Introduzione

giovedì 20 novembre 2025 17:29

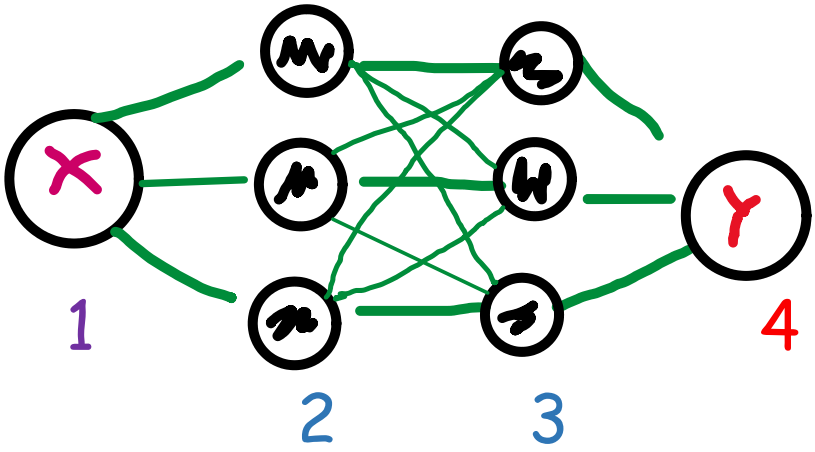
Questi appunti sono il diario di bordo del mio percorso nel Machine Learning. L'obiettivo non è ripetere formule o riscrivere manuali, ma costruire da zero modelli di intelligenza artificiale realmente funzionanti.
L'approccio è sempre lo stesso: partire dal problema, progettare l'architettura, addestrare il modello e verificarne il comportamento nel mondo reale.
La teoria è ridotta all'essenziale, solo ciò che serve per capire cosa sta succedendo sotto il cofano, ed è affiancata subito da codice PyTorch reale.
Questo non è un libro "chiuso". È una raccolta di appunti tecnici che evolve insieme alle competenze, ai progetti e agli errori. Ogni capitolo nasce da esperimenti concreti, non da esercizi astratti.

Capitolo 1:

Capitolo 2 Natural Language Processing:

Capitolo 1: Come è formata una rete neurale

Una rete neurale può essere vista come una struttura stratificata di neuroni artificiali che cooperano per trasformare un input grezzo in un output significativo. La loro organizzazione ricorda vagamente quella del cervello umano, ma il funzionamento è puramente matematico: ogni neurone esegue una piccola trasformazione e la rete intera nasce dalla combinazione di migliaia di queste trasformazioni elementari.



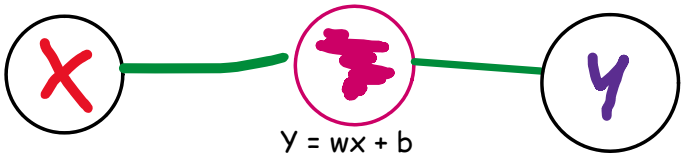
(1). Strati 2 e 3 sono degli strati **hidden** ovvero gli strati non visibili, sono strati dove l'informazione viene elaborata fino ad arrivare allo strato di output (y).

Questa struttura è formata da neuroni (strati 3 e 4), Ogni neurone calcola combinazioni lineari degli input tramite i pesi e ogni neurone può specializzarsi in pattern diversi, contribuendo alla generazione del dato in uscita.

Ogni neurone produce un output e l'output del primo neurone diventerà così l'input del neurone dello strato successivo. In questo tipo di layer ogni neurone è connesso a tutti quelli dello strato precedente (**Fully Connected Layer**). L'informazione quindi entra da (x) e viene passata a diversi neuroni nello strato hidden(1).

I neuroni per andare ad elaborare le informazioni usano dei **parametri**. Questi parametri vanno a decretare quanto e come un'informazione in ingresso è importante per la produzione dell'output e questi parametri vengono appresi durante l'**ADDESTRAMENTO**.

Esempio di rete neurale con uno strato di input (x), un neurone e uno strato di output (y).



Un neurone lineare può essere visto come la versione più semplice possibile di un

neurone artificiale. Riceve un valore in ingresso x, lo moltiplica per un peso w e aggiunge un termine chiamato bias $\mathbf{b}$. L'equazione che descrive questo comportamento è:
Il peso controlla quanto l'input influisce sull'output: valori più grandi amplificano l'effetto dell'ingresso, mentre valori piccoli lo riducono.
Il bias, invece, permette al neurone di spostare il risultato verso l'alto o verso il basso indipendentemente dal valore di x. Insieme determinano come il neurone "risponde" ai dati che riceve.
Nelle reti neurali reali, dopo la combinazione lineare viene quasi sempre applicata una funzione di attivazione. Questo passaggio introduce non linearità e permette alla rete di apprendere relazioni più complesse che una semplice trasformazione lineare non potrebbe rappresentare.

Funzioni di attivazione

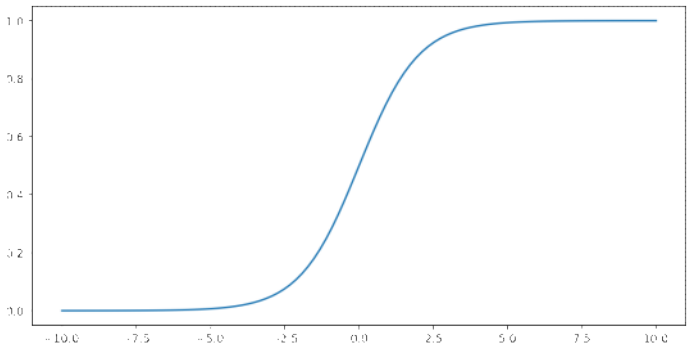
Una funzione di attivazione in una rete neurale è un'operazione matematica applicata all'output di un neurone. Determina se un neurone debba essere attivato o meno, introducendo non linearità nel modello, il che consente alla rete di apprendere schemi complessi. Senza queste funzioni, una rete neurale si comporterebbe come un modello di regressione lineare.
Le funzioni di attivazione hanno diversi scopi:

1. Permettono alle reti neurali di catturare relazioni non lineari dei dati, essenziali per risolvere compiti complessi
2. Limitano l'output dei neuroni a un intervallo specifico (dopo lo vedremo), prevenendo da valori estremi che possono ostacolare il processo di apprendimento
3. Durante la retropropagazione (backpropagation), le funzioni di attivazione aiutano nel calcolo dei gradienti.

Tre funzioni matematiche comunemente utilizzate come funzioni di attivazione sono sigmoide, tanh e ReLU.

La funzione sigmoidale (discussa sopra) esegue la seguente trasformazione sull'input , producendo un valore di output compreso tra 0 e 1:

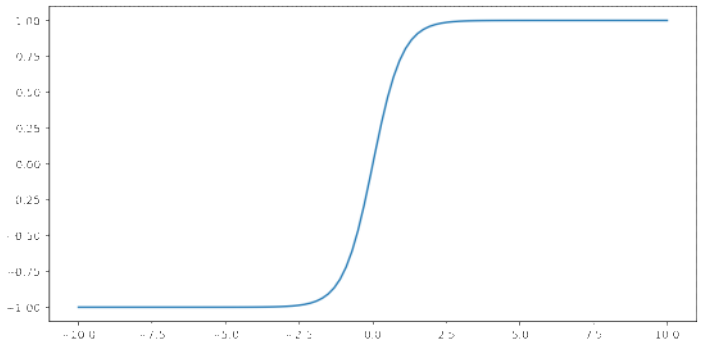
$$F(x) = \frac{1}{1 + e^{-x}}$$



La funzione tanh (abbreviazione di "tangente iperbolica") trasforma l'input per produrre un valore di output compreso tra -1 e 1:

$F(x) = \tanh(x)$

Ecco un grafico di questa funzione:

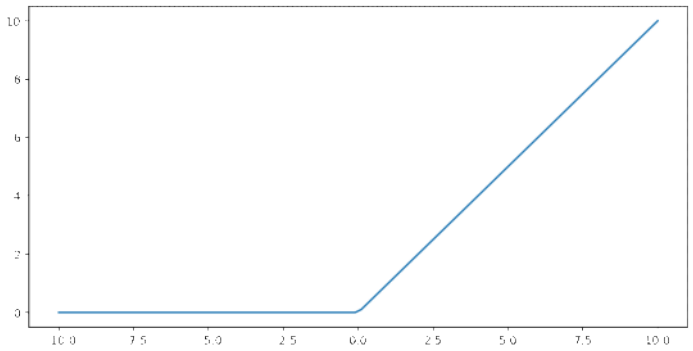


La funzione di attivazione **unità lineare rettificata** (o **ReLU**, per abbreviare) trasforma l'output utilizzando il seguente algoritmo:

- Se il valore di input è minore di 0, restituisce 0.
- Se il valore di input è maggiore o uguale a 0, restituisce il valore di input.

La funzione ReLU può essere rappresentata matematicamente mediante la funzione max():

$F(x) = \max(0, x)$



La funzione di attivazione Softmax trasforma un vettore di numeri in una distribuzione di probabilità, in cui ogni valore rappresenta la verosimiglianza di una particolare classe. È particolarmente importante per i problemi di classificazione multi-classe.

- Ogni valore di output è compreso tra 0 e 1.
- La somma di tutti i valori di output è uguale a 1.
  Questa proprietà rende Softmax ideale per scenari in cui ogni neurone di output rappresenta la probabilità di una classe distinta.

Funzione di attivazione Softmax
Il Problema
Immagina che la tua rete neurale stia classificando un'immagine e produca questi numeri:
Gatto: 4.2
Cane: 1.8
Uccello: -0.5
Domanda: Cosa significano questi numeri? Non sono probabilità!

La Soluzione: Softmax

La funzione Softmax trasforma numeri qualsiasi in una **distribuzione di probabilità**:

- Tutti i valori diventano positivi (tra 0 e 1)
- La somma totale è sempre 100% (= 1)

Come Funziona (Passo-Passo)

Input: Punteggi Grezzi

```
Gatto:   4.2
Cane:    1.8
Uccello: -0.5
```

Step 1: Esponenziazione

Eleviamo tutto a potenza di e (≈2.718) per rendere positivo:

python

```python
import math
gatto_exp = math.exp(4.2)   # = 66.686
cane_exp = math.exp(1.8)    # = 6.050
uccello_exp = math.exp(-0.5) # = 0.607
```

Perché l'esponenziale?

1. Rende tutti i numeri positivi
2. Amplifica le differenze (66 vs 6 è più evidente di 4.2 vs 1.8)

Step 2: Normalizzazione

Dividiamo ogni valore per la somma totale:

python

```python
totale = 66.686 + 6.050 + 0.607  # = 73.343
prob_gatto = 66.686 / 73.343    # = 0.909 → 90.9%
prob_cane = 6.050 / 73.343      # = 0.083 → 8.3%
prob_uccello = 0.607 / 73.343   # = 0.008 → 0.8%
```

Output: Probabilità

```
Gatto:   90.9% ✓
Cane:    8.3%
Uccello: 0.8%
```

Verifica: 90.9 + 8.3 + 0.8 = 100% ✓

Proprietà Chiave

1. Somma = 1 (100%)

```python
print(probabilita.sum().item())  # 1.0
```

Ogni valore è una "fetta della torta". Tutte le fette fanno la torta intera.

2. Valori tra 0 e 1

python

```python
print(probabilita.min().item())  # ≥ 0
print(probabilita.max().item())  # ≤ 1
```

Impossibile avere probabilità negative o superiori al 100%.

3. Amplifica Differenze

```python
# Input: differenza piccola
input1 = torch.tensor([2.0, 2.1])
print(F.softmax(input1, dim=0))
# tensor([0.4750, 0.5250]) ← Quasi 50/50
```

```
# Input: differenza grande
input2 = torch.tensor([2.0, 4.0])
print(F.softmax(input2, dim=0))
# tensor([0.1192, 0.8808])  ← Molto più netto!
```

Più la rete è "sicura" (punteggi distanti), più la probabilità si concentra su una classe.

