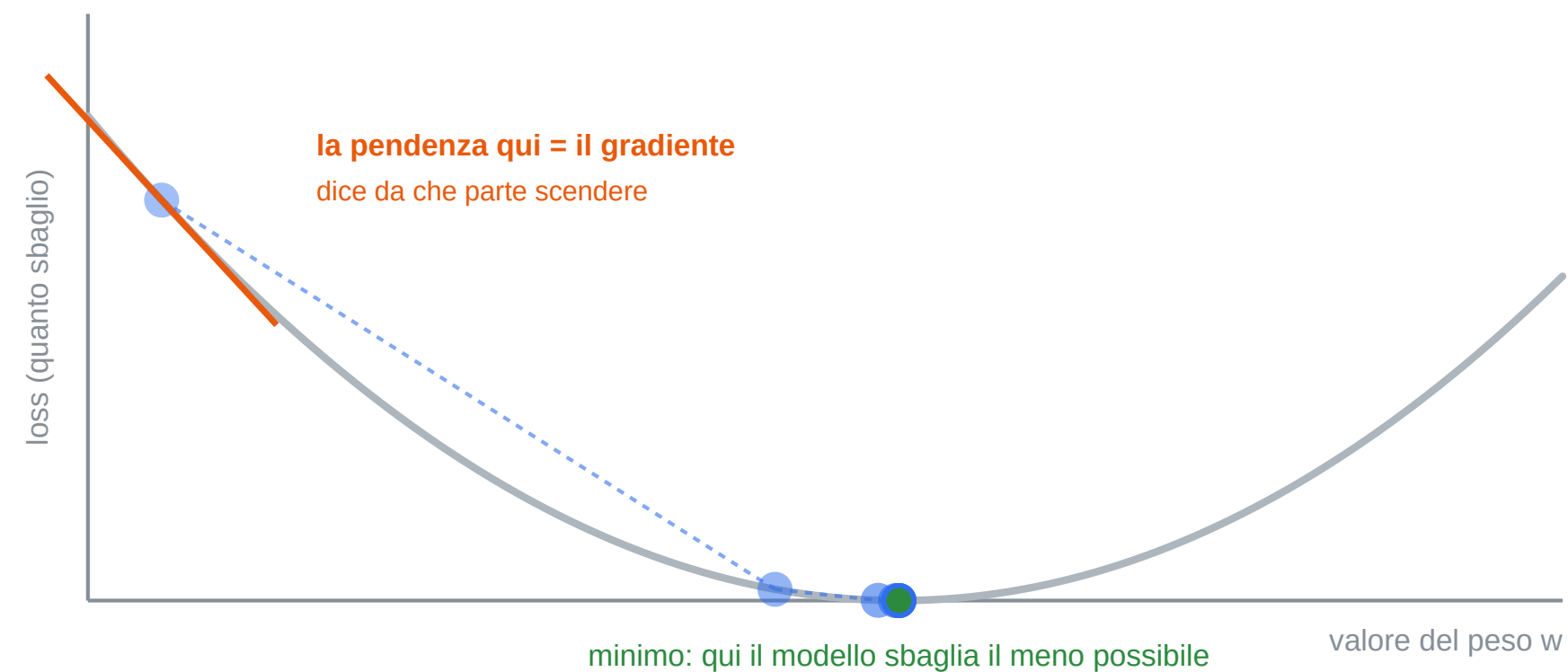
# LA DISCESA DEL GRADIENTE

*Sei su una collina, nella nebbia.*

*Vuoi arrivare in fondo ma non vedi la valle, ma solo la pendenza nelle immediate vicinanze.*

1. senti da che parte scende
2. fai un passo in quella direzione
3. ripeti

**Discesa del gradiente: seguo la pendenza, un passo alla volta**



*Ogni pallina rappresenta un passo del training.*

*La direzione dell'aggiornamento dipende dal gradiente, mentre la sua dimensione dipende dal learning rate e dall'algoritmo di ottimizzazione.*

*Vicino a un minimo, il gradiente può diventare piccolo e gli aggiornamenti possono quindi diventare più piccoli.*



# LA DISCESA DEL GRADIENTE

Il gradiente risponde, per ogni parametro, a una domanda:



“SE AUMENTO DI POCHISSIMO QUESTO PARAMETRO,  
LA LOSS SALE OPPURE SCENDE?”

Se la derivata è positiva, l'aggiornamento di gradient descent tende a diminuire il parametro.

Se è negativa, tende ad aumentarlo.

Il processo viene applicato a tutti i parametri del modello.



# IL LEARNING RATE

Tornando all'esempio della collina, quanto grande lo facciamo il passo?

E' qui che entra in gioco il...



## LEARNING RATE

Un **iperparametro** (ovvero un parametro deciso da chi scrive il codice e non auto-regolato dal modello) che determina, appunto quanto deve essere grande ogni passo con cui il modello modifica i propri parametri durante il training.

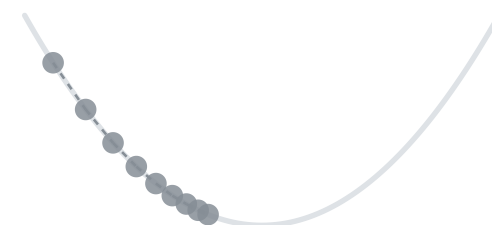


## LE DIMENSIONI CONTANO

Se è troppo alto, il modello rischia di saltare oltre il minimo; se è troppo basso, impara molto lentamente.

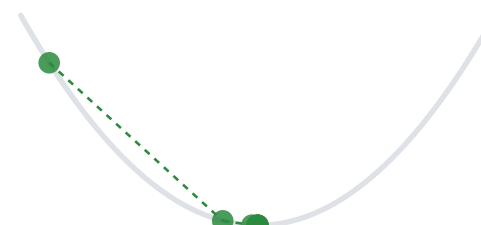
Il learning rate è la lunghezza del passo

troppo piccolo



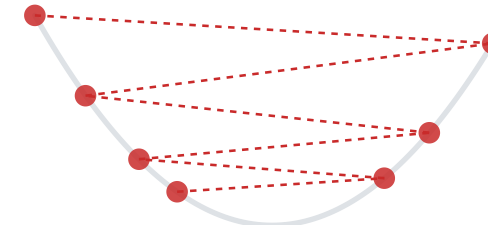
arrivi, ma dopo ore

giusto



poche decine di passi

troppo grande



può saltare oltre il minimo e rendere il training instabile





# LA BACKPROPAGATION

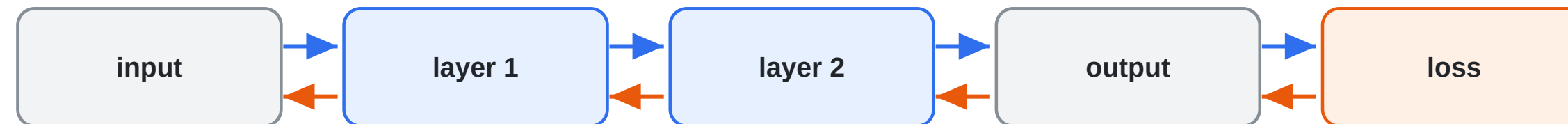


## BACKPROPAGATION

La backpropagation è il meccanismo con cui la rete neurale, dopo aver calcolato la loss, torna indietro attraverso i vari layer precedenti per calcolare come la loss cambia rispetto a ciascun parametro.

**Forward: calcolo. Backward: attribuzione della colpa.**

FORWARD → quanto vale?



← BACKWARD di chi è la colpa?



# OPTIMIZER

chi modifica i gradienti durante la backpropagation?



## OPTIMIZER

L'optimizer è l'algoritmo che usa i gradienti calcolati dalla backpropagation per modificare i parametri della rete, cercando di ridurre la loss.



# AUTOGRAD

La buona notizia, è che PyTorch, una delle librerie Python più usate per il ML, ci fornisce uno strumento utile:



## AUTOGRAD

Autograd è lo strumento di PyTorch che automatizza il calcolo dei gradienti che servono alla backpropagation.

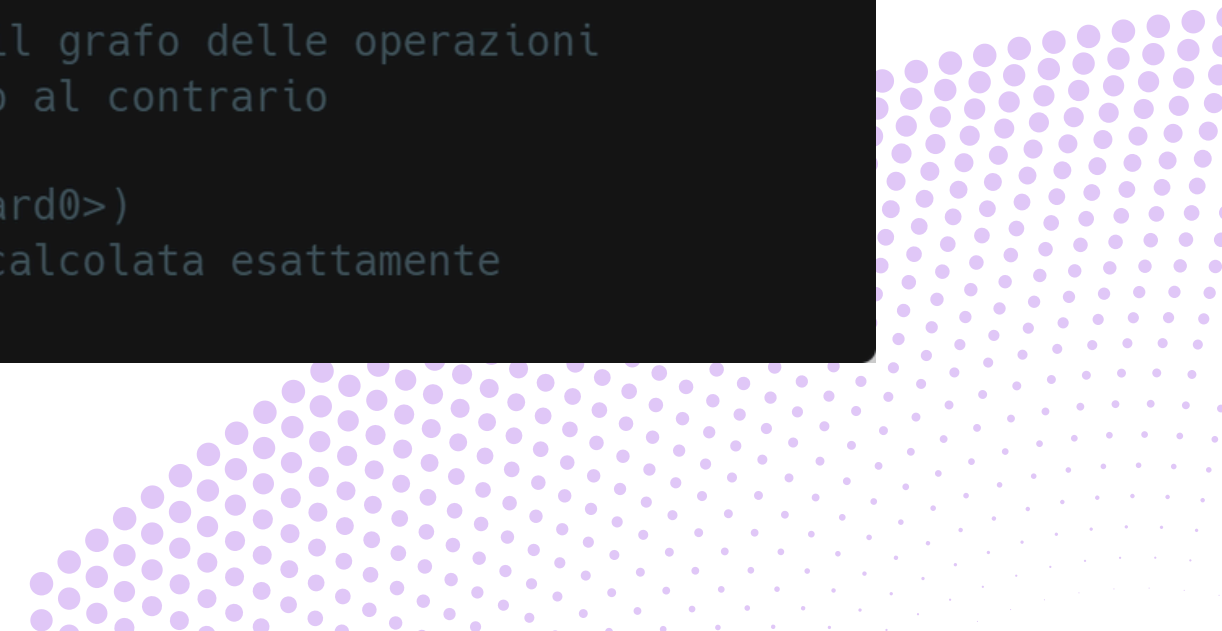


```
import torch

# requires_grad=True dice a PyTorch: "tieni traccia di cosa faccio con questo"
w = torch.tensor([2.0], requires_grad=True)

loss = (w - 3.0) ** 2 # forward: costruisce il grafo delle operazioni
loss.backward() # backward: percorre il grafo al contrario

print(loss) # tensor([1.], grad_fn=<PowBackward0>)
print(w.grad) # tensor([-2.]) <- d(loss)/dw calcolata esattamente
```



# EPOCH



## COS'E' UN'EPOCA

Un'epoca è un passaggio completo attraverso tutti i dati del training set.



## FUNZIONAMENTO

Durante ogni epoca, i dati vengono elaborati in batch e i pesi della rete vengono aggiornati tramite:

Forward -> Loss -> Backpropagation -> Optimizer



## LIMITI

- La quantità di epochs, essendo un iperparametro, è impostabile dallo sviluppatore.
- Non sempre avere tante epoche porta a risultati migliori: troppe epoche potrebbero causare **overfitting**.
- L'epoch è un concetto del training, non dell'inferenza.

### ESEMPIO

10.000 esempi di training  
Batch size = 100

- 100 batch per epoch
- 100 aggiornamenti dei pesi per epoch

5 epoch

- 500 aggiornamenti complessivi





# OVERFITTING & UNDERFITTING

Due dei rischi durante la fase di training



## OVERFITTING

Quando un modello impara troppo bene i dati di training, finendo per memorizzarne i dettagli invece di imparare pattern generalizzabili.

Risultato: ottime performance sui dati già visti, ma peggiori su dati nuovi.



