Deep Learning

Daniele Avolio

November 7, 2023

Contents

1 Introduzione **5**

2 Deep Learning 101 **5**
2.1 Architetture e strumenti nel deep learning . . . 5
2.2 Libri utili . . . 5
2.3 Strumenti che useremo . . . 6
2.4 Schema generale di un problema di deep learning . . . 6
2.5 Perché si usa il termine "Tensore"? . . . 6
2.6 AI vs DL . . . 7

3 Introduzione alle Reti Neurali **8**
3.1 Il modello di McCulloch-Pitts . . . 8
3.2 Modello di Rosenblatt . . . 8
3.3 Esempio con 2 neuroni . . . 9
3.4 Rappresentazione le funzioni logiche . . . 10
3.4.1 AND . . . 10
3.4.2 Il problema dello XOR . . . 11
3.5 Gli ingredienti di una rete neurale . . . 12
3.5.1 Il grafo g . . . 12
3.5.2 La funzione di loss . . . 13
3.5.3 Funzione di loss per Regressione . . . 14
3.5.4 Funzione di loss per classificazione . . . 14
3.6 L'ottimizzatore o . . . 14
3.6.1 Il metodo di discesa del gradiente . . . 15
3.6.2 Il metodo di discesa del gradiente stocastico . . . 16
3.6.3 Linee guida sul learning rate . . . 17

4 Classificazione Binaria **19**
4.1 Caricamento e gestione del Dataset . . . 20
4.2 Definizione della Rete Neurale . . . 21
4.3 Plotting del modello . . . 23
4.4 Training del modello e valutazione . . . 24
4.4.1 Validation Set . . . 24
4.5 Prediction . . . 28
4.6 Come risolvere problemi di accuracy bassa? . . . 28
4.7 Early Stopping . . . 29

5 Classificazione Multiclasse **29**
5.1 Descrizione del dataset - Reuters . . . 29
5.1.1 Come processiamo l'output? . . . 29
5.1.2 Softmax activation function . . . 30

6 Regressione **32**
6.1 Boston Housing Price . . . 32
6.1.1 Normalizzare i dati . . . 32
6.1.2 Costruzione della rete . . . 33
6.1.3 Validation con pochi data point . . . 33
6.1.4 Visualizzare i risultati . . . 34
6.2 Overfitting . . . 34
6.2.1 Regolarizzazione . . . 34
6.3 Dropout . . . 36

7 Convolutional Neural Networks **37**
7.1 Il primo problema con le immagini . . . 37
7.2 Il secondo problema . . . 37
7.3 Come fa la rete a riconoscere i pattern . . . 37
7.4 Convoluzione . . . 38
7.5 Padding . . . 39
7.6 Strides . . . 39
7.7 Esempio 1 . . . 39
7.8 Pooling . . . 40
7.8.1 Max Pooling . . . 40
7.9 Multiclass Classification Example . . . 41
7.10 Nozioni alla lavagna . . . 42

8 Reti Neurali Convoluzionali Pre-allenate **43**
8.1 Come si usa il transfer learning? . . . 43

9 Oltre il modello sequenziale **45**
9.1 Multi input e multi output . . . 45
9.2 Functional API . . . 45

10 Adanvced Keras **47**
10.1 Subclassing . . . 47

11 Serie Temporali — Time Series **48**
11.1 Recurrent Neural Network RNN . . . 48
11.1.1 L'utilizzo delle RNN . . . 49
11.2 Long Short Term Memory (LSTM) . . . 51

12 Lab: Introduzione Python **55**
12.1 MatPlotLib . . . 55
12.1.1 Plots . . . 55
12.1.2 Sub-plots . . . 55
12.2 NumPy . . . 56

13 Lab: Reti Neurali da zero **58**
13.1 Introduzione . . . 58
13.2 Esempio pratico . . . 58
13.2.1 Il grafo . . . 59
13.2.2 La funnzione loss . . . 59
13.2.3 L'ottimizzatore . . . 60
13.2.4 Discesa del gradiente . . . 60
13.2.5 Inizializzatore . . . 60
13.2.6 Fix Point Procedure . . . 60
13.3 Esempio da zero . . . 60
13.3.1 La funzione Sigmoid . . . 60
13.4 Tangente Iperbolica TanH . . . 61
13.5 ReLu . . . 62
13.6 Scalare i valori . . . 63
13.7 Plottare la loss . . . 64
13.8 Importante cosa su Gradient Descent . . . 65

14 TensorFlow e Keras **66**
14.1 One Hot Encoding . . . 66
14.2 Variabili correlate . . . 66
14.3 Accuracy come loss . . . 66
14.4 Epoche e batch size . . . 66
14.5 Overfitting e come evitarlo . . . 67
14.6 Migliorare le performance di un modello . . . 67

15 Lab: Convolutional Neural Networks **68**
15.1 Cross Entropy vs Accuracy . . . 68
15.2 Workflow per image classification . . . 69
15.2.1 Bilanciare le classi . . . 69

16 Autoencoders **71**
16.1 Nota su PCA . . . 71
16.2 Autoencoders . . . 71
16.2.1 Gli steps . . . 72
16.2.2 Applicazioni . . . 72
16.3 Autoencoders e Convolution . . . 73

17 VAE (Variational Autoencoder) **74**
17.1 Regularization term e Reparametrization trick . . . 75

18 Recurrent Neural Network e Natural Language Processing (RNN e NLP) **78**
18.1 RNN e LSTM . . . 78
18.2 NLP . . . 80
18.3 Come si lavora con i Word Embedding? . . . 80
18.3.1 Encoder . . . 80

1 Introduzione

Info esame: Tecnicamente, per ora rimane un progetto e potrebbe essere di gruppo. Ancora non ci sono informazioni precisissime.

2 Deep Learning 101

In questo corso affronteremo diverse tematiche, il che può sembrare assurdo se ci si pensa.

- Classificazione (binaria, cioè sì o no)
- Multi-class classification (non più binaria)
- Regressione (il guessing viene fatto su un valore numerico)
- Gestione di immagini e riconoscimento
- Serie numeriche (predizioni di mercato e trend)
- Classificazione di testi

2.1 Architetture e strumenti nel deep learning

- Autoencoder
  - Tutti i possibili tipi
  - Qui si fa anche **Clustering** e **Anomaly detection**
- Architetture generative
  - Tutti i possibili tipi
- XAI: Explainable AI

2.2 Libri utili

- "Deep Learning in Python"
- "Tensorflow tutorial"

2.3 Strumenti che useremo

- Tensorflow
  - High-level più di altri
- Keras
  - High-level API basato su Tensorflow
  - Ci saranno cose che non possiamo fare con Keras perché è troppo ad alto livello

2.4 Schema generale di un problema di deep learning

Abbiamo delle coppie $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ dove x_i è un vettore di features e y_i è un valore numerico (regressione) o una classe (classificazione).

$$y_i = f(x_i)$$

Non conosciamo la funzione f, quindi dobbiamo impararla.

$$y = \alpha x + \beta$$

Con una rete neurale puoi approssimare praticamente qualsiasi funzione.

Una rete neurale permette di collegare un input di dati a una funzione di output.

Abbiamo diversi tipi di reti neurali a seconda del tipo di problema che vogliamo risolvere. È importante essere in grado di selezionare l'architettura giusta per risolvere il problema.

Definiamo alcuni concetti che useremo:

- N: rete neurale
- w: valori dei pesi della rete neurale
- f: funzione di output della rete neurale

$$f \in N(w)$$

2.5 Perché si usa il termine "Tensore"?

Un *tensore* non è altro che una matrice.

- 0D tensor: scalar
- 1D tensor: vector
- 2D tensor: matrix
- 3D tensor: tensor

2.6 AI vs DL

- AI: è un ampio insieme di tecniche per risolvere problemi che richiedono "intelligenza".
  - Esempio: Stockfish, un programma che gioca a scacchi.
- DL: È un sottoinsieme di AI che si concentra sull'*astrazione*.
  - L'astrazione consiste nel fornire una funzione che traduce dati di input in dati di output senza conoscere la funzione stessa.
  - È un approccio induttivo: fornisci un input e ti aspetti un output, senza conoscere la funzione che li collega.
  - Questo è completamente diverso dall'AI basata sulla logica.

3 Introduzione alle Reti Neurali

3.1 Il modello di McCulloch-Pitts

Un modello pensato dai due tizi qui presenti,

- Warren McCulloch
- Walter Pitts

Era formato da:

- Un insieme di neuroni
- Un insieme di connessioni tra i neuroni
- Un insieme di pesi associati alle connessioni
- Una funzione di attivazione
- Una funzione di output

Quindi immaginiamo $x_1, x_2, \ldots, x_n$ che vengono dati come input, e quello che viene fuori è un valore di $y \in \{0, 1\}$. Questo è un modello di classificazione binaria.

In soldoni, in deep learning si usa un vettore di numeri per tirare fuori un altro vettore di numeri.

Nella maggior parte del tempo, però, non lavoriamo solamente con i dati.

- immagini
- Testo
- Audio
- ...

Non ci sono concetti di foto, video, immagini. L'unico concetto che esiste è quello dei numeri.

3.2 Modello di Rosenblatt

Qui il modello è leggermente diverso. Gli elementi in questo modello sono i seguenti:

- Valori di input: x_i

- Funzione di attivazione: ϕ
- Pesi degli archi: w_i
- Bias: b

$$h(x|w,b) = h(\sum_{i=1}^{l} w_i \cdot x_i - b) = h(\sum_{i=1}^{l} w_i \cdot x_i) = sign(w^T x) \tag{1}$$

Nota: Quando il **bias** non viene specificato, allora si assume che sia 0.

I pesi w_i sono collegati archi che vanno da x_i al prossimo neurone. Sia il valore di input che il peso sono **numeri reali**. Non sono lo stesso valore, hanno solamente il formato di *reale* che è uguale tra loro. Ciò che viene fatto è solamente la somma della prodotto tra ogni peso w_i e x_i **meno** il bias.

La funzione di attivazione: Dipende. Ogni funzione che ha *2 stati* va bene per noi. Una funzione di attivazione può essere una qualsiasi che in un punto ha valore 1 e in un altro ha valore -1.

3.3 Esempio con 2 neuroni

.

Immaginiamo di avere un piano cartesiano con una retta che interseca in 2 punti.

$$sign(w_1 \cdot x_1 + w_2 \cdot x_2) \tag{2}$$

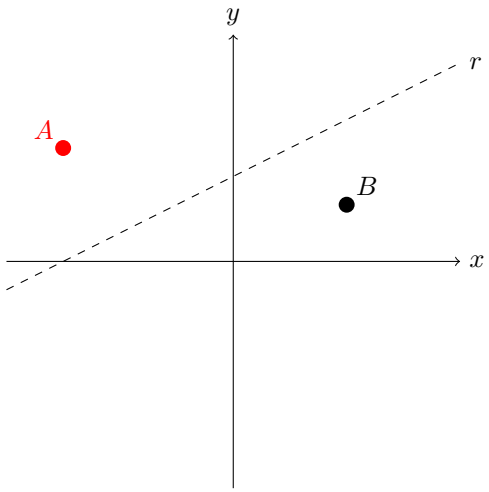
Il parametro della funzione non è altro che **l'equazione di una retta.**

In particolare, se consideriamo la reta che separa gli spazi del piano, vediamo che la retta è **capace di separare 2 punti nel piano.**

Cosa abbiamo:

- Un neurone
- 2 valori in input
- 2 pesi

Con la funzione di attivazione **sign** si avrà come output una retta che separa



dei punti.

Nota: Quando è utile avere un valore output che non è una separazione di punti in un piano? Nel caso della **regressione**.

Oltre la funzione sign:

- Funzione di attivazione lineare
- Funzione di attivazione sigmoid
- Funzione di attivazione Tanh

3.4 Rappresentazione le funzioni logiche

:

3.4.1 AND

Immaginiamo di avere:

- $x_1, x_2 \in \{0, 1\}$
- Bias= -30
- Funzione di attivazione: Logistica o Sigmoid

Come facciamo a rappresentare un AND?

$$h(x) = g(-30 + 20x_1 + 20x_2) \tag{3}$$

x_1	x_2	$h(x)$
0	0	1
0	1	0
1	0	0
1	1	1

Lo stesso ragionamento vale per:

- OR
- NOT
- (NOT x_1) AND (NOT x_2)

3.4.2 Il problema dello XOR

Il percettrone non può imparare regioni che non sono linearmente separabili.

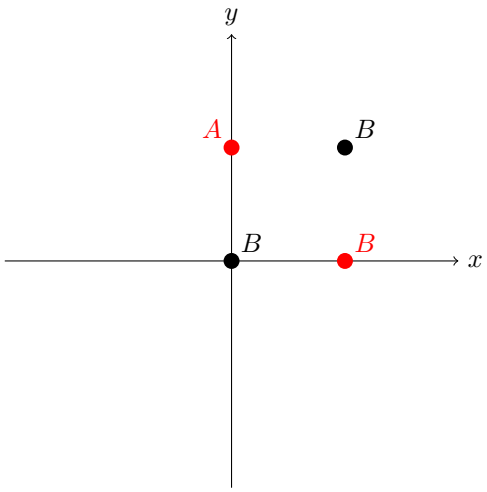


Figure 1: XOR non possibile con 1 percettrone

Come vediamo qui non possoamo tracciare una retta per dividere i due punti. In questo caso ci serve una funzione **non lineare**.

La soluzione a questo problema è **aggiungere LAYER** alla rete neurale. Aggiungere un layer significa aggiungere un neurone successivamente ad un altro.

Maggiore è il numero di layer, maggiore diventa la potenza espressiva della rete. Praticamente possiamo catturare qualsiasi cosa aggiungendo layer alla rete. Questo è **il vero potere del deep learning**.

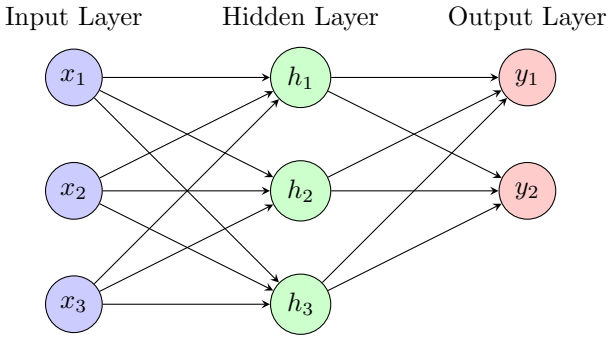


Figure 2: Neural Network con 1 hidden layer

3.5 Gli ingredienti di una rete neurale

3.5.1 Il grafo g

Un grafo $g = \{N, E\}$ è un grafo diretto pesato con label.

Ogni nodo $i \in N$ viene chiamato **neurone** o **percettrone**. Per ogni nodo ci sono 2 targhette:

- Un valore a_i che viene chiamato **attivazione**
- Una funzione di attivazione f_i che viene applicata all'attivazione, che produce un output z_i

Ogni arco $e = \{j \in N \to i \in N\} \in E$ associato con un peso $w_{j,i}$.

Ogni nodo i è anche coinvolto con un arco speciale, con un nodo fantasma, che viene chiamato **bias** b_i.

Nota: $z_i = f_i(a_i)$. E $a_i = b_i + \sum_{j:j\to i\in E} w_{j,i} z_j$

La combinazione dei neuroni connessi costruisce il grafo. I nodi che condividono gli stessi input sono raggruppati in **layers**.

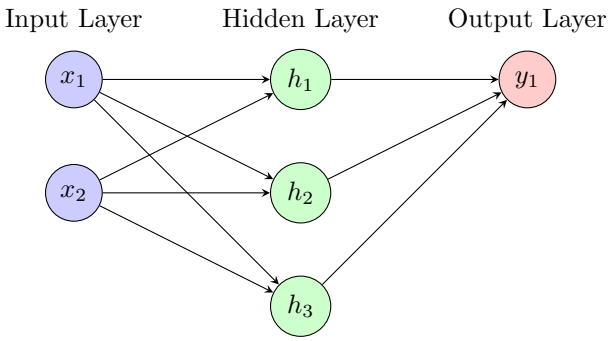


Figure 3: Neural Network con 1 hidden layer

Il risultato finale dipende da tutti i parametri del grafo, e anche dal tipo di funzione di attivazione che viene usata.

> Ma come facciamo a scegliere il corretto valore dei parametri del grafo?

3.5.2 La funzione di loss

Il grafo è un operatore algebrico non lineare: $g(\vec{x}|W, B)$. In questo grafo non sappiamo il valore di **B e W**. La fase di apprendimenti di una rete consiste nel trovare il **migliore** valore per W e per B. Ma cosa intendiamo con **migliore?**.