Formula Matematica

Per un vettore **z** = [z_1, z_2, ..., z_n], la Softmax è definita come:

$$\sigma(z_i) = \frac{e^{(z_i)}}{\sum_{j=1 \text{ to } N} e^{(z_j)}}$$

Dove:

- **$e^{(z_i)}$**: Esponenziale del valore i-esimo
- **$\sum e^{(z_j)}$**: Somma di tutti gli esponenziali (per normalizzare)

In PyTorch:

```
# PyTorch fa tutto questo automaticamente:
probabilities = F.softmax(logits, dim=0)
```

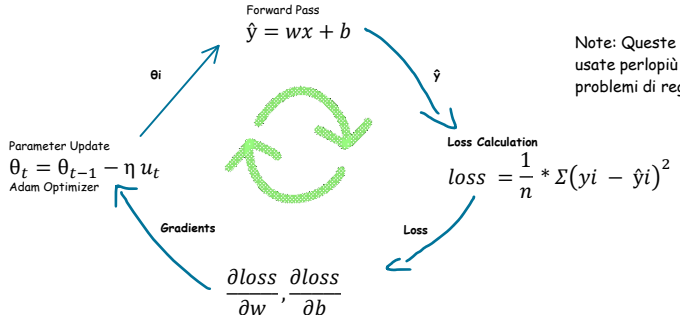
Quando Si Usa?

Classificazione multi-classe (es: riconoscere gatti, cani, uccelli)

- Output finale della rete
- Serve per interpretare le predizioni

Ciclo di training

Come detto nell'introduzione abbiamo dei parametri **weight** e **bias**.
Questi parametri devono essere appresi tramite il ciclo di addestramento.
Per effettuare questi cicli abbiamo bisogno di molti **dati di training.**
Il modello apprende tramite questo ciclo di apprendimento.



Il primo step (**Forward Pass**) e serve a calcolare l'output previsto (y).
Nella prima iterazione dell'addestramento il modello non è preciso e l'output sarà errato, perché il ciclo di addestramento è appena iniziato.

Il secondo step (**Loss Calculation**) va a prendere il nostro output previsto generato e lo confronta con quello reale (noi sappiamo quale dovrebbe essere l'output reale per ciascun input e questo tipo di allenamento si chiama allenamento supervisionato).
Nell'equazione stiamo andando ad effettuare una differenza tra il valore reale (yi) e il valore previsto dal modello (ŷ) e lo eleviamo al quadrato, operazione la facciamo per tutti gli output previsti, confrontandoli con il valore reale e facendo poi una media matematica, in questo modo abbiamo calcolato **l'errore.**
L'errore rappresenta quanto è stato errato il nostro modello nel calcolo dei nostri valori, lo scopo del nostro ciclo è **ridurre l'errore,** ridurre il più possibile la **differenza** tra gli output **reali** e quelli **previsti** dal **modello.**

Nel terzo step (**backpropagation**) andiamo a calcolare qual è l'influenza di ciascun parametro nella produzione dell'errore (loss), andiamo a prendere i vari parametri e otteniamo dei gradienti.

Nel quarto step (**Parameter Update**) mettiamo in atto l'**Adam Optimizer**, un **algoritmo** che va a prendere i parametri definiti generalmente con theta (θ) che possono star a significare (w) o (b) o qualsiasi altro parametro.
Per ottenere il parametro aggiornato va a prendere quello precedente e sottrae questo termine (η), il termine (u) va a tenere conto dei gradienti, quindi più un parametro avrà inciso sull'errore più dovrà essere modificato e di questo ne tengono traccia i gradienti (u).
Il termine Eta (η) tiene conto del **learning rate**, la dimensione del passo con cui vengono aggiornati i parametri (step), più è alto e più drasticamente verranno modificati i parametri per ottenere un miglioramento e sarà veloce l'apprendimento.
Se abbiamo un learning rate troppo alto potremmo incorrere in delle instabilità.

A questo punto abbiamo i parametri aggiornati e il ciclo ricomincia, facciamo una nuova formula di Forward Pass con i parametri nuovi (θ_i), otteniamo un nuovo output previsto, lo confrontiamo con quello reale e l'errore si presume sia leggermente minore al secondo passaggio. Otteniamo i gradienti, aggiorniamo i parametri e otteniamo i nuovi. Questo ciclo verrà ripetuto migliaia di volte anche mostrando più volte gli stessi dati di training finché il modello non apprende a produrre risultati più soddisfacenti.

Immagina di essere su una montagna al buio.
Il gradiente è la direzione della massima pendenza.
Nel nostro caso:
- Montagna = funzione di errore
- Obiettivo = scendere a valle (errore minimo)
- Gradiente = direzione in cui scendere
Matematicamente: è la derivata che indica "quanto cambia l'errore se modifico questo parametro".

Progetto 1: Regressione Lineare per la Temperatura

Primo Progetto: Regressione Lineare per la Temperatura

Prima di Iniziare: Cosa Sono i Tensori?

Nel codice che segue vedrai torch.tensor(). Ma cos'è esattamente un tensore?

Tensore vs Lista Python

```
# Lista Python normale
temperature = [55, 60, 65, 70, 75]
print(temperature[0])  # 55
```

Una lista Python può contenere qualsiasi cosa (numeri, stringhe, oggetti), ma è **lenta** per operazioni matematiche.

python

```
# Tensore PyTorch
temperature_tensor = torch.tensor([55, 60, 65, 70, 75])
print(temperature_tensor[0])  # tensor(55)
```

Un tensore è una struttura dati ottimizzata per:

- **Calcoli matematici veloci** (su GPU)
- **Calcolo automatico dei gradienti** (per l'addestramento)
- **Operazioni parallele** (su migliaia di numeri insieme)

Confronto Pratico

```
import torch
import time
# Liste Python
lista_a = list(range(1000000))
lista_b = list(range(1000000))
start = time.time()
lista_c = [a + b for a, b in zip(lista_a, lista_b)]
print(f"Liste Python: {time.time() - start:.4f} secondi")
# Tensori PyTorch
tensor_a = torch.arange(1000000)
tensor_b = torch.arange(1000000)
start = time.time()
tensor_c = tensor_a + tensor_b
print(f"Tensori PyTorch: {time.time() - start:.4f} secondi")
```

Output tipico:

```
Liste Python: 0.1234 secondi
Tensori PyTorch: 0.0012 secondi  ← 100x più veloce!
```

Dimensioni dei Tensori

I tensori possono avere più dimensioni:

```
# 0D: Scalare (numero singolo)
scalare = torch.tensor(42)
print(scalare.shape)  # torch.Size([])
# 1D: Vettore (lista di numeri)
vettore = torch.tensor([1, 2, 3, 4])
print(vettore.shape)  # torch.Size([4])
# 2D: Matrice (tabella)
matrice = torch.tensor([[1, 2, 3],
                        [4, 5, 6]])
print(matrice.shape)  # torch.Size([2, 3])  ← 2 righe, 3 colonne
# 3D: "Cubo" (es: immagine RGB)
cubo = torch.randn(3, 224, 224)  # 3 canali, 224×224 pixel
print(cubo.shape)  # torch.Size([3, 224, 224])
```

Perché unsqueeze(1)?

Nel codice vedrai:

```
x = x.unsqueeze(1)
```

Questo aggiunge una dimensione. Perché serve?

```
# Prima: vettore 1D
x = torch.tensor([55, 60, 65, 70, 75])
print(x.shape)  # torch.Size([5])
```

```
# Dopo unsqueeze: matrice 2D con 1 colonna
x = x.unsqueeze(1)
print(x.shape)  # torch.Size([5, 1])
print(x)
# tensor([[55],
#         [60],
#         [65],
#         [70],
#         [75]])
```

I modelli PyTorch si aspettano input 2D: [numero_esempi, features]

- Nel nostro caso: 5 esempi, 1 feature per esempio

dtype: Tipo di Dati

```
torch.set_default_dtype(torch.float32)
```

Questo dice a PyTorch di usare numeri decimali a 32 bit (compromesso tra precisione e velocità).

Tipi comuni:

- torch.float32 (default): Numeri decimali normali
- torch.float64: Più preciso ma più lento
- torch.int64: Numeri interi
- torch.bool: True/False

Cosa Significa "Plottare"?

```
plt.scatter(x, y)
```

"Plottare" = Disegnare un grafico per visualizzare i dati

È come trasformare una tabella di numeri in un'immagine:

```
# Dati: due colonne
x = [55, 60, 65, 70, 75]
y = [13, 16, 19, 22, 25]
# Grafico: punti su piano cartesiano
plt.scatter(x, y)
plt.xlabel("X (Temperature Inventate)")
plt.ylabel("Y (Temperature Reali)")
plt.show()
```

Perché è utile?

- Vedi immediatamente se c'è una relazione tra X e Y
- Scopri valori anomali (punti strani)
- Capisci se serve un modello lineare o complesso

Ora procediamo con il codice, andremo a creare una **rete neurale** molto semplice. Lo creeremo tramite un **Jupyter Notebook** in **VSCode** usando **pytorch**.
Immaginiamo di avere questo dispositivo che misura delle temperature inventate in Fahrenheit (lista x), ad ogni temperatura inventata il nostro modello dovrà dirci qual è la temperatura corrispondente, perciò alleniamo il modello a ricevere in input una temperatura x e dare la temperatura giusta y.

```
import torch
```

```
x = torch.tensor([55, 65, 75, 54, 67, 78, 53, 65, 76, 60, 70, 81, 51, 68],
dtype=torch.float32).unsqueeze(1) # Temperature inventate date da un dispositivo
y = torch.tensor([13, 18, 24, 12, 20, 26, 12, 18, 24, 14, 21, 28, 12, 20],
dtype=torch.float32).unsqueeze(1) # Temperature reali misurate
```

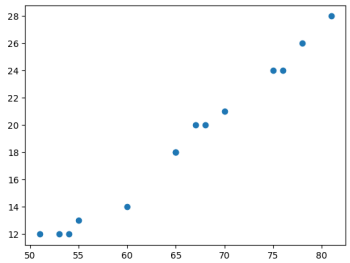
Come vedete abbiamo utilizzato torch per creare dei **tensori**, questi sono simili alle classiche liste di Python ma con funzionalità ben specifiche.
Nel tensore x abbiamo le temperature inventate (input), mentre nel tensore y troviamo le temperature effettive (output).
Definiamo il datatype standard per i tensori per ottenere massima compatibilità e performance.
La funzione unsqueeze la usiamo per dare un'altra dimensione al nostro tensore, perché il nostro modello che creeremo successivamente non vuole tensori **unidimensionali**, con questa funzione andiamo a rendere il tensore bidimensionale con 1 colonna.

Ora nella successiva istruzione "plottiamo" i dati di training usando la libreria **matplotlib**.

```
from matplotlib import pyplot
pyplot.scatter(x, y)
```

"plottare" dei dati significa **rappresentare graficamente dei dati numerici** per analizzarli e interpretarli, evidenziando tendenze, relazioni e valori anomali.

Ecco l'output generato da pyplot. Sulle ascisse abbiamo x (temperatura inventata), sulle ordinate invece abbiamo y (temperatura effettiva).



Il grafico mostra che i dati di allenamento hanno effettivamente una relazione lineare tra x e y.

DEFINIZIONE DEL MODELLO, LOSS FUNCTION E OPTIMIZER

Andiamo a definire un modello lineare con 1 input, 1 output e un solo neurone e visualizziamo gli output prodotti dal modello.

```
tensor([[42.8796
],

[50.5250],

[58.1704],

[42.1151],

[52.0541],

[60.4640],

[41.3506],

[50.5250],

[58.9349],

[46.7023],
```

Ecco la risposta del nostro modello che come possiamo vedere non è accurata, questo output rappresenta il nostro output previsto (ŷ).
Il primo input della lista era 55 e ha previsto 26.14, mentre il nostro valore reale (yi) è 13.

```
[54.3477],

[62.7576],

[39.8215],

[52.8186]],
grad_fn=<AddmmBa
ckward0>)
```

Questo succede perché il modello quando viene istanziato per la prima volta i parametri dell'equazione **y = wx + b** rispettivamente **weight** e **bias** vengono generati **randomicamente**. Infatti se proviamo a rieseguire la cella avremo output diversi perché appunto i parametri vengono rigenerati portando ad un output diverso ogni volta che il modello viene **istanziato**.
Possiamo però avere output uguali fissando la variabilità casuale di questi parametri in questo modo:

```
torch.manual_seed(42)
```

Il **seed** è il seme da cui germina la **casualità** del nostro modello, di solito viene scelto in modo casuale ma di solito si usa il valore 42 per motivi filosofici. Ora ogni qualvolta il modello viene istanziato abbiamo gli stessi valori in output

Adesso andiamo a definire la **loss function** e l'**optimizer**.

```
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

Per la loss function andiamo a definire tramite torch l'algoritmo che useremo ovvero **MSELoss (Mean Squared Loss)**, definendola in questo modo torch andrà ad attuare la formula definita nel secondo step del ciclo di training.

Andiamo ad utilizzare l'**Adam Optimizer**, anche di questo algoritmo la formula viene eseguita da pytorch. All'optimizer passiamo 2 parametri, i parametri su cui deve lavorare e il nostro learning rate (η) impostato inizialmente 0.01

NOTA: La **loss function** è una funzione matematica per misurare la differenza tra il valore **previsto** (ŷ) da un modello e il valore **reale** (yi) o desiderato.

CICLO DI ADDESTRAMENTO

Adesso andiamo a scrivere il codice del nostro ciclo di addestramento, seguendo l'ordine delle funzioni del ciclo di addestramento Ciclo di training.