## UNDERFITTING

il modello non riesce a imparare abbastanza dai dati, risultando impreciso anche sul training set.



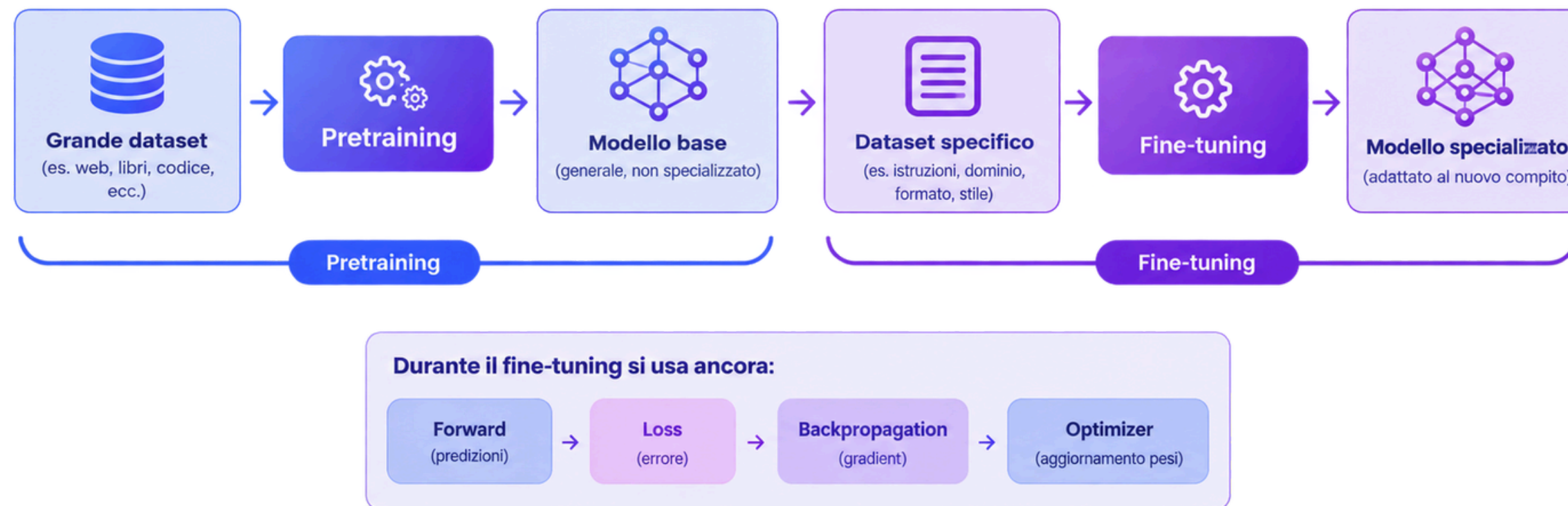


# IL FINE TUNING



## FINE TUNING

Il fine-tuning consiste nel continuare il training di un modello già addestrato, utilizzando un dataset più piccolo e specifico per adattarlo a un determinato compito, comportamento o dominio.



# RICAPITOLANDO

COME FUNZIONA IL LOOP DI TRAINING?



# VALUTAZIONE DEL MODELLO



## 1) PREDIZIONE

Il modello produce un risultato a partire dagli input.



## 2) LOSS

Misuriamo quanto la predizione è distante dal risultato atteso.



## 3) GRADIENTE

Calcoliamo come ogni peso contribuisce all'errore e in quale direzione modificarlo.



## 4) OPTIMIZER

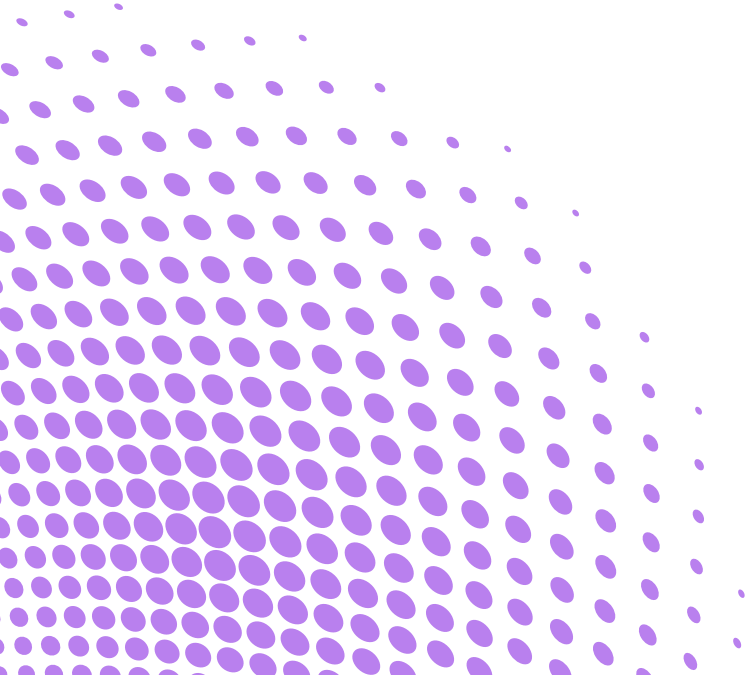
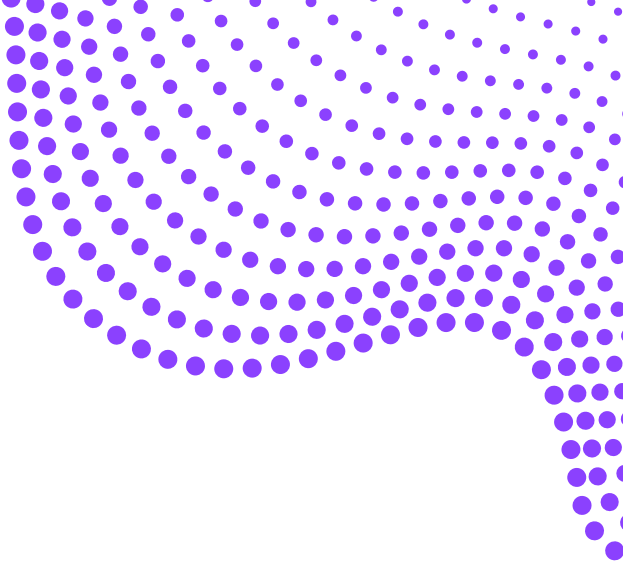
Usa queste informazioni per aggiornare i pesi.



## ↻) ....& repeat

Ripetiamo il processo su molti esempi, per minimizzare la loss e ottenere un modello che generalizzi bene su nuovi dati





**DAL TESTO AI NUMERI**

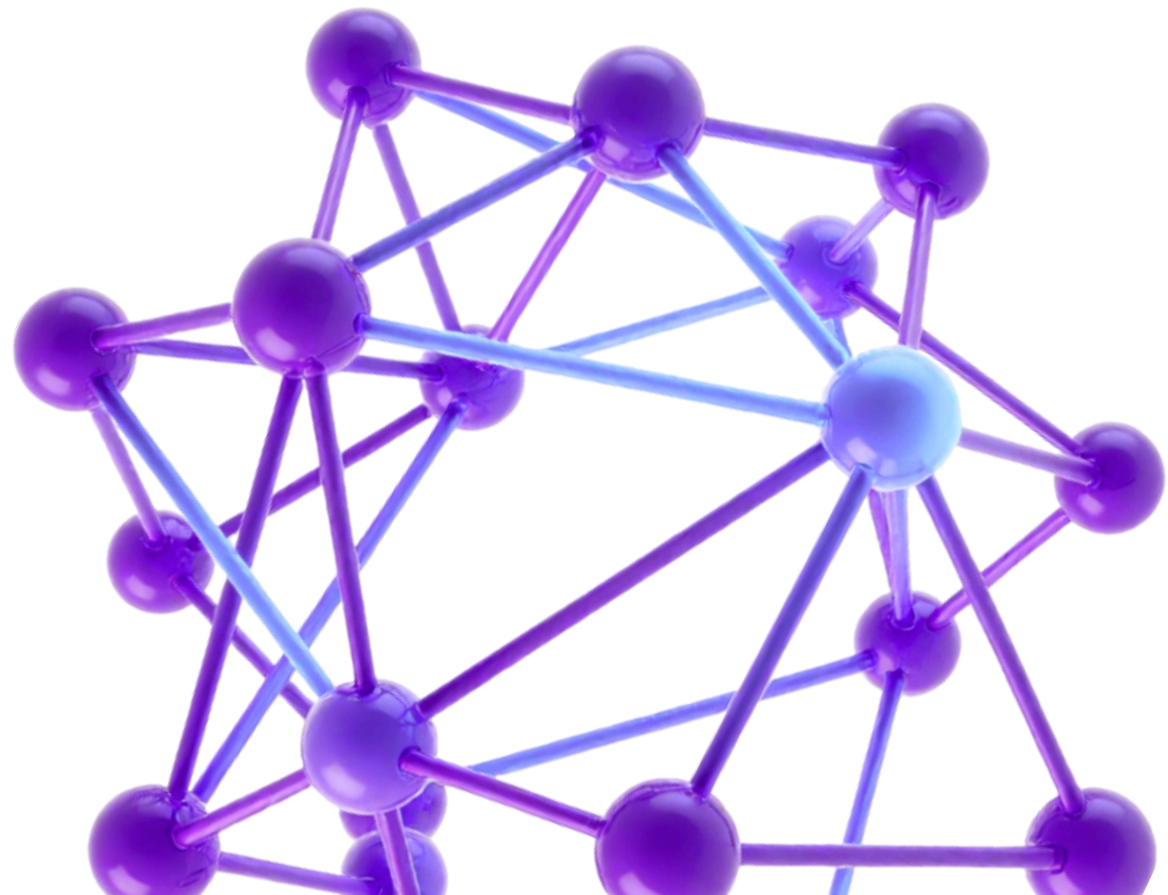


# LA TOKENIZZAZIONE

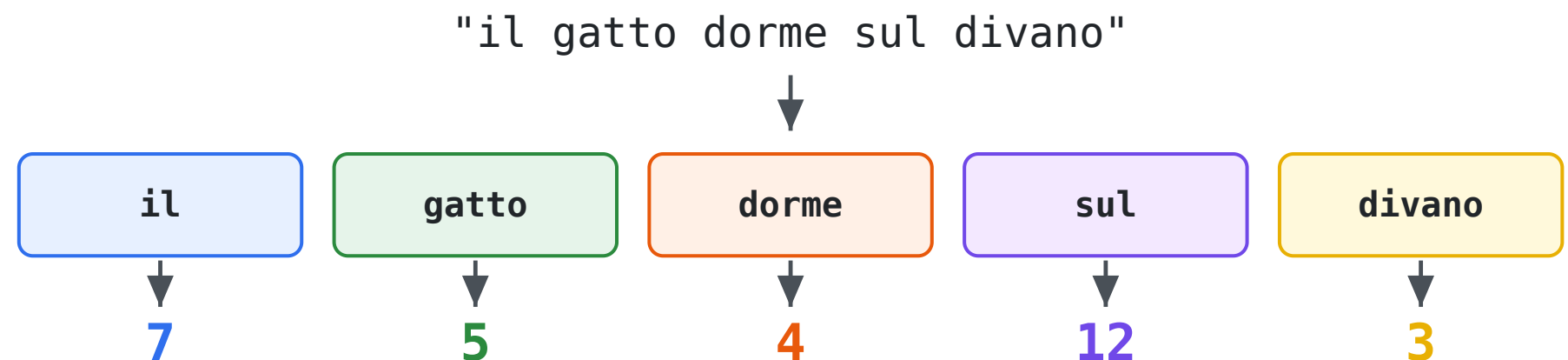
La tokenizzazione è il processo con cui il testo viene suddiviso in piccoli pezzi chiamati token, che il modello può elaborare.