Formalmente, l'unico modo che abbiamo per definire la conezione di migliore, è quello di *approssimare al meglio la funzione che vogliamo trovare.*

Consideriamo il valore di una funzione F su x_i e il vero valore di y_i. Noi vogliamo minimizzare la differenza tra $F(x_i)$ e y_i.

$$\min_{W,B} \frac{1}{n} \sum_{i=1}^{n} loss(\vec{y_i}, g(x_i|W, B)) \tag{4}$$

con:

- $loss(\vec{y_i}, g(x_i|W, B))$ è una funzione che misura la differenza tra y_i e $g(x_i|W, B)$
- n è il numero di esempi
- W e B sono i parametri del grafo
- $g(x_i|W, B)$ è il valore di output del grafo
- $\vec{y_i}$ è il valore di output vero

La funzione di **loss** dipende dal tipo di task che dobbiamo svolgere.

3.5.3 Funzione di loss per Regressione

Mean Absolute Error:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - g(\vec{x_i}|W,B)| \tag{5}$$

- Considera tutti gli errori con lo stesso peso
- **Non è differenziabile in 0**

Mean Squared Error:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - g(\vec{x_i}|W,B))^2 \tag{6}$$

- Gli errori più grandi hanno un peso maggiore
- **È differenziabile in 0**
- **È più sensibile agli outliers**

Domanda: quale si usa tra le due? Dall'approccio greedy, **usa entrambe**. **Esistono anche altre funzioni:**

- Smooth Absolute Error
- Huber Loss

3.5.4 Funzione di loss per classificazione

Binary Cross Entropy [BCE] Viene usata se $y_i \in \{0,1\}, g(\vec{x}|W,B) \in [0,1]$.

$$BCE = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(g(\vec{x_i}|W,B)) - (1-y_i)\log(1-g(\vec{x_i}|W,B)) \tag{7}$$

Categorical Cross Entropy [CCE]

$$CCE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{i,j}\log(g(\vec{x_i}|W,B))_j \tag{8}$$

3.6 L'ottimizzatore o

Come risolviamo il problema di:

$$\min_{W,B}\frac{1}{n}\sum_{i=1}^{n} loss(\vec{y_i}, g(\vec{x_i}|W,B)) \tag{9}$$

Il problema lo risolviamo calcolando il **gradiante** della funzione loss, lo poniamo uugale a 0 e controlliamo se siamo in un punto di **massimo, minimo o punto di sella**.

$$\frac{\partial loss}{\partial W} = 0 \tag{10}$$

Abbiamo bisogno di un metodo iterativo per trovare una soluzione. Ci vuole un'**euristica**. Tipicamente, siamo soddisfatti di un **minimo locale**, e si utilizza, appunto, il **metodo di discesa del gradiente**.

Differenza minimo locale e globale

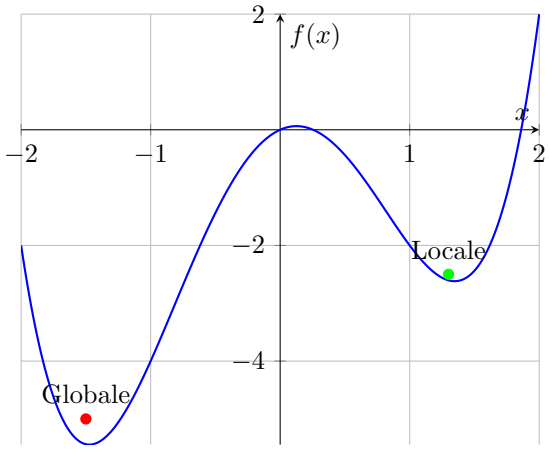


Figure 4: Differenza tra minimo locale e globale

3.6.1 Il metodo di discesa del gradiente

Sia $F(\vec{x})$ una funzione differenziabile.

- $F(\vec{x})$ decresce più veloce nella direzione del gradiente negativo
- $F(\vec{x})$ cresce più veloce nella direzione opposta al gradiente

$$a^{new} = a^{old} - \eta \cdot \nabla F(a^{old}) \tag{11}$$

Il parametro η viene chiamato **learning rate** e determina il comportamento del'ottimizzazione. Da notare che e' l'unico **parametro**.

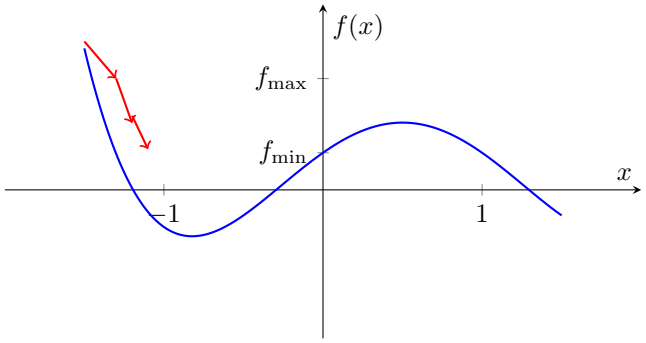


Figure 5: Discesa del gradiente

Ci sono ancora problemi: Questo metodo non è esente da problemi, quindi abbiamo:

- Dipendentemente dal punto di inizio, abbiamo un risultato diverso. Dobbiamo capire che, giustamente, se ci accontentiamo di un minimo locale, non avremo quasi mai lo stesso minimo per ogni allenamento della rete.

Nota: con **topologia** intendiamo il numero di layer e il numero di neuroni per layer.

Diciamo che in generale, gli steps per allenare una rete sono:

```
for each topology T in C_t
    for each η in C_η
        for each initialization I in C_I
            Applica la discesa del gradiente
```

3.6.2 Il metodo di discesa del gradiente stocastico

La differenza del metodo di discesa del gradiente stocastico è che, invece di calcolare il gradiente su tutti i dati, si calcola il gradiente su un sottoinsieme di dati.

Ciò che viene fatto, per ogni step, invece di prendere la derivata di ogni loss function e poi fare i calcoli, prendo **un sample del mio dataset**, tipicamente chiamato **batch**. Ogni volta che calcolo la discesa del gradiente, calcolo la derivata **NON PER TUTTO IL DATASET**, ma solamente del **batch**. Questo OVVIAMENTE non mi assicura che ottimizzare ogni batch mi ottimizza anche l'intero dataset, ma non c'è modo di lavorare sull'intero dasaset, poiché questo è troppo grande e richiede troppo tempo.

Nonostante tutto, utilizzando questa tecnica **si ottengono risultati comunque accettabili**.

Il *workflow* allora cambia in:

```
for each topology T in C_t
    for each η in C_η
```

```
        for each initialization I in C_I
         for each batch size b ∈ C_b
           Applica la discesa del gradiente stocastica
```

3.6.3 Linee guida sul learning rate

Epoca: Un'epoca è un passaggio dell'approssimazione della funzione

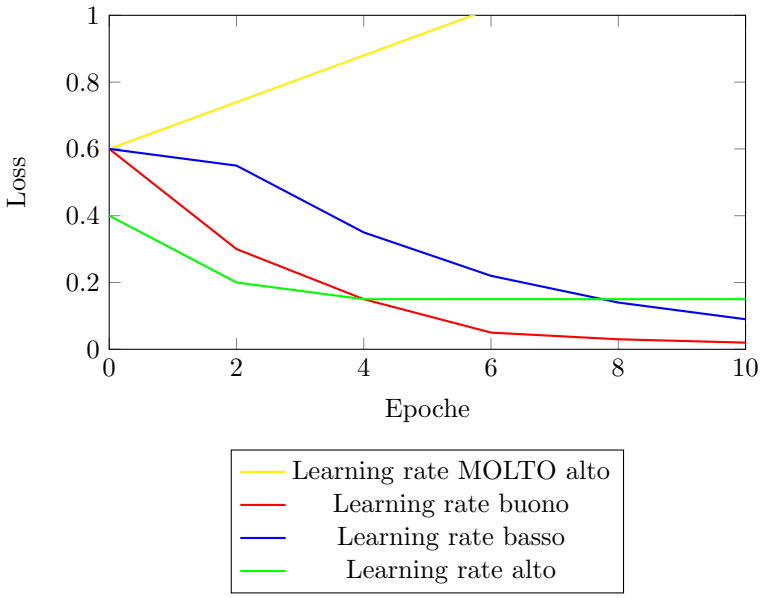


Figure 6: Esempio di andamento della funzione di loss in base al learning rate

Analisi dei learning rate:

- learning rate basso: troppo lento!
- learning rate alto: fermo con un minimo locale immediatamente
- learning rate MOLTO alto: non converge mai
- learning rate buono: decresce sempre e eventualmente converge

Nota: Il learning rate è un parametro che va **tunato**.

- Se il learning rate è troppo alto, non si riesce a convergere poiché si salta il minimo
- Se il learning rate è troppo basso, si rischia di non convergere mai

Il nostro obiettivo è quello di avere un learning rate che sia **giusto** e che permetta di avere un andamento della funzione di loss come quello in rosso, cioé avere una buona diminuizione della loss andando avanti con le epoche. Sempre decrescente e con la distanza minore dall'asse delle x.

Domanda: Vogliamo sempre la loss uguale a 0? **No.** Perché? Perché a differenza che in statistica dove vogliamo avere un errore il minimo. In deep learning non è così. Se la loss è 0, allora vuol dire che la rete ha imparato a memoria i dati, e non è quello che vogliamo. Vogliamo che la rete sia capace di generalizzare.

In particolare, noi **vogliamo fare predizioni**, sopratutto su **istanze ancora non viste.** Se avessimo una loss uguale a 0, probabilmente non saremmo in grado di avere delle predizioni decenti, poiché la rete non è in grado di generalizzare. Avendo la loss non a 0 rendiamo la rete capace di poter introdurre istanze ancora non viste in futuro.

Quindi, per statistica **loss = 0 : nice**, per **deep learning: insomma**.

4 Classificazione Binaria

Descrizione di cosa andremo a fare: utilizziamo il dataset IMDb Dataset. Questo dataset è un Database compilato dagli utenti del sito. Le caratteristiche, in particolare, sono:

- 50.000 recensioni
- circa 50% positive e 50% negative
- *25.000* sono usate per il **training** e *25.000* sono usate per il **testing**. Anche queste sono il 50% positive e 50% negative.

La task che faremo su questo dataset prende il nome di **sentimental analysis**, ovvero analisi del sentimento.

Assunzione: Ciò che stiamo facendo ha senso solamente se assumiamo *che nel futuro avremo punti con una distribuzione abbastanza simile a quelli utilizzati per il training.*

La partizione di **test** è una partizione che non viene utilizzata per la fase di training, ma viene utilizzata successivamente per controllare se il modello si sta comportando bene nella predizione dei valori. Ci sono delle misure che hanno range $[0, 1]$ che ci permettono di capire quanto il modello si sta comportando bene.

Nota: il *test set* non viene fornito quando si lavora nel deep learning, altrimenti verrebbe usato per ottimizzare direttamente il modello. **Non si conosce inizialmente.**

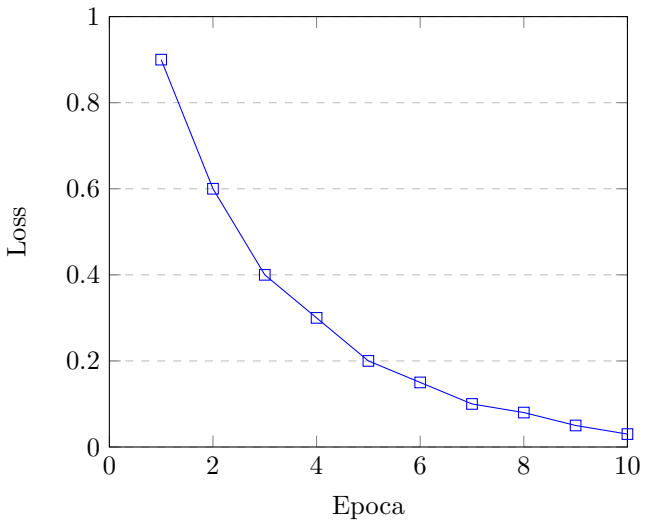


Figure 7: Grafico di una curva di loss che diminuisce all'aumentare delle epoche.

4.1 Caricamento e gestione del Dataset

```
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.
    load_data(num_words=10000)
```

Listing 1: Caricamento del dataset IMDb Dataset.

- training_data: sono gli input del training
- training_labels: sono gli output del training
- test_data: sono gli input del test
- test_labels: sono gli output del test

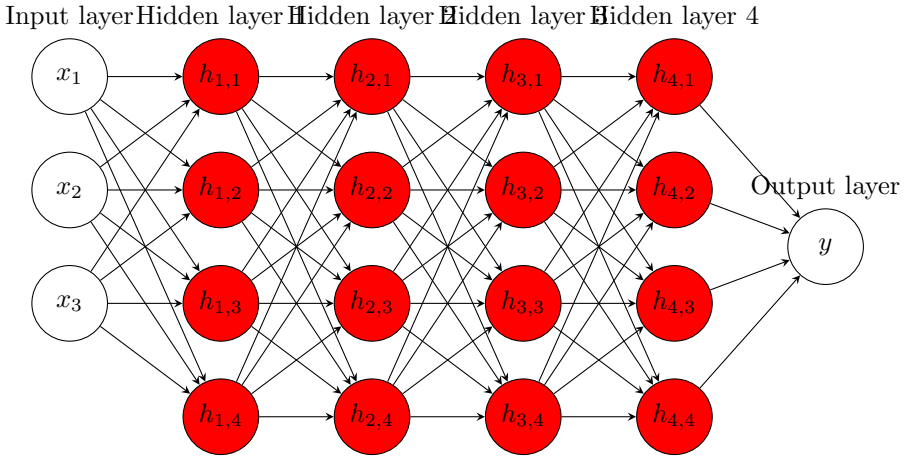


Figure 8: Neural network with 3 input nodes, 4 hidden layers with 3 nodes each, and 1 output layer with 1 node.

Quali sono i primi problemi che vanno gestiti in questo caso? Abbiamo il problema che i **dati sono parole e non numeri**. Quindi dobbiamo trasformare le parole in numeri.

Ogni parola viene **trasformato** in un numero. Ci troviamo però con delle recensioni che sono delle **sequenze di parole**, e quindi abbiamo delle **sequenze di numeri**. La soluzione è quelal di usare un **set di parole codificate.** Mi spiego:

Immaginiamo di avere 10.000 parole encodate. Salvo queste 10.000 parole in un array e associo ad ogni parola un indice di questo array.

0	1	2	3	4	5	6	7	8	9

Ora, cosa manca? **Manca l'ordine**. L'unica cosa che abbiamo è quindi una indicizzazione delle parole, ma non abbiamo alcuna informazione dell'ordine. **Manca anche totalmente la semantica**.