```
for epoch in range(3000): #Per ciascuna epoca nel range 1000
    optimizer.zero_grad() #Azzerro i gradienti
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    loss.backward() #Calcola i gradienti
    optimizer.step() #Aggiorna i pesi
    if epoch % 100 == 0:
        print(f'Epoca {epoch}, Loss: {loss}')
```

Definiamo il ciclo di addestramento con un **for**, questo for va a dire che per ogni 1000 **epoche** esegue il ciclo di addestramento.
Con la funzione **optimizer.zero_grad()** andiamo ad azzerare i **gradienti** del ciclo precedente.
Andiamo poi a definire una variabile **y_pred** che rappresenta l'output **previsto (Forward Pass)**.

Si definisce epoca una ripetizione in cui il modello apprende tutti i dati del nostro dataset

Dopo questa istruzione andiamo a calcolare la loss tramite pytorch, quindi inizializziamo una variabile loss che conterrà il risultato della formula generato da

pytorch con il comando **loss_fn**, come previsto dalla formula passiamo 2 parametri rispettivamente l'output **previsto** (y_pred) e l'output **effettivo** (y) ovvero il tensore generato all'inizio (temperature effettive).

Dopodiché dobbiamo confrontare la loss con i vari parametri ad ottenerne i gradienti per valutare quanto ogni parametro abbia inciso sull'errore.
Chiamiamo la funzione **backward** sulla variabile **loss.**

Ora passiamo al 4 step del nostro ciclo dove aggiorniamo i parametri, utilizziamo quindi la nostra variabile **optimizer** dichiarata precedentemente e andiamo a chiamare la funzione **step** questa funzione serve proprio a fare un passaggio di aggiornamento dei parametri.

L'if serve a stampare i **progressi** ogni 100 epoche e visualizzare epoca attuale e loss function.

Qui troviamo l'output generato dopo il ciclo di addestramento.

```
Epoca 0, Loss: 1044.3514404296875
Epoca 100, Loss: 6.815361022949219
Epoca 200, Loss: 6.2898993492126465
Epoca 300, Loss: 5.671482086181641
Epoca 400, Loss: 5.007758617401123
Epoca 500, Loss: 4.340624809265137
Epoca 600, Loss: 3.7006661891937256
Epoca 700, Loss: 3.1093404293060303
Epoca 800, Loss: 2.580324411392212
Epoca 900, Loss: 2.1207163333892822
Epoca 1000, Loss: 1.7322407960891724
```

Possiamo notare che nella prima epoca, la loss function è un valore molto alto
quindi la differenza tra l'output previsto e quello reale è enorme.
Man mano che il ciclo prosegue questo valore diminuisce quindi la differenza sarà minore e il valore sarà molto più vicino a quello reale.

Se aumentiamo il range nel for eseguirà molte più **ripetizioni**, quindi facendo più step apprenderà più **informazioni**.
Se aumentiamo il **learning rate** decrescerà molto di più ma se aumentiamo il learning rate dobbiamo aumentare il numero di **ripetizioni** evitando instabilità.
Un esempio con 3000 **epoche** e un **learning rate** di 0.1:

```
Epoca 2800, Loss: 0.4438796937465668
Epoca 2900, Loss: 0.44381776452064514
```

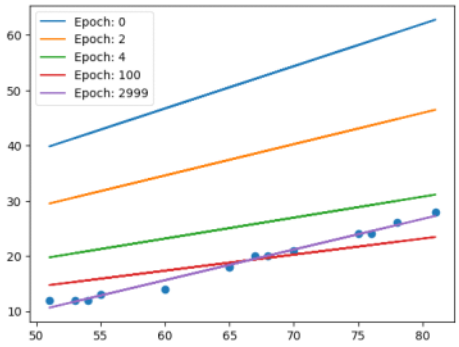
Il modello apprende e arriva ad un valore basso ma inizia a "**fermarsi**" su un valore, oltre questo valore non si può più scendere se non di **poco.**

Visualizzazione della curva di apprendimento

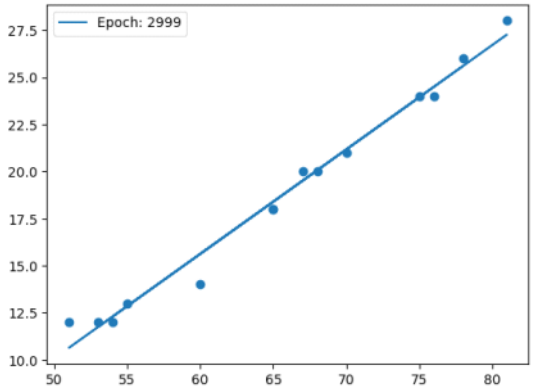
Adesso andiamo a visualizzare la curva di apprendimento.

```
    if epoch in [0, 2, 4, 100, 2999]:
        pyplot.plot(x, y_pred.detach(), label = f'Epoch: {epoch}')
pyplot.scatter(x,y)
pyplot.legend()
```

Plottiamo questi dati nelle **epoche 0,2,4,100,2999** e visualizziamo la relazione tra **x (input) e la y prevista**, usiamo **detach** per staccare i gradienti nella rappresentazione. Stampiamo il grafico con lo **scatter** e attiviamo la **legenda.**



Ad epoca 0 notiamo che la linea è molto **alta** quindi completamente diverso dal valore delle x, nella seconda epoca abbiamo un abbassamento del **bias**. Nell'ultima migliaia notiamo che va a modificare il **peso(weight)** che regola la pendenza della linea.
Se plottiamo solo l'ultima epoca (2999) vediamo che l'approssimazione è quasi corretta.



Notiamo che il modello riesce a **prevedere** quasi perfettamente le nostre **x**

Inferenza modello

Adesso andiamo ad effettuare l'**inferenza** della rete neurale, ovvero il processo che il modello utilizza per trarre conclusioni da dati nuovi.

```
current_temperature = torch.tensor([70], dtype=torch.float32).unsqueeze(1)
model(current_temperature)
```

Questa cella va a dare **70** come input, salvato su una variabile e va a chiamare la risposta del modello. Riceviamo una risposta accurata e coerente con la **y** iniziale. Quindi il modello ha appreso molto bene dalle nostre informazioni.

```
tensor([[21.1672]], grad_fn=<AddmmBackward0>)
```

Ora andiamo a stampare i parametri del modello rispettivamente **peso** e **bias**.

```
for param in model.parameters():
    print(param)
```

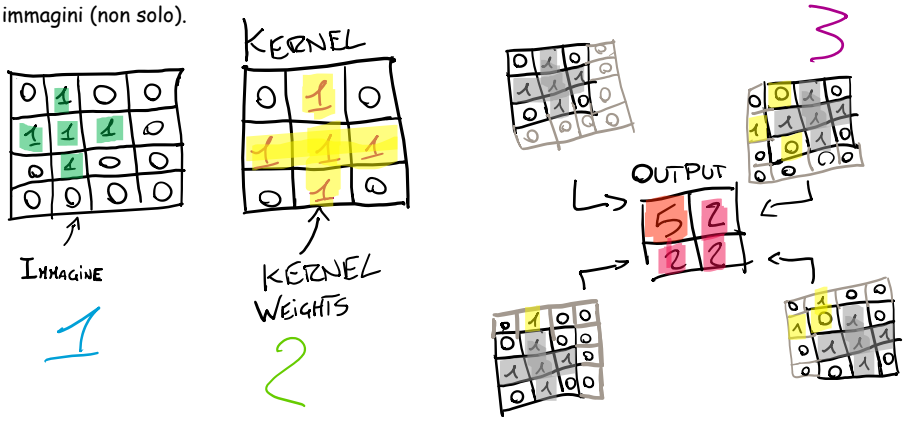
L'output che riceviamo è questo:

```
Parameter containing:
tensor([[0.5537]], requires_grad=True)
Parameter containing:
tensor([-17.5949], requires_grad=True)
```

Il primo valore (**0.55**) è il **peso** il secondo (**-17.6**). Ritornando all'equazione iniziale (**Pagina 1**). Realizzando l'equazione troveremo **y** ovvero la **temperatura reale**.

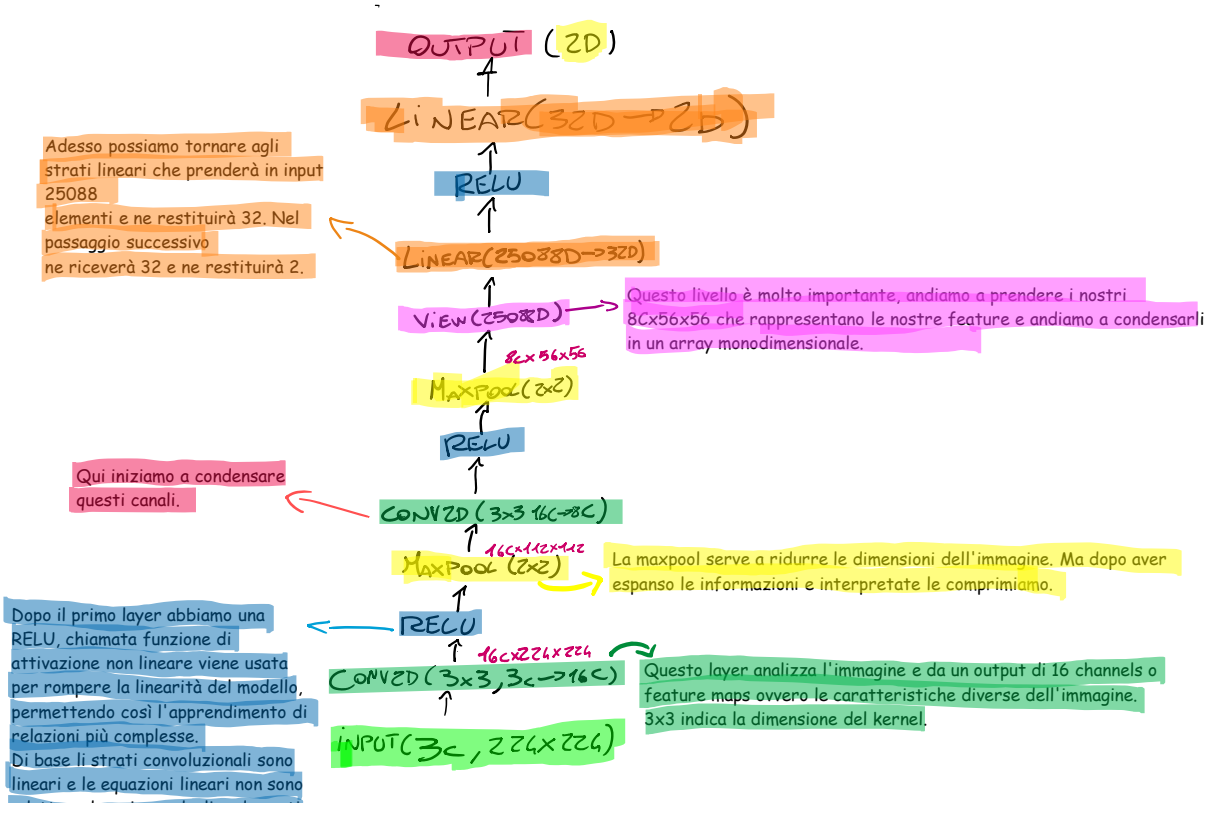
1.1: Riconoscimento Visivo con CNN

Se nello scorso progetto abbiamo lavorato con dei numeri, in questo lavoreremo con le immagini, creeremo un classificatore di immagini che restituisca una probabilità. Per far ciò abbiamo bisogno di **strati convoluzionali**. Sono strati fatti specificatamente per la gestione delle immagini (non solo).



Usiamo questo layer perché riesce a rilevare le feature (caratteristiche) locali della nostra immagine come bordi, angoli, texture, forme, vedendo proprio come sono disposti i pixel. L'output che produce questo convolutional layer viene chiamato channel o feature, può anche produrre più canali. Ogni canale è capace di catturare una caratteristica diversa es.(bordo).
Per farlo usa un filtro chiamato **kernel**, solitamente si usa la matrice 3x3 (Figura 2) nel kernel troviamo i pesi del kernel, i parametri che poi devono essere appresi dal modello, ciò richiederà uno sforzo in più a livello di calcolo.
Questa matrice va a scansionare (Figura 3) tutte le diverse parti dell'immagine producendo un output, matematicamente va a calcolare un prodotto scalare del kernel stesso con la matrice scansionata dell'immagine (Figura 1).
Noi partiamo da un'immagine 4x4 e nella Figura 3 notiamo che l'output è solo 2x2, quell'output viene chiamato feature o caratteristica.
Ora vediamo la struttura della rete neurale.

Lo stride è il passo con cui il filtro si sposta sull'input durante l'operazione di convoluzione. Un valore di stride più grande comporta uno spostamento maggiore del filtro e quindi una riduzione della dimensione dell'output. Al contrario, un valore di stride più piccolo comporta uno spostamento minore del filtro e quindi un'output di dimensioni maggiori. La formula per calcolare la dimensione dell'output della convoluzione in base al valore di stride è:



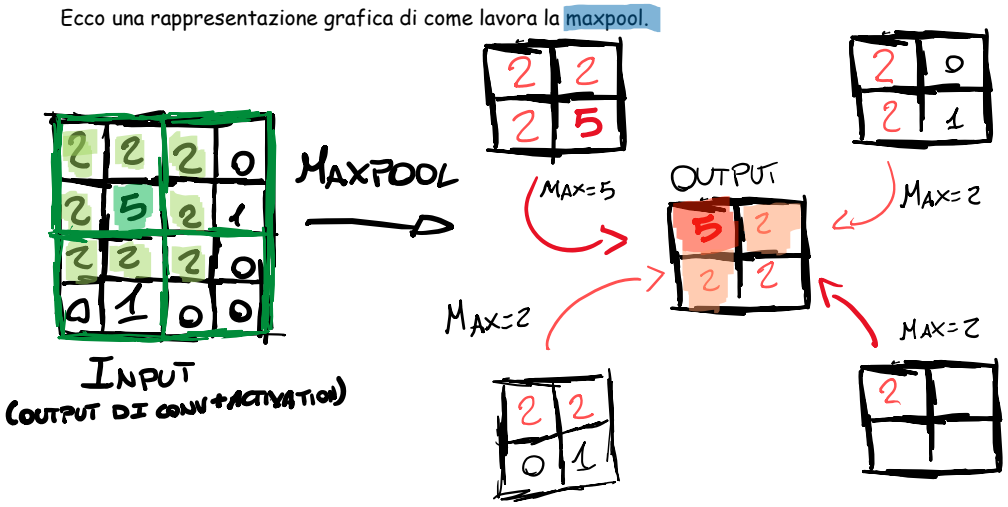
NOTA: Usiamo un array monodimensionale perché vogliamo che il nostro modello restituisca una probabilità.

permettendo così l'apprendimento di relazioni più complesse.
Di base li strati convoluzionali sono lineari e le equazioni lineari non sono adatte a descrivere degli andamenti complessi come quelli di un immagine.

INPUT(3c, 224x224)

3x3 indica la dimensione del kernel.

Ecco una rappresentazione grafica di come lavora la maxpool.

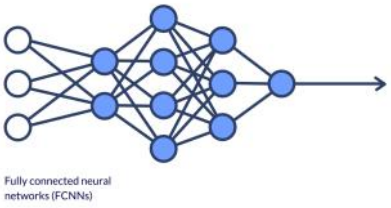


La maxpool prende l'input che in questo caso è un canale di feature maps in output da uno strato di convoluzione, come vediamo in output va a prendere solo le feature principali del nostro canale e le inserisce dentro una matrice 2x2.

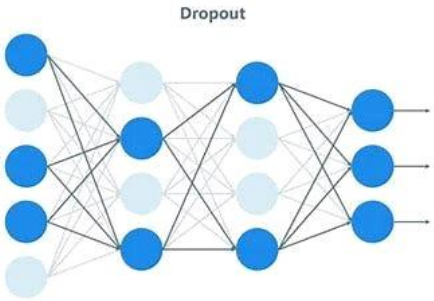
Evitare l'Overfitting: Il Dropout

Quando una rete neurale è molto grande e si allena a lungo, rischia di andare in **Overfitting** (sovra-adattamento). I neuroni iniziano a "fidarsi" troppo l'uno dell'altro, creando dipendenze complesse che funzionano solo sui dati di training ma falliscono nel mondo reale. È come un lavoro di gruppo dove uno studente fa tutto e gli altri si limitano a copiare.
Il **Dropout** (letteralmente "lasciar cadere") risolve questo problema in modo drastico ma efficace.
Il concetto: Durante l'addestramento, ad ogni passaggio, disattiviamo ("spegniamo") casualmente una percentuale di neuroni (solitamente il 20% o il 50%).
L'effetto: I neuroni non possono più affidarsi ciecamente ai loro vicini, perché potrebbero essere spenti in qualsiasi momento. Ogni neurone è costretto a imparare caratteristiche utili in modo indipendente. Questo crea una rete molto più robusta e democratica, capace di generalizzare meglio su dati mai visti.

Nota: Durante l'inferenza (quando usiamo il modello), il Dropout viene disattivato automaticamente e tutti i neuroni partecipano al 100%.



Vediamo una rete **Fully Connected** (completamente connessa). Ogni neurone dello strato precedente è collegato a tutti i neuroni dello strato successivo. Tutti i nodi sono attivi e partecipano all'elaborazione dell'informazione. In questa configurazione, i neuroni possono sviluppare una "co-dipendenza", affidandosi eccessivamente ai segnali dei vicini per correggere gli errori, il che porta spesso all'**Overfitting**.



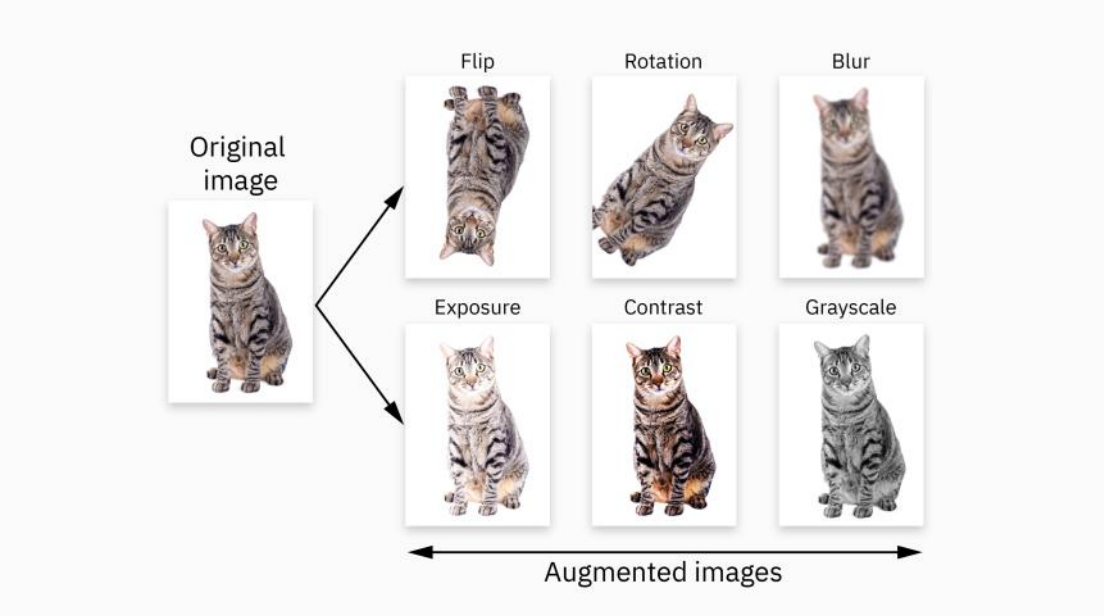
Vediamo la stessa rete a cui è stato applicato il **Dropout**. I nodi grigi rappresentano i neuroni che sono stati **temporaneamente disattivati** (spenti) per questo specifico passaggio di training.

- Notiamo che non solo il neurone è spento, ma anche **tutte le connessioni (frecce)** in entrata e in uscita da esso sono rimosse.
- La rete appare più "snella" e costringe i neuroni attivi (quelli colorati) a farsi carico del lavoro extra, imparando caratteristiche più robuste e indipendenti.

Nota Bene: La disattivazione è casuale e cambia ad ogni singolo step di aggiornamento. Un neurone spento ora potrebbe essere acceso nel passaggio successivo.

Tecniche per migliorare l'apprendimento: Data Augmentation

Uno dei problemi principali nel Deep Learning è avere pochi dati. Se mostriamo alla rete sempre le stesse 100 foto di gatti, lei imparerà a riconoscere *quei* gatti specifici, non il concetto universale di "gatto".
La **Data Augmentation** è una tecnica per aumentare artificialmente il nostro dataset senza dover scattare nuove foto. L'idea è applicare trasformazioni casuali alle immagini di training ogni volta che vengono passate al modello.
Come funziona: Ogni volta che l'immagine viene caricata, subisce una modifica random: può essere ruotata di pochi gradi, ribaltata orizzontalmente (come allo specchio), leggermente zoomata o modificata nella luminosità.
Perché è geniale: Per il computer, un gatto ruotato di 15° è una matrice di pixel completamente diversa dall'originale. In questo modo, il modello non vede mai due volte l'immagine identica e impara che la forma del gatto è indipendente dal suo orientamento o dalla luce. È come se il nostro dataset diventasse virtualmente infinito.



Progetto 2: Classificatore MNIST

Oggi andiamo ad usare le conoscenze apprese prima con il riconoscimento e la classificazione delle immagini. Andremo a costruire un modello in cui noi passeremo un'immagine di un numero scritto a mano da un umano presa da un dataset online e il modello restituisce a quale numero corrisponde.
Iniziamo importando le librerie necessarie.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
```

Andiamo ad effettuare la parte iniziale dove convertiamo le immagini in tensori e prepariamo i dati.

```
# Convertire immagini MNIST in tensori di 4 dimensioni (n di immagini, altezza, larghezza e
canali)
transform = transforms.ToTensor()
# Dati di training
train_data = datasets.MNIST(root = 'cnn_data', train = True, download=True,
transform=transform)
# Dati di test
test_data = datasets.MNIST(root = 'cnn_data', train = False, download=True,
transform=transform)
# Dataloader
train_loader = DataLoader(train_data, batch_size=10, shuffle=True)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

Il dataset è preso dalla libreria e installato nella directory del progetto nella cartella cnn_data

splittiamo i dati in dati di allenamento e dati di "test", lo facciamo perché se noi quando passeremo l'immagine, diamo in pasto un'immagine che abbiamo usato in allenamento lui saprà già qual è, mentre se noi dividiamo i dati in training e set non apprenderà in modo meccanico solo le immagini del training ma diamo immagini nuove contenute nei dati di test. Così vedremo se anche con immagini che non ha mai visto riesce a dare la risposta accurata.

Perché mescoliamo solo i dati di train e non quelli di test? Perché altrimenti il modello apprende sempre nello stesso ordine e non sarà capace di contestualizzare, quindi per ogni 10 immagini (batch_size) andiamo a mescolare queste immagini. I dati di test non c'è bisogno di mescolarli.

Adesso inizializziamo il modello e tutti i suoi layers

```
# Classe del modello
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        # Fully connected layer
        self.fc1 = nn.Linear(5*5*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2) # 2 x 2 Kernel e Stride 2

        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)

        x = x.view(-1, 16*5*5) # Numero negativo cosi possiamo variare la
dimensione del batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

# Creiamo un istanza del modello
torch.manual_seed(41)
model = ConvolutionalNetwork()
```

Significa che il layer **riceve 1 canale in input** non un singolo dato ma un'intera mappa 2D. Applica **6 filtri convoluzionali distinti**. Ogni filtro ha **kernel 3×3**. Lo stride è 1. Il risultato è **6 feature map in output**.
Il secondo layer riceve **6 canali in input**, cioè le 6 feature map prodotte prima. Ogni filtro ora è **3×3×6**, perché deve guardare tutti i canali. Applica **16 filtri diversi** e produce **16 feature map in output**.

Dopo i layer convoluzionali otteniamo 16 feature map di dimensione 5×5.
Prima di passare ai layer fully connected, queste mappe vengono flattenate in un unico vettore di 5·5·16 = 400 valori.
Il primo layer fully connected prende questo vettore e lo proietta in 120 neuroni, costruendo una rappresentazione più astratta.
Il secondo layer riduce ulteriormente la dimensionalità da 120 a 84.
L'ultimo layer mappa gli 84 neuroni in 10 valori di output, uno per ciascuna classe.

Nel metodo forward definiamo come i dati attraversano la rete.
L'input x passa prima nel primo layer convoluzionale, seguito da una funzione di attivazione ReLU, che introduce non linearità.
Subito dopo applichiamo un max pooling 2×2 con stride 2 per ridurre la dimensione spaziale e rendere le feature più robuste.
Il risultato passa poi nel secondo layer convoluzionale, di nuovo seguito da ReLU e da un altro max pooling, riducendo ulteriormente la dimensione delle feature map.
A questo punto flatteniamo l'output in un vettore di dimensione 16·5·5. Usiamo -1 per lasciare a PyTorch il calcolo automatico della dimensione del batch.
Il vettore entra nei layer fully connected: prima da 400 a 120 neuroni, poi da 120 a 84, entrambi con ReLU.
Infine, l'ultimo layer fully connected produce 10 valori di output, che rappresentano i logits delle classi.

Adesso dopo aver preparato i dati, definito l'architettura di rete e istanziato il modello andiamo a creare il ciclo di training del nostro modello.