Un token può essere una parola, una parte di parola, un carattere o un simbolo.  
Questa decisione viene presa da un algoritmo chiamato "tokenizer"

Ogni token viene poi convertito in un numero (ID), che è ciò che la rete neurale riceve effettivamente in input.



```
"il gatto dorme" → ["il", "gatto", "dorme"] → [7, 42, 128]
```



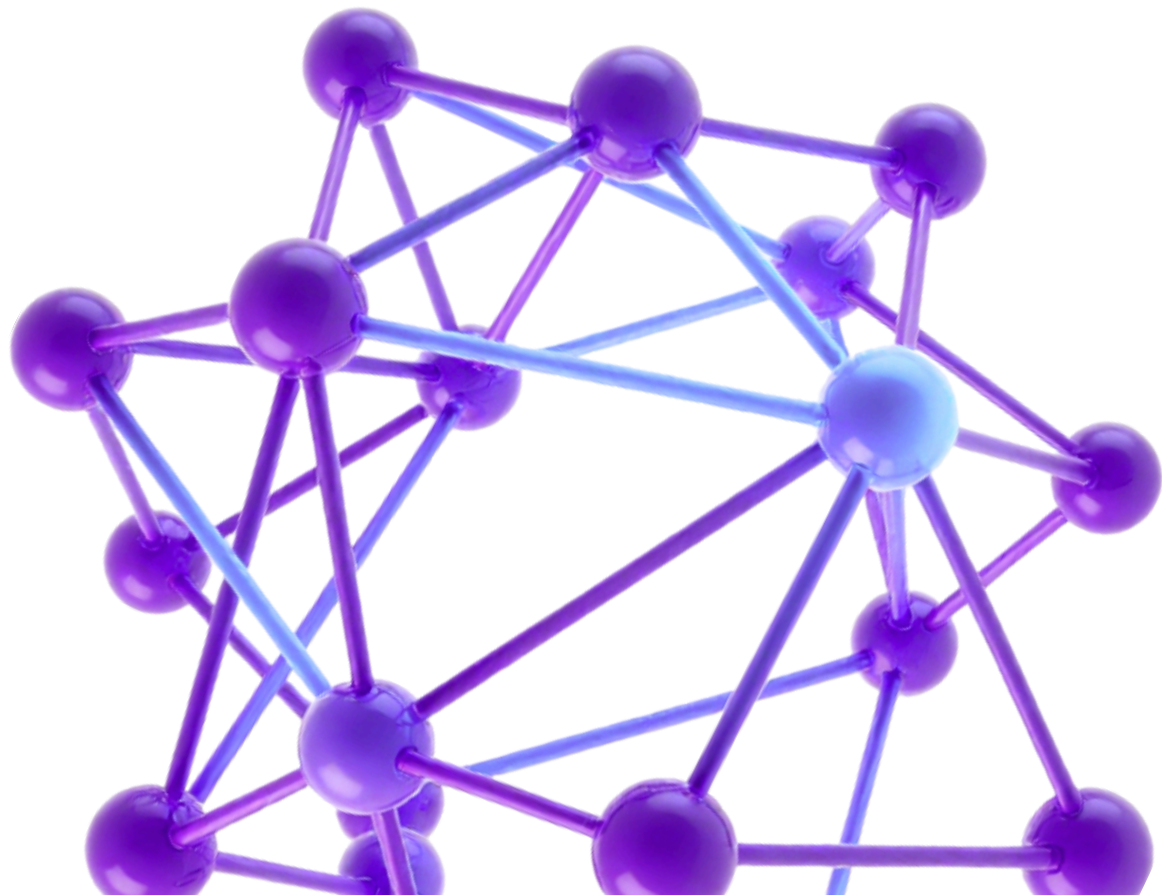


# IL TOKENIZER

Viene costruito e addestrato prima del training del modello

## COME VIENE COSTRUITO

- Prima del training del modello viene definito e costruito un tokenizer.
- A seconda dell'algoritmo utilizzato, il tokenizer individua un insieme di token e costruisce un vocabolario associando a ciascuno un ID.
- Durante l'utilizzo del modello, applica quelle regole per trasformare il testo in una sequenza di token ID.



```
testo = "il gatto dorme sul divano il cane dorme sul tappeto"

parole = testo.split()
vocabolario = sorted(set(parole)) # ['cane', 'divano', 'dorme', 'gatto', ...]
stoi = {p: i for i, p in enumerate(vocabolario)} # string -> int
itos = {i: p for p, i in stoi.items()} # int -> string

ids = [stoi[p] for p in parole]
print(vocabolario) # 8 parole distinte -> vocab_size = 8
print(ids) # [4, 3, 2, 6, 1, 4, 0, 2, 6, 7]
print("".join(itos[i] + " " for i in ids[:4])) # 'il gatto dorme sul '
```

# EMBEDDING

Un ID da solo non basta: rappresenta un token ma non mappa il significato della parola in modo gestibile dalla rete neurale.

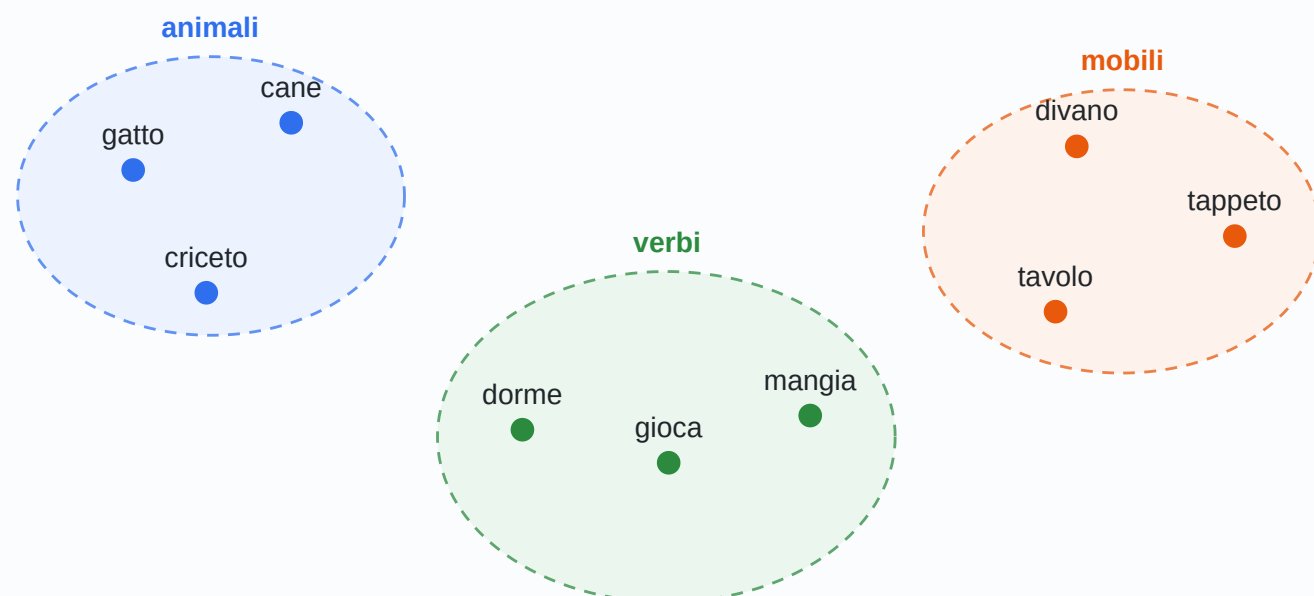
L'embedding trasforma ogni token ID in un vettore di numeri, cioè una rappresentazione matematica che il modello può elaborare.



```
"gatto" → 1847 → [0.21, -0.43, 0.87, ...]
```

## Lo spazio degli embedding (schema concettuale)

*parole usate in contesti simili finiscono vicine*



Durante il training, il modello impara i valori degli embedding.

Come conseguenza dell'addestramento, token che vengono utilizzati in contesti simili possono sviluppare rappresentazioni simili.

Lo spazio degli embedding può quindi catturare alcune regolarità statistiche presenti nei dati.



```
gatto → [ 0.21, -1.03, 0.44, ... ]  
cane → [ 0.19, -0.98, 0.51, ... ] ← simili!  
divano → [-1.44, 0.07, -0.62, ... ] ← lontano
```

# COME FA IL MODELLO A PREVEDERE IL PROSSIMO TOKEN?

Il modello è addestrato a produrre un punteggio per ciascun token del vocabolario, indicando quanto lo considera compatibile con il contesto.

Esempio:

```
input:  
"Il gatto dorme sul"
```

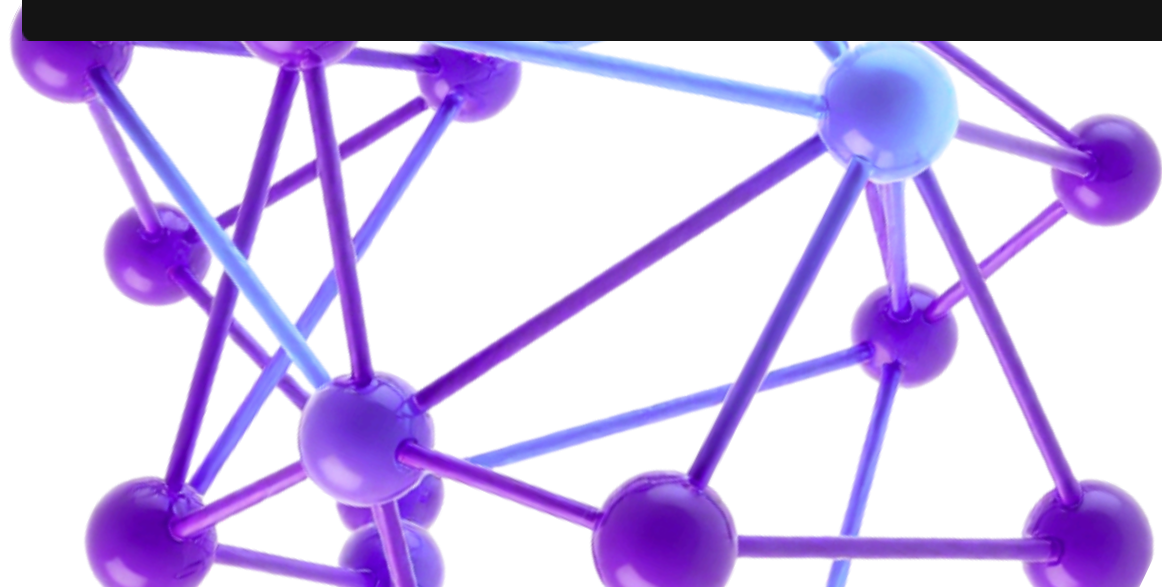
```
Il modello analizza il contesto e produce una probabilità per ogni possibile token:
```

```
# divano → 42%  
# letto → 28%  
# pavimento → 12%  
# tetto → 1%  
# ecc...
```

```
Il modello sceglie quindi un token secondo queste probabilità, ad esempio: "divano"
```

I punteggi vengono poi convertiti in probabilità tramite la funzione **softmax**. Il token successivo viene scelto o campionato da questa distribuzione.