La struttura dell'input è quindi un **livello con 10.000 nodi di input.s**

Input

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
# Create an all-zero matrix of shape (len(sequences), dimension)
results = np.zeros((len(sequences), dimension))
for i, sequence in enumerate(sequences):
    results[i, sequence] = 1. # set specific indices of results[i]
    to 1s
return results
# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

Output

```
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

4.2 Definizione della Rete Neurale

Codice iniziale della definizione della Rete

```
    from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,))
    )
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
               metrics=['accuracy'])
```

Il **modello sequenziale** è quello migliore da dove iniziare. Cosa significa però modello sequenziale? Il modello sequenziale ha il concetto di **layer**. Ha la funzione **add** che aggiunge un layer sopra gli altri layer che sono già esistenti.

Primo Layer:

layers.Dense: un layer denso significa che ogni nodo di quel layer è collegato con ogni nodo del layer precedente. **16** è il numero di neuroni che si vogliono

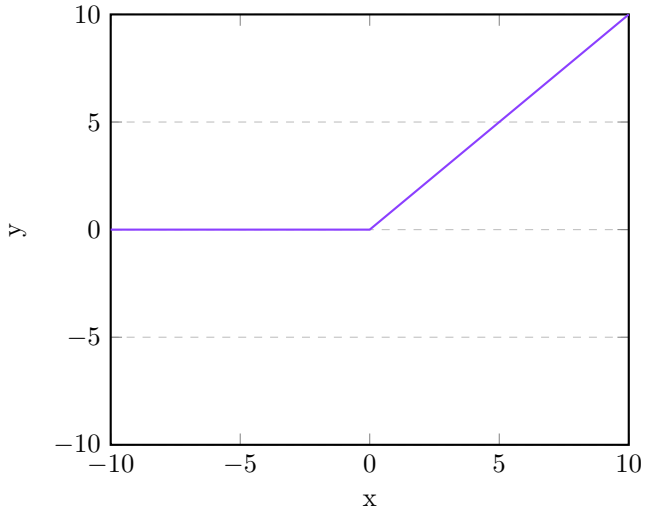
attivare in quel layer. input_shape è la dimensione dell'input. **10000** è la dimensione dell'input. Notare che si ha una **virgola** nell'input dopo il 10.000 e indica che *che stiamo aspettando una sequenza di vettori, ognuno di dimensione 10.000* Cioè, praticamente stiamo dicendo **10.000** sono le parole per ogni recensione, e con la virgola stiamo dicendo che **non sappiamo quante recensioni abbiamo**.E' comodo quando non abbiamo un numero fisso di example del dataset da processare.

Quante connessioni ci sono?

$$16 \cdot 10000 + 16 = 160016$$

dove sono 16 i bias.

Cosa significa relu? ReLu è la funzione di attivazione.



E' una funzione di attivazione artificiale che fino a 0 è 0, e poi cresce linearmente. E' una funzione che si usa molto in deep learning.

Secondo layer:

Nel secondo layer abbiamo altri **16** nodi connessi con i precedenti 16 nodi. In tutto abbiamo

$$16 \cdot 16 + 16 = 272$$

connessioni, con 16 che sono i bias.

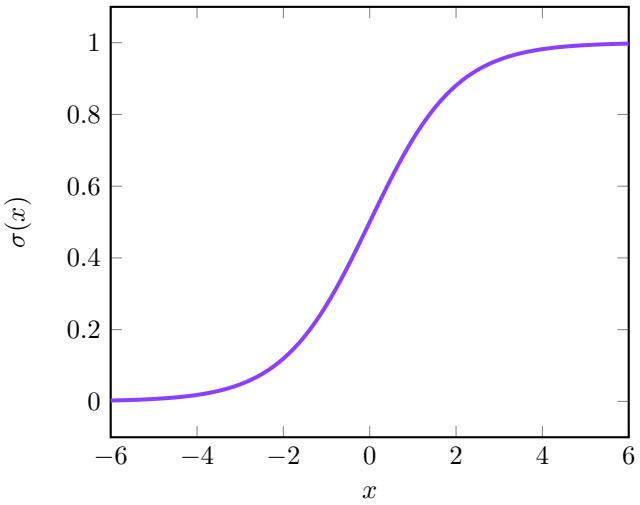
Layer di output:

In questo layer abbiamo un unico nodo connesso con i 16 nodi precedenti. In tutto abbiamo

$$16 \cdot 1 + 1 = 17$$

connessioni, con 1 che è il bias. La funzione di attivazioe è la **sigmoid**, che è una funzione che ha range $[0, 1]$ che indica la probabilità che il dato appartenga alla classe 1.

Diciamo che è stato osservato che la *sigmoid è meglio nell'output.* Perché? Eh funziona cosi a quanto pare lol.



Compilazione del modello

In questa sezione si definisce il **loss** e l'**optimizer**.

L'ottimizzatore sarebbe, praticamente, la *discesa del gradiente*. Ci sono vari ottimizzatori ed è molto buono per te stesso provare gli ottimizzatori e vedere quale funziona meglio per il tuo problema. Letteralmente il deep learning :) Per quanto riguarda la loss abbiamo visto nella scorsa sezione le funzioni di loss che conosciamo.

La **metrica** è quella che viene usata per valutare effettivamente il modello. In questo caso si usa l'**accuracy** che praticamente è la percentuale di classificazioni corrette.

4.3 Plotting del modello

E' un plotting molto base e non molto fancy, ma fa il suo lavoro diciamo

```
from keras.utils import plot_model
plot_model(model, show_shapes=True, show_layer_names=True)
```

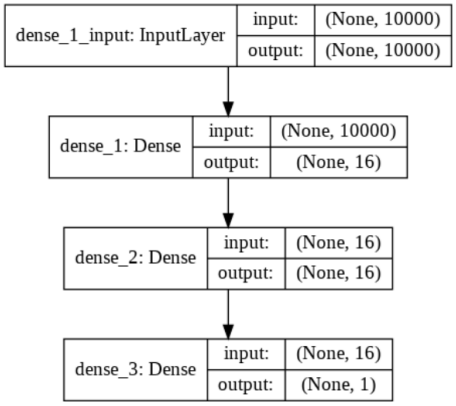


Figure 9: Plotting del modello.

4.4 Training del modello e valutazione

Entra in gioco il concetto di **validation set**.

4.4.1 Validation Set

Quando ho un modello, come faccio a sapere se la rete che sto allenando si sta allenando in modo corretto? Tutto questo sapendo che **non abbiamo accesso al test set**. Pensiamo al **training set** e pensiamo ad un altro **split** interno al training set:

- Training set parziale
- Validation set

Quindi se avessimo 50.000 di grandezza del dataset:

- 25.000 test set
- 25.training set
  - 10.000 validation set
  - 15.000 training set parziale

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Praticamente è come se usassimo il validation come se fosse un test set. Quindi andremo a plottare 2 curve:

- La loss sulle epoche
- L'accuracy sulle epoche

La loss ci da un'idea di quanto il modello si sta allenando bene. Se ci sono problemi, la figura è strana e non segue un andamento corretto, si modifica. Ma la cosa importante è la **validation accuracy**, che DEVE essere crescere in modo monotono e deve avvicinarsi a 1 il più possibile.

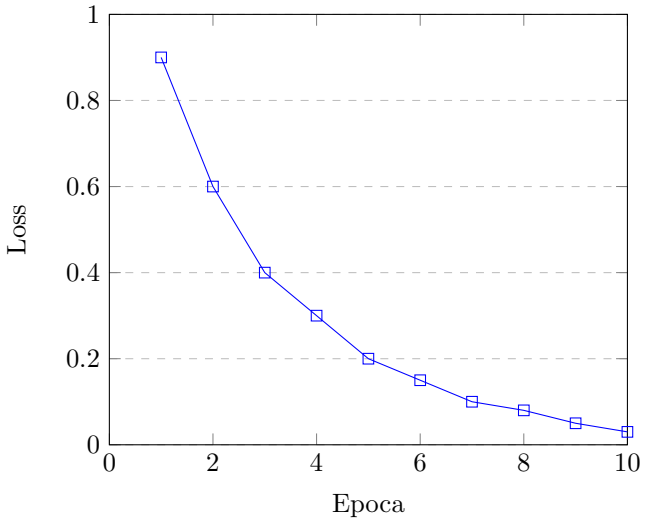


Figure 10: Grafico di una curva di loss che diminuisce all'aumentare delle epoche.

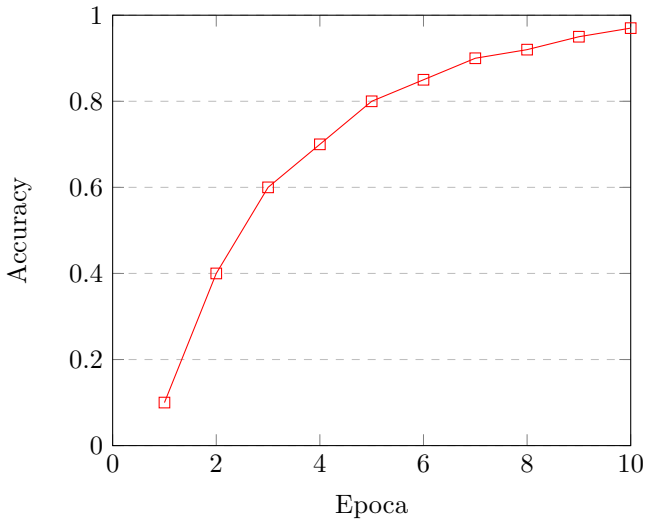


Figure 11: Grafico di una curva di accuracy che aumenta all'aumentare delle epoche.

Ma anche se avessimo queste curve inìcon questa conformazione, **ancora non ci dice niente**. Il validation è in un senso insensato per l'output. Ciò che importerà sarà il **test set** che non è stato usato per il training.

Training code

```
history = model.fit(partial_x_train,
            partial_y_train,
            epochs=20,
            batch_size=512,
            validation_data=(x_val, y_val))
```

Plotting delle curve

Notare che questo è un plotting molto base e non molto fancy, e in futuro utilizzeremo **Tensorboard** che aggiornerà in tempo reale i grafici

```
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Notare che *history* è un dizionario e si può accedere come indicato a riga *1 e 2* come accedere a loss e val_loss.

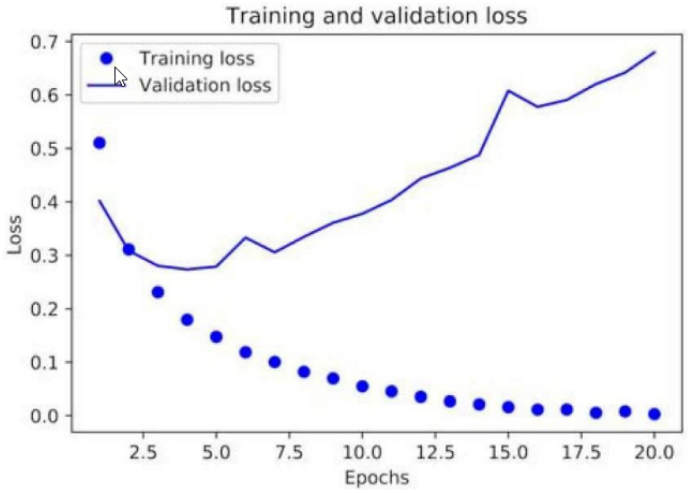


Figure 12: Rete sbagliata

Un grafico del genere ci mostra una rete che **funziona male**!.

```
plt.clf() # clear figure
acc = history_dict['binary_accuracy']
val_acc = history_dict['val_binary_accuracy']
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

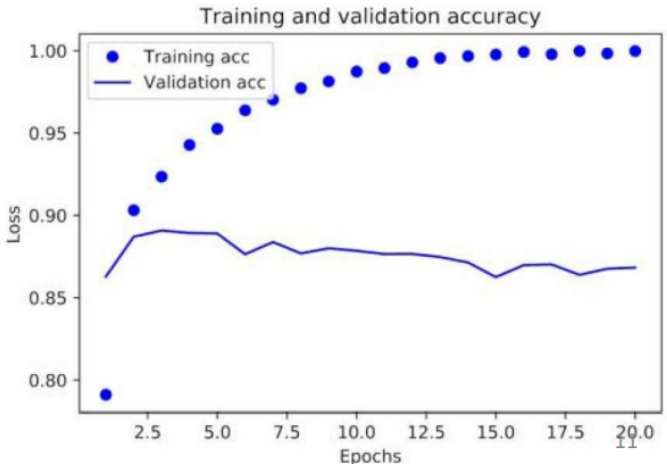


Figure 13: Overfitting

In questo caso la rete ha un problema di **overfitting!**.

4.5 Prediction

```
model.predict(x_test)
```

$array([[0.91966152], [0.86563045], [0.99936908], ..., [0.45731062], [0.0038014], [0.79525089]], dtype = float32$

4.6 Come risolvere problemi di accuracy bassa?

Se quando andiamo a fare la validazione della nostra rete, come sappiamo come modificare qualcosa per rendere la rete migliore? Ci sono alcuni approcci per farlo:

- Cambiare la topologia
- Diminuire le epoche
- Cambiare il numero di nodi per qualche layer
- FORSE cambiare anche il learning rate

Si noti che facendo questo vanno interpretati i grafici per capire come sta andando la rete. Quando cominciamo a vedere che la **validation accuracy NON STA SOLAMENTE CRESCENDO** ma ci sono dei punti in cui diminuisce, siamo sicuri che **c'è qualche problema di mezzo.** Bisogna capire anche su cosa lavorare. Se cambiando i dati spesso la **validation accuracy** mostra problemi, allora bisogna lavorare per quello e cercare di trovare un modo per fare in modo che il problema non si presenti.

RECAPPONE: Se notiamo che nella LOSS ci sono problemi, possiamo tunare il **learning rate** e altri parametri per cercare di risolvere questi problemi. Ma la cosa più importante è **la metrica**. Se notiamo che la **metrica non rispecchia un andamento di effettivo apprendimento**, capiamo che la rete non si sta comportando nel modo corretto e non sta funzionando.

4.7 Early Stopping

E' un metodo che consiste nel vedere quando **la loss** comincia ad avere dei comportamenti sospetti. Quando si **ferma** la rete in un dato momento, si impedisce alla rete di **overfittare** nella maggior parte dei casi.

5 Classificazione Multiclasse

Questa sezione sarà molto corta, poiché praticamente è la stessa cosa della classificazione binaria, ma il layer finale della rete non sarà composto da un solo nodo, ma da **un numero di nodi pari al numero di classi da classificare**.

5.1 Descrizione del dataset - Reuters

Il dataset che verrà utilizzato è il **Reuters Dataset**, che è un dataset di **news** che sono state classificate in **46 categorie**. Ogni categoria ha almeno 10 esempi nel training set.

Preprocessing dell'input

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

5.1.1 Come processiamo l'output?

Nel deep learning, **gli output categorici** non vengono mai mappati ad una scala numerica. Si utilizza una tecnica che si chiama **one hot encoding**.

One Hot Encoding:

Consiste nel creare tante variabili numeriche quanti sono i valori della variabile categorica, e per quel dato example viene assegnato

- 1: se il valore della variabile categorica è quello

- 0: se il valore della variabile categorica non è quello, quindi in tutti gli altri casi

Da notare che il one hot encoding viene fatto sia per le labels di output, ma non è sbagliato farlo anche per gli attributi in alcuni casi.

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
# Our vectorized training labels
one_hot_train_labels = to_one_hot(train_labels)
# Our vectorized test labels
one_hot_test_labels = to_one_hot(test_labels)

#OPPURE ALLO STESSO MODO

from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Nota: nel one hot encoding DOBBIAMO AVERE 1 SOLO VALORE per il dato example, e tutti gli altri non devono essere attivi. La domanda è, quindi, **quale activation function usiamo?**

Immaginiamo questa situazione: Abbiamo y_1, y_2, y_3 che sono i valori di output. Vogliamo solamente uno dei tre. Se normalizzassimo i dati in questo modo:

- $p_1 = \frac{y_1}{y_1+y_2+y_3}$
- $p_2 = \frac{y_2}{y_1+y_2+y_3}$
- $p_3 = \frac{y_3}{y_1+y_2+y_3}$

E se sommiamo $p_1 + p_2 + p_3$ otteniamo 1. Quindi, in questo caso, possiamo usare la **softmax** come activation function.

5.1.2 Softmax activation function

Se prendiamo un vettore $\vec{y} = (y_1, y_2, y_3)$, la softmax è definita come:

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{3} e^{y_j}} \tag{12}$$

In output abbiamo: $\vec{p} = (p_1, p_2, p_3)$, dove p_i è la probabilità che il dato example appartenga alla classe i.

Nota: quando dividiamo qualcosa, abbiamo **sempre** un problema riguardo la stabilità numerica. La divisione è molto critica, poiché **potrebbe essere vicina a 0**. Sappiamo che se usiamo $e_1^y + e_2^y + e_3^y$ difficilmente si avvicina a 0.

```
    from keras import models
    from keras import layers
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu', input_shape
    =(10000,)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(46, activation='softmax'))
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

Nota: Dobbiamo avere un numero di nodi di output quanto il numero di classi, ma **altra cosa**, il numero di nodi interni deve essere sicuramente maggiore di 46, altrimenti ci sarebbe una perdita di informazioni.

Nota 2: La funzione di attivazione dell'ultimo layer è una **softmax**.

Validation set e Training set

```

x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Loss

```
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Accuracy

```
plt.clf() # clear figure
acc = history.history['acc']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Acc')
plt.legend()
plt.show()
```

Nota: L'accuracy è importante dipendentemente dall'utilizzo che bisogna farne. Ad esempio, *in ambito medico* vogliamo **minimizzare i falsi negativi**. Questo implica che la misura che usiamo dipende dall'applicazione che se ne fa.

6 Regressione

Parlando di **regressione**, si ha un problema impostato allo stesso modo di quelli di classificazione binaria o multiclasse, ma l'*output* è un **nuemro reale**.

Nota: quando all'interno della nostra rete abbiamo dei valori categorici, sappiamo che per gestirli utilizziamo la tecnica del **one hot encoding**. Ma quando abbiamo valori numerici, come gestiamo questi dati? Potremmo avere dati che hanno *scale diverse, unità di misura diverse*, ecc.... Questi rendono abbastanza complicato il lavoro della rete. Per questo motivo, è necessario **normalizzare** i dati.