```
#Loss FN Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
# Allenamento
# Variabili
epochs = 5
train_losses = []
test_losses = []
train_correct = []
test_correct = []
# Loop
for i in range(epochs):
    trn_corr = 0
    tst_corr = 0
    # train
    for b,(x_train, y_train) in enumerate(train_loader):
        b += 1 # Inizializziamo i nostri batches a 1
        y_pred = model(x_train) # Prendiamo la y iniziale dal nostro set di
addestramento
```

```
        loss = criterion(y_pred, y_train) # Confrontiamo la predizione con le
risposte corrette in y_train

        predicted = torch.max(y_pred.data, 1)[1] # Aggiungere il numero delle
predizioni corrette. Indicizzato al primo punto
        batch_corr = (predicted == y_train).sum() # Quando è stato corretto dal
suo batch. True = 1, False = 0
        trn_corr += batch_corr # Tiene traccia mentre procediamo con
l'allenamento
        # Aggiornamento Parametri
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print
        if b%600 == 0:
            print(f'Epoch: {i} Batch: {b} Loss: {loss.item()}')
    train_losses.append(loss)
    train_correct.append(trn_corr)
```

Dopo aver definito la loss function e l'optimizer, inizializziamo le variabili necessarie a monitorare l'allenamento.
Impostiamo il numero di epoche, ovvero quante volte il modello vede l'intero dataset di training, e creiamo delle liste per salvare loss e accuratezza.

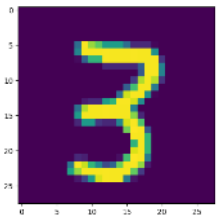
Per ogni epoca iteriamo sui batch del dataloader di training.
Per ogni batch il modello produce una predizione y_pred a partire dagli input x_train.
La loss viene calcolata confrontando queste predizioni con le etichette reali y_train usando la CrossEntropyLoss.
Successivamente calcoliamo quante predizioni sono corrette confrontando la classe con probabilità massima con la label reale. Questo ci permette di tracciare le performance durante l'allenamento.
A questo punto eseguiamo il classico ciclo di ottimizzazione: azzeriamo i gradienti, calcoliamo i gradienti tramite backpropagation e aggiorniamo i pesi usando l'optimizer Adam.
Periodicamente stampiamo la loss per monitorare l'andamento dell'allenamento.
Al termine di ogni epoca salviamo la loss e il numero di predizioni corrette.

Ora dopo aver finito l'allenamento, andiamo a preparare l'immagine per essere passata al modello.

```
img_idx = 5214
# Prendiamo un'immagine
test_data[img_idx] # tensore con un immagine e mostrerà il label
# Prendiamone solo i dati
test_data[img_idx][0]
# Ridimensioniamo
test_data[img_idx][0].reshape(28, 28)
# Mostrare l'immagine
plt.imshow(test_data[img_idx][0].reshape(28, 28)
)
```

Questo indice è abbastanza casuale possiamo mettere qualsiasi visto che il dataset è molto vasto.

OUTPUT:



Preparata e mostrata l'immagine passiamola al modello e riceviamo delle probabilità.

```
# Passiamo l'immagine al modello
model.eval()
with torch.no_grad():
    new_pred = model(test_data[img_idx][0].view(1,1,28,28)) #batch_size di 1, 1 canale dei colori, dimensione
28*28
# probabilità
new_pred
tensor([[-0.0991, -0.0449,  0.0486,  0.0004,  0.0598,  0.0327, -0.0668, -0.1152,
         0.0228, -0.0117]])
```

Se eseguiamo questo blocco di codice otterremo le nostre probabilità per ogni classe (numero) infatti ne abbiamo 10.
Nel prossimo blocco andiamo a prendere la probabilità massima con torch.

```
new_pred.argmax()
```

Otteniamo come risposta il tensore con il label del dataset corretto.
`tensor(3).`

Adesso abbiamo appena finito di sviluppare il nostro primo classificatore di immagini MNIST.

Progetto 2.1: Classificazione visiva

Dopo aver visto la teoria e la struttura della nostra rete neurale adesso possiamo procedere con la scrittura del codice e l'addestramento. Abbiamo preparato ben 2 dataset, uno contiene 6000+ immagini di **gatti** e l'altro 6000 + immagini di **cani**, il nostro modello deve restituire una probabilità, ovvero la probabilità che l'immagine che diamo in pasto al modello sia un cane o un gatto. Proseguiamo con importare le librerie necessarie:

```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from PIL import Image
from sklearn.metrics import confusion_matrix, precision_score, recall_score,
accuracy_score
torch.manual_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Qui abbiamo semplicemente importato le librerie e successivamente dichiarato una variabile device che contiene il dispositivo su cui stiamo lavorando. Se è disponibile la GPU (maggiori performance) usiamo quello altrimenti usiamo la CPU (molto più lenta).

Dopo questo passaggio procediamo con la creazione di tutti gli attributi d'istanza, questi rappresenteranno i nostri layer.

```
class Net(nn.Module):
    # Metodo costruttore / Schema della rete neurale
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(8* 56 * 56, 32)
        self.fc2 = nn.Linear(32, 2)
```

Andiamo a definire una classe Net andando ad ereditare dalla classe Module messa a disposizione da pytorch. Andiamo a definire poi il metodo costruttore (2 riga). Andiamo a definire poi la super classe, per poi creare i nostri attributi d'istanza.

Qui andiamo a definire il primo livello della struttura il Conv2D, rispettivamente deve avere 4 parametri, ovvero i canali d'ingresso (3), i canali d'uscita (16), la dimensione del kernel(3x3) e il padding (1) questo serve ad evitare che dopo il passaggio dall'input all'output si perde 1 pixel nel bordo, quindi per evitare che dopo ciascun layer l'immagine venga ridotta progressivamente aggiungiamo dei pixel ai bordi del channel.

Andiamo a definire gli strati lineari (view) usiamo fc1/2(full connected layer) ovvero che tutti i neuroni sono connessi. Mandando come input 25088 (come nello schema) e come output 32. Nel secondo abbiamo 32 in input e 2 in output (Ultimo strato dello schema), questi output rappresenteranno dei vettori monodimensionali e il primo elemento mostra la probabilità che un immagine mostri un gatto, mentre il secondo la probabilità che mostri un cane.

```
def forward(self, x):
    out = self.pool(F.relu(self.conv1(x)))
    out = self.pool(F.relu(self.conv2(out)))
    out = out.view(out.size(0), -1)
    out = F.relu(self.fc1(out))
    out = self.fc2(out)
    return out
```

Il metodo forward rappresenta il Forward Pass (Pagina 2). Passiamo i parametri self e x (immagine del gatto).

Ora andiamo ad appiattire le nostre immagini in una sola dimensione chiamando il metodo view su out, chiamiamo il metodo size.

Ora entriamo negli ultimi passaggi, quindi passiamo alla RELU il primo strato lineare, successivamente passiamo all'ultimo strato lineare. Infine ritorniamo out.

Dobbiamo applicare all'input il primo layer e

Dobbiamo applicare all'input il primo layer e farlo entrare nella RELU (Riga 2 e 3).
Uscito dalla prima RELU deve entrare nella pool (self.pool), ora l'immagine non sarà più 224x224 ma sarà stata dimezzata dalla Max Pool.

Nella cella successiva andiamo a ridimensionare l'immagine
a prendere i dati di trading dalla cartella nella root per poi suddividere il dataset in due parti, dati di training e dati di validation.

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(), # Trasformiamo l'immagine in un tensore
])
# Peschiamo i dati di training
dataset = datasets.ImageFolder(root='dataset', transform=transform)
print(dataset.class_to_idx)
# Suddividiamo il dataset in training e validation
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size,
val_size])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Ridimensioniamo l'immagine e le trasformiamo in **tensori**

Definiamo il percorso della cartella dataset e stampiamo quali classi ha trovato nella cartelle

Dividiamo randomicamente

Andiamo a creare i DataLoader ovvero degli strumenti di pytorch che servono a caricare e passare i dati progressivamente al ciclo di addestramento, dividendoli in batch per non passarli tutti (modello più pesante e non sarebbe capace di generalizzare poiché non vedrebbe tanti dati che cambiano). Usiamo la proprietà shuffle che mescola i dati ad ogni epoca.

Qui facciamo uno split e lo dividiamo in dati di allenamento e dati di validazioni. Lo dividiamo per poter interrogare il modello e valutarlo, per vedere se performa bene.

Esempio: Diamo ad uno studente degli esercizi (training data), ma ci teniamo per noi degli esercizi per valutarlo, perché dando tutti gli esercizi impara specificatamente e non riesce a generalizzare, non capendo davvero cosa succede perciò tiriamo fuori una parte dei dati e lo interroghiamo.

```
model = Net()
model = model.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = 0.001)
model.train()

for epoch in range(15):
    running_loss = 0.0 # Rappresenta la loss del singolo batch
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images) # Prima volta in cui passiamo le immagini nella rete
        loss = loss_fn(outputs, labels) # Calcoliamo la loss
```

Andiamo qui ad istanziare il modello, usando la classe Net dichiarata in precedenza, passiamo il modello al device (GPU o CPU) e lo impostiamo in modalità train. Inoltre definiamo la loss function e l'optimizer

```
        optimizer.zero_grad()
        outputs = model(images) # Prima volta in cui passiamo le immagini nella rete
        loss = loss_fn(outputs, labels) # Calcoliamo la loss
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_loss = running_loss / len(train_loader) # Loss media su tutti i batch
    print(f"Epoch {epoch}, Loss: {avg_loss}")
```

Qui effettuiamo il ciclo di addestramento come mostrato nel 2 capitolo, con leggeri differenze. In questo ciclo andiamo a definire una variabile running loss ovvero quanta loss c'è nel singolo batch (campione di dati) mentre nel secondo ciclo andiamo a dire che per ogni immagine e ogni labels (se è un gatto o un cane) prendendoli dal nostro train loader deve effettuare il ciclo di addestramento (Capitolo 2). Il primo step del ciclo passa alla GPU/CPU le immagini e i label, la variabile output va ad analizzare le nostre immagini tramite il nostro forward (classe net metodo forward), quindi adesso il nostro modello ci darà una y prevista, andiamo a calcolare la loss function, otteniamo i gradienti con la backpropagation, aggiorniamo i parametri e andiamo a sommare a running loss l'item della loss.
Nella riga sottostante andiamo a fare una media di loss stampandola insieme alle epoche.

Adesso avviando il modello inizierà l'addestramento e inizierà ad apprende i dati delle immagini.

```
Epoch 0, Loss: 0.6860524713993073
Epoch 1, Loss: 0.6737722635269165
Epoch 2, Loss: 0.647117656469345
Epoch 3, Loss: 0.6017773270606994
Epoch 4, Loss: 0.5721716940402984
Epoch 5, Loss: 0.5076314270496368
Epoch 6, Loss: 0.4637900412082672
Epoch 7, Loss: 0.41112600862979887
Epoch 8, Loss: 0.3859466940164566
Epoch 9, Loss: 0.2936780959367752
Epoch 10, Loss: 0.24542222470045089
Epoch 11, Loss: 0.20702120363712312
Epoch 12, Loss: 0.14328788593411446
Epoch 13, Loss: 0.13495455980300902
Epoch 14, Loss: 0.08506143242120742
```

Qui visualizziamo la loss function nelle varie epoche.