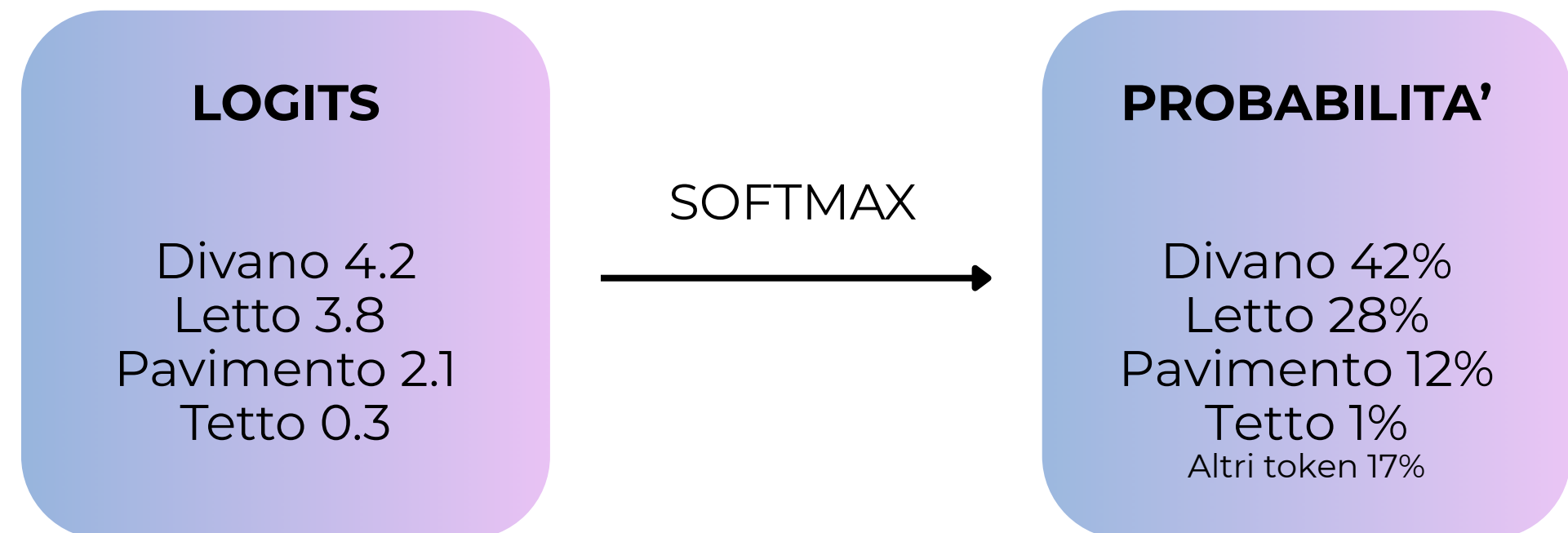
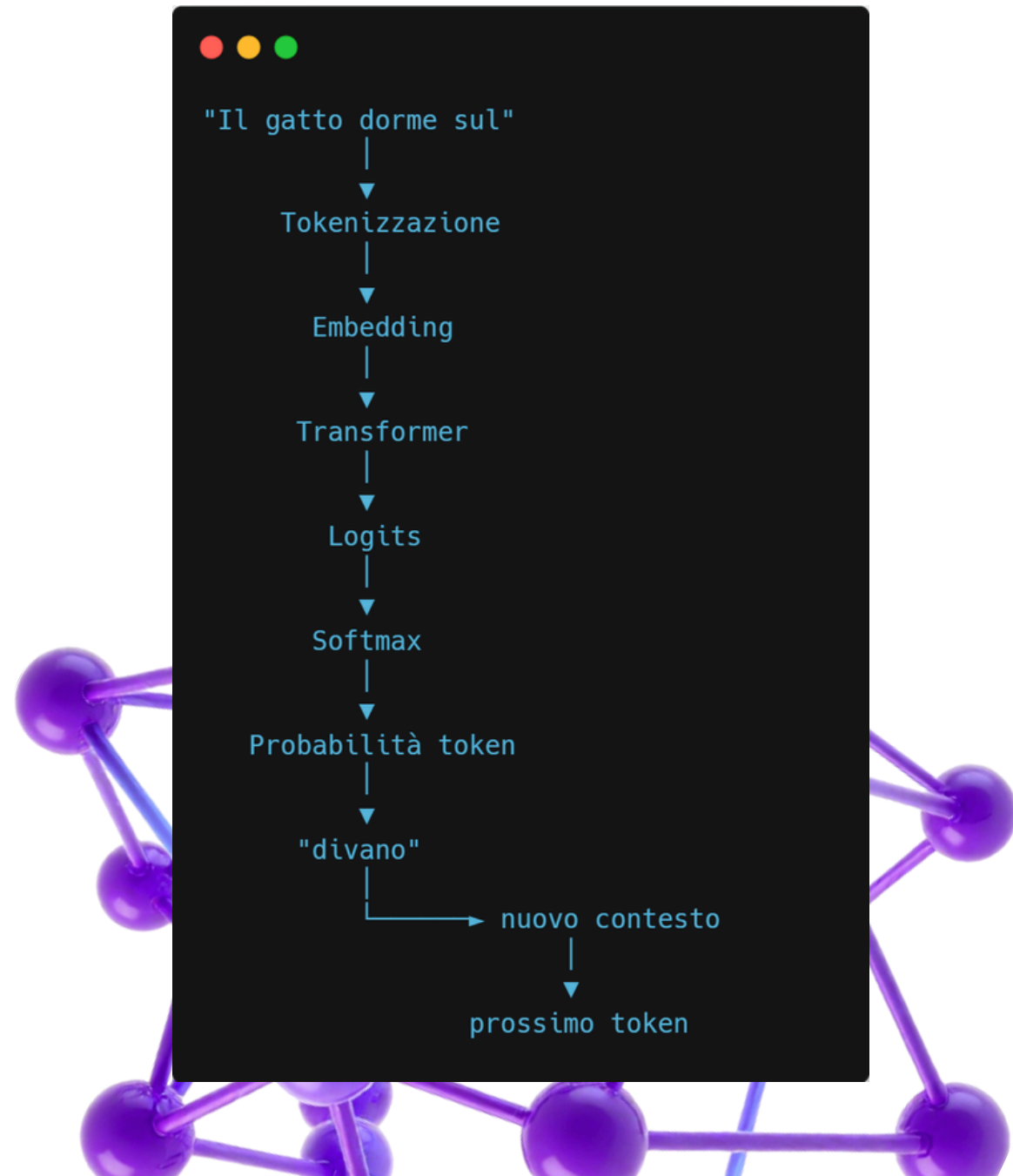
Il processo può quindi essere ripetuto utilizzando il nuovo token come parte del contesto.



# SOFTMAX

La softmax è una funzione che trasforma una serie di valori (logits) in una distribuzione di probabilità, facendo sì che tutti i valori siano tra 0 e 1 e che la loro somma sia 1.

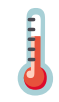
La softmax permette al modello di trasformare i suoi punteggi in probabilità per il prossimo token.





# TEMPERATURA

La temperatura modifica la distribuzione dei logits prima della softmax



BASSA

Con una temperatura bassa, la distribuzione tende a concentrarsi sui token con i punteggi più alti.



ALTA

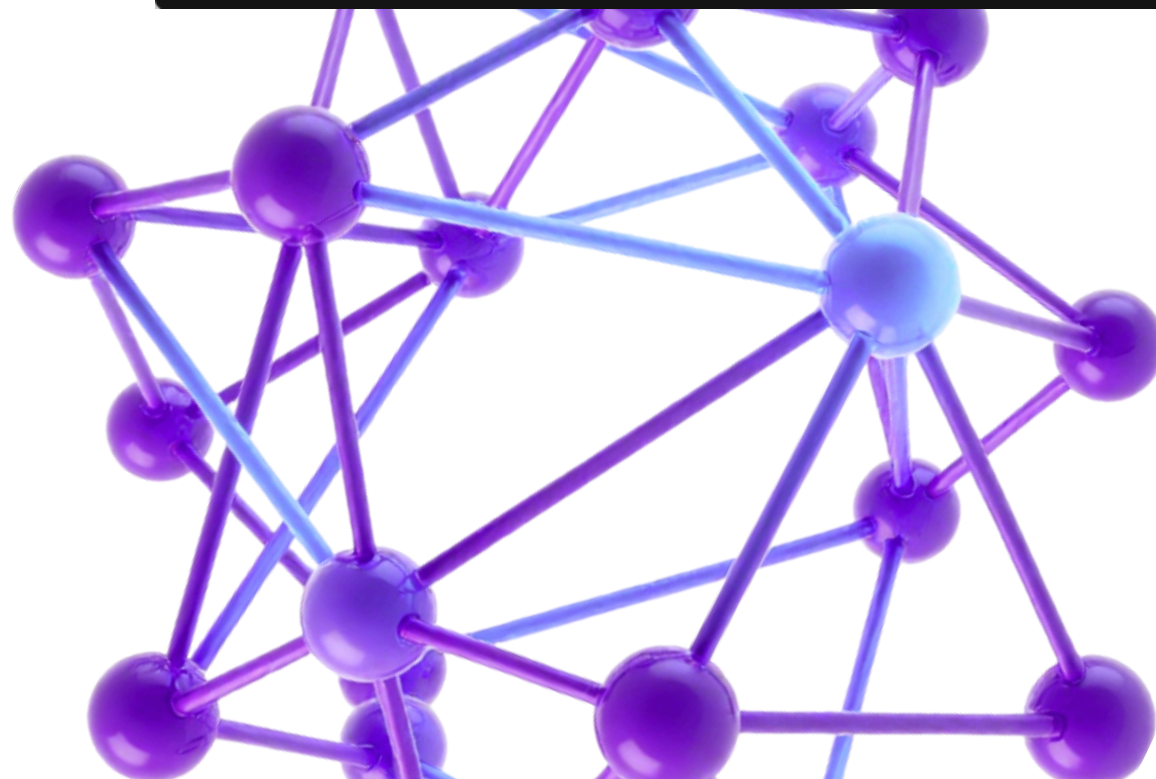
Con una temperatura alta, la distribuzione tende a diventare più uniforme e quindi aumenta la varietà dei possibili output.



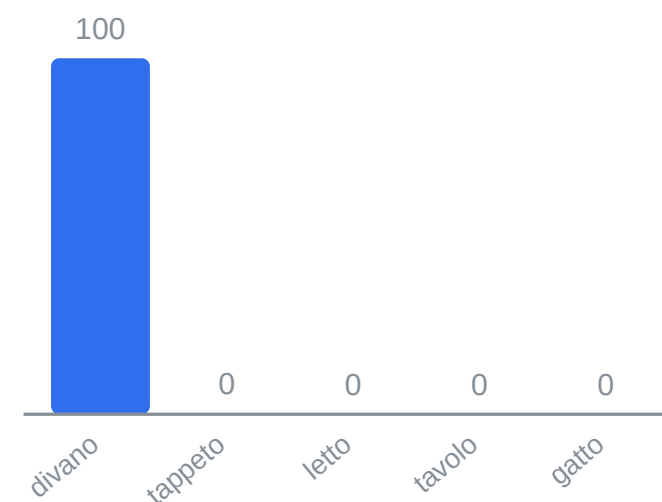
```
probs = softmax(logits / temperatura)

# T = (o tendente a) 0 : sceglie sempre la più probabile -> ripetitivo, robotico
# T = 1 : le probabilità così come sono
# T > 1 : appiattisce -> creativo, e poi delirante
```

Nb: la temperatura non cambia ciò che il modello ha imparato, cambia il modo in cui viene utilizzata la distribuzione prodotta dal modello.

