Laboratorio 2 - Data Science

Pivi Riccardo

Ottobre 2025

1 Punto 1: Creare i Dati

Sono stati creati 200 dati (x,y) nel modo seguente:

```
N = 200

X = np.random.rand(N,1)
Y = np.sin(2*np.pi*X) + np.random.uniform(-0.2,0.2, size=(N,1))
```

Si riporta il grafico [Fig.1] del confronto dei dati con la funzione $y = \sin 2\pi x$

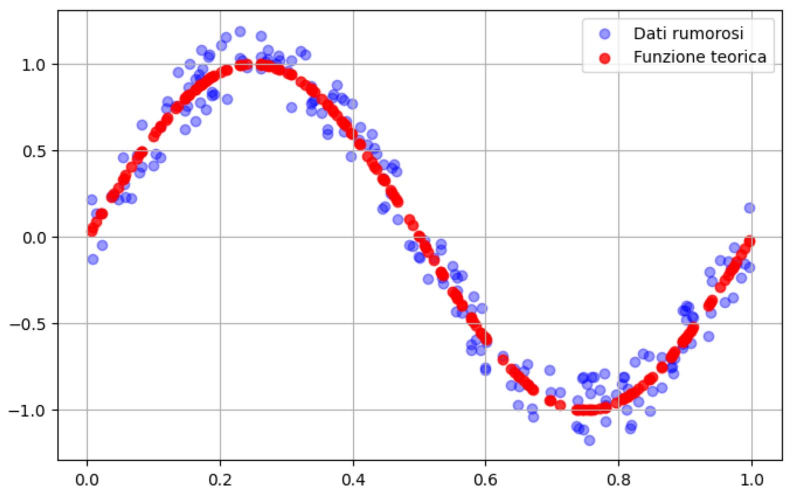

Figura 1: Dati - in blue- generati a partire da $sin(2\pi x)$ con l'aggiunta di un rumore uniforme. In rosso la funzione $sin(2\pi x)$.

2 Modello Polinomiale

```python
def pol(n, X=None):
    if X is None:
        X = np.linspace(0, 1, 100) # default: 100 punti fra 0 e 1

    # Coefficienti casuali tra -0.5 e 0.5
    coef = np.random.uniform(-0.5, 0.5, size=n+1)
    p = np.polynomial.Polynomial(coef)
    Z = p(X)

    return Z, coef
```

Si riporta una prova del metodo per creare vari polinomi:

```python
n = np.linspace(1,4,4)
for i in n:
    i = int(i)
    X = np.linspace(-3,3, 10000)
    Z, _ = pol(i,X)

    plt.scatter(X, Z, s=10, color='red', alpha=0.3, label=f'Polinomio di {i} grado')
    plt.grid(True)
    plt.legend()
    plt.show()
```

Si riportano i risultati grafici [Fig.2]:

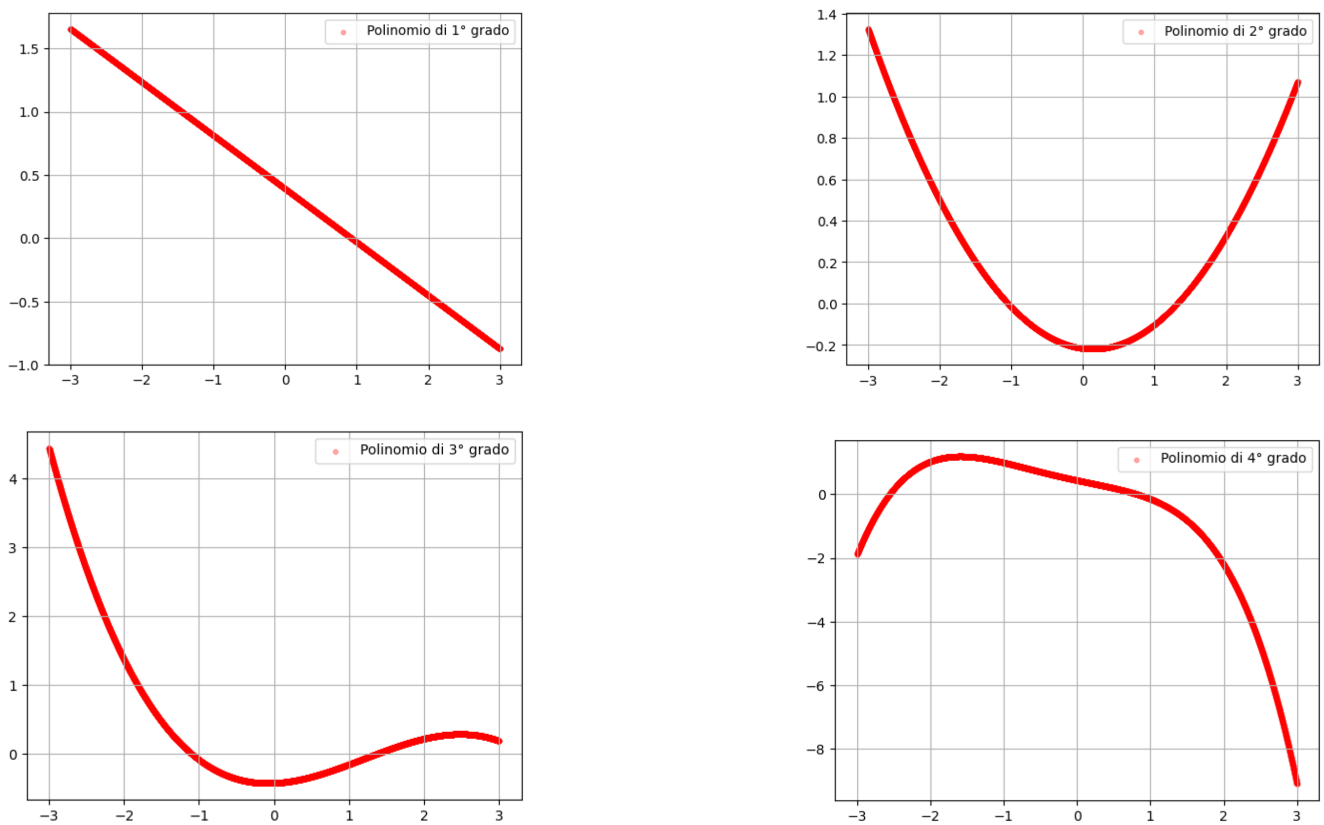

Figura 2: Polinomi di grado 1, 2, 3 e 4 in [-3,3]

Come ci si attende aumentare il grado modifica il numero di zeri del polinomio. Inoltre modifica la sua curvatura: aumentando il grado aumentano il numero di derivate che possiamo svolgere sul polinomio prima di ottenere il risultato nullo.

3 Discesa del Gradiente

Definite le funzioni costo e gradiente:

```python
def J(omega, Y, X, N):
    p =np.polynomial.Polynomial(omega)
    elements = (Y - p(X))**2
    return np.sum(elements)/(2*N)


def grad_J(omega, Y, X, N):
    M = len(omega)
    grad = np.zeros(M)
    error = (Y - np.polynomial.Polynomial(omega)(X))
    for j in range(M):
```

```
        # derivata rispetto a omega_j
            grad[j] = np.sum(error * X**j)
            grad[j] = -grad[j] / N

    return grad
```

Si è implementata la discesa del gradiente per un modello polinomiale di grado $n = 10$ iterata per un numero di $steps = 80000$ con vari valori del tasso di apprendimento $\eta = [0.1; 0.01; 0.001]$:

```
etas = [0.1, 0.01, 0.001]
steps = 80000
n = 10

results = {} # per salvare (omega_finale, J_history) per ogni eta

# Calcolo una sola volta per ogni eta
for eta in etas:
    p, omega = pol(n, X)
    J_history = []

    for i in range(steps + 1):
        omega = omega - eta * grad_J(omega, Y, X, N)
        cost = J(omega, Y, X, N)
        J_history.append(cost)

    results[eta] = (omega, J_history)

#Dati e polinomi
plt.figure()
plt.scatter(X, Y, color='red', alpha=0.4, label='Dati')

for eta, (omega, _) in results.items():
    plt.scatter(X, np.polynomial.Polynomial(omega)(X), alpha=0.6, label=f'Polinomio eta={eta}')

plt.legend()
plt.title('Dati e polinomi')
plt.grid(True)
plt.show()

# Funzione di costo
fig, ax = plt.subplots()

for eta, (_, J_history) in results.items():
    ax.plot(range(steps + 1), J_history, label=f'eta={eta}')

ax.set_xlabel('Step')
ax.set_ylabel('J(omega) - Funzione di costo')
ax.set_title('Evoluzione della funzione di costo')
ax.grid(True)
ax.legend()
plt.show()
```