Persistenza del Modello: Salvare i "pesi"

Una rete neurale appena inizializzata è "stupida": i suoi pesi (i numeri nelle matrici) sono casuali. Tutta l'intelligenza acquisita durante le ore di addestramento risiede nel preciso aggiustamento di questi numeri.
Tuttavia, la memoria RAM è volatile. Se chiudiamo il programma Python, perdiamo tutto il lavoro fatto. Per usare il modello in futuro senza riaddestrarlo, dobbiamo salvare il suo stato.
State Dict: In PyTorch, il "cervello" del modello è conservato in un dizionario chiamato state_dict. Questo mappa ogni strato della rete (es. conv1, fc1) ai suoi parametri (tensori di pesi e bias).
Serializzazione: Salvare il modello significa scrivere questo dizionario su un file fisico (solitamente con estensione .pth o .pt).
Inferenza futura: Quando vorremo usare l'IA tra un mese, ci basterà istanziare una rete "vuota" (con pesi casuali) e sovrascriverli caricando il file .pth. In un istante, la rete recupererà tutta la sua "esperienza".
Ecco come salvare i pesi di un modello.

```
torch.save(model.state_dict(), "image_classifier.pth")
```

Per caricare i pesi invece:

```
model.load_state_dict(torch.load("C:\\Users\\rdarc\\Desktop\\Reti Neurali
\\Codice\\image_classifier.pth", weights_only=True))
```

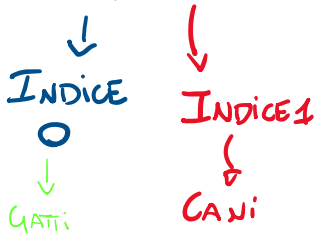
Inferenza del classificatore di immagini

Dopo aver istanziato il modello, allenato su un dataset di immagini adesso andiamo a visualizzare la risposta del modello passandogli delle immagini, che ritraggono o un cane o un gatto, il modello restituirà una percentuale.

```
image = Image.open('dataset\\cats\\cat (3).jpg').convert('RGB')
image_tensor = transform(image).unsqueeze(0).to(device)
model.eval()
with torch.no_grad():
    output = model(image_tensor)
    probs = torch.softmax(output, dim=1)
print(output)
print(f"{probs*100}%")
```

In questo blocco di codice andiamo ad aprire l'immagine, con il metodo Image.Open(percorso), convertendolo in RGB, quindi un input a 3 canali (come definito nell'architettura). Dopodiché andiamo a chiamare il nostro metodo transform sull'immagine, aggiungiamo la dimensione con unsqueeze e la passiamo alla GPU.

```
tensor([[ 0.9958, -4.2078]],
device='cuda:0')
tensor([[99.4533,  0.5467]],
device='cuda:0')%
```



Qui invece andiamo ad impostare il modello in modalità evaluation ovvero di valutazione.
Con il metodo with torch.no_grad() andiamo a dire che non vogliamo i gradienti perché siamo in modalità di valutazione non di allenamento.
Nelle seguenti righe andiamo a passare al modello il tensore dell'immagine.
Infine stampiamo le probabilità

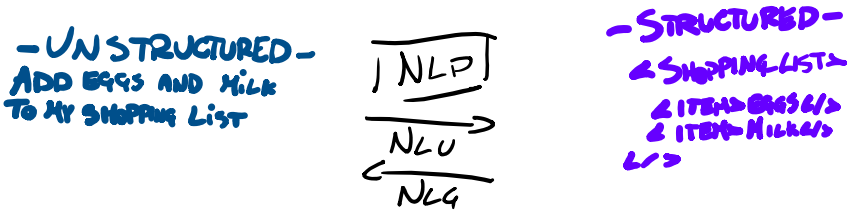
L'output che vediamo è la probabilità che l'immagine passata (cat (3).jpg) sia un gatto o un cane. Pytorch suddivide il nostro dataset in classi e divide ognuno per indici, nel nostro caso 0 è l'indice del dataset contenente i gatti e 1 è l'indice del dataset contenente i cani.

Noi passando l'immagine di un gatto otteniamo sull'indice 1 99.3469% di probabilità che sia un gatto e lo 0.6531% che sia un cane. Il modello risponde perfettamente

Capitolo 2: NLP (Natural Language Processing)

Nel capitolo 1 abbiamo lavorato con le immagini ovvero un dato statico e organizzato in uno spazio 2D. Le CNN estraevano feature spaziali dove i pixel vicini sono i più importanti.
Il Linguaggio Naturale (testo), al contrario è un dato sequenziale.
L'informazione non è statica: l'ordine delle parole è tutto. Per esempio "Il gatto mangia il topo" è diverso da "Il topo mangia il gatto".

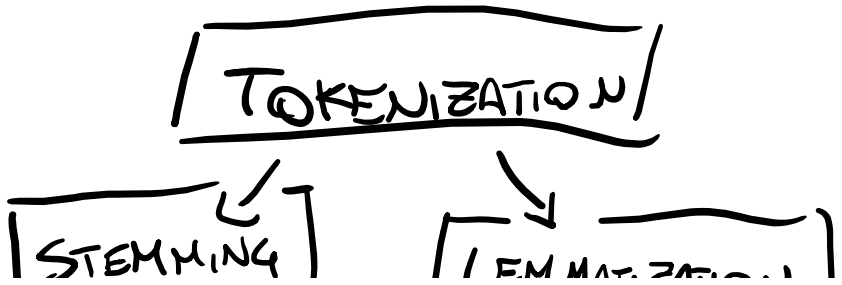
L'NLP, o Elaborazione del Linguaggio Naturale, è un ramo dell'intelligenza artificiale che permette ai computer di comprendere, interpretare e generare il linguaggio umano.
L'NLP utilizza una combinazione di linguistica computazionale, machine learning e deep learning per analizzare il linguaggio. Attraverso una serie di passaggi, come la tokenizzazione (suddivisione del testo in unità), l'analisi sintattica e la disambiguazione semantica, le macchine "digeriscono" il linguaggio per estrarne significato.

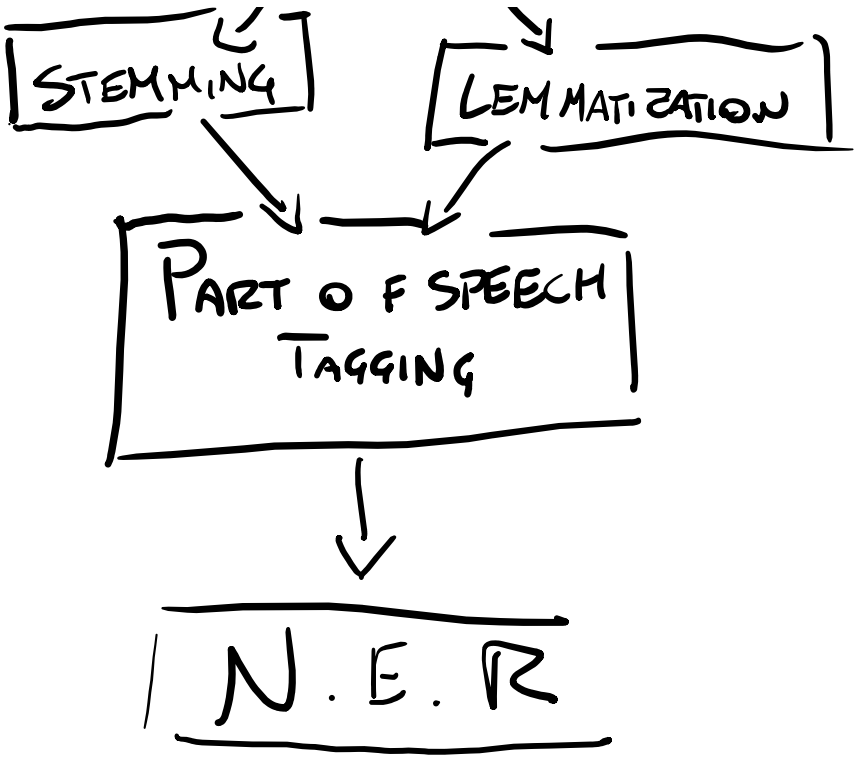


Abbiamo due tipi di dati, quelli strutturati e quelli non strutturati e il nostro NLP risiede proprio nel mezzo, traducendo da dati non strutturati a strutturati, quando noi andiamo da dati strutturati a non strutturati si chiama NLU (Natural Language Understanding), mentre viceversa è chiamato NLG (Natural Language Generation).
Ci sono diversi casi in cui l'NLP ci torna utile:

1) Machine Translation, i traduttori di lingue
2) Virtual Assistant/ Chatbot, Siri o Alexa
3) Sentiment Analysis, come se valutassimo se una recensione è positiva o meno
4) Spam Detection, ad esempio se un email è potenziale spam o no

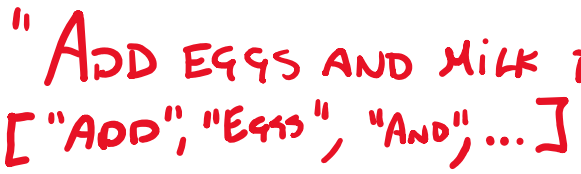
Ci sono parti fondamentali dell'NLP e sono




STEMMING
LEMMATIZATION
PART OF SPEECH TAGGING
N.E.R

Tokenizzazione

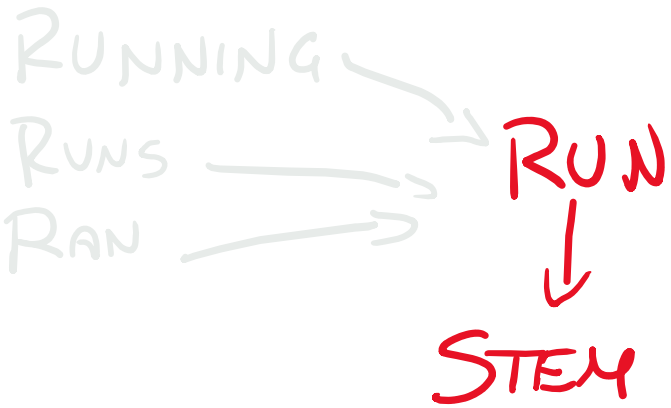
Il primo step dell'NLP è chiamato tokenizzazione, il suo funzionamento è quello di prendere una stringa di dati non strutturati e spezzarla in pezzi, se noi consideriamo la frase di prima "add eggs and milk to my shopping list", sono 8 parole e saranno 8 tokens (Word Tokenization) ovvero divide il testo in parole singole, utilizzato dai chatbot e assistenti virtuali.



Character Tokenization: segmenta il testo in singoli caratteri. Utile per le lingue senza confini di parole chiari o per attività che richiedono un'analisi granulare.
Subword Tokenization: suddivide il testo in unità più grandi dei caratteri ma più piccole delle parole. Utile per le lingue con morfologia complessa o per la gestione di parole fuori dal vocabolario.

Stemming & Lemmatization

Lo stemming e la lemmatizzazione costituiscono una fase delle pipeline di text mining che convertono i dati di testo non elaborati in un formato strutturato per l'elaborazione automatica. Sia la derivazione che la lemmatizzazione eliminano gli affissi dalle forme flesse delle parole, lasciando solo la radice.
Gli stemmer eliminano i suffissi delle parole confrontando i token delle parole in ingresso con un elenco predefinito di suffissi comuni.

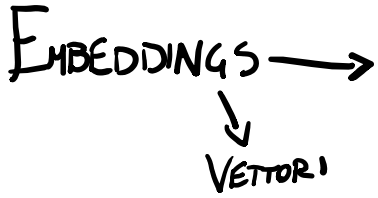


Ma lo stemming non funziona bene per ogni token, ad esempio:

University e Universal ad esempio non derivano da Universe. Per questi casi specifici abbiamo la lemmatizzazione.
La lemmatizzazione prende il token e impara il suo significato attraverso la definizione del dizionario, poiché la lemmatizzazione mira a produrre forme base del dizionario, richiede un'analisi morfologica più solida rispetto allo stemming.

Embedding

I word embeddings sono incorporamenti di parole, sono un modo per rappresentare le parole come vettori in uno spazio multidimensionale, in cui la distanza e la direzione tra vettori rispecchiano la somiglianza e le relazioni tra le parole corrispondenti

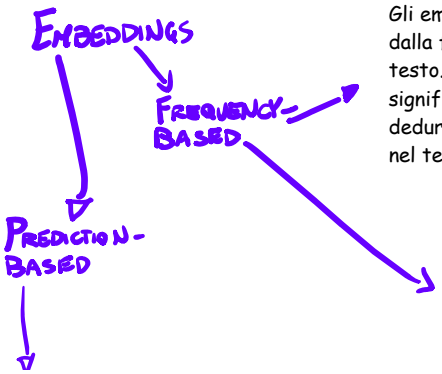


Questi embeddings sono molto presenti nell'NLP, Specificatamente nella text classification e N.E.R, anche ad esempio nella Spam Detection. Aiutano nei compiti di somiglianza tra parole e analogia delle parole, un altro esempio sono i Q&A.