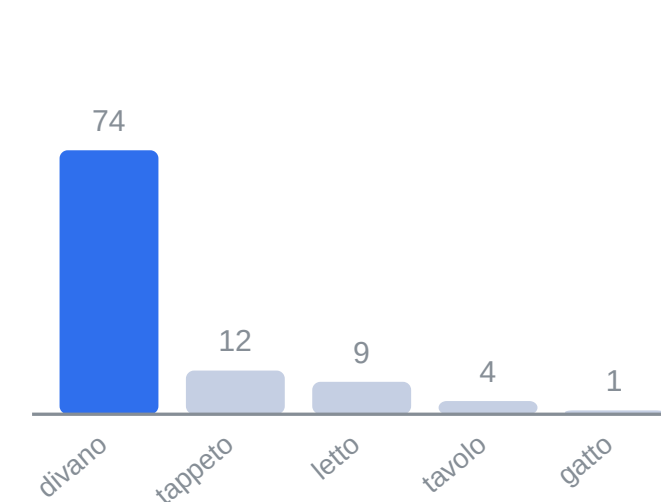


**T = 0.3**



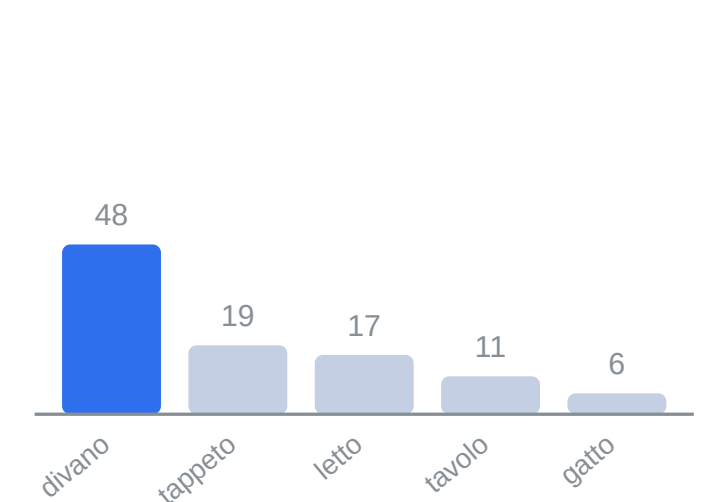
*quasi sempre la stessa parola*

**T = 1.0**



*le probabilità così come sono*

**T = 2.0**



*tutto diventa possibile*



Too hot to handle

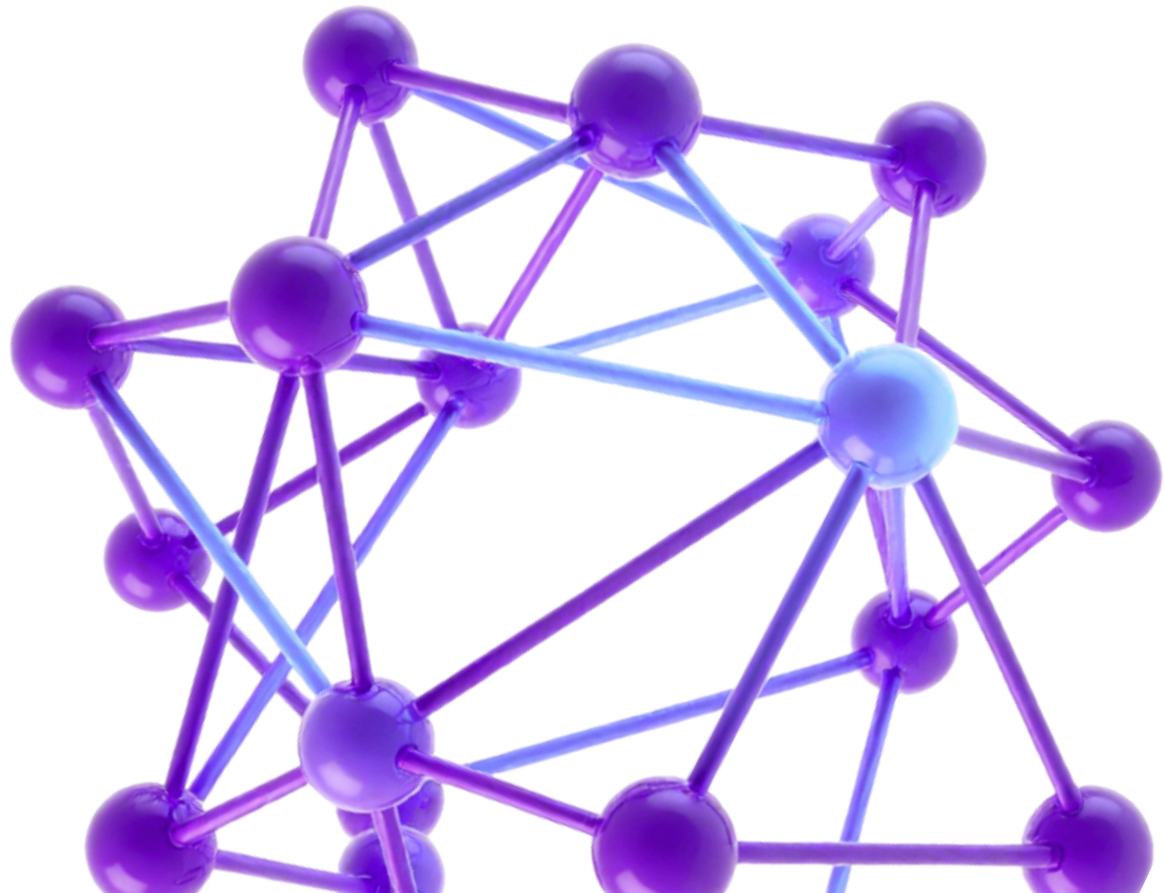
# E PER QUESTO, A VOLTE, ALLUCINA

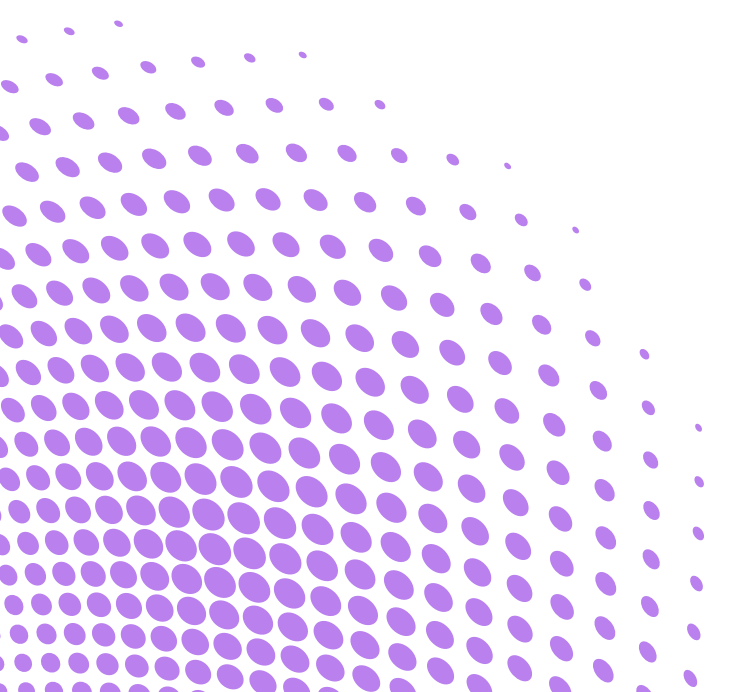
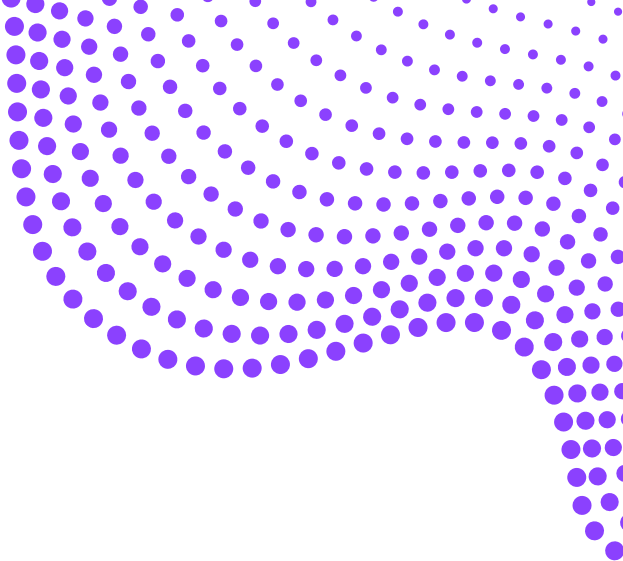
Il modello genera testo scegliendo token sulla base della distribuzione che produce.

Ma probabilità non significa verità: un modello può assegnare alta probabilità a una continuazione plausibile ma non corretta.

Per questo può generare informazioni inventate o non supportate dai dati disponibili.

Esistono tecniche per ridurre il problema, come verifica esterna, RAG e tool use.





**E O R A...**  
**PASSIAMO AL CODICE**

# IL PROGETTO

Un modello super semplice per l'autocompletamento delle frasi



## STRUTTURA DEL PROGETTO

(core del progetto)

Unica dipendenza: **torch**



```
src/  
├── data.py # corpus, vocabolario, dataset 78 righe  
├── model.py # l'architettura 45 righe  
├── train.py # il ciclo di training 117 righe  
├── generate.py # inferenza 140 righe  
└── demo.py # menu interattivo per il talk 98 righe
```



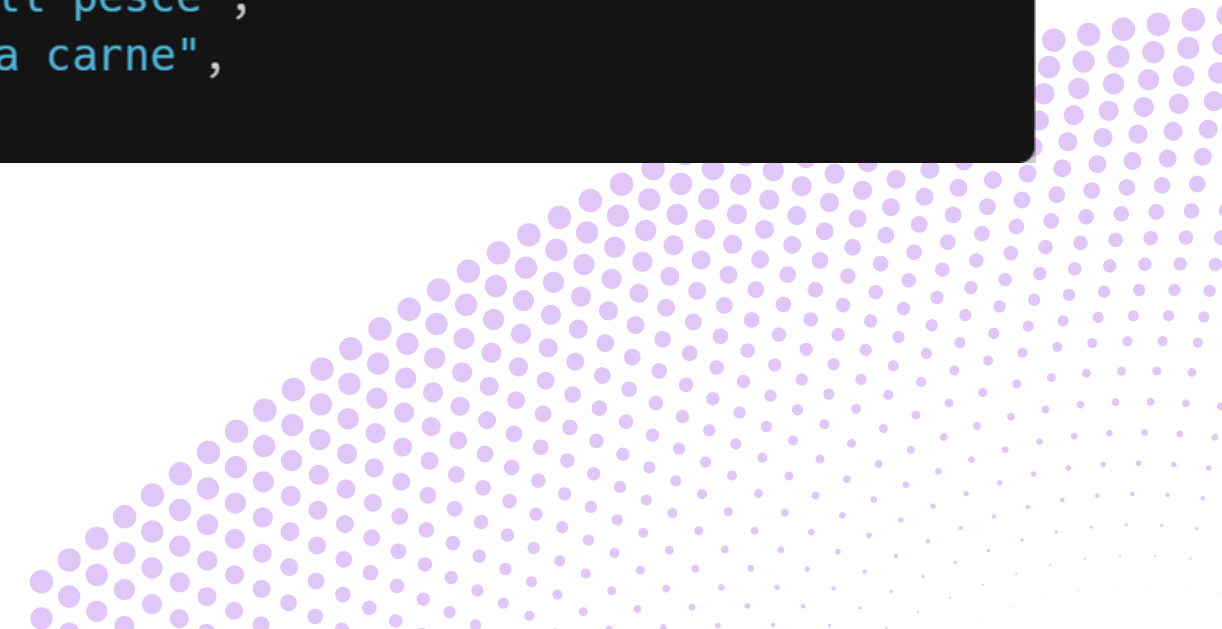
## IL DATASET

enorme (scherzo, sono letteralmente 6 frasi)

Per la precisione: 6 frasi, 32 parole di cui 14 diverse



```
CORPUS = [  
    "il gatto dorme sul divano",  
    "il cane dorme sul tappeto",  
    "il gatto gioca con la palla",  
    "il cane gioca con la palla",  
    "il gatto mangia il pesce",  
    "il cane mangia la carne",  
]
```



# IL PROGETTO

Un modello super semplice per l'autocompletamento delle frasi



## IL VOCABOLARIO

3 righe di codice per mappare le 14 parole



```
parole = sorted(set(" ".join(corpus).split()))  
stoi = {parola: i for i, parola in enumerate(parole)}  
itos = {i: parola for parola, i in stoi.items()}
```

stoi = string to integer; itos = integer to string



```
# 0 cane | 1 carne | 2 con | 3 divano | 4 dorme | 5 gatto | 6 gioca  
# 7 il | 8 la | 9 mangia | 10 palla | 11 pesce | 12 sul | 13 tappeto
```





# L'APPRENDIMENTO

Associare una parola alla parola successiva



## COPPIE SEQUENZIALI

Il dato etichettato che serve per allenare un modello linguistico è il testo stesso, letto con un offset di uno.  
Il modello si addestra sulle 32 parole formando 26 coppie di esempio di training  
(si perde l'ultima parola della frase dato che dopo di lei non c'è nulla)