6.1 Boston Housing Price

Il dataset **Boston Housing Price** è un dataset che contiene 506 esempi di case nella zona di Boston. Ogni esempio è composto da 13 *feature* che descrivono la casa e il prezzo della casa. Questo dataset è stato utilizzato per la prima volta nel 1978, ma è ancora utilizzato per testare i modelli di regressione, ed è proprio quello che faremo noi.

```
1  from keras.datasets import boston_housing
2  (train_data, train_targets), (test_data, test_targets) =
   boston_housing.load_data()
```

Listing 2: Caricamento del dataset

6.1.1 Normalizzare i dati

La normalizzazione consiste nel portare i valori numerici di un dataset tutti sulla stessa scala. Abbiamo due modi per fare questo processo di normalizzazione:

- MinMax Normalization: La MinMax Normalization è una tecnica di normalizzazione dei dati che consiste nel portare tutti i valori di un dataset su una scala compresa tra 0 e 1. Questa tecnica è utile quando i dati hanno scale diverse e si vuole portarli tutti sulla stessa scala per facilitare l'elaborazione da parte della rete neurale. Per applicare la MinMax Normalization, si utilizza la seguente formula per ogni valore del dataset: Viene usata ma **non è proprio approrpiata.**

$$x_{norm} = \frac{(x - x_{min})}{(x_{max} - x_{min})} \tag{13}$$

- Normalizzazione Statistica: Si utilizzano la **media** e la **deviazione standard**. Questa tecnica è utile quando i dati hanno scale diverse e si vuole portarli tutti sulla stessa scala per facilitare l'elaborazione da parte della rete neurale.

```
1          mean = train_data.mean(axis=0)
2          train_data -= mean
3          std = train_data.std(axis=0)
4          train_data /= std
5
6          test_data -= mean
7          test_data /= std
8
```

Questo porta un **centramento** intorno allo 0. **Nota importante:** Da notare le righe *6 e 7* che applicano la normalizzazione dei dati **anche sul test set**. Questa cosa si fa? La risposta è **NON ABBIAMO MAI I TEST DATA.**

Praticamente stiamo facendo l'assunzione che i dati vengano dalla **stessa distribuzione**. Poiché i risultati del test set probabilmente saranno su una scala diversa rispetto a quelli che abbiamo dal training set e i risultati che otteniamo dal nostro modello potrebbero essere su una scala diversa rispetto a quella del training. Quindi, questo va fatto solo se si ha la certezza che i dati provengano dalla stessa distribuzione.

6.1.2 Costruzione della rete

```
1  def build_model():
2      # Because we will need to instantiate
3      # the same model multiple times,
4      # we use a function to construct it.
5      model = models.Sequential()
6      model.add(layers.Dense(64, activation='relu',
7                  input_shape=(train_data.shape[1],)))
8      model.add(layers.Dense(64, activation='relu'))
9      model.add(layers.Dense(1))
10     model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
11     return model
```

Piccole note su questo codice:

1. L'ultimo livello ha un solo nodo di output
2. Non ha una funzione di attivazione
3. Sta usando la metrica **MAE**: Mean Absolute Error e la loss **MSE**: Mean Squared Error

6.1.3 Validation con pochi data point

Spiegazione al volo della K-Fold validation: La K-Fold validation è una tecnica di validazione che consiste nel dividere il dataset in **K parti** e utilizzare una di queste parti come **validation set** e le altre come **training set**.

6.1.4 Visualizzare i risultati

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(
    num_epochs)]

import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1),
    average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

Nota: Solitamente i primi punti in una task di regressione **sono fuori scala**. Conviene scartarli e non contarli nella big picture finale.

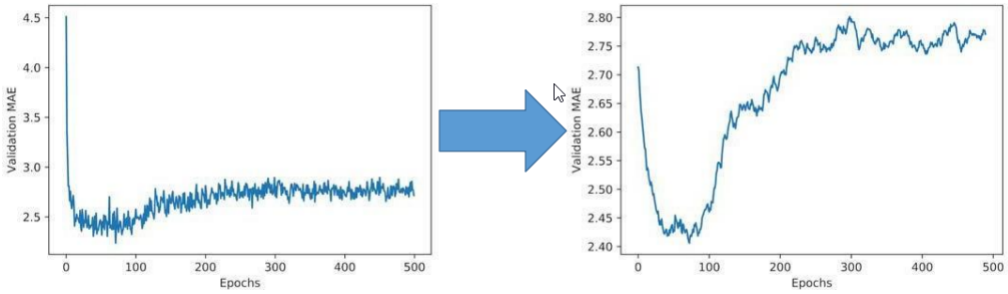


Figure 14: Grafico MAE

6.2 Overfitting

Formalizziamo al volo il problema:

$$\min_{w}[loss(y, NN(x|w))] \tag{14}$$

Cioe vogliamo minimizzare in funzione dei pesi la funzione che ha come parametri la **vera y** e la **y predictata** usando il nostro input **x** e i pesi **w**.

La tecnica che vediamo è la **regolarizzazione**.

6.2.1 Regolarizzazione

La regolarizzazione è una tecnica che non fa altro che **modificare la funzione che vogliamo approssimare**. Questo rende i pesi della rete più piccoli, il che rende la distribuzione dei valori *più regolare.*

$$\min_{w}[loss(y, NN(x|w)) + R(w)] \tag{15}$$

Abbiamo due tipi di regolarizzazione:

- L1 Regularization: *vecchia loss function* $+ \lambda \sum_i |w_i|$, cioè aggiunge la somma dei valori assoluti dei pesi.
- L2 Regularization: *vecchia loss function* $+ \lambda \sum_i w_i^2$, cioè aggiunge la somma dei valori dei pesi al quadrato.

Queste due tecniche rendono *il modello più sensibile la noise e alla varianza dei dati.*

- L2: Rende i pesi più piccoli
- L1: Rende i pesi più sparsi (più 0, in pratica)

```
from keras import regularizers
l2_model = models.Sequential()
l2_model.add(layers.Dense(8, kernel_regularizer=regularizers.l2
    (0.001),
    activation='relu'
    , input_shape=(10000,)))
l2_model.add(layers.Dense(8, kernel_regularizer=regularizers.l2
    (0.001),
            activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))
```

Nota: si può fare una combinazione tra L1 e L2.

```
from keras import regularizers
# L1 regularization
regularizers.l1(0.001)
# L1 and L2 regularization at the same time
regularizers.l1_l2(l1=0.001, l2=0.001)
```

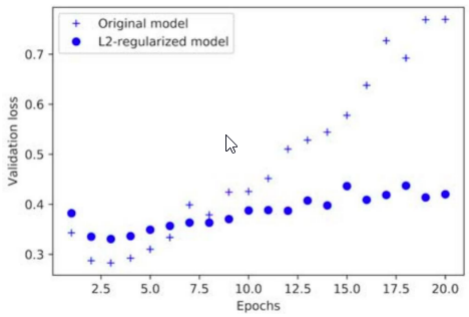


Figure 15: Grafico Regularization L2

Quindi, alla fine, **early stopping** e **diminuire grandezza rete** non vengono fatti con leggerezza e non sono spesso soluzioni applicabili.

6.3 Dropout

In teoria, questa tecnica prevede il rimuovere **durante il training** alcuni nodi della rete in modo casuale seguendo una specifica distribuzione (si setta il valore di 0). L'effetto è quello di rendere la rete più confusa, e quindi più robusta. **Quando si fa il testing** si va ad utilizzare l'intera rete senza troncamenti di connessioni.

Per quale motivo funziona? La rispota è **BOH**. Il fatto è che **funziona** ed è uno standard. Quindi praticamente in ogni NN si utilizza la tecnica del *dropout*.

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape
    =(10000,)))
#Probabilita di rendere 0 un nodo del layer con probabilita 0.5
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))
dpt_model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['acc'])
```

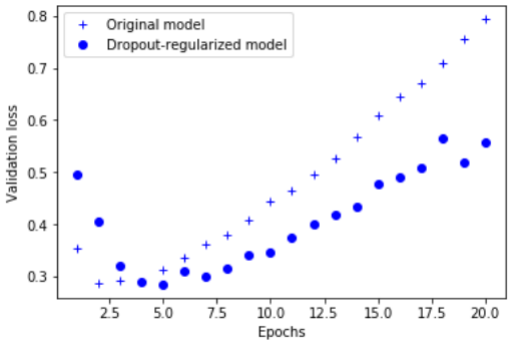


Figure 16: Grafico Dropout

7 Convolutional Neural Networks

7.1 Il primo problema con le immagini

Le reti neurali convoluzionali vengono utilizzate e sono diventate famose per l'elaborazione di immagini. Quando parliamo di *fully connected networks* parliamo di reti dove ogni neurone è connesso con tutti i neuroni del layer successivo. Questo significa che se abbiamo un'immagine di **100x100 pixel**, avremo **10000 neuroni** nel layer di input.

$$\sum_{i=1}^{k} d_h \cdot d_{h-1} \tag{16}$$

per una rete con k layer e dimensioni d per ogni layer h.

Questo numero di parametri è *eccessivamente elevato*. Il numero di connessioni raggiungerebbe un magnitudo altissimo contando le connessioni con i layer successivi.

7.2 Il secondo problema

Spatial pattern e spatial hierarchies. (Da finire non ho capito).

7.3 Come fa la rete a riconoscere i pattern

La rete per riconoscere le immagini non può analizzare in totale l'immagine, sarebbe eccessivamente costoso per i motivi precedenti.

L'intuizione: riconoscere pattern all'interno dell'immagine di **dimensioni ridotte**. Quando intendiamo dimensioni ridotte, intendiamo anche **3x3** pixels. Questo è il motivo per cui le reti convoluzionali sono chiamate **convoluzionali**: perché utilizzano la convoluzione per riconoscere i pattern.

L'algoritmo ad alto livello è il seguente:

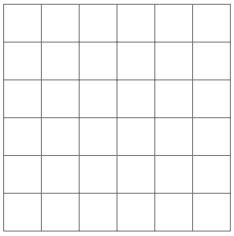
1. Prendere un'immagine
2. Analizzare piccole porzioni dell'immagine
3. Costruire elementi dell'immagine trovando elementi che si ripetono (pattern)
4. Astrarre i pattern combinandoli tra loro
5. Riconoscere l'immagine

La nozione di **astrazione** e **connessione spaziale** è fondamentale ed è l'intuizione dietro le reti convoluzionali.

7.4 Convoluzione

Parliamo dell'idea dietro la convoluzione. Partiamo dagli elementi fondamentali.

- Immagine in scala di grigiù
- Matrice di convoluzione **(kernel)**: Gli elementi all'interno della matrice sono **parametri**



Nota: nelle reti neurali convoluzionali non citiamo mai i neuroni.
Obiettivo: Creare una nuova immagine.
Ma come si applica? Come funziona la convoluzione?
Dobbiamo effettuare una moltiplicazione tra una sotto-matrice dell'immagine principale e la matrice di convoluzoine (cioè il kernel). Ciò che ne viene fuori è un **unico punto**.

a	b	c	d
e	f	g	j
i	j	k	l

w1	w2
w3	w4

Per calcolare il primo punto con la convoluzione avremmo:

$$aw_1 + bw_2 + ew_3 fw_4 \tag{17}$$

Questo per ogni **shift** del kernel nella matrice dell'immagine, andando a calcolare i prossimi elementi:

$$bw_1 + cw_2 + fw_3 + gw_4 \tag{18}$$

E continuando.
Qual è il numero di parametri?
Nota: Se nella feature map **troviamo un pattern**, ad esempio un quadrato 2x2, allora **sicuramente** questo sarà anche un **pattern più grande nella matrice principale**, poiché

7.5 Padding

Il "Padding" è una tecnica utilizzata nell'ambito del Machine Learning, in particolare nell'elaborazione delle immagini. Quando si applica una convoluzione utilizzando una "feature map" 3x3 a un'immagine, essa tende a restringersi di 2 pixel lungo ciascuna dimensione. Questo può comportare la perdita di informazioni ai bordi dell'immagine.

Per evitare questo effetto di restringimento, è possibile utilizzare il "padding". Il padding consiste nell'aggiungere una cornice esterna all'immagine con una larghezza e un'altezza adeguate. In altre parole, si aggiungono pixel extra intorno all'immagine prima di eseguire la convoluzione.

L'obiettivo del padding è preservare le dimensioni originali dell'immagine durante la convoluzione, consentendo così di mantenere tutte le informazioni presenti ai bordi. Questo è particolarmente importante in applicazioni di Machine Learning che coinvolgono il rilevamento di bordi, oggetti o caratteristiche chiave nelle immagini.

7.6 Strides

Il concetto di **stride** nelle reti neurali convoluzionali (CNN) è fondamentale per comprendere come i filtri scorrono attraverso i dati di input. Il *stride* rappresenta la distanza tra le posizioni in cui il filtro viene applicato. Un **stride** maggiore porta a una riduzione della dimensione dell'output, poiché il filtro salta più spazio alla volta. D'altro canto, uno *stride* minore comporta una maggiore sovrapposizione tra le regioni in cui il filtro viene applicato, mantenendo una dimensione di output più grande.

Comunque **riduci il numero di feature maps**.

Nota generale: Si possono usare diversi **kernels** e ogni kernel darà come risultato una **nuova feature map**.

7.7 Esempio 1

Immaginiamo di avere un input 32×32 come immagine.

- Si usano 6 kernel
- Danno come risultato 6 feature maps 28x28
- Si sono persi 4 pixels sia in altezza che larghezza
- Il kernel era 5x5
- Il numero di parametri era $6 \cdot (5x5)$

Dopo questo si utilizza una tecnica che si chiama Pooling

7.8 Pooling

Il "Pooling" è una tecnica ampiamente utilizzata nell'ambito dell'apprendimento automatico, specialmente nell'elaborazione delle immagini. Il suo obiettivo principale è ridurre la dimensione dei dati mantenendo le caratteristiche più importanti.

Il Pooling è un'operazione che riduce la dimensione di una rappresentazione mantenendo le caratteristiche più rilevanti. Questo processo è spesso utilizzato dopo la fase di convoluzione in una rete neurale convoluzionale (CNN).

7.8.1 Max Pooling

Il Max Pooling suddivide una regione in celle e restituisce il valore massimo all'interno di ciascuna cella. Questo aiuta a conservare le caratteristiche più rilevanti, come i bordi o le texture.

Il Pooling ha diversi vantaggi:

- Riduzione della dimensione dei dati, consentendo un addestramento più veloce.
- Riduzione del rischio di overfitting, poiché si mantengono solo le caratteristiche più importanti.
- Maggiore invarianza spaziale, poiché il Pooling considera la presenza delle caratteristiche, indipendentemente dalla loro posizione esatta nell'immagine.

Il Pooling è una tappa fondamentale in molte architetture di reti neurali convoluzionali. Viene utilizzato per creare rappresentazioni più compatte e significative dei dati, consentendo alle reti neurali di apprendere con successo le caratteristiche più importanti delle immagini o dei dati in ingresso.

- Si applica il pooling sulle feature map di prima con una matrice 2×2
- Si passa da 28×28 a 14×14

E si ripete il processo di applicare la convoluzione sulle 6 feature map 14×14, portano ad avere 16 feature map 10×10. Si riapplica ancora il pooling e si avranno 16 feature map 5×5.

Alla fine si unisce tutto quanto insieme e si ha una **rete neurale standard**.

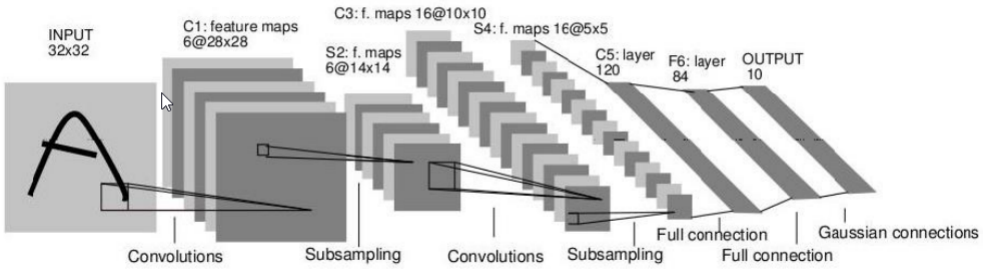


Figure 17: Esempio di rete neurale convoluzionale

7.9 Multiclass Classification Example

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models, optimizers
# CIFAR_10 is a set of 60K images 32x32 pixels on 3 channels
IMG_CHANNELS = 3
IMG_ROWS = 32
IMG_COLS = 32
#constant
BATCH_SIZE = 128
EPOCHS = 20
CLASSES = 10
VALIDATION_SPLIT = 0.2
OPTIM = tf.keras.optimizers.RMSprop()

#define the convnet
def build(input_shape, classes):
    model = models.Sequential()

    #32 e' il numero di kernel, 3x3 e' la grandezza del kernel
    model.add(layers.Convolution2D(32, (3, 3), activation='relu'
    , padding='valid', input_shape=input_shape))

    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Dropout(0.25))

    #Flatten e' necessario per passare da 3D a 1D (Diventa un
    vettore)
    model.add(layers.Flatten())

    #Qui diventa una rete neurale standard
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.Dropout(0.5))

    #Qui abbiamo tanti nodi di output quante sono le classi del
    dataset
```

```
33      model.add(layers.Dense(classes, activation='softmax'))
34      return model
35
36  # data: shuffled and split between train and test sets
37  (X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()
38  # normalize
39  X_train, X_test = X_train / 255.0, X_test / 255.0
40  # convert to categorical
41  # convert class vectors to binary class matrices
42  y_train = tf.keras.utils.to_categorical(y_train, CLASSES)
43  y_test = tf.keras.utils.to_categorical(y_test, CLASSES)
44  model=build((IMG_ROWS, IMG_COLS, IMG_CHANNELS), CLASSES)
45  model.summary()
46
47  # use TensorBoard,
48  callbacks = [
49      # Write TensorBoard logs to `./logs` directory
50      tf.keras.callbacks.TensorBoard(log_dir='./logs')
51  ]
52  # train
53  model.compile(loss='categorical_crossentropy', optimizer=OPTIM,
        metrics=['accuracy'])
54
55  model.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
56  validation_split=VALIDATION_SPLIT, verbose=VERBOSE, callbacks=
        callbacks)
57
58  score = model.evaluate(X_test, y_test,
59  batch_size=BATCH_SIZE, verbose=VERBOSE)
60  print("\nTest score:", score[0])
61  print('Test accuracy:', score[1])
```

7.10 Nozioni alla lavagna

Quante immagini ci servono per trainare una NN?