Cos'è una Sliding Context Window? Un metodo per elaborare flussi di dati continui spostando una "finestra" di dimensioni fisse sui dati

I word embeddings sono creati da modelli addestrati su un ampio corpus di testo, per esempio su tutto Wikipedia.
Il processo inizia dal pre-processare il testo come ad esempio Tokenizzazione, Stemming & Lemmatization. Una finestra di contesto scorrevole identifica le parole che aiutano il contesto consentendo al modello di imparare le relazioni tra le parole. Il modello è allenato per predire in base al contesto, posizionare semanticamente le parole simili vicine l'una con l'altra nello spazio multidimensionale, i parametri sono regolati per minimizzare gli errori di predizione. Come appare?

Quindi ogni dimensione ha un valore numerico e crea un vettore 3D per ogni parola. Questi valori rappresentano le posizioni delle parole in uno spazio vettoriale 3d continuo. Queste parole hanno significato o contesto simile e ognuna ha una rappresentazione del vettore simile. Per esempio Apple e Orange rappresentano una relazione semantica tra loro e i valori dei vettori rispecchiano questa relazione. Invece se notiamo Happy e Sad hanno direzioni opposte tra loro nello spazio per il loro significato contrastante. Ovviamente gli attuali embedding di parole hanno centinaia di dimensioni non solo tre, questo consente di catturare relazioni più intricate e sfumature di significato.
Ci sono due approcci fondamentali di come il word embedding genera le rappresentazioni delle parole.



Gli embedding frequency-based derivano dalla frequenza di parole in un corpus di testo. Sono basate sull'importanza o il significato di una parola, ciò si può dedurre da quanto frequentemente appare nel testo una parola.

Uno di questi frequency-based è chiamato TF-IDF (Term Frequency Inverse Document Frequency). TF-IDF va a menzionare le parole che sono frequenti all'interno di un documento specifico ma sono rari nell'intero corpus.

Uno di questi Frequency-Based è chiamato TF-IDF (Term Frequency Inverse Document Frequency). TF-IDF va a menzionare le parole che sono frequenti all'interno di un documento specifico ma sono rari nell'intero corpus.
Ad esempio se abbiamo un documento sul caffè andrà ad evidenziare le parole come espresso o cappuccino che appaiono spesso in quel documento ma raramente in altri argomenti. Mentre parole come "the", "and" che appaiono quasi sempre in documenti avranno punteggi bassi nel TF-IDF.

Catturano le relazioni semantiche e contestuali tra le parole. Ad esempio nella frase "the dog is barking loudly e "the dog is wagging its tail", questo modello di embedding associa la parola "dog" con "bark", "tail". Questo consente a questi modelli di create un'unica rappresentazione fissa di un cane. Questi modelli eccellono nel separare le parole con un significato vicini e possono gestire i vari sensi in cui la parola può essere usata.

Ci sono vari modelli per generare word embeddings ma il più famoso è il "Word2vec" sviluppato da Google nel 2013, questo modello ha due architetture.

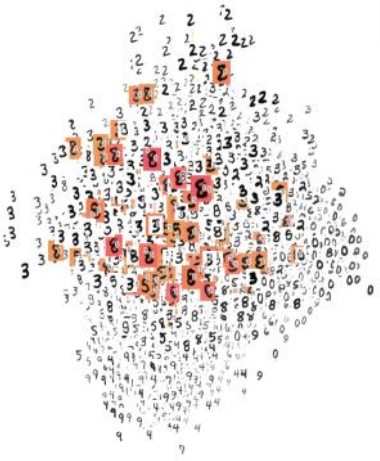
Il CBOW predice una parola bersaglio in base alle parole del contesto circostante.

Skip-Gram fa l'opposto ovvero predice le parole di contesto, data da una parola target

Un altro modello molto popolare è il GLOVE, che sta per Global Vectors for Word Representation, creato dalla Stanford University nel 2014. Usa la co-occorrenza statistica per creare vettori di parole.

Questi modelli sono ancora considerati molto validi nell'NLP. Ma mentre i tradizionali word embeddings assegnano un vettore fisso per ogni parola, i transformers model usano un altro tipo di embedding chiamato Contextual Based Embedding. Questo modello cambia la rappresentazione della parola basato sul contesto circostante. Ad esempio la parola "bank" ha diverse rappresentazioni nella frase. Questa "sensibilità di contesto" consente a questi modelli di catturare molte sfumature di significato e relazioni tra parole. Questo modello ha dato tanti miglioramenti in vari cambi del NLP.

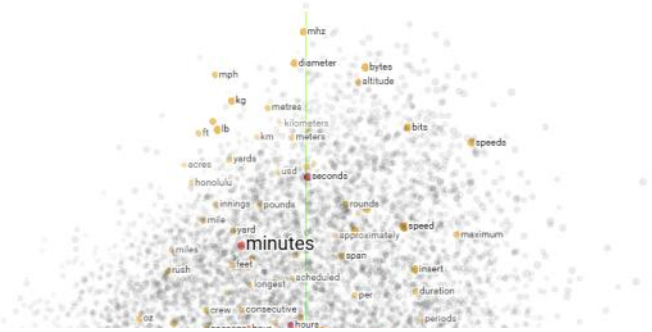
Co-occorrenza: frequenza con cui due o più parole appaiono insieme in un testo o corpus, indicando una forte relazione semantica o contestuale tra di esse.



Questa figura rappresenta uno spazio vettoriale applicato al database di immagini di un Classificatore MNIST, come possiamo vedere le immagini arancioni sono quelle "vicine" fra loro proprio perché hanno lo stesso numero.
La stessa cosa l'abbiamo con le parole che vengono posizionate però in una mappa semantica.

Qui invece possiamo vederlo con le parole, ho evidenziato qui la parola minutes (ho usato il Word2vec da 10.000 punti), possiamo vedere che le parole vicine alla parola minutes hanno una semantica vicina a quella della parola "minuti". Ad esempio:

- Pounds (unità di misura)
- Mph, ft, lb, km, meters
- Long, duration, years hours

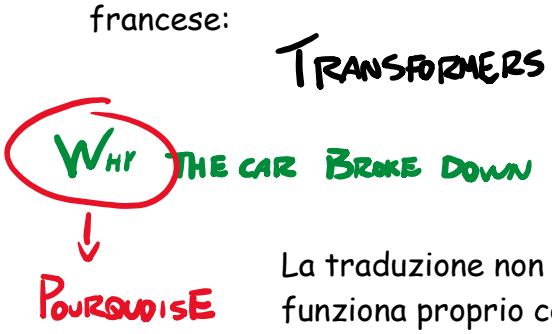


- Pounds (unità di misura)
- Mph, ft, lb, km, meters
- Long, duration, years hours

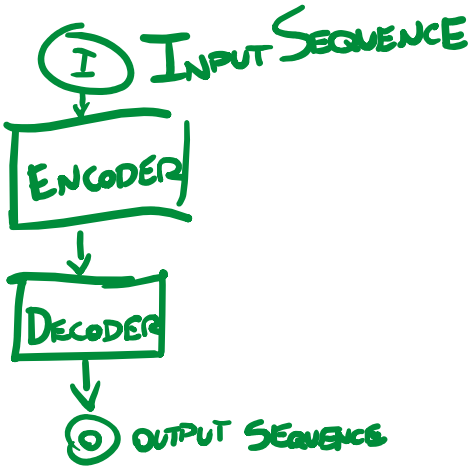
Sono tutte parole che hanno un significato vicino a quello della parola.

Transformers

Il compito dei transformers è letteralmente trasformare una sequenza in un'altra sequenza. Ad esempio prendiamo una frase da tradurre da inglese a francese:

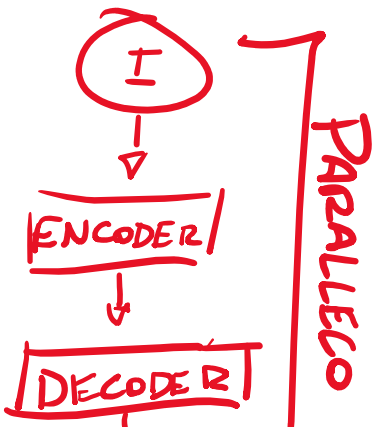


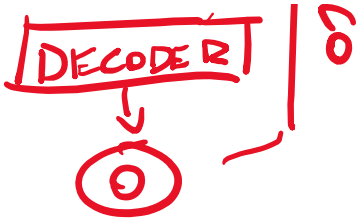
La traduzione non funziona proprio così ma è un esempio di transformers.



Il lavoro dei transformers avviene attraverso l'apprendimento sequenza a sequenza, dove i transformers prendono una sequenza di token, in questo caso la nostra frase, per poi predire la prossima parola nella sequenza di output. Questo accade iterando attraverso tutti i livelli di encoder e passando questi encodings al prossimo livello di encoder. Il decoder prende questi encodings e utilizza il loro contesto derivato per generare la sequenza di output.
I transformers hanno un tipo di allenamento semi-supervised. Con questo termine intendiamo che sono stati pre allenati in una maniera non supervisionata poi hanno passato una fase di Fine-Tuning attraverso apprendimento supervisionato per migliorare la loro prestazione.

I transformers usano un "Attention Mechanism" e questo fornisce contesto attorno agli oggetti nella input sequence, con l'esempio di sopra la traduzione parte dal "why" perché è l'inizio della nostra frase, ma i transformer cercano di identificare il contesto che dà il significato ad ogni parola nella frase. Proprio questo meccanismo che dà un passo in più ai transformer rispetto alle Reti Neurali RNN che eseguono compiti in sequenza i transformers eseguono varie sequenze in parallelo.




DECODER
δ
0

Positional Encoding

Abbiamo visto che gli Embedding trasformano le parole in vettori di numeri basati sul significato. "Gatto" sarà matematicamente vicino a "Felicita".
Tuttavia, c'è un problema enorme: al Transformer (l'architettura che useremo) non importa l'ordine.
Se passiamo i vettori di embedding puri, per la rete queste due frasi sono identiche:

1. *"Il cane morde l'uomo"*
2. *"L'uomo morde il cane"*

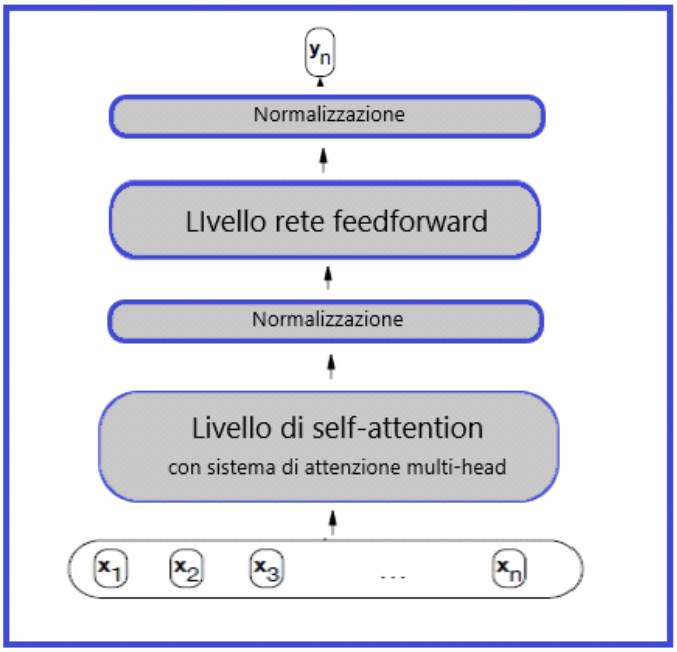
Perché? Perché a differenza delle vecchie reti (RNN) che leggevano una parola alla volta in sequenza, il Transformer legge tutto insieme in parallelo.
È come un insegnante che corregge tutti i compiti contemporaneamente invece che uno alla volta: perde il senso di chi ha consegnato prima.
La soluzione sarebbe quella di sommare la Posizione dobbiamo "timbrare" ogni parola con un numero che indica la sua posizione.
Non aggiungiamo una nuova colonna, ma sommiamo un vettore di posizione al vettore della parola.

$$Input = Embedding(Parola) + Embedding(Posizione)$$

(Immagina un grafico dove al vettore della parola "Cane" viene sommato un vettore che rappresenta "Posizione 2". Il risultato è un vettore leggermente modificato che contiene entrambe le info).
Implementazione in PyTorch.
Nei modelli GPT moderni, invece di usare complicate funzioni sinusoidali (come nel paper originale del 2017), spesso si usano dei Learnable Embeddings. La rete impara da sola qual è il miglior vettore per rappresentare la "posizione 1", la "posizione 2", ecc.