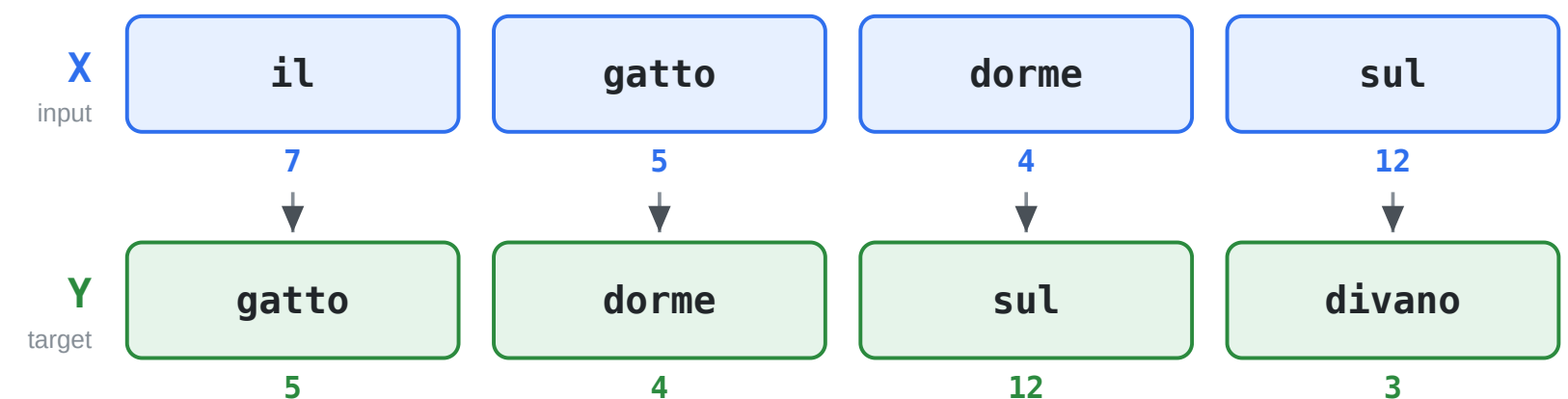
```
for i in range(len(ids) - 1):  
    coppie.append((ids[i], ids[i + 1]))
```

src/data.py , funzione build\_dataset()



```
X = [7, 5, 4, 12, 7, 0, 4, 12, ...] il gatto dorme sul il cane ...  
Y = [5, 4, 12, 3, 0, 4, 12, 13, ...] gatto dorme sul divano ...
```

Il target è il testo stesso, spostato di una posizione



# IL MODELLO

inizializzazione + 2 righe di forward



## COSA FA, IN BREVE

Cerca il vettore della parola in una tabella, e lo moltiplica per una matrice



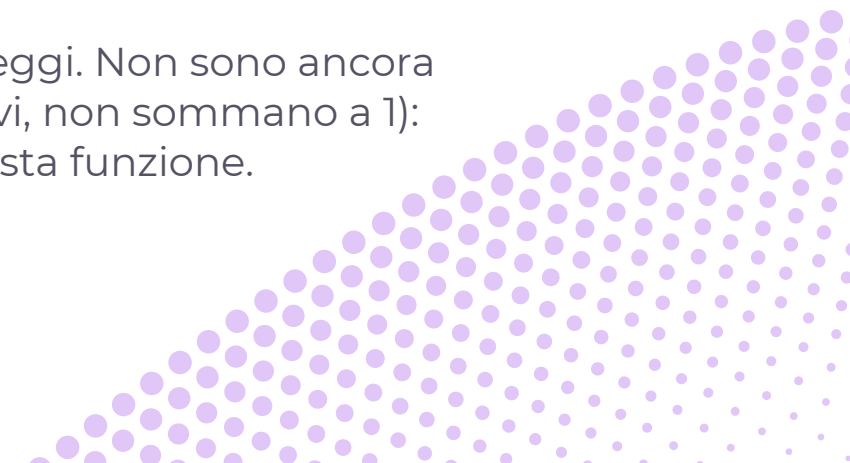
```
class TinyLM(nn.Module):
    def __init__(self, vocab_size, embedding_dim=8):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, vocab_size)
    def forward(self, x):
        e = self.embedding(x) # (batch, 8) significato
        logits = self.linear(e) # (batch, 14) punteggi
        return logits
```

src/model.py



## COSA FA, NEL DETTAGLIO

- **\_\_init\_\_** - cosa possiede il modello. Quando crei il modello, si costruisce due cose e le tiene in memoria:
  - **self.embedding**: una tabella con 14 righe (una per parola del vocabolario) e 8 colonne. Ogni riga sono 8 numeri che "rappresentano" quella parola.
  - **self.linear**: un gruppo di 14 neuroni, uno per ogni parola del vocabolario. Ogni neurone ha 8 pesi (uno per ciascuno degli 8 numeri in ingresso) più un bias.
- **forward** - cosa fa quando gli dai una parola. È la funzione che viene chiamata ogni volta che il modello deve fare una previsione. Riceve x, cioè l'id di una parola (per esempio 5 = "gatto"), e fa due passi:
  - **e = self.embedding(x)**: va nella tabella, prende la riga 5, ottiene 8 numeri. È una lettura, come `tabella[5]`.
  - **logits = self.linear(e)**: passa quegli 8 numeri ai 14 neuroni. Ognuno fa la sua somma pesata e produce un numero. Escono 14 numeri: un punteggio per ogni parola del vocabolario, che dice "quanto è probabile che sia lei la prossima".
- **return logits** - restituisce i 14 punteggi. Non sono ancora probabilità (possono essere negativi, non sommano a 1): quello lo fa la softmax, fuori da questa funzione.



# È UN MODELLO LINEARE

perché... keep it simple

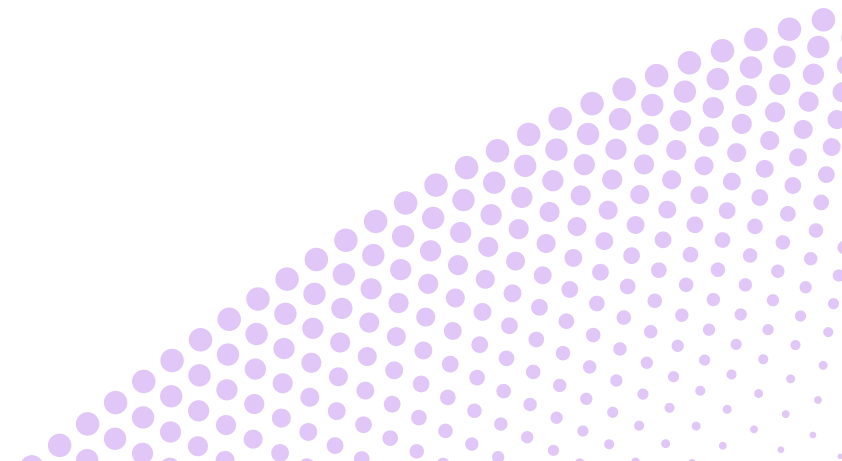


## MANCA L'ATTIVAZIONE

Ricordate il punto in cui spiegavo le **funzioni di attivazione**?

“Se manca l'attivazione, tutti i layer di neuroni collassano in uno solo”.

Qui abbiamo solo una matrice, ma data la semplicità del modello e lo scopo didattico ed esemplificativo, ci va bene così.



# IL LOOP DI TRAINING

```
for epoch in range(epochs):  
    optimizer.zero_grad() # 1. azzera i gradienti  
    logits = model(X) # 2. forward su tutto il dataset  
    loss = criterion(logits, Y) # 3. quanto abbiamo sbagliato  
    loss.backward() # 4. la correzione per ogni peso  
    optimizer.step() # 5. applica la correzione
```

src/train.py , funzione train\_model()



## CHI SCEGLIE LA QUANTITA' DI EPOCICHE?

è un iperparametro, per cui lo sceglie il dev.

nel codice di esempio è settato a 200.

in pratica, gira 200 volte e per ciascuna aggiusta i vari parametri di volta in volta.

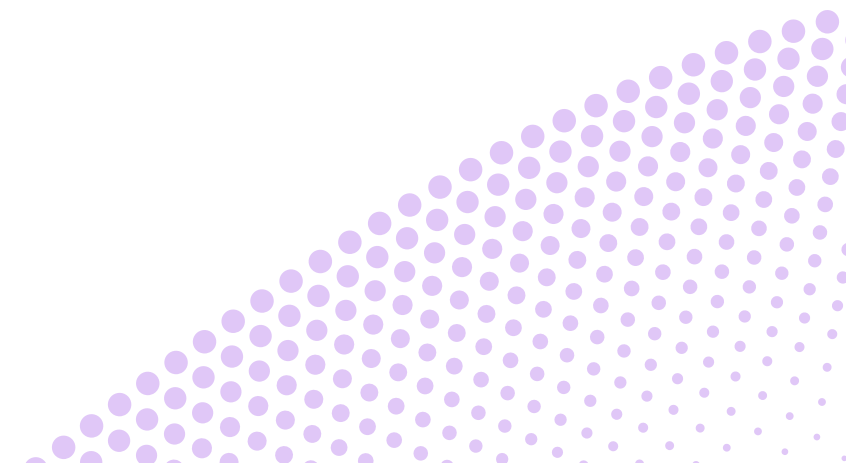




# **BENCHMARK**

del training

vediamo come ha performato



# FINO A DOVE E' SCESA LA LOSS

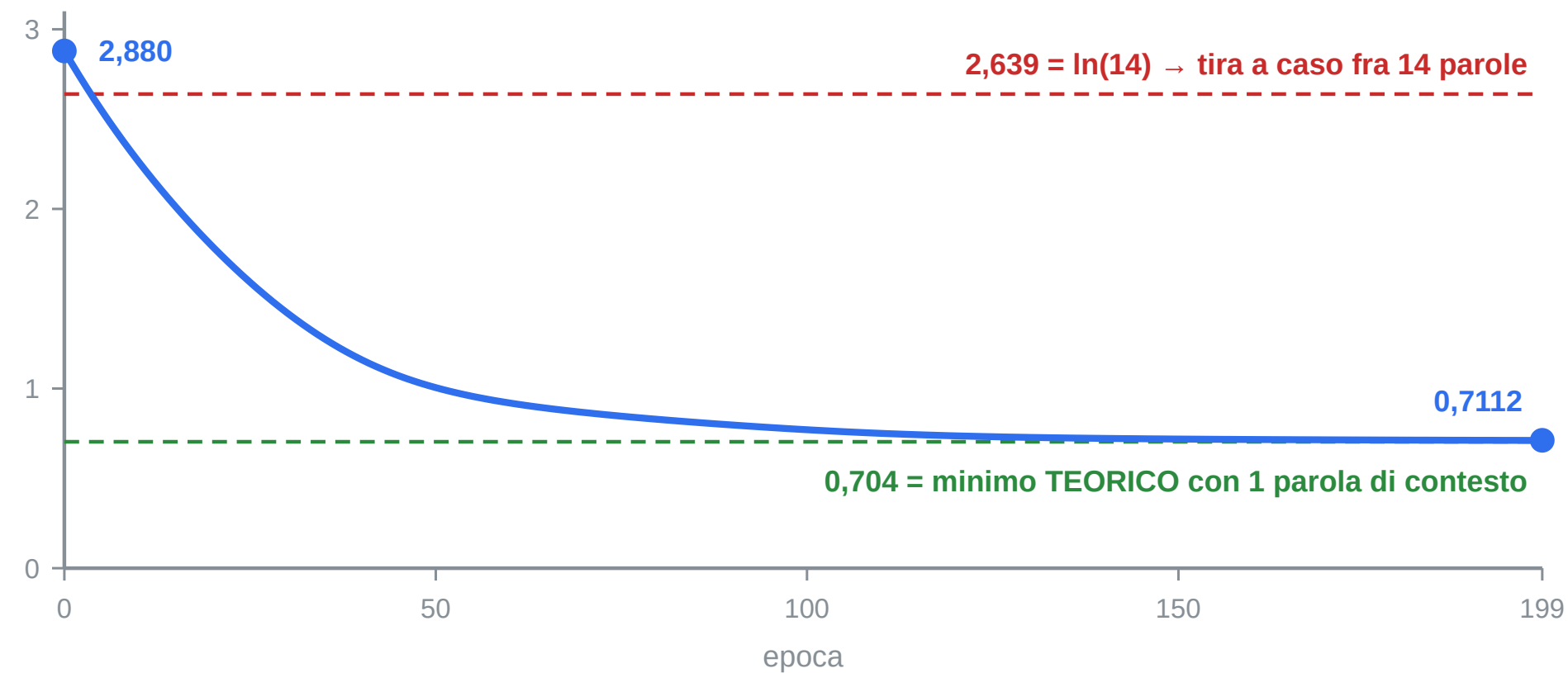
nel nostro progetto



## SNAPSHOT DEL TRAINING

- partendo da zero con il training, inizia a 2,64
- a fine training, è arrivato a 0,704

il training è terminato ad esaurimento epoche (300), in 3 secondi



# COSA HA IMPARATO

a fine training

## Il modello ha imparato le statistiche del corpus

a sinistra le probabilità prodotte dal modello allenato; a destra le frequenze contate a mano nelle sei frasi