Prendiamo il concetto di posizione dell'oggetto nell'immagine, se volessimo parlare di *object recognition*. Le reti neurali dovrebbero essere *immuni* al concetto di traslazione. La posizione, quindi, non dovrebbe influire molto sul risultato finale. Per quanto riguarda **la scala** dell'oggetto, qui è diverso. Conviene avere diverse immagini dello stesso oggetto con dimensioni diverse, poiché, in questo caso, influirebbe sul risultato finale della classificazione. Anche per quanto riguarda la **rotazione**, si ha lo stesso discorso della scala. Anche in quest caso conviene avere più immagini dello stesso oggetto con **rotazioni diverse**.

> Per ogni immagine, inserire altre immagini con scala e rotazione diversa dell'oggetto.

Esempio del mondo reale: I segnali stadali di velocità. Per poterli riconoscere c'è la necessità di avere diverse immagini a divers a distanza per avere un allenamento corretto della rete.

8 Reti Neurali Convoluzionali Pre-allenate

Motivazione principale: E' quello di ridurre il modello e riutilizzarlo in altri contesti, senza dover effettuare ancora un allenamento della rete.

Prendiamo l'esempio di **VCG-16** che è una rete allenata su un dataset di immagini di una vecchia competizione. La cosa da capire è che quando vogliamo utilizzare la rete **bisogna preparare l'input** per essere inserito nella rete. Ad esempio, le immagini sono in $224x224$.

```
# prebuild model with pre-trained weights on imagenet
model = VGG16(weights='imagenet'
, include_top=True)
model.compile(optimizer='sgd'
, loss='categorical_crossentropy')

# resize into VGG16 trained images' format
# add a dummy iniziatl axis
im = cv2.resize(img, (224, 224))
plt.imshow(im)
im = np.expand_dims(im, axis=0)
im.astype(np.float32)
print(im.shape)
```

Domanda 8.1. *(Come usiamo la rete?)*

Se volessimo, ad esempio, classificare *modelli di laptop*, questa rete può tornare utile nella classificazione? Come facciamo? La risposta **non è banale.**

La risposta è **si**. Il nucleo della questione è quello di utilizzare **il primo layer** della rete già allenata. Questa è l'idea principale del **transfer learning**. La motivazione è quella che i livelli dopo i primi sono troppo specifici per il nostro scopo, mentre i primi layer sono più generici e sono quellli che si occupano di estrarre le features.

8.1 Come si usa il transfer learning?

Definizione 8.1. *(Feature extraction)*

Il concetto principale è quello di **utilizzare i primi layer** della rete, prendendo solamente l'output che si ottiene ,circa, a metà della rete. Questo significa che per ogni immagine del nostro dataset si estraggono le **features** e le si salvano, creando cosi un dataset alternativo formato dalle features.

Successivamente si utilizzano queste feature nel quale viene fatto l'allenamento, e quindi il task sarà quello di **classificare le features** nella task che ci interessa.

Definizione 8.2. *(Fine Tuning)*

Il concetto di **fine tuning** è quello di utilizzare i primi layer della rete regolarmente, come se fossero sempre funzionanti, e successivamente utilizzare una nuova rete collegata con gli ultimi nodi utilizzati fino a quel punto e allenarla con il nostro dataset. In questo caso diciamo che **congeliamo** i primi layer della rete.

Per unire un modello con un altro, in Keras è molto semplice perché si possono trattare i modelli come layer.

```
1
2     model = VGG16(weights='imagenet', include_top=True)
3
4     new_model = Sequential()
5     new_model.add(model)
6     new_model.add(Dense(2, activation='softmax'))
7
8     new_model.compile(optimizer='sgd', loss='
      categorical_crossentropy')
```

In Keras, un modello può essere congelato impostando il parametro **trainable** su **False**.

```
1     model.trainable = False
```

9 Oltre il modello sequenziale

Il modello sequenziale in Keras è un'implementazione semplice e intuitiva di una rete neurale, ma ha dei limiti in termini di flessibilità e complessità. In particolare, il modello sequenziale è limitato a reti neurali feedforward, ovvero reti neurali in cui l'informazione fluisce in una sola direzione, senza cicli o connessioni ricorrenti.

Ci sono molti problemi di deep learning che richiedono reti neurali più complesse, come ad esempio le reti neurali convoluzionali per l'elaborazione di immagini o le reti neurali ricorrenti per l'elaborazione di sequenze. In questi casi, il modello sequenziale può essere troppo limitato per modellare efficacemente i dati.

Inoltre, il modello sequenziale non supporta la condivisione di pesi tra i layer, il che può essere un'importante tecnica di regolarizzazione e può aiutare a ridurre il numero di parametri del modello. Infine, il modello sequenziale non supporta la definizione di grafi di calcolo arbitrari, il che può essere necessario per alcune applicazioni avanzate di deep learning.

Un'alternativa è quella del **multi input**.

9.1 Multi input e multi output

Quando abbiamo più input, non possiamo utilizzare il modello sequenziale. Ad esempio, abbiamo dati eterogenei che hanno bisogno di essere elaborati in modi diversi tra loro. Ci possono essere vari approcci che permettono di gestire questi casi:

- Il merging dei moduli, quando più input che sono eterogenei
- Concatenate, quando gli input sono omogenei
- Multiple Output: In questo caso abbiamo un solo input ma più output, e quindi ci saranno regressori diversi in base alla feature da predire.
- Inception: L'architettura a inception è una rete neurale convoluzionale profonda che è stata introdotta per la prima volta nel 2014. L'idea principale è quella di utilizzare filtri di dimensioni diverse per estrarre le features e poi **concatenare** le features estratte per avere un output finale.
- Residual: E' un archiettura che sfrutta il concetto di **residual learning**, ovvero l'idea di aggiungere un layer che è un'identità rispetto all'input.

9.2 Functional API

Se ci pensiamo, un modello è un qualcosa che *prende in input qualcosa* e ritorna fuori *un output*. Concettualmente, è la stessa cosa di una **funzione**. Possiamo, quindi, considerare dei moduli come delle funzioni.

$$\begin{aligned} \mathcal{O}_1 &= M_1(I_1) \\ \mathcal{O}_2 &= M_2(I_2) \\ \mathcal{O}_3 &= Q(\mathcal{O}_1, \mathcal{O}_2) \end{aligned} \tag{19}$$

dove Q è una funzione che prende in input due moduli e ritorna un output. Vediamo un esempio di come funzionano e come usare queste *functional API*

Esempio 9.1. *(Functional API)*

```
1
2  #Vogliamo avere questa rete
3  # (input: 784-dimensional vectors)
4  # [Dense (64 units, relu activation)]
5  # [Dense (64 units, relu activation)]
6  # [Dense (10 units, softmax activation)]
7  # (output: logits of a probability distribution over 10 classes)
8
9  inputs = keras.Input(shape=(784,))
10
11 # Qui, invece di sequenzializzare i layer, li colleghiamo
12 dense = layers.Dense(64, activation="relu")
13 x = dense(inputs)
14
15 #Abbiamo passato l'input a questo layer e otteniamo x come output
16 #Aggiungiamo altri layer alla rete
17
18 x = layers.Dense(64, activation="relu")(x)
19 outputs = layers.Dense(10)(x)
20
21 #Possiamo specificare, ora, il modello
22 model = keras.Model(inputs=inputs, outputs=outputs, name="Modello
       funzionale")
```

Nota: In questo momento stiamo solamende creando la struttura della rete, ma non stiamo ancora definendo i pesi.

10 Adanvced Keras

10.1 Subclassing

Definizione 10.1. *(Subclassing)*

Il subclassing in Keras è una tecnica avanzata per la creazione di modelli di deep learning personalizzati. Consiste nell'estendere la classe "tf.keras.Model" e definire il modello all'interno del metodo "init" e il passaggio in avanti all'interno del metodo "call". In questo modo, è possibile creare modelli di deep learning altamente personalizzati e flessibili, con la possibilità di definire qualsiasi tipo di struttura di rete e di utilizzare qualsiasi tipo di operazione di calcolo.

In soldoni, il subclassing ci permette di definire:

- Loss personalizzate
- Layer personalizzati
- Metriche personalizzate
- Modelli personalizzati

Chiaramente questo aumenta la complessità del codice, ma aumenta la flessibilità e la possibilità di avere un controllo maggiore di quello che ci fornisce di base Keras.

Ad esempio nelle **Generative Adversarial Networks - GAN** il concetto di training viene fatto in modo diverso, poiché non si usa la **stochastic gradient descent**. Nelle GAN anche l'ordine di apprendimento delle reti è importante; allenare contemporaneamente potrebbe portare a risultati strani e non convenienti. Ci sono tecniche e scenari specifici che hanno apprendimenti anch'essi specifici. Ovviamente, come detto prima, aumenta la complessità ma aumenta anche la flessibilità.

HA FATTO UN BORDELLO DI ROBA DALLA PAGINA DI TENSOR FLOW QUINDI BOH GUARDA La

11 Serie Temporali — Time Series

Le serie temporali vengono usate in campi come la **predizione di prezzi di azioni**, che richiedono una conoscenza dei trend passati per funzionare. Le reti neurali Feed Forward non considerano gli stati temporali. Si utilizza, infatti, un'altra architettura di rete neurale: **Recurrent Network - RNN**

11.1 Recurrent Neural Network RNN

Una RNN itera sugli elementi mantenendo uno stato che contiene informazioni di ciò che è stato visto finora. Questo stato viene passato avanti ad ogni iterazione.

Definizione 11.1. *RNN*

Una RNN è un grafo con cicli. I percettroni beneficiano del feedback dei loop. L'output di un percettrone al tempo t è coinvolto nel calcolo dell'output di un percettrone al tempo $t+1$.

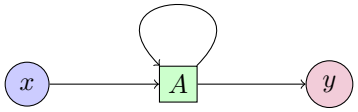


Figure 18: RNN

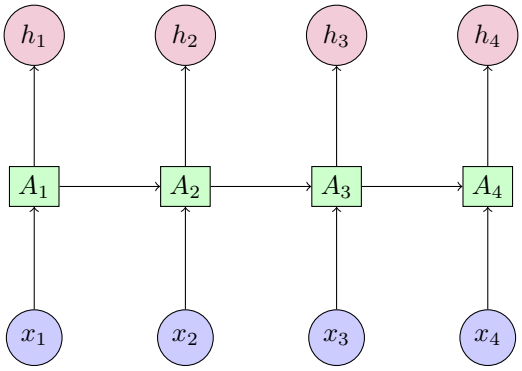


Figure 19: Unfolded RNN

Ci sono 2 equazioni da tenere a mente:

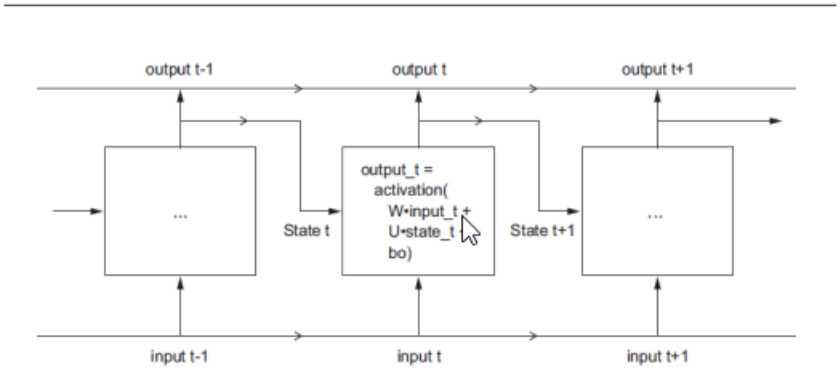


Figure 20: RNN

Rete Elman:

$$\begin{aligned} h_t &= \sigma(W_h x_t + U_h h_{t-1} + b_h) \\ y_t &= \sigma(W_y h_t + b_y) \end{aligned} \tag{20}$$

Rete Jordan:

$$\begin{aligned} h_t &= \sigma(W_h x_t + U_h y_{t-1} + b_h) \\ y_t &= \sigma(W_y h_t + b_y) \end{aligned} \tag{21}$$

Dove:

- x_t = Vettore di input
- y_t = Vettore di output
- h_t = Vettore di layer nascosto
- W, U, b = Pesi e Bias
- σ_h, σ_y = Funzioni di attivazione

11.1.1 L'utilizzo delle RNN

Alla fine, tutto ciò che noi andiamo a fare è **aggiungere dei layer** alla nostra rete.

```
model.add(SimpleRNN(32, return_sequences=True))#Return a list
model.add(SimpleRNN(32, return_sequences=True))#Return a list
model.add(SimpleRNN(32, return_sequences=True))#Return a list
model.add(SimpleRNN(32)) # last layer only returns last output

model.summary()
```

Un esempio di codice di rete è:

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000 # number of words to consider as features
maxlen = 500
batch_size = 32

(input_train, y_train), (input_test, y_test) = imdb.load_data(
    num_words=max_features)

input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)

from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(16))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop'
            , loss='binary_crossentropy'
            , metrics=['acc'])