Si riportano i risultati grafici [Fig.3]:

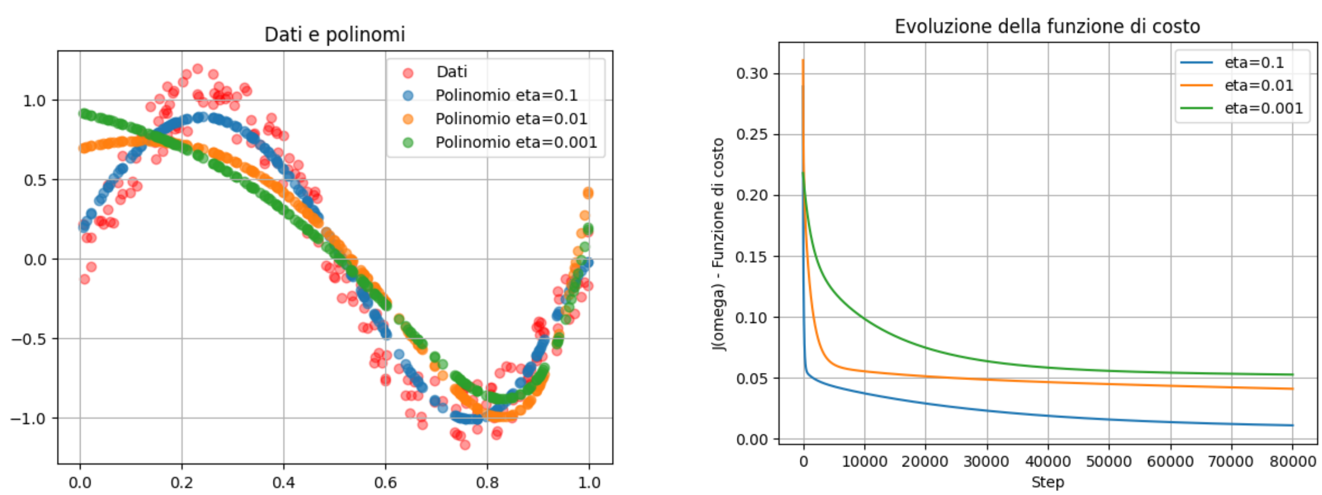

Figura 3: La figura a sinistra mostra i polinomi ottenuti dopo la discesa del gradiente, confrontandolo con i dati (in rosso). La figura a destra mostra l'andamento delle funzioni di costo durante la discesa del gradiente. In entrambi i grafici il blu indica un η di 0.1, arancione di 0.01, verde di 0.001.

Il tasso di apprendimento di $\eta = 0.1$ ha raggiunto un miglior risultato nell'apprendere i dati.

4 Monitorare la Convergenza

Si è considerato un unico valore di $\eta = 0.1$ e ne si è monitorata la convergenza.

```python
eta=0.1
steps=80000
n=10
p, omega = pol(n,X)
J_history = []

for i in range (steps+1):
    omega = omega - eta*grad_J(omega,Y,X,N)
    cost = J(omega, Y, X, N)
    J_history.append(cost)
    #Grafico Dati - Polinomio
    if (i % 8000 == 0 or i%80000 == 1):
        plt.scatter(X, Y, color='red',alpha=0.4, label='Dati')
        plt.scatter(X, np.polynomial.Polynomial(omega)(X),color='blue', label=f'Iterazione {i}')
        plt.legend()
        plt.grid(True)
        plt.show()

# Grafico della funzione di costo
plt.scatter(range(steps+1), J_history)
plt.xlabel('Step')
plt.ylabel('J(omega)')
plt.title('Evoluzione della funzione di costo')
plt.grid(True)
plt.show()

print(omega)
```

Si riportano i risultati grafici [Fig.4] e [Fig.5]:

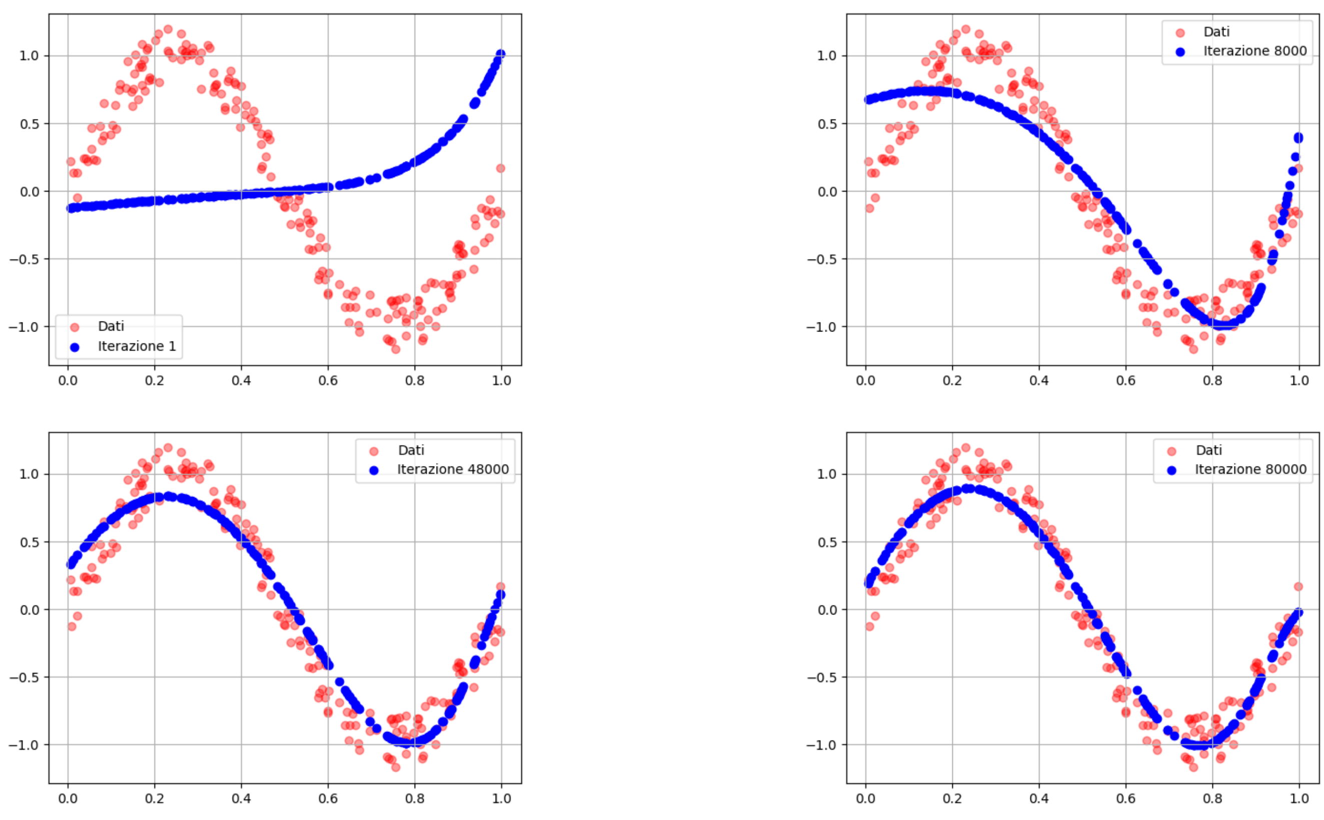

Figura 4: Varie iterazioni dell'implementazione della discesa del gradiente con $\eta = 0.1$. In rosso i dati, in blu il polinomio ottenuto in quella iterazione.