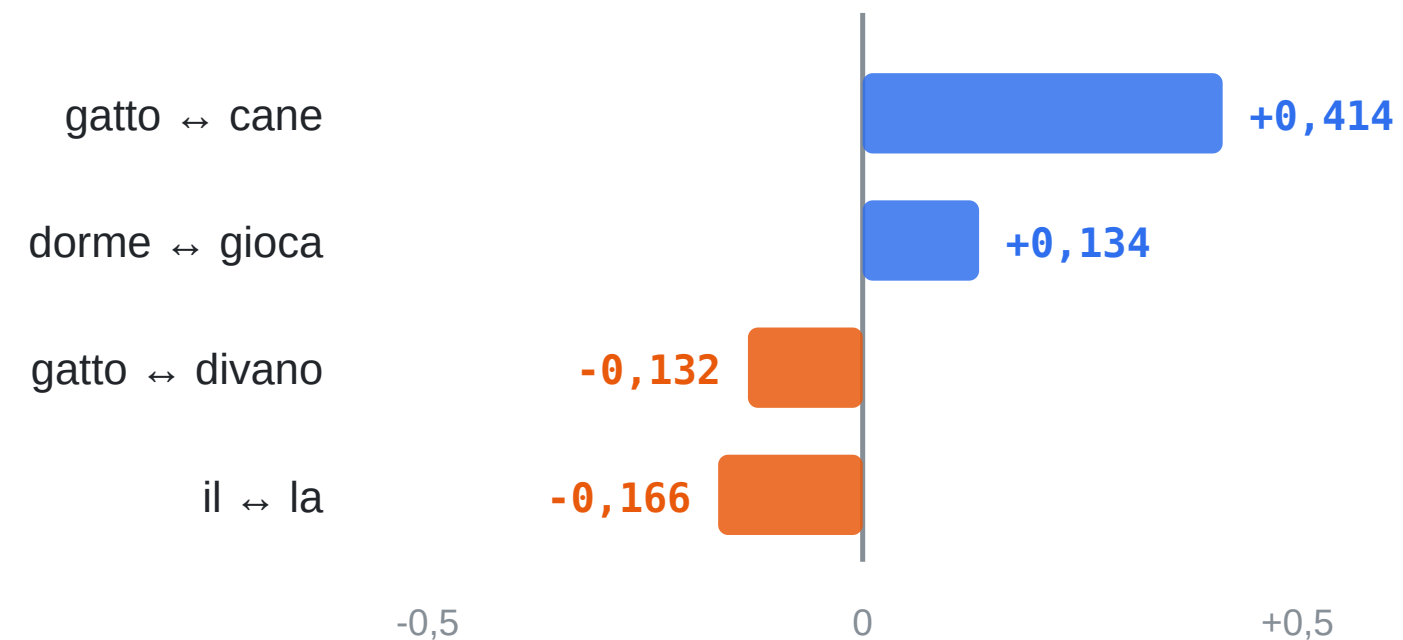
| dopo la parola... | il modello dice |                              | nel corpus era |                                   |
|-------------------|-----------------|------------------------------|----------------|-----------------------------------|
| dorme             | sul             | <div><div></div></div> 98,8% | sul            | <div><div></div></div> 2/2 = 100% |
| sul               | divano          | <div><div></div></div> 49,9% | divano         | <div><div></div></div> 1/2 = 50%  |
|                   | tappeto         | <div><div></div></div> 49,4% | tappeto        | <div><div></div></div> 1/2 = 50%  |
| il                | gatto           | <div><div></div></div> 42,6% | gatto          | <div><div></div></div> 3/7 = 43%  |
|                   | cane            | <div><div></div></div> 42,5% | cane           | <div><div></div></div> 3/7 = 43%  |
|                   | pesce           | <div><div></div></div> 14,2% | pesce          | <div><div></div></div> 1/7 = 14%  |
| gatto             | mangia          | <div><div></div></div> 33,2% | mangia         | <div><div></div></div> 1/3 = 33%  |
|                   | gioca           | <div><div></div></div> 33,2% | gioca          | <div><div></div></div> 1/3 = 33%  |
|                   | dorme           | <div><div></div></div> 33,1% | dorme          | <div><div></div></div> 1/3 = 33%  |

Nessuno ha scritto queste percentuali. Sono uscite dal loop di training, partendo da 238 numeri casuali.

# SIMILARITA' TRA PAROLE

come l'embedding ha associato le parole tra loro in base al pattern

Similarità tra embedding, misurata sul modello allenato



*Partivano tutti da numeri casuali. Nessuno ha detto al modello che gatto e cane sono animali.*

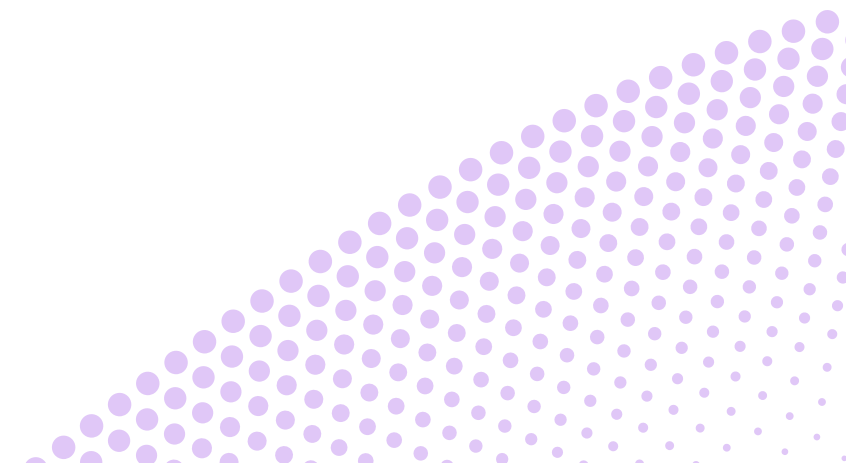
Non significa necessariamente che il modello abbia sviluppato una rappresentazione semantica di "animale": nel nostro dataset la similarità deriva dai contesti in cui le parole compaiono.



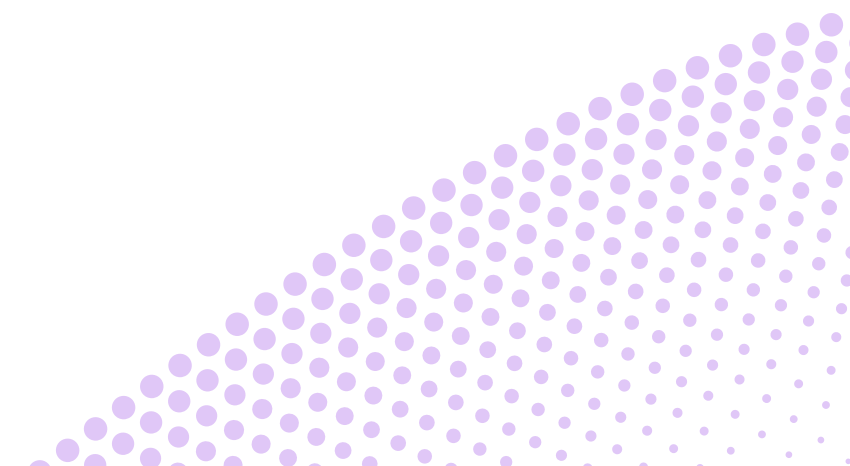
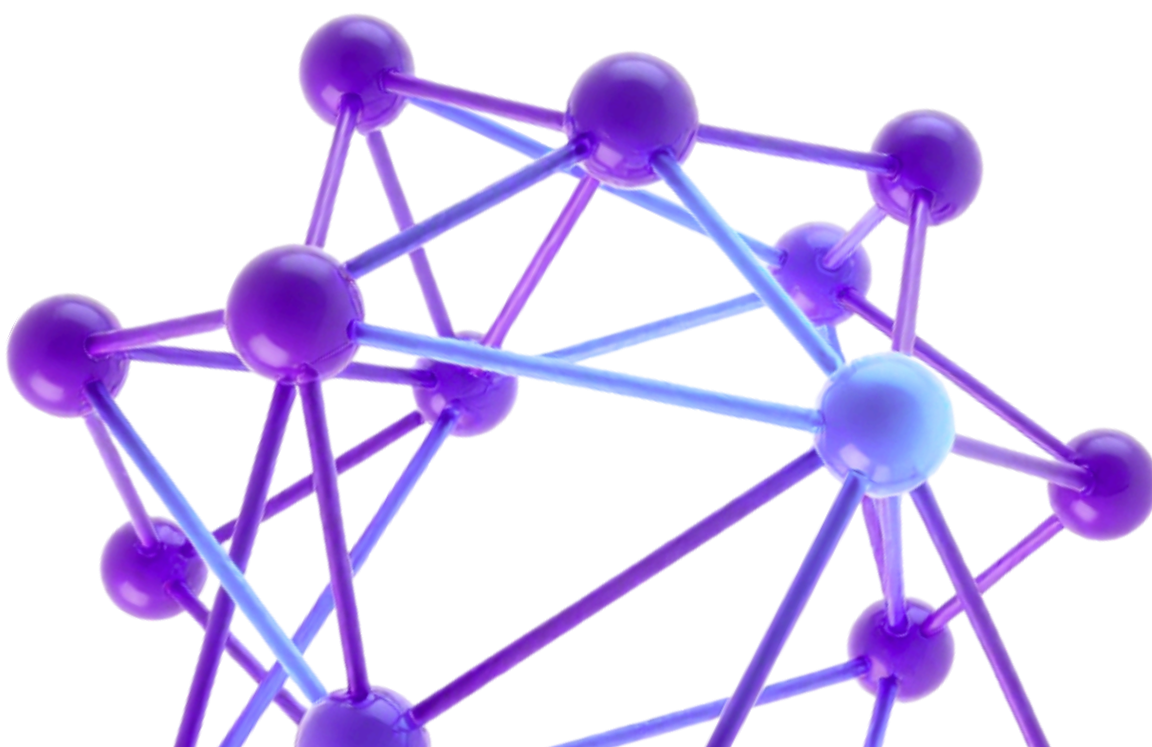
# PROVIAMOLO