history = model.fit(input_train, y_train,
            epochs=10,
            batch_size=128,
            validation_split=0.2)
```

Questo codice ha un problema: **non funziona**. Il problema è la **back propagation** attraverso il tempo. La normale backpropagation non funziona e ha bisogno di un cambiamento per adattarsi al **feedback loop**. Immaginiamo di avere una funzione del genere:

$$f(f(f(f(f(f(f(f(f(f(x_0|x_1,\ldots,x_n)))))))))) \tag{22}$$

Praticamente si ha un numero di derivate ripetuto. Poi, dalle slides, dopo tutti i calcoli si arriva ad avere un risultato in cui la derivata ha valore che **tende sempre a 0**. Come se si trovasse sempre un minimo globale. Quindi, questo approccio non funziona. Il **gradiente sparisce**

Motivazione: la sequenza è troppo lunga.

Definizione 11.2. *(Spiegazione del problema)*

Il problema della dipendenza a lungo termine è una sfida nell'addestramento delle reti neurali artificiali, in particolare le reti neurali ricorrenti (RNN). Si riferisce alla difficoltà che queste reti hanno nell'apprendere a collegare informazioni o contesti da passaggi precedenti nella sequenza a passaggi successivi.

Ad esempio, considera un modello di linguaggio che cerca di prevedere la parola successiva in una frase. Se la frase è "Sono cresciuto in Francia... parlo fluentemente —", il modello deve ricordare il contesto della "Francia" da molto prima nella frase quando arriva al punto vuoto, così può riempirlo con "francese". Questa è una dipendenza a lungo termine.

Le RNN fanno fatica con questo a causa del problema dei "gradienti che svaniscono". Durante la retropropagazione, i gradienti spesso diventano sempre

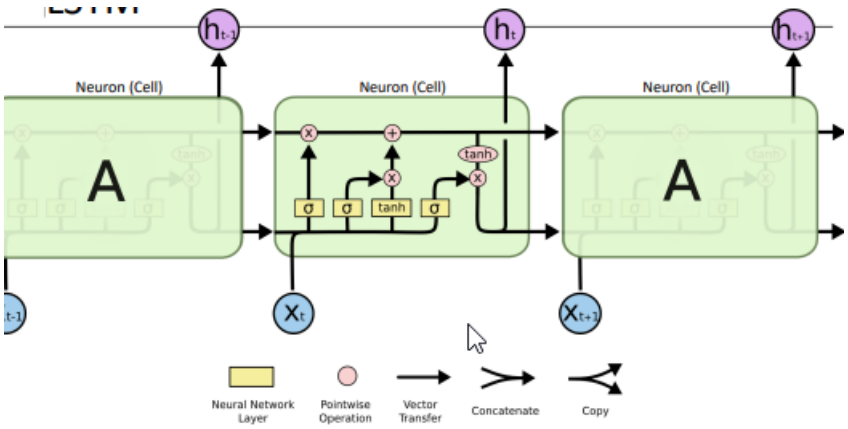


Figure 21: LSTM

più piccoli man mano che vengono propagati all'indietro nel tempo. Ciò significa che gli aggiornamenti ai pesi che collegano i passaggi precedenti nella sequenza a quelli successivi possono essere molto piccoli, e la rete può non riuscire a imparare queste dipendenze a lungo termine.

Una proposta di soluzione è quella di **cambiare la funzione di attivazione**.

Soluzione: Long Short Term Memory (LSTM)

11.2 Long Short Term Memory (LSTM)

Queste RNN speciali sono reti capaci di risolvere il problema delle dipendenze a lungo termine. In paticolare, sono state progettate per **controllare la sparizione del gradiente** e **evitare proprio il problema della dipendenza a lungo termine**

Riguardo l'archiettura, parliamo di una cosa importante.

Lo stato della cella salva un'informazione interna della catena **Storico interno**. Questa informazione, se rilevante, può essere **propagata**. Per controllare il flow di queste informazioni si una un **gate**. Servono, appunto, per decidere quali informazioni far passare.

Ci sono formule da ricopiare

- **Forget Gate**: decide quali informazioni scartare e quali tenere

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

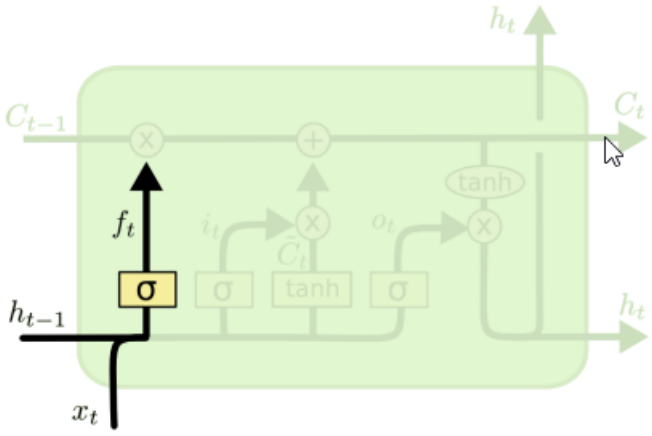


Figure 22: Forget Gate

- **Input Gate**: decide quali informazioni passare e quanto devono influenzare lo storico interno

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_1) \\ \tilde{C}_t &= tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned} \tag{23}$$

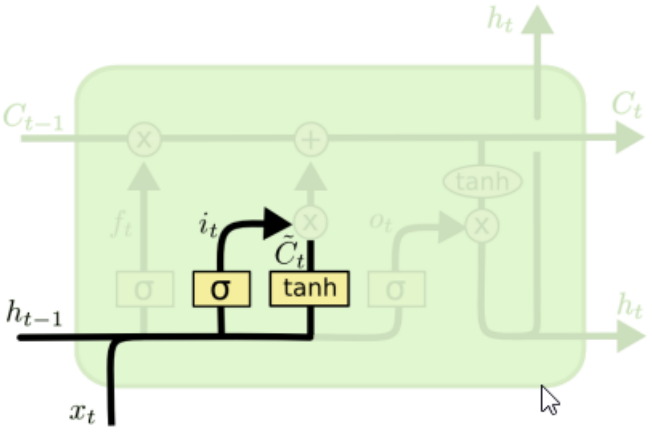


Figure 23: Input Gate

- **Aggiornamento dello stato interno**: Gate che risolvono il problema della dipendenza a lungo termine e del gradiente che svanisce

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

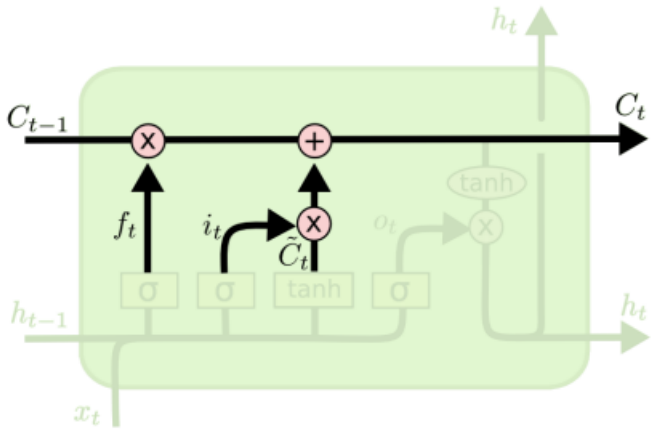


Figure 24: Update State

- **Output Gate**: L'output dipende dallo stato interno della cella. La LSTM controlla quanto l'output deve essere influenzato dallo storico interno

$$\begin{aligned} o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * tanh(C_t) \end{aligned} \tag{24}$$

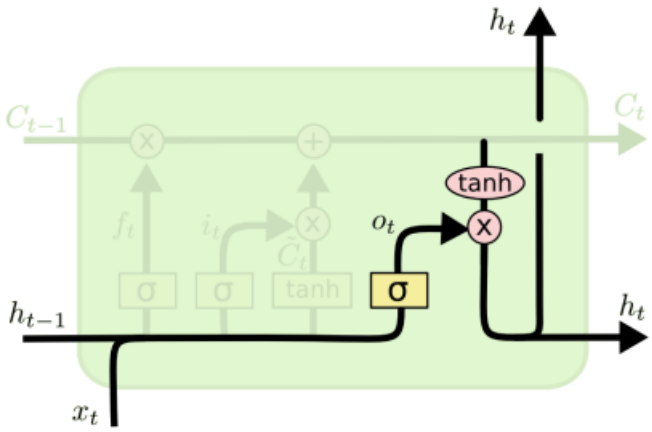


Figure 25: Output Gate

Ricordiamo che gli elementi sono:

- x_t = Vettore di input
- y_t = Vettore di output
- h_t = Vettore di layer nascosto
- W, U, b = Pesi e Bias
- σ_h, σ_y = Funzioni di attivazione

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop'
              ,loss='binary_crossentropy'
              ,metrics=['acc'])

              history = model.fit(input_train, y_train,
epochs=10,
batch_size=128,
validation_split=0.2)
```

Capitoli di laboratorio

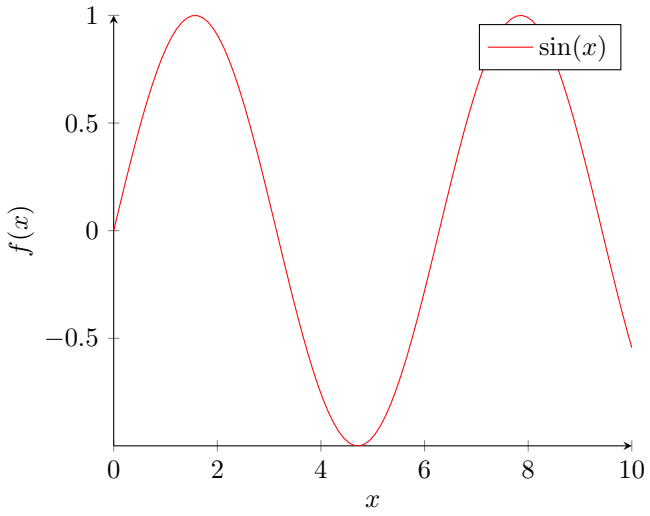
12 Lab: Introduzione Python

12.1 MatPlotLib

12.1.1 Plots

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 10, 100)
5 plt.plot(x, np.sin(x))
6 plt.show()
```

Non c'è molto da dire, il codice è autoesplicativo. La funzione `plot` prende in input due array, uno per l'asse delle ascisse e uno per l'asse delle ordinate. In questo caso, `x` è un array di 100 punti equidistanti tra 0 e 10, mentre `np.sin(x)` è un array di 100 punti che rappresentano il seno dei punti di `x`. La funzione `show` mostra il grafico.

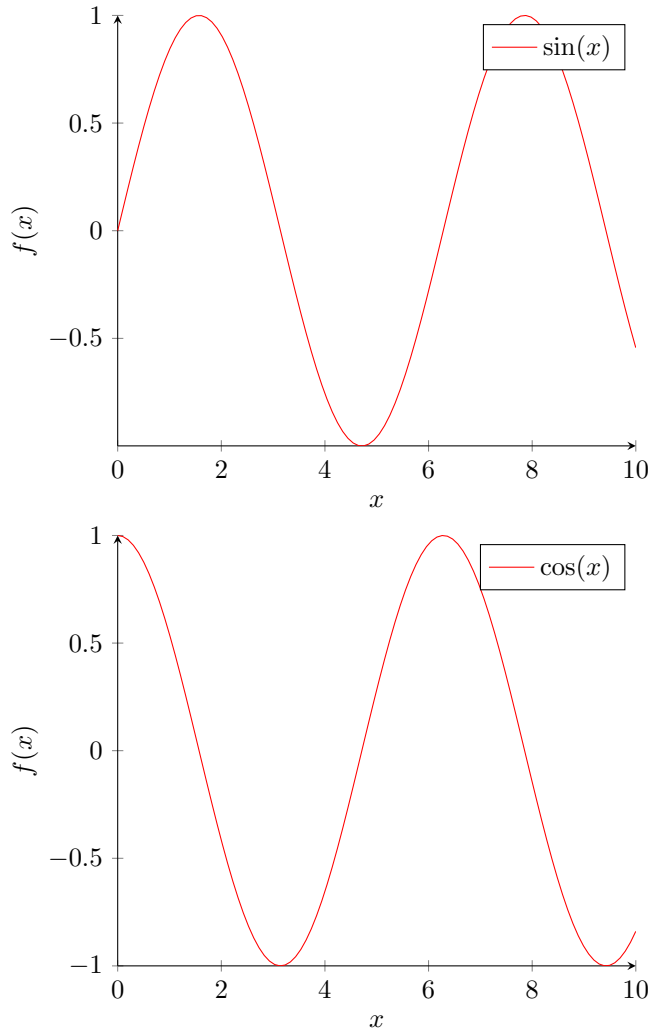


12.1.2 Sub-plots

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 10, 100)
5 plt.subplot(2, 1, 1)
6 plt.plot(x, np.sin(x))
```

```
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x))
plt.show()
```

La funzione `subplot` prende in input tre parametri: il numero di righe, il numero di colonne e l'indice del subplot corrente. Nel caso di questo esempio, il subplot corrente è il primo, quindi viene mostrato il grafico del seno. Poi viene mostrato il secondo subplot, che è quello del coseno.



C'è anche qui poco da dire, il codice è autoesplicativo.

12.2 NumPy

La libreria NumPy è una libreria per Python che permette di lavorare con array multidimensionali. Per importare la libreria, basta scrivere `import numpy as`

`np`.

Mi rompo le palle in maniera assurda di scrivere tutti gli esempi. Quindi questo capitolo penso sia abbastanza inutile.

E' possibile trovare il codice di **numPy** a questo link: `www.ciao.it`

13 Lab: Reti Neurali da zero

13.1 Introduzione

Partiamo dicendo una cosa molto importante: **per quale motivo usiamo la backpropagation?**

$$\frac{y-b}{x} = w \tag{25}$$

Ma possiamo scriverlo come:

$$(y-b) \cdot x^{-1} = w \tag{26}$$

Ora però, se consideriamo:

- y il vettore risultante
- w la matrice dei pesi
- x il vettore di input

Calcolare l'inversa di **x** non è una cosa cosi poco costosa, anzi. Per questo motivo utilizziamo il training delle reti come la backpropagation e la discesa del gradiente.

Ora, andiamo più a fondo. Facciamo un esempio più pratico.

13.2 Esempio pratico

Immaginiamo di avere un dataset di 2 features e una label da identificate.

x_1	x_2	y
1	2	0
2	3	0
3	4	0
4	5	0
5	6	0
6	7	1
7	8	1
8	9	1
9	10	1
10	11	1

Table 1: Dataset di esempio

Ogni livello di una rete neurale può essere rappresentato attraverso le **matrici**.

Ma passiamo alla definizione formale:

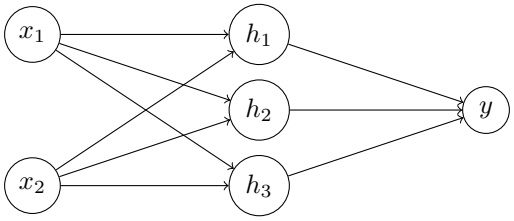


Figure 26: Rete neurale di esempio

Definizione Rete Neurale: Una rete neurale è una tupla:

$$NN = \{g, l, o, i, fpp\} \tag{27}$$

con:

- g: il grafico
- l: la funzione loss
- o: l'ottimizzatore
- i: l'inizializzatore
- fpp: la fix point procedure

Nota 1: l'ottimizzatore intende in quale modo si performa la **discesa del gradiente.**

Nota 2: l'inizializzatore è la funzione che inizializza i pesi della rete neurale. Ci possono essere diversi algoritmi per gli inizializzatori, ma non c'è modo di sapere quale funziona meglio, poiché non si può sapere nelle reti neurali.

13.2.1 Il grafo

Il solito grafo che abbiamo visto in precedenza è un grafo aciclico diretto. Ha dei nodi che sono i percettroni, che hanno degli archi composti dai pesi, che hanno una funzione di attivazione, che prendono in input un valore e ne sputano fuori uno chiamando la funzione di attivazione sull'input.

Nota: la funzione di attivazione è una funzione non lineare.

13.2.2 La funnzione loss

L'obiettivo della funzione loss è quello di misurare la distanza tra il valore predetto e il valore reale.

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{28}$$

13.2.3 L'ottimizzatore

L'ottimizzatore ci permette di trovare una soluzione ottimale **non ottima**, cioé trovare **il minimo della funzione di loss**. La tecnica che si usa è quella delle **discesa del gradiente**.

13.2.4 Discesa del gradiente

La discesa del gradiente sfrutta un parametro chiamato **learning rate** che permette di capire quanto la discesa del gradiente deve essere veloce. Se il learning rate è troppo alto, la discesa del gradiente potrebbe non convergere, se è troppo basso, la discesa del gradiente potrebbe convergere troppo lentamente.

Nota: il learning rate è un ottimizzatore *molto naive*.

13.2.5 Inizializzatore

Il metodo di inizializzazione può cambiare di molto il risultato della rete neurale. In pratica assegna un valore ai *pesi* e ai *bias*. Alcuni metodi sono:

- Inizializzazione a 0: Questo ha alcuni problemi, perché non si ha diversificazione tra i nodi e i nodi nascosti diventano simmetrici.
- inizializzazione costante: stessi problemi della precedente
- Inizializzazione Random: Si fa seguendo una distribuzione uniforme oppure una distribuzione normale.

13.2.6 Fix Point Procedure

La procedura per il training di una rete neurale è un processo che si basa su alcuni steps:

```
1
2 net CustomNeuralNetwork(...)
3 initialize_weights_and_biases (net)
4 optimizer = myOptimizer(...)
5 loss_function = myLoss Function(...)
6 epochs = ... # the number of dataset scans
7 history = [] # a list containing the loss evolution
8 for epoch in range(epochs):
9     optimizer.reset() # it may have an internal status
10    loss = loss_fuction (out_target, net(input)) back_propagation (
      loss, optimizer, net)
11    history.append(loss)
```

13.3 Esempio da zero

13.3.1 La funzione Sigmoid

Spendiamo qualche parola sulla funzione sigmoid, che è una funzione molto importante per le reti neurali.

In particolare, la funzione avrà un valore di attivazione compreso tra 0 e 1 e, in particolare, quando l'input è 0, la funzione ha valore 0.5. Se ha un valore basso, sarà zero e chiaramente se sarà alto avrà valore 1.

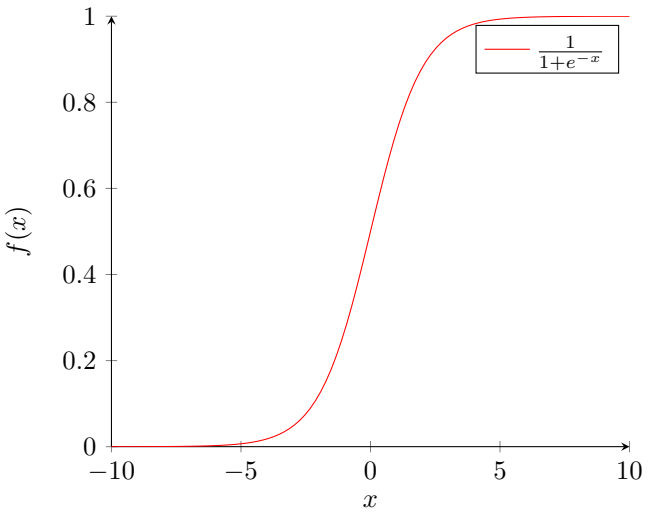


Figure 27: Funzione Sigmoid

Detto questo, **per quale motivo è importante?** Se pensiamo all'*inizializzazione*, che tipi di valori è meglio avere? Avere tutti i valori a zero porterebbe a valori simmetrici e quindi a problemi di convergenza. Vogliamo dei valori di pesi che siano **distribuiti uniformemente intorno allo 0.**

13.4 Tangente Iperbolica TanH

La funzione TanH è una funzione che ha un valore di attivazione compreso tra -1 e 1 e, in particolare, quando l'input è 0, la funzione ha valore 0. Se ha un valore basso, sarà -1 e chiaramente se sarà alto avrà valore 1.