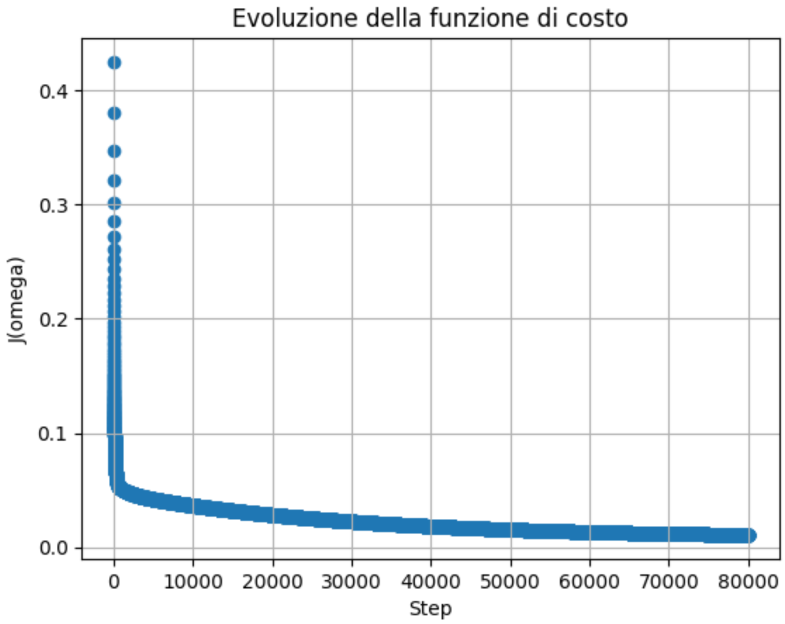

Figura 5: In blu l'andamento della funzione costo durante le iterazioni a $\eta = 0.1$.

5 Mini-Batch SGD

Si è implementato in primo luogo una funzione in grado di creare i batches:

```python
def create_batches(X, Y, batch_size):
    N_train = len(X)
    X_batches = []
    Y_batches = []

    for i in range(0, N_train, batch_size):
        X_batches.append(X[i:i + batch_size])
        Y_batches.append(Y[i:i + batch_size])

    return X_batches, Y_batches
```

Si è suddiviso il dataset in Train e Test e si riporta graficamente tale suddivisione [Fig.6]:

```python
test_ratio = 0.2
N_test = int(N * test_ratio)
N_train = N-N_test
indices = np.arange(N)
np.random.shuffle(indices)
X_train, Y_train = X[indices[N_test:]], Y[indices[N_test:]]
X_test, Y_test = X[indices[:N_test]], Y[indices[:N_test]]
#Plot Dati
plt.scatter(X_train, Y_train,color='blue', label='Dati Train')
plt.scatter(X_test, Y_test,color='green', label='Dati Test')
plt.legend()
plt.grid(True)
plt.show()
```

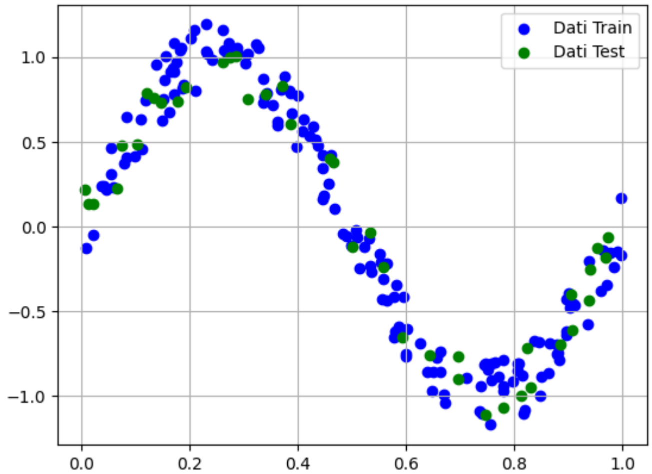

Figura 6: Grafico dei dati di partenza ora suddivisi in due gruppi. In blu i dati di train, in verde i dati di test.

Successivamente si è implementata la Mini-Batch SGD con varie dimensioni dei batches, ma ad $\eta = 0.1$, grado del polinomio $n = 10$ e $steps = 40000$

```
eta = 0.1
steps = 40000
n=10
p, omega0 = pol(n,X)
results = {}

batch_sizes = [1, 10, 20, 40, 80, 160]

for batch_size in batch_sizes:
    #partire dalle stessa situazione iniziale
    J_history=[]
    omega=omega0
    num_batches = int(N_train / batch_size)

    for i in range(steps+1):
        if(i%(num_batches)==0):
            indices = np.arange(N_train)
            np.random.shuffle(indices)
            X_train = X_train[indices]
            Y_train = Y_train[indices]
            X_batches, Y_batches = create_batches(X_train, Y_train, batch_size)
        index_batches=i%num_batches
        current_batch_size = len(X_batches[index_batches])
        omega = omega - eta * grad_J(omega,Y_batches[index_batches],X_batches[index_batches],current_batch_size)
        cost = J(omega, Y_train, X_train, N_train) #calcolato sui dati di train
        J_history.append(cost)

    results[batch_size] = J_history
```