Self-Attention

Il trasformer si basa sulla tecnica di self-attention che consente alle reti di analizzare e ponderare l'importanza di ogni parte di una sequenza di dati rispetto a tutte le altre parti della stessa sequenza.
Questo è particolarmente utile per comprendere il contesto di una parola in una frase, poiché **una parola può influenzare o essere influenzata da qualsiasi altra parola,** indipendentemente dalla distanza nella sequenza. Ogni parola o simbolo nella sequenza viene convertito in un **vettore numerico (embedding)** che rappresenta il suo significato in uno spazio multidimensionale.



Blocco del Transformer

HumAI.it

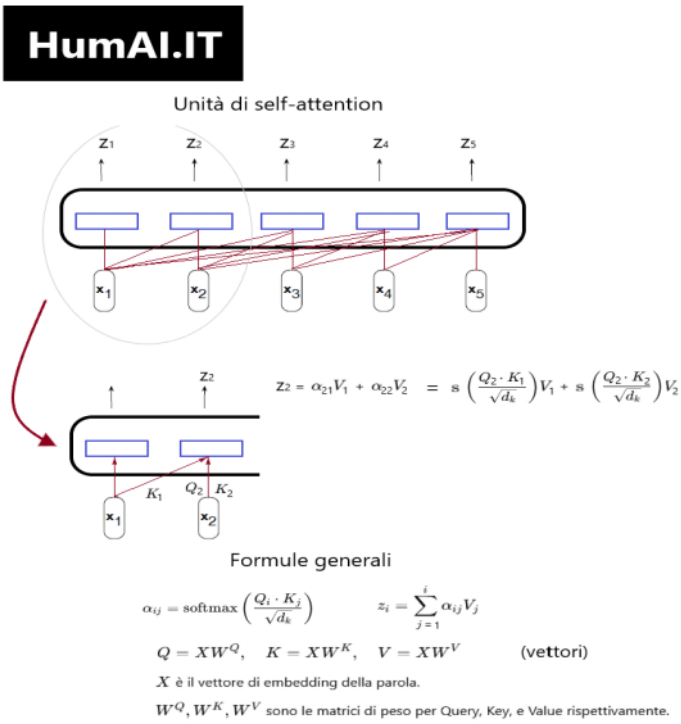
Il processo di self-attention si può suddividere in:

1. Calcolo dei punteggi di attenzione
    a. Per ogni parola, calcoliamo il prodotto scalare tra il suo vettore Query e i vettori Key di tutte le parole nella sequenza. Questo ci dà una misura di quanto ogni parola nella sequenza è rilevante per la parola corrente.
    $score(Qi, Kj) = \frac{Qi * Kj}{\sqrt{dk}}$
    Dove dk è la dimensione del vettore Key, e la divisione per $\sqrt{dk}$ serve a stabilizzare i gradienti durante l'addestramento.
2. Applicazione della Softmax:
    a. Convertiamo i punteggi in probabilità usando la funzione softmax.

$$aij = softmax\left(\frac{Qi * Kj}{\sqrt{dk}}\right)$$

Le aij rappresentano il peso di attenzione che la parola i dà alla parola j.

3. Combinazione dei valori (V):
   a. Utilizziamo questi pesi di attenzione (aij) per fare una media ponderata dei vettori Value(V). Questo ci dà il nuovo rappresentante della parola corrente considerando il contesto di tutte le parole.
   $zi = \Sigma aijVj$
   Il vettore zi è una combinazione dei valori Vj dove i pesi sono determinati dalla rilevanza calcolata tramite i vettori Query e Key.



Progetto 4: Sentiment Analysis

Dopo aver capito come una rete può comprendere il testo e il linguaggio naturale adesso andiamo a realizzare un modello capace di prendere una frase e capire se il fine di questa frase sia positivo o negativo.
Caso d'uso: "Oggi il concerto è stato fantastico!", la frase ha un fine positivo.
Caso d'uso 2: "Oggi non posso venire al concerto", la frase ha un fine negativo.
Useremo la libreria tiktoken per effettuare la Tokenizzazione.
Il flusso di lavoro è questo:

1. Creare la lista di parametri del modello.
2. Preparare i dati su quei lavorare.
3. Andare a tokenizzare.
4. Creare i batch per il training.
5. Allenare il modello.
6. Istanziare il modello con i suoi livelli.
7. Inferenza del modello.

Questi sono gli step da seguire per realizzare questo primo progetto sulla lavorazione del testo.

```
import torch.nn as nn
import torch.optim as optim
import tiktoken
config = {
    "embed_dim": 128,
    "num_heads": 8,
    "max_len": 100,
    "num_classes": 1,
    "batch_size": 2,
    "num_epochs": 5,
    "lr": 0.001
}
encoder = tiktoken.get_encoding("cl100k_base")
```

Con queste righe andiamo a definire vari parametri e ad inizializzare l'encoder per il tokenizer.
Embed_dim, rappresenta la dimensione del vettore che descrive ogni token dopo l'embedding.
Num_heads, il numero di teste per l'attenzione,
Max_len, è il numero massimo che può ricevere e mandare in output.
Num_classes, 1 classe per la risposta (label).
Batch_size, il numero del batch nel training.
Num_epochs, il numero delle epoche.
Lr, è il learning rate.
Perfetto adesso passiamo alla preparazione dei dati e degli esempi.

```
{"text": "Mi è piaciuto molto il film", "label": 1},
```

Ecco come appare una riga del nostro dataset, quindi andiamo a creare una serie di esempi e assegnare ad ognuno di essi un "label" questo label ci servirà a far capire al nostro modello se è positiva la nostra frase o negativa (1 positiva 0 negativa).
Quindi prepariamo gli esempi e mandiamoli all'encoder.

```
import json
import torch
with open("train_data.json", "r", encoding="utf-8") as f:
    data = json.load(f)
texts = []
labels = []
for item in data:
    texts.append(item["text"])
    labels.append(item["label"])
inputs = []
targets = []
for text, label in zip(texts, labels):
```

```
    tokens = encoder.encode(text)
    if len(tokens) < config["max_len"]:
        tokens += [0] * (config["max_len"] - len(tokens))
    else:
        tokens = tokens[:config["max_len"]]
    inputs.append(tokens)
    targets.append(labels)
inputs = torch.tensor(inputs, dtype=torch.long)
targets = torch.tensor(labels, dtype=torch.float)
```

In questo snippet di codice andiamo a fare i seguenti passaggi:

1. Andiamo ad aprire il json e carichiamo il contenuto in una variabile data.
2. Poi inizializziamo due variabili in cui appenderemo i due tipi di dati ovvero la frase contenuta in texts e il label quindi positivo o negativo, tutto ciò per ogni frase in data.
3. Poi andiamo a inizializzare altre due variabili che hanno lo stesso obiettivo effettivamente ma devono ricevere i dati finali in questo caso, inputs riceverà le frasi tokenizzare mentre targets i labels che sono già dei numeri.
4. Poi in un for andiamo a dire che per ogni text e ogni label, se la lunghezza della frase e minore della lunghezza massima dichiarata nella configurazione aggiunge 0 alla fine del token, altrimenti limita la lunghezza.
5. Andiamo ad aggiungere i tokens alla lista input e i label a targets e trasformiamo in tensori da passare al modello.

Dopo questo passaggio abbiamo il batching e lo shuffling.
Durante l'addestramento, i dati vengono **rimescolati casualmente** prima di essere forniti al modello.
Lo shuffling serve a evitare che il modello impari dipendenze legate all'ordine dei dati nel dataset. Se le frasi positive o negative fossero raggruppate, il modello potrebbe adattarsi a quell'ordine invece che al contenuto del testo.
Rimescolando i dati a ogni epoca, ogni batch contiene esempi diversi e più rappresentativi della distribuzione reale.
Successivamente creiamo i batch che ci serviranno nell'allenamento.

La dimensione del batch è un iperparametro di discesa del gradiente che controlla il numero di campioni di addestramento su cui elaborare prima che i parametri interni del modello vengano aggiornati.

```
batch_size = config["batch_size"]
dataset_size = inputs.size(0)
indices = torch.randperm(dataset_size)
shuffled_inputs = inputs[indices]
shuffled_targets = targets[indices]
for start_idx in range(0, dataset_size, batch_size):
    end_idx = start_idx + batch_size
    batch_inputs = shuffled_inputs[start_idx:end_idx]
    batch_targets = shuffled_targets[start_idx:end_idx]
```

Andiamo a prendere le due "misure" che ci serviranno ovvero quella del dataset e quella del batch, mescoliamo gli indici con randperm e gli assegniamo a delle nuove variabili mescolate.

Proseguiamo con la definizione di tutti i layer.

```
import torch.nn as nn
import torch
class Model(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_heads, max_len = 100):
        super().__init__()

        self.embedding = nn.Embedding(encoder.n_vocab, embed_dim)

        self.pos_embedding = nn.Parameter(torch.randn(1, max_len, embed_dim))

        encoder_layer = nn.TransformerEncoderLayer(
            d_model = embed_dim,
            nhead = num_heads,
            batch_first = True
        )
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers = 1)

        self.fc = nn.Linear(embed_dim, 1)

    def forward(self, x):
        x = self.embedding(x)
```

```
        seq_len = x.size(1)

        x = x + self.pos_embedding[:, :seq_len, :]

        x = self.transformer_encoder(x)

        x = x.mean(dim=1)

        x = self.fc(x)

        return x
model = Model(
    encoder.n_vocab,
    config["embed_dim"],
    config["num_heads"],
    config["max_len"]
)
```

Creiamo una classe chiamata modello in cui andiamo tramite pytorch a inizializzare i livelli e l'architettura del modello.

1. Embedding layer
2. Positional embedding
3. Transformer Layer
4. Layer di output

Nel forward passiamo i nostri dati in vari livelli, prima facciamo l'embedding, aggiungiamo le informazioni di posizione e dopodiché passiamo tutto al transformer.
Adesso passiamo allenamento del nostro modello seguendo il classico ciclo. Ciclo di training

```
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
num_epochs = 20
for epoch in range(num_epochs):
    indices = torch.randperm(dataset_size)
    shuffled_inputs = inputs[indices]
    shuffled_targets = targets[indices]
    for start_idx in range(0, dataset_size, batch_size):
        end_idx = start_idx + batch_size
        batch_inputs = shuffled_inputs[start_idx:end_idx]
        batch_targets = shuffled_targets[start_idx:end_idx]
        optimizer.zero_grad()
        outputs = model(batch_inputs)
        loss = criterion(outputs.squeeze(), batch_targets)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
OUTPUT:
Epoch 1/20, Loss: 0.6868
Epoch 2/20, Loss: 0.6552
Epoch 3/20, Loss: 0.5603
Epoch 4/20, Loss: 0.2165
Epoch 5/20, Loss: 0.0920
Epoch 6/20, Loss: 0.2450
Epoch 7/20, Loss: 0.0115
Epoch 8/20, Loss: 0.0076
Epoch 9/20, Loss: 0.0062
Epoch 10/20, Loss: 0.0056
Epoch 11/20, Loss: 0.0047
Epoch 12/20, Loss: 0.0040
Epoch 13/20, Loss: 0.0043
Epoch 14/20, Loss: 0.0046
Epoch 15/20, Loss: 0.0034
Epoch 16/20, Loss: 0.0035
Epoch 17/20, Loss: 0.0029
Epoch 18/20, Loss: 0.0037
Epoch 19/20, Loss: 0.0027
Epoch 20/20, Loss: 0.0028
```

Ora passiamo all'inferenza:

```
new_sentence = "Il concerto è stato bellissimo"
tokens = encoder.encode(new_sentence)
input_tensor = torch.tensor([tokens], dtype=torch.long)
```

```
model.eval()
with torch.no_grad():
    output = model(input_tensor)
    prob = torch.sigmoid(output)
if prob.item() > 0.5:
    print("Predizione: Positivo, Probabilità:", prob.item())
else:
    print("Predizione: Negativo, Probabilità:", prob.item())
OUTPUT:
Predizione: Positivo, Probabilità: 0.5992962718009949
```

Adesso abbiamo completato il nostro primo modello che elabora il testo.
Qui troverete i file del progetto.
https://drive.google.com/file/d/11uC4WrrUEhMq7Bjbww5ZQ9AHFR3gYlXt/view?usp=sharing