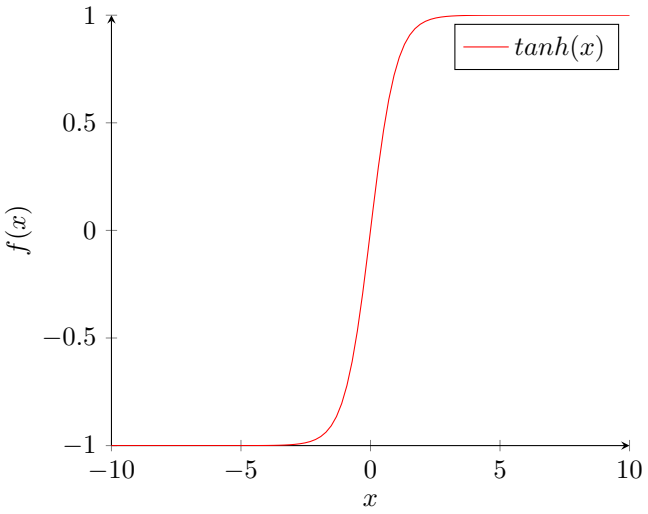


Figure 28: Funzione TanH

13.5 ReLu

La funzione ReLu è una funzione che ha un valore di attivazione pari a 0 quando l'input è negativo e ha un valore di attivazione pari all'input quando l'input è positivo.

Parole di Adornetto: E' buona? Si. E' stabile? Si. Perché? Boh.

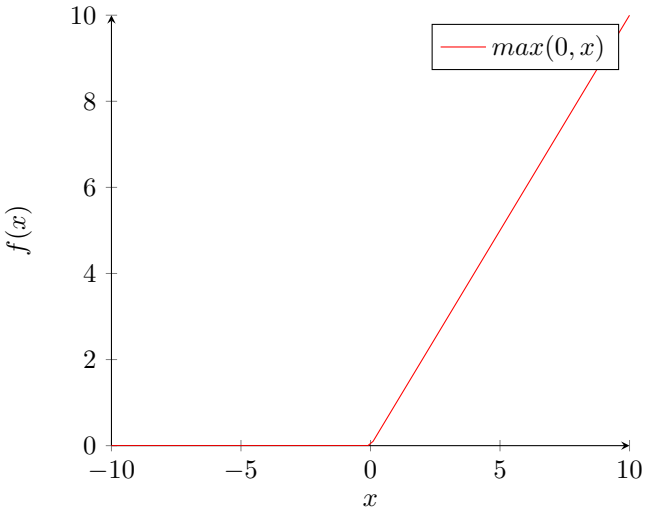


Figure 29: Funzione ReLu

13.6 Scalare i valori

Skip molto avanti riguardo l'esempio, ma si parla di scalare i dati.

Lo scaling dei dati è un'operazione importante nel machine learning perché i dati possono essere su scale diverse e questo può causare problemi durante l'allenamento del modello. Ad esempio, se abbiamo due variabili di input, una che varia da 0 a 1 e l'altra che varia da 0 a 1000, la seconda variabile avrà un impatto molto maggiore sull'output del modello rispetto alla prima. Ciò può portare a problemi come l'overfitting, in cui il modello si adatta troppo ai dati di addestramento e non generalizza bene sui dati di test.

Per risolvere questo problema, si utilizzano tecniche di scaling dei dati per portare tutte le variabili su una scala comune. Ci sono diverse tecniche di scaling, come la normalizzazione e la standardizzazione, che possono essere utilizzate a seconda del tipo di dati e del modello utilizzato.

Le funzioni di attivazione, come la funzione ReLU, possono anche essere influenzate dalla scala dei dati di input. Se i dati non sono scalati correttamente, la funzione di attivazione potrebbe produrre valori che non permettono un allenamento corretto del modello. Lo scopo dello scaling dei dati è quello di mantenere i dati intorno allo 0, in modo che le funzioni di attivazione possano produrre valori che permettono un allenamento corretto del modello.

- Standardizzazione: La standardizzazione è una tecnica di scaling dei dati che assume che i dati siano distribuiti normalmente all'interno di ogni feature e li scala in modo che la distribuzione abbia una media uguale a 0 e una deviazione standard uguale a 1. Questa tecnica funziona bene

quando i dati hanno una distribuzione normale, ma non funziona bene se i dati hanno una distribuzione non normale.

- Normalizzazione: La normalizzazione è una tecnica di scaling dei dati che scala i valori di ogni feature in modo che siano compresi tra 0 e 1. Questa tecnica funziona bene quando i valori di input non hanno una distribuzione normale. Ad esempio, se i valori di input sono compresi tra 0 e 1000, la normalizzazione li porterà su una scala compresa tra 0 e 1.

13.7 Plottare la loss

La funzione loss è una misura dell'errore del modello durante l'allenamento. L'obiettivo dell'allenamento è quello di minimizzare la funzione loss, ovvero di ridurre l'errore del modello. Durante l'allenamento, il modello viene eseguito su un set di dati di addestramento e la funzione loss viene calcolata per ogni esempio di addestramento. L'errore totale del modello è la somma di tutte le funzioni loss per ogni esempio di addestramento.

L'allenamento del modello avviene in epoche, ovvero in cicli di esecuzione su tutti i dati di addestramento. L'obiettivo è che la funzione loss diminuisca ad ogni epoca, ovvero che l'errore del modello diminuisca man mano che il modello viene addestrato su più dati. Se la funzione loss non diminuisce ad ogni epoca, significa che il modello non sta imparando abbastanza dai dati di addestramento e potrebbe essere necessario modificare l'architettura del modello o i parametri di addestramento.

Se la funzione di loss arriva a 0, vuol dire che **siamo in minimo globale**, che non è comune.

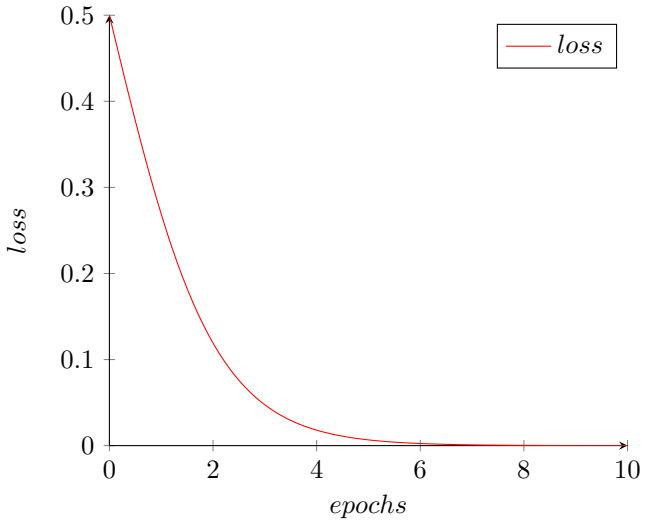


Figure 30: Funzione loss

Se invece finiami con un grafico che arriva fino a 0.1 e poi si stabilizza, potremmo essere in un **minimo locale**, oppure semplicemente la topologia della Rete permette di avere questo risultato come massimo risultato.

13.8 Importante cosa su Gradient Descent

Lo usiamo per un motivo particolare:

> Più andiamo indietro nei layer della rete, maggiori saranno i termini che abbiamo già calcolato, rendendo i calcoli più efficienti.

Questo è più semplice rispetto ad invertire una matrice

14 TensorFlow e Keras

Onestamente non so cosa scrivere in questo capitolo. Se trovo qualcosa di interessante la scrivo.

La lezione è fatta su un notebook. Quindi scriverò le notebook

14.1 One Hot Encoding

Sulle categoriche lo si fa quando si ha **una classificazione di classe**. Ad esempio, se abbiamo una categorica che viene rappresentata numericamente, **overall condition of the house**, se ha una scala da 1 a 10, in questo caso **NON SERVE** il one hot encoding.

Se invece fosse stato qualcosa del tipo **House Color = 1,2,3**, in questo caso si.

Diciamo che serve quando **NON ABBIAMO UNA SCALA NUMERICA o ORDINE**

14.2 Variabili correlate

Se abbiamo due variabili correlate, ad esempio **Overall Condition** e **Overall Quality**, in questo caso **una delle due deve sparire.** Le motivazioni sono due:

- Ridurre la complessità del modello
- Evitare che la rete neurale NON RIESCA a dare il peso corretto alla variabile, poiché potrebbe dividere il peso in 50% tra le due.

14.3 Accuracy come loss

Non usiamo l'accuracy come loss function perché dobbiamo usare una funzione che sia **differenziabile**, e l'accuracy non lo è.

Nel nostro esempio usiamo **la binary cross entropy**.

14.4 Epoche e batch size

Capiamo questa cosa: ad ogni **epoca** dobbiamo scorrere l'intero dataset. Ma ad ogni **iterazione di training NON CI SERVE l'intero dataset**. Quindi, ad ogni epoca, dobbiamo scorrere il dataset **numero di iterazioni** volte.

Ad esempio, se abbiamo una batch size di 32, e abbiamo 1000 dati, allora abbiamo 32 iterazioni per epoca.

14.5 Overfitting e come evitarlo

La definizione migliore di overfitting sentita: **dobbiamo imparare la distribuzione dei dati e non i dati stessi.**

Il nostro modello deve essere capace di **generalizzare**. In un grafico dove abbiamo una validation loss, se la validation comincia a risalire dopo un certo periodo di tempo non è capace di generalizzazione e vuol dire che non ha imparato bene dai dati la loro distribuzione.

14.6 Migliorare le performance di un modello

Ci sono varie tecniche per migliorare le performance.

- Aumentare il numero di epoche
- Aumentare il numero di neuroni
- Aumentare il numero di layer
- Cambiare la funzione di attivazione

Attenzione: Se si esagera con questi valori si può andare in overfitting.

Una tecnicha è quella della **regolarizzazione**. L'abbiamo vista anche in **figura 15**.

```
1     model_3 = Sequential([
2     Dense(1000, activation='relu', kernel_regularizer=regularizers.
      l2(0.01), input_shape=(10,)),
3     Dropout(0.3),
4     Dense(1000, activation='relu', kernel_regularizer=regularizers.
      l2(0.01)),
5     Dropout(0.3),
6     Dense(1000, activation='relu', kernel_regularizer=regularizers.
      l2(0.01)),
7     Dropout(0.3),
8     Dense(1000, activation='relu', kernel_regularizer=regularizers.
      l2(0.01)),
9     Dropout(0.3),
10    Dense(1, activation='sigmoid', kernel_regularizer=regularizers.
      l2(0.01)),
11 ])
```

Nel notebook si vedono 3 grafici:

- La rete normale: si comporta bene
- La rete più complessa: si comporta male perché overfitta
- La rete più complessa regolarizzata: è la migliore

15 Lab: Convolutional Neural Networks

15.1 Cross Entropy vs Accuracy

Domanda 15.1. *(Per quale motivo usiamo la cross entropy nella multi-class classification? Perché non accuracy?)*

Se prendessimo un esempio di classificazione con 3 classi $1, 2, 3$ e due classificatori. Immaginiamo di avere degli examples che hanno **true class label**:

- 0 0 1
- 0 1 0
- 1 0 0

E avessimo due classificatori che hanno come risultato:

Classificatore 1:

Example 1	Example 2	Example 3
0.1	0.1	0.8
0.1	0.8	0.1
0.4	0.5	0.1

Classificatore 2:

Example 1	Example 2	Example 3
0.3	0.3	0.4
0.3	0.4	0.3
0.4	0.5	0.1

Entrambi hanno un'**accuracy** del 0.66 ma il **classificatore 1** è più sicuro delle sue predizioni. La **cross entropy** è una funzione che misura la **distanza** tra due distribuzioni di probabilità. In questo caso misura la distanza tra la distribuzione di probabilità del classificatore e la distribuzione di probabilità delle **true class label**. La cross entropy è definita come:

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{29}$$

Dove p è la distribuzione di probabilità delle **true class label** e q è la distribuzione di probabilità del classificatore.

15.2 Workflow per image classification

```
img_size=(50,50)

images = []
age = []
gender = []
for img in os.listdir(path):
  ages = img.split("_")[0]

  if int(ages) <18 or int(ages)>25:
    continue

  genders = img.split("_")[1]
  img = cv2.imread(str(path)+"/"+str(img))
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img, img_size, interpolation = cv2.INTER_AREA)
  images.append(np.array(img))
  age.append(np.array(ages))
  gender.append(np.array(genders))
```

Listing 3: Workflow per image classification

In questo caso, vogliamo creare un modello per classificare **gender** e **age**. Facciamo un po' di modifiche alla dimensione delle immagini e le salviamo ciò che ci serve in un array.

```
age = np.array(age,dtype=np.int64)
images = np.array(images) / 255.
gender = np.array(gender,np.uint64)

n=9
idx_toplot = np.random.randint(len(images), size=n)
plt.figure(figsize=(10, 10))
for i in range(n):
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(images[idx_toplot[i]])
  plt.title(f'Age={age[idx_toplot[i]]}, Gender={gender[idx_toplot[i
    ]]}')
  plt.axis('off')
```

Listing 4: Workflow per image classification

15.2.1 Bilanciare le classi

Se ci sono problemi di **unbalance** tra le classi di un dataset, ciò che si può fare per bilanciare le classi è **aumentare** il numero di esempi della classe con meno esempi. In questo caso, possiamo fare **data augmentation** per aumentare il numero di esempi per ogni classe che ha meno esempi.

```
data_augmentation = tf.keras.Sequential(
  [
    layers.RandomFlip("horizontal",input_shape=(img_size[0],
    img_size[1],3)),
    layers.RandomRotation(0.1),
```

```
    layers.RandomZoom(0.1),
  ]
)

# Find the class to be augmented and the major one
minor_class = min(gender_class_counts, key=gender_class_counts.get)
major_class = max(gender_class_counts, key=gender_class_counts.get)

# Select the images belonging to the class to augment
img_seeds = images[gender==minor_class]

# Select random indexes for the images to augment
n_augment = gender_class_counts[major_class]-gender_class_counts[
    minor_class]
idx_to_aug = np.random.randint(img_seeds.shape[0], size=n_augment)

new_images = []
new_gender = []
plt.figure(figsize=(10, 10))
for i in range(n_augment):
  augmented_image = data_augmentation(img_seeds[idx_to_aug[i]])
  new_images.append(augmented_image)
  new_gender.append(minor_class)

  if i<9:
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_image)
    plt.axis("off")
```

Listing 5: Data augmentation

A questo punto, dopo aver **aumentato** il numero di esempi della classe minore, arriviamo ad avere un dataset bilanciato tra le due classi. Per quanto riguarda il modello, solita roba.

```
x_train_age, x_test_age, y_train_age, y_test_age = train_test_split
    (images, age, random_state=42)

x_train_gender, x_test_gender, y_train_gender, y_test_gender =
    train_test_split(balanced_images, balanced_gender, random_state
    =42)

gender_model = Sequential()

gender_model.add(Conv2D(32, kernel_size=3, activation='relu',
    input_shape=(img_size[0],img_size[1],3)))
gender_model.add(MaxPool2D(pool_size=3, strides=2))

gender_model.add(Conv2D(64, kernel_size=3, activation='relu'))
gender_model.add(MaxPool2D(pool_size=3, strides=2))

gender_model.add(Conv2D(128, kernel_size=3, activation='relu'))
gender_model.add(MaxPool2D(pool_size=3, strides=2))

gender_model.add(Flatten())
gender_model.add(Dropout(0.2))
gender_model.add(Dense(128, activation='relu'))
gender_model.add(Dense(1, activation='sigmoid', name='gender'))
```

```
20
21 gender_model.compile(optimizer='adam', loss='binary_crossentropy',
       metrics=['accuracy'])
22
23 history_gender = gender_model.fit(x_train_gender, y_train_gender,
24                         validation_data=(x_test_gender,
       y_test_gender), epochs=10)
25
26 ############################
27 gender_model.save(save_model_path+'gender_model.h5')
28 ############################
```

Listing 6: Modello

Domanda 15.2. *(Per quale motivo utilizziamo la funzione **save**?)*

Perché vogliamo salvare il modello che abbiamo creato in un file. In questo caso, il modello viene salvato in un file **.h5**. Salviamo **i pesi della rete** per poter continuare in un secondo momento l'addestramento della rete.