Per studiare come influisca la dimensione dei batches sulla velocità di convergenza, stabilità e rumorosità del metodo si mette in evidenza come cambi la funzione costo.

```
#Grafico delle funzioni costo in scala log
plt.figure(figsize=(10,6))

for batch_size, J_history in results.items():
    plt.plot(range(len(J_history)), J_history, label=f'batch={batch_size}', linewidth=1)

plt.yscale('log')
plt.xlabel('Step')
plt.ylabel('J() - Funzione di costo - scala logaritmica')
plt.title('Confronto delle funzioni di costo per diversi batch size')
plt.grid(True, which='both')
plt.legend()
plt.show()

#Grafico: Normalizzazione per epoche
plt.figure(figsize=(10,6))
for batch_size, J_history in results.items():
    # ogni epoca = passaggio completo sui N_train
    num_batches = int(N_train / batch_size)
    #rinormalizzo il vettore delle iterazione per il numero di batches
    epochs = np.arange(len(J_history)) / num_batches
    plt.plot(epochs, J_history, label=f'batch={batch_size}', linewidth=1)
plt.xlabel('Epoche')
```

```python
plt.ylabel('J() - Funzione di costo')
plt.title('Funzione di costo normalizzata per epoche')
plt.grid(True, which="both", alpha=0.5)
plt.legend()
plt.show()
```

Risultati grafici:

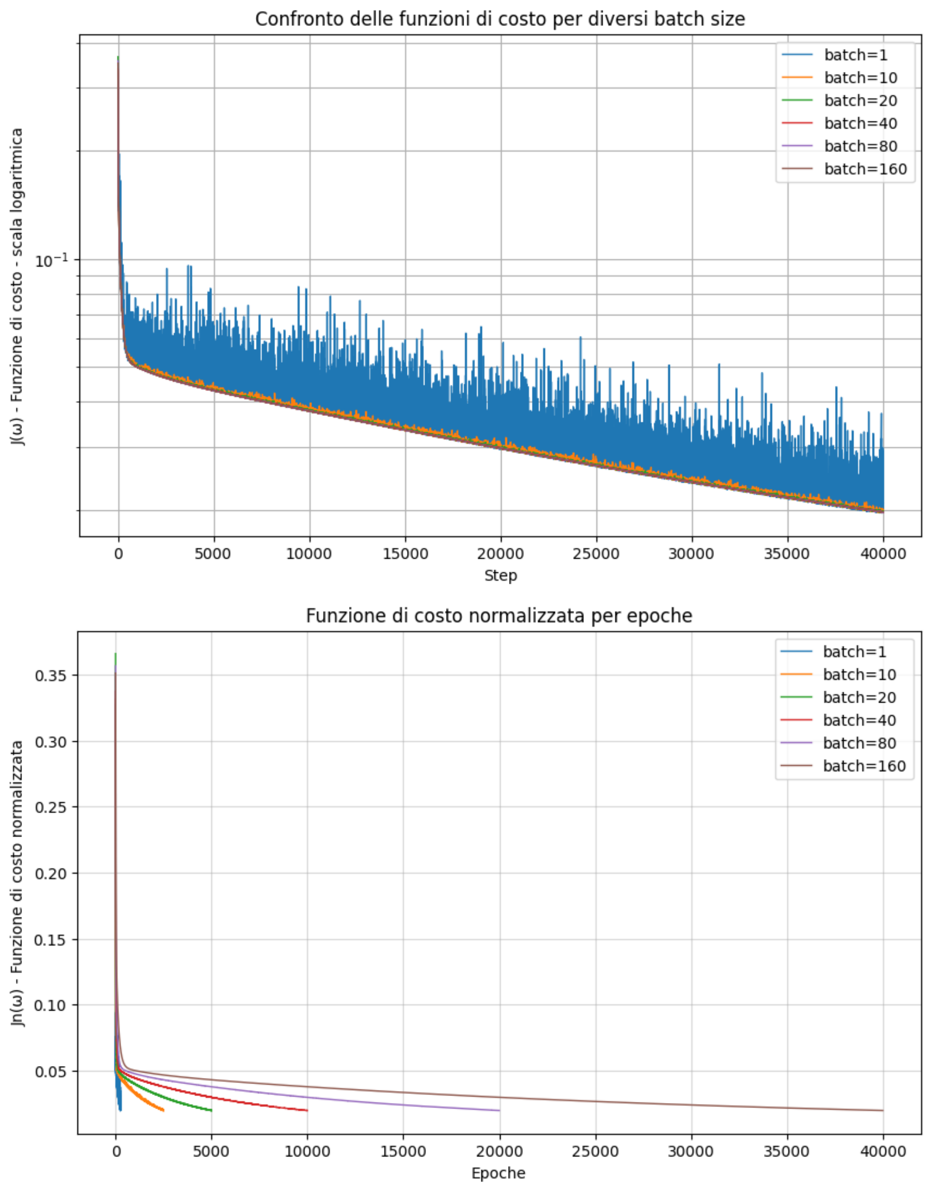

Figura 7: Il primo grafico mostra l'andamento della funzione di costo in scala logaritmica per varie dimensioni dei batch. Il secondo grafico mostra la funzione di costo normalizzata per epoche.