lanciamo `python -m src.generate "il gatto dorme"`

e vediamo cosa risponde

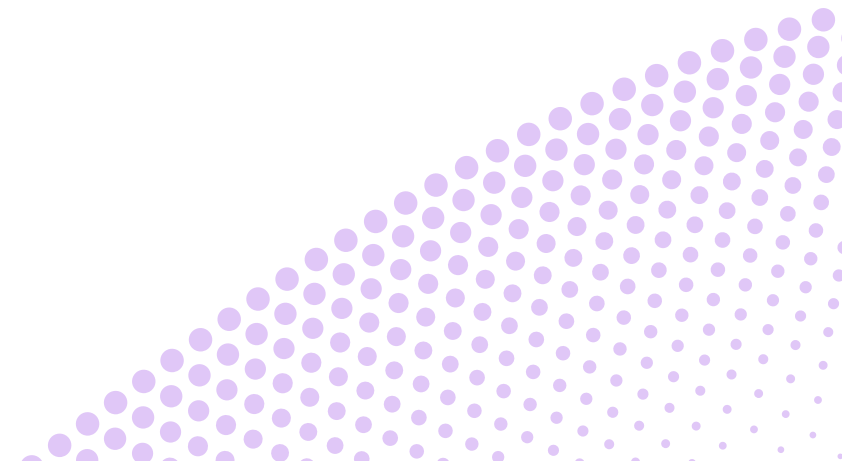
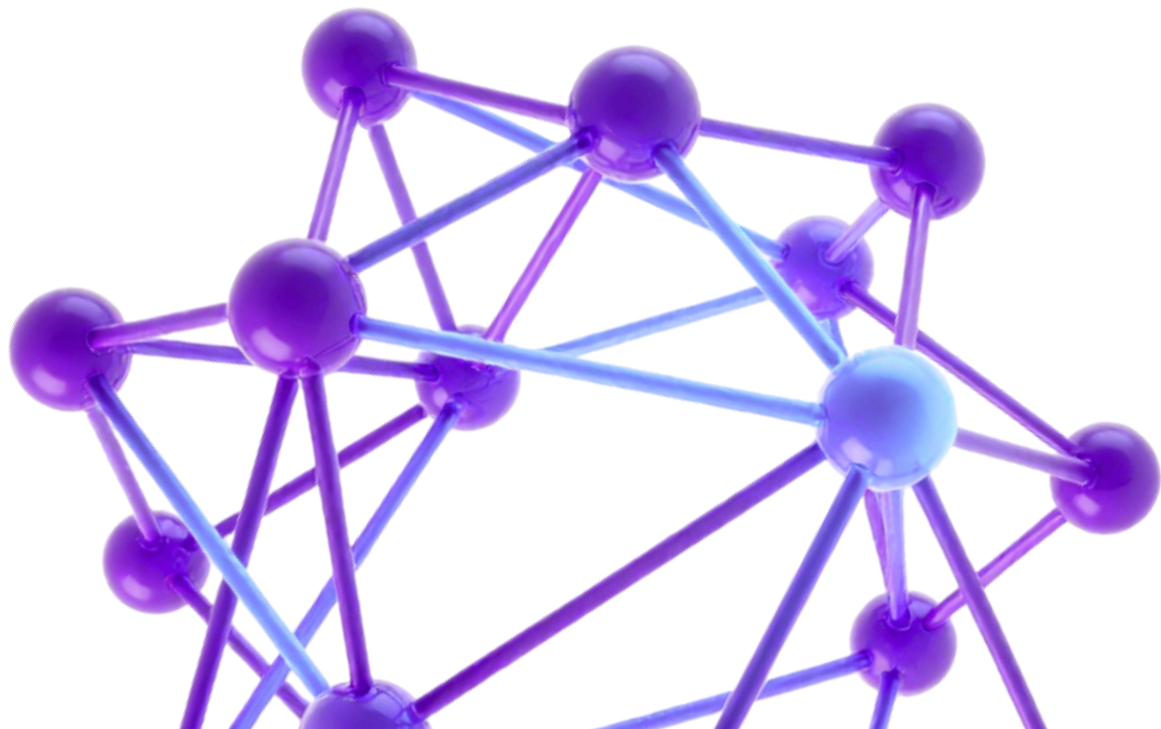


# **COSA SEPARA QUESTO MODELLO DA CHATGPT?**



# OVVIAMENTE TANTISSIMO

ma  
andiamo con ordine



# I TRANSFORMERS

no, non loro



© Hasbro

# I TRANSFORMERS

il motore per l'elaborazione e comprensione del contesto



## COS'E' UN TRANSFORMER

Il Transformer è un'architettura di rete neurale progettata per elaborare sequenze di token e modellare le relazioni tra loro attraverso meccanismi di attention.



## COSA CONTIENE

- **Self Attention**, ovvero mette in relazione tra loro i diversi token del contesto
- **Feed Forward Network** - processo per il quale elabora e trasforma le informazioni
- **Layer Multipli** che ripetono questi processi per costruire rappresentazioni sempre più complesse



# ATTENTION

per predire la prossima parola, guarda le precedenti e decidi quanto pesa ciascuna

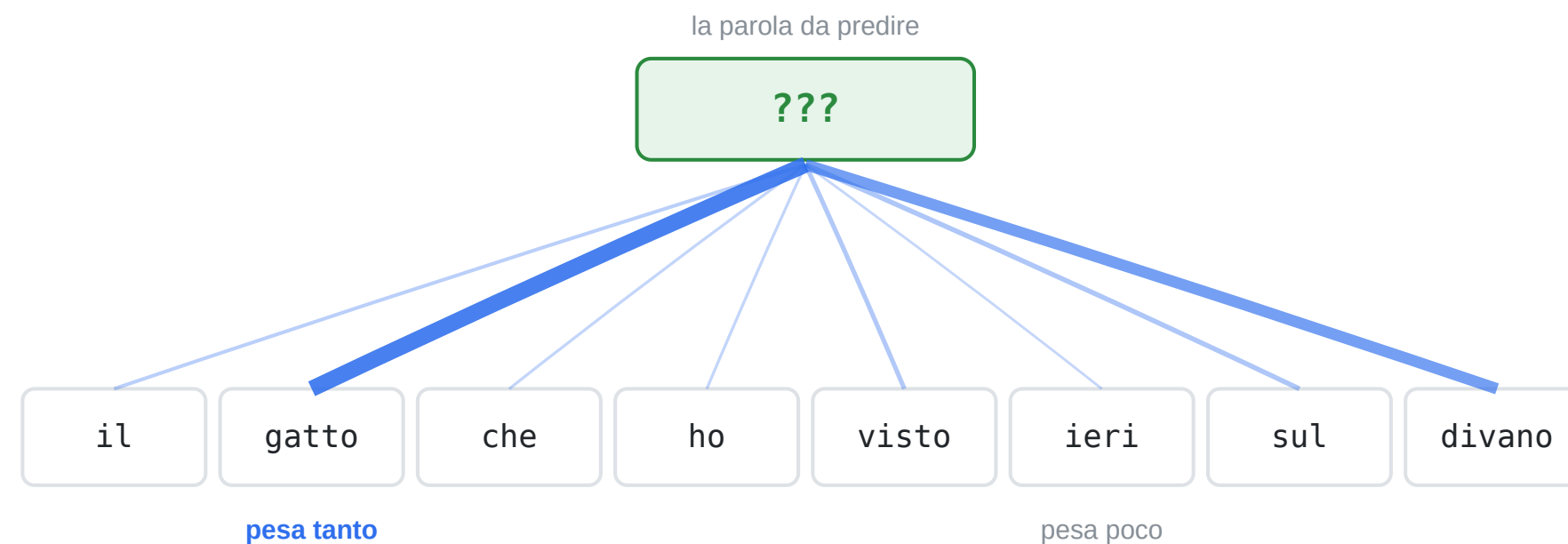


## COS'E' L'ATTENTION

Il meccanismo che permette al modello di determinare quali token del contesto sono più rilevanti per elaborare ciascun token.

Nel caso di un LLM autoregressivo, per predire il prossimo token, l'attention permette di combinare informazioni provenienti dai token precedenti, assegnando loro pesi diversi.

**Attention: pesa TUTTE le parole precedenti**



# CONTESTO

maggiore contesto = maggiori informazioni per il nostro modello (se sono rilevanti)



## COS'E' IL CONTESTO

Il contesto è l'insieme dei token disponibili al modello per comprendere ciò che sta elaborando e prevedere il token successivo.



## CONTEXT WINDOW

è la quantità massima di contesto che il modello può considerare



## COME VIENE GESTITO

Durante l'inferenza, il contesto viene rappresentato come una sequenza di token. Per evitare di ricalcolare continuamente le informazioni sui token già elaborati, il modello può conservarne le Key e Value nella "KV cache".

# E MOLTO, MOLTO ALTRO

più dati, più parametri, architettura migliore, training,...

|                                 | TinyLM              | GPT-3                  |
|---------------------------------|---------------------|------------------------|
| CAMBIA — di ordini di grandezza |                     |                        |
| parametri                       | 238                 | 175.000.000.000        |
| contesto                        | 1 parola            | migliaia di token      |
| dati di training                | 32 parole           | ~300 miliardi di token |
| architettura                    | embedding + 1 layer | 96 blocchi Transformer |
| training                        | 3 secondi, laptop   | mesi, migliaia di GPU  |



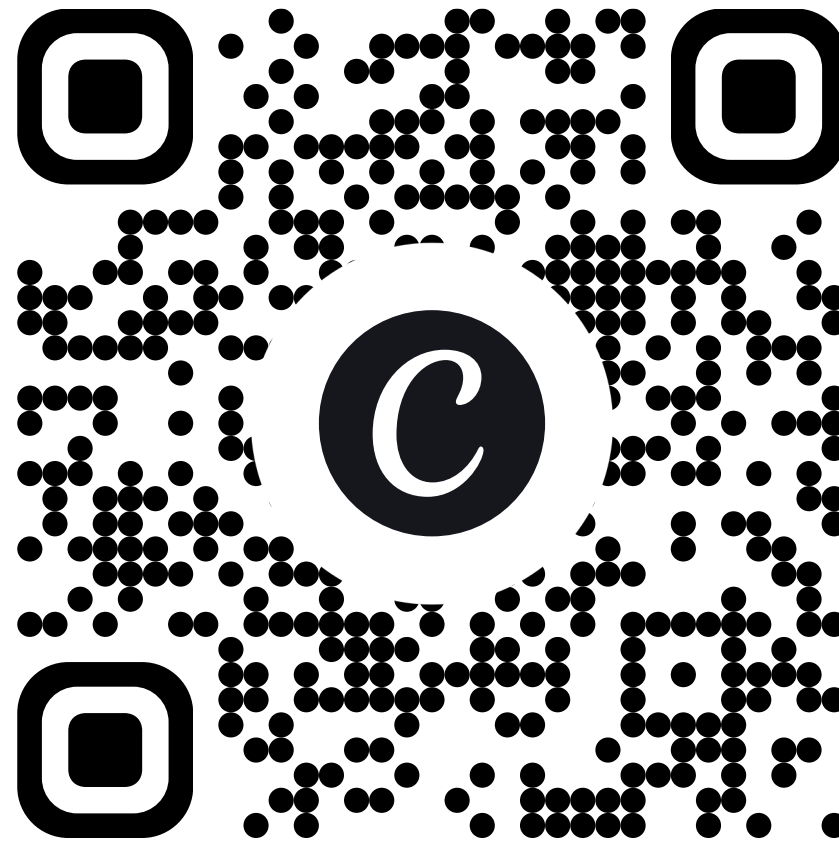
## MA FORSE, QUALCOSA IN COMUNE CE L'HANNO...

e forse, è quello che ci possiamo portare a casa con questo breve ma intenso tour

Il principio di training, la loss, la backpropagation, l'ottimizzazione e la predizione del prossimo token sono concetti che ritroviamo anche nei modelli molto più grandi.

# REPOS

verrà reso visibile assieme alla pubblicazione delle slides



<https://github.com/filippoZancanaro/tiny-llm-demo>

# SPECIAL THANKS



Guglielmo Bartelloni  
*Software Engineer*  
<https://bartelloni.it>

Per lo studio in pair



Elisa Reale  
*Head of AI*

Per la review alle slides





# GRAZIE PER L'ATTENZIONE



**Filippo Zancanaro**  
Software Engineer

[linkedin.com/in/filippoZancanaro](https://linkedin.com/in/filippoZancanaro)  
[zancanaro.filippo.97@gmail.com](mailto:zancanaro.filippo.97@gmail.com)  
[github.com/filippoZancanaro](https://github.com/filippoZancanaro)



Pizza Time!