Osservazione 15.1. *(Cosa impara la rete?)*

La parte di convoluzione CNN impara a riconoscere le **features** delle immagini. La parte di **Dense** impara a classificare le immagini in base alle features che ha imparato la parte di convoluzione.

16 Autoencoders

16.1 Nota su PCA

La PCA (Principal Component Analysis) è una tecnica di riduzione della dimensionalità che si basa sulla decomposizione in autovalori della matrice di covarianza.

Ora, immaginiamo di avere tre input x_1, x_2, x_3. Un **neurone** può essere visto come una combinazione lineare di questi tre input, ovvero:

$$y = w_1x_1 + w_2x_2 + w_3x_3 \tag{30}$$

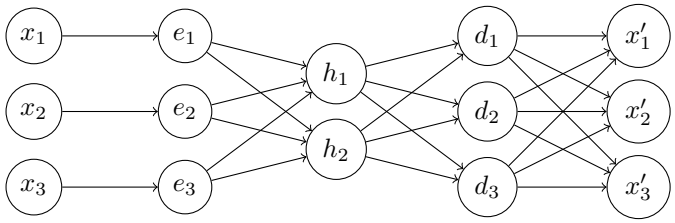
16.2 Autoencoders

E' un architettura che permette di **proiettare** il nostro input in uno **spazio dimensionale più piccolo**.

Avendo un input x_1, x_2, x_3, la rete ci darà come *output* la **ricostruzione** dell'input.

Obiettivo: minimizzare la differenza tra l'input e la sua ricostruzione

LATENT REPRESENTATION ENCODER DECODER



$$Loss(\bar{x}, \bar{x'}) \tag{31}$$

L'**autoencoder** è praticamente l'unione tra **encoder, latent representation e decoder**.

Domanda 16.1. *(Per quale motivo usiamo le NN e non le PCA?)*

La risposta è che le **PCA** sono lineari, e quindi meno complesse. Per questo motivo, le NN sono più flessibili e possono essere usate per risolvere problemi più complessi.

16.2.1 Gli steps

1. Encoder: Impara una **rappresentazione compatta** dell'input
2. Decoder: Ricostruisce l'input

16.2.2 Applicazioni

Gli autoencoders possono essere usati in vari campi.

Ad esempio, forzare $x = r$. Questo porta ad un **rischio di overfitting elevato**. Diminuisce la dimensione e apprendimenti delle feature.

La vera potenza però è nell'**astrarre i dati** invece di impararli in modo perfetto. Stiamo parlando di **produrre dati** che potrebbero appartenere al dominio di input. Questo rientra in **data generation, data completion**.

Gli autoencoders non sono perfetti e soffrono di alcuni problemi. Alcuni sono:

- Overfitting
- Grandezza del codice

Se vuoi copiarti le slides e aggiungere qualcosa.

16.3 Autoencoders e Convolution

Ricordiamo il concetto di fare sampling nello spazio latente. Questo è fondamentale per la generazione di nuovi dati.

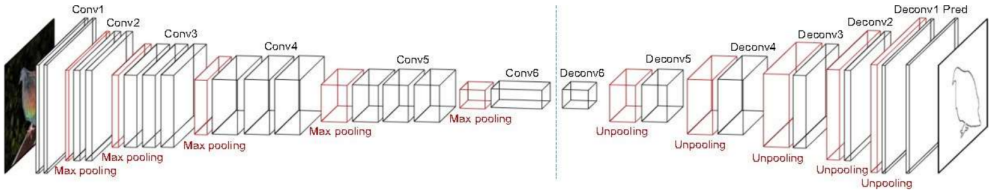


Figure 31: Convolutional Autoencoder

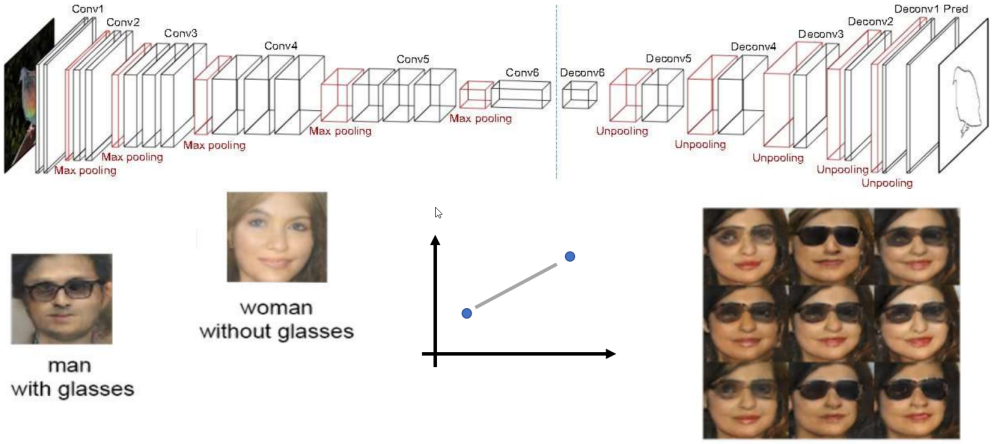


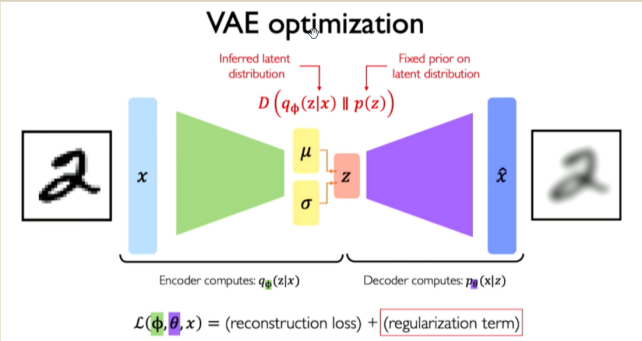
Figure 32: Convolutional Autoencoder sampling

Praticamente, questa tecnologia è alla base per i **Deep Fakes!**
Altre informazioni: Notebook su autoencoders.

17 VAE (Variational Autoencoder)

I VAE permettono di costruire un **latent space continuo**; Permette di prendere uno spazio e un sample da quello spazio e saremo certi che i punti all'interno seguono la distribuzione di quello spazio.

Nota: Se prendiamo un input, lo codifichiamo, quando sarà nel **latent space** non sarà più un punto, ma sarà una distribuzione di punti. Non si codificano punti, quindi, ma **distribuzioni** (coppie di *mean e standard deviation* μ, σ). La distribuzione sarà **normale**.



$$D(q_\theta(z|x)||p(z)) \tag{32}$$

Elementi:

- Inferred Latent Distribution: Distribuzione che hanno i dati di input
- Fixed Prior Latent Distribution: Distribuzione che vogliamo avere nel latent space

Ad esempio, nei normali Autoencoder, abbiamo che la distribuzione di input e la distribuzione di z che viene fuori non hanno niente a che fare. Cioè, non sappiamo niente a riguardo e non ci interessa. Nei VAE, invece, vogliamo che la distribuzione di z sia **normale**, perché con la distribuzione normale si lavora in modo più stabile.

1. Input X
2. Decodifica X
3. Si hanno due vettori di medie e varianze
4. Si fa sampling e si ottiene z
5. Si decodifica z

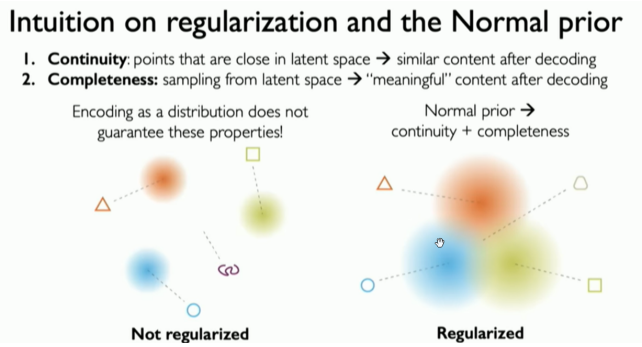


Figure 33: Esempio di VAE

6. Si ottiene il sample generato X'

L'output di questa architettura prende il nome di **generated sample**.
Parlare di differenza tra regolarized e non regolarized dopo (inizio notebook)

17.1 Regularization term e Reparametrization trick

Il **Regularization Term** serve per fare in modo che la rete apprenda la distribuzione nel **latent space**. Questo permtte di forzare di avere $\mu = 0 \wedge \sigma = 1$
Il regularization term si chiama KL-Divergence e si calcola come segue:

$$\frac{1}{2}\sum_{j=0}^{k-1}(\sigma_j + \mu_j^2 - 1 - log(\sigma_j)) \tag{33}$$

Domanda 17.1. *(Qual è la condizione per trainare una NN?)*

Che la funzione sia **differenziabile**. Ma abbiamo un problema. Se dal tra **mean e standard deviation** facciamo sampling, otteniamo una distribuzione. Questo **non è differenziabile**. E qui entra in gioco il **reparametrization trick**. Il trick è quello di **wrappare** questo processo in una funzione che sia differenziabile.

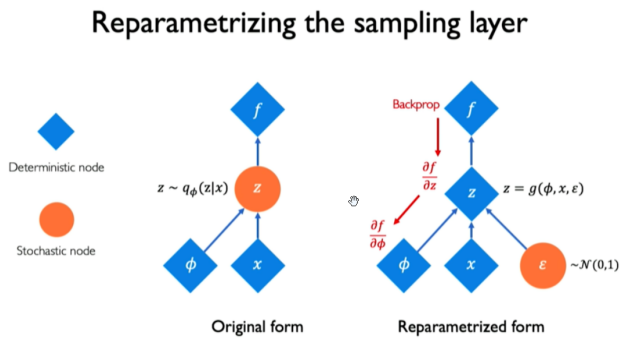


Figure 34: Reparametrization trick

Oltre all'input, si aggiunge un elemento ε che è un sample da una distribuzione normale, lo consideriamo come **noise**. Questo noise è **non deterministico** ed è differenziabile.

In questo modo, la funzione è differenziabile e si può fare backpropagation, andando a cambiare i pesi dell'**encoder** e del **decoder**.

Senza di questo non potremmo farlo perché z è un **random process** e non è differenziabile.

```
from tensorflow import keras

class Sampling(keras.layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a
    digit."""

    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim)
    )
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

from tensorflow.keras.datasets import fashion_mnist
import numpy as np

(x_train, _), (x_test, _) = fashion_mnist.load_data()
mnist_fashion = np.concatenate([x_train, x_test], axis=0)
mnist_fashion = np.expand_dims(mnist_fashion, -1).astype("float32")
     / 255

mnist_fashion.shape
```

Listing 7: Esempio di VAE

```

latent_dim = 50
#50 medie e 50 deviazioni standard

encoder_inputs = keras.Input(shape=(28, 28, 1))
```

```
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="
    same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="
    same")(x)
x = layers.Flatten()(x)
x = layers.Dense(100, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name=
    "encoder")
encoder.summary()
```

Listing 8: Esempio di VAE

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2,
    padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2,
    padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid"
    , padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder
    ")
decoder.summary()
```

Listing 9: Esempio di VAE

```

class VAE(keras.Model):

    def __init__(self, encoder, decoder, **kwargs):
        super(VAE,self).init(kwargs)

        self.encoder = encoder
        self.decoder = decoder
        self.totallosstracker = keras.metrics.Mean(name="totalloss"
    )
        self.reconstructionlosstracker = keras.metrics.Mean(name="
    reconstructionloss")
        self.kllosstracker = keras.metrics.Mean(name="klloss")

    def trainstep(self, data):

        with tf.GradientTape() as tape:

            zmean, zlogvar, z = self.encoder(data)

            reconstruction = self.decoder(z)

            reconstructionloss = tf.reducemean(
                tf.reducesum(keras.losses.binarycrossentropy(data,
    reconstruction), axis=(1, 2)))

            klloss = -0.5 * (1 + zlogvar - tf.square(zmean) - tf.
    exp(zlogvar))
```

```
            klloss = tf.reducemean(tf.reducesum(klloss, axis=1))

            totalloss = reconstructionloss + klloss

        grads = tape.gradient(totalloss, self.trainableweights)
        self.optimizer.applygradients(zip(grads, self.
    trainableweights))
        self.totallosstracker.updatestate(totalloss)
        self.reconstructionlosstracker.updatestate(
    reconstructionloss)
        self.kllosstracker.updatestate(klloss)

        return {
            "loss": self.totallosstracker.result(),
            "reconstructionloss": self.reconstructionlosstracker.
    result(),
            "klloss": self.kllosstracker.result(),
        }
```

Listing 10: Esempio di VAE

18 Recurrent Neural Network e Natural Language Processing (RNN e NLP)

18.1 RNN e LSTM

Stiamo pensando in uno spazio a tre dimensioni:

$$Input, Output, Time$$

Nelle RNN ogni neurone prende in input l'output del neurone precedente.

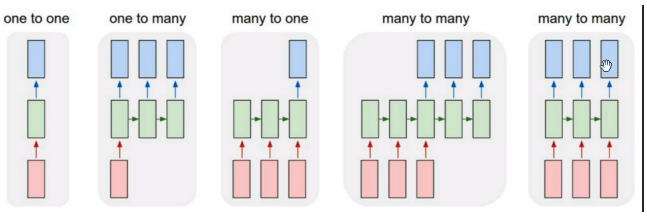


Figure 35: RNN

- **One-to-one** - classic feed forward neural network architecture, with one input and we expect one output. - **One-to-many** - e.g. image captioning.

We have one image as a fixed size input and the output can be words or sentences which are variable in length. - **Many-to-one** e.g. sentiment classification. The input is expected to be a sequence of words or even paragraphs of words. The output can be a regression output with continuous values which represent the likelihood of having a positive sentiment. - **Many-to-many** - machine translation like in Google translate. The input could an English sentence which has variable length and the output will be the same sentence in a different language which also has variable length. The last many to many model can be used for video classification on frame level. Feed every frame of a video into the neural network and expect an output right away. However, since frames are generally dependent on each other, it is necessary for the network to propagate its hidden state from the previous to the next. Thus, we need recurrent neural network for this kind of task.

Si utilizza un'architettura particolare per le RNN, chiamata **Long Short Term Memory (LSTM)**.

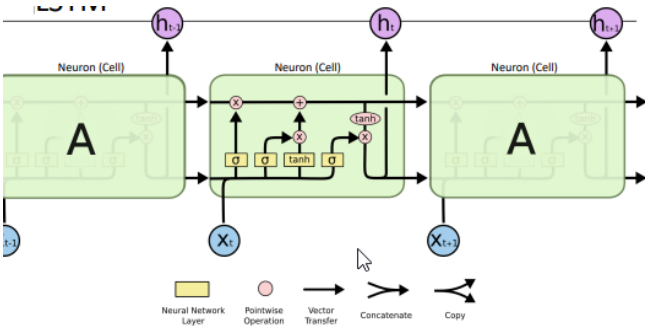


Figure 36: LSTM

- **Forget Gate** - decides which information from long term memory be kept or discarded and this is done by multiplying the incoming long term memory (upper incoming arrow) by a forget vector generated by the current input and incoming short memory (lower arrow).

- **Input Gate** - decides what information will be stored in long term memory. It only works with the information from the current input and short term memory from the previous step. At this gate, it filters out the information from variables that are not useful.

- **Output Gate** - it takes the current input, the previous short term memory and newly computed long term memory to produce new short term memory which will be passed on to the cell in the next time step. The output of the current time step can also be drawn from this hidden state.

Cose su LSTM ma boh

18.2 NLP

Le reti neurali lavorano con **numeri**. Nel natural language processing si lavora con dei **testi** che sono delle **sequenze** di parole.

Proposta 1: One Hot Encoding: Si potrebbe pensare ad un **one hot encoding**, ma questo sarebbe troppo costoso, perché se dovessimo pensare ad un OHE di un vocabolario di 1000 parole, questo crescerebbe in modo esponenziale

Proposta 2:Ogni parola un unico numero: Problema da prendere dal notebook

Proposta finale: Word Embedding: Questa proposta è vincente perché permette di avere una rappresentazione delle parole mantenendo un'importante proprietà: la **somiglianza** tra esse. Praticamente un embedding è un vettore di valori in floating point. Invece di hard-codare i valori del vettore, si trattano come valori da **allenare**.

cat	0.2	0.1	
on	0.1	0.9	
mat	0.1	0.9	

Table 2: Esempio di Word Embedding

18.3 Come si lavora con i Word Embedding?

18.3.1 Encoder

```
VOCAB_SIZE = 1000
#Per usare testo si usa un layer TextVectorization
encoder = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))

#Parole piu usate nel dataset
vocab = np.array(encoder.get_vocabulary())
vocab[:20]

#Come sono encodate le parole
encoded_example = encoder(example)[:3].numpy()
encoded_example

#Ad esempio, l'encoding viene fatto stampando l'indice della parola
     nel vocabolario
```

Listing 11: Esempio di Word Embedding