6 Riflessioni Finali

I grafici di [Fig.7] ci permettono di fare alcune valutazioni. Se la dimensione di ogni batch è molto piccola la funzione costo è molto rumorosa, mentre per dimensioni sempre maggiori essa diventa sempre più liscia. Questo permette di comprendere come la funzione costo durante le iterazioni sia più stabile nel caso di grandi batch. Il secondo grafico invece di permette di vedere come una dimensione piccola dei batch permetta di velocizzare la discesa del gradiente, ovvero in meno iterazioni riesce a far diminuire il valore della funzione di costo. Questo a discapito della sua stabilità come detto precedentemente.

6.1 Riflessioni sulla Complessità del modello

Durante il laboratorio ho sempre utilizzato $n = 10$ come grado del polinomio del modello. Si mostra un confronto fra errore commesso e complessità del modello considerando $steps = 40000$ e facendo variare il grado del modello monitorando l'ultimo valore della funzione J(ω)

```python
 eta = 0.1
steps = 40000
ns = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
J_last_train=[]
J_last_test=[]

for n in ns:
    p, omega = pol(n, X)
    for i in range(steps + 1):
        omega = omega - eta * grad_J(omega, Y_train, X_train, N_train)

        # Calcola errore su entrambi
        cost_train = J(omega, Y_train, X_train, N_train)
        cost_test = J(omega, Y_test, X_test, N_test)
        if(i==steps):
            J_last_train.append(cost_train)
            J_last_test.append(cost_test)
            print(f"fatto grado {n}")
```

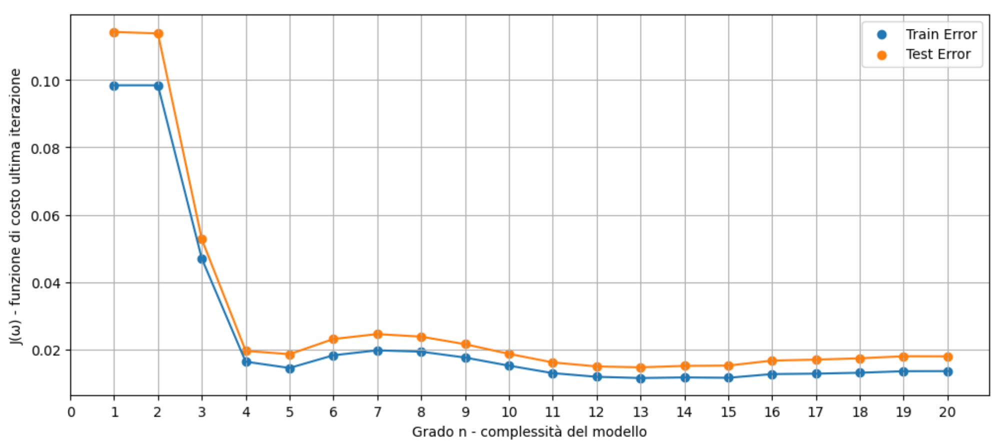

Figura 8: Il risultato grafico dello studio fra costo all'ultima iterazione e grado del polinomio usato come modello. In blu il calcolo del costo sui dati di Train, mentre in arancione sui dati di Test.

I primi modelli con grado troppo piccolo presentano un errore maggiore, poichè si tratta di una regione di underfit. Il grado 10 utilizzato durante la prova non presenta un'elevata differenza fra errore dei dati di train e test, quindi qualitativamente non è una zona di overfit. Si nota che per n > 12 l'errore di test inizia a crescere rispetto al train, suggerendo overfitting. Si può comunque dedurre in modo qualitativo che anche un polinomio di grado 5 avrebbe comunque appreso con un errore piccolo i dati, riducendo di molto i calcoli svolti dal calcolatore.