UNIVERSITÀ
DI PARMA

Corso Laurea Magistrale in Scienze Informatiche

Adam:
Tecniche Avanzate di Ottimizzazione nei Modelli di Machine Learning

Studente:
Rocco Persiani

Professoressa:
Eleonora Iotti

Anno Accademico 2023/2024

UNIVERSITÀ DI PARMA

Obiettivi del Progetto

Studiare la famiglia degli algoritmi di ottimizzazione Adam, implementando l'algoritmo di discesa del gradiente Adam su un Multi-Layer Perceptron a uno strato e a più strati, confrontando le prestazioni di quest'ultimo con l'algoritmo Stocastich Gradient Descent (SGD).

INDICE

UNIVERSITÀ DI PARMA

Nel campo del machine learning e dell’intelligenza artificiale, uno degli obiettivi principali è quello di sviluppare modelli che possano apprendere dai dati e fare previsioni accurate.

Il processo di apprendimento richiede l’ottimizzazione dei parametri del modello in modo che minimizzino una certa funzione di costo.

- La funzione di costo guida l’aggiornamento dei parametri del modello durante il processo di addestramento.

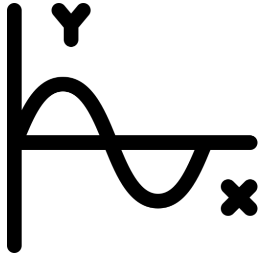
- Con l’aumentare delle dimensioni dei dati e della complessità dei modelli, si è cercato di sviluppare nuovi algoritmi che potessero rispondere in modo efficace alle nuove esigenze

L'algoritmo Stocastich Descend Gradient (SGD) si presta ad aggiornare iterativamente i parametri di un modello di machine learning in modo da ridurre gradualmente l'errore di previsione del modello durante l'addestramento.

Il gradiente è una misura che quantifica la pendenza di una funzione, ovvero indica la direzione di massima crescita o massima diminuzione in uno spazio multidimensionale.

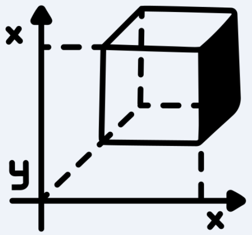
- L'algoritmo quantifica quindi il movimento verso il basso del gradiente di una funzione di costo.

- Il punto minimo di una funzione di costo corrisponde ai valori dei parametri per i quali essa raggiunge il suo valore più basso



La famiglia di algoritmi Adam è stata sviluppata per ottimizzare funzioni obiettivo stocastiche attraverso l'uso dei gradienti del primo ordine.

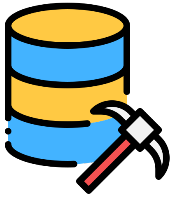
Per funzioni stocastiche si intendono funzioni che possono variare nel tempo o essere influenzate da rumore o varianza.



- L'introduzione di questi ottimizzatori ha il fine di rispondere a diverse sfide e limitazioni incontrate con l'uso del semplice algoritmo Stocastich Descend Gradient (SGD).

- Adam fa uso di stime adattive dei gradienti di ordine inferiore per migliorare efficienza e velocità della convergenza durante l'ottimizzazione

- Il dataset su cui siamo andati ad analizzare le prestazioni dei vari modelli presi in esame è il MNIST (Modified National Institute of Standards and Technology).



Esso è composto da cifre, che vanno da 0 a 9, scritte a mano con un training set composto da 60.000 esempi e un test set di 10.000 esempi.

- MNIST è comunemente utilizzato come benchmark per valutare l'accuratezza e le prestazioni di algoritmi di machine learning, in particolare per problemi di classificazioni di immagini

Pseudo-codice Adam:

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

- $\boldsymbol{\beta}_1$ controlla il decadimento esponenziale della media dei gradienti passati
- $\boldsymbol{\beta}_2$ gestisce il decadimento esponenziale della media dei quadrati dei gradienti
- ϵ è un termine di stabilizzazione che previene la divisione per zero

L'algoritmo proposto è una tecnica avanzata per aggiornare i parametri di un modello in modo efficiente durante l'addestramento, esso :

- Calcola il gradiente della funzione di costo rispetto ai parametri del modello
- Mantiene due stime dei momenti del gradiente
- Utilizza i parametri $\boldsymbol{\beta}_1, \boldsymbol{\beta}_2$ che aiutano l'algoritmo ad adattarsi dinamicamente alle variazioni nei gradienti durante l'ottimizzazione
- Combina le stime adattive dei momenti del gradiente per l'aggiornamento dei parametri del modello

I valori di default dei parametri che sono utilizzati per le stime adattive sono :

- $\boldsymbol{\beta}_1 = 0.9$
- $\boldsymbol{\beta}_2 = 0.999$
- $\epsilon = 1e\text{-}8$

Adamax, è un’estensione di Adam che a differenza di quest’ultimo utilizza la norma infinita per la stima dei momenti di secondo ordine anziché la norma quadratica

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
  $u_0 \leftarrow 0$ (Initialize the exponentially weighted infinity norm)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
    $\theta_t \leftarrow \theta_{t-1} - (\alpha/(1 - \beta_1^t)) \cdot m_t/u_t$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

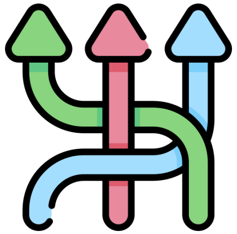
- La modifica rende l’algoritmo particolarmente robusto in presenza di gradienti di magnitudine variabile

Le differenze sostanziali che apporta Adamax rispetto ad Adam sono:

- Dal punto di vista computazionale Adamax è generalmente meno costoso poiché il calcolo della norma infinita è più semplice rispetto a quello della radice quadrata di Adam

- Per l'uso invece si preferisce Adamax in situazioni in cui i gradienti subiscono variazioni molto significative

- Il valore di default del parametro α è impostato a 0.002



In generale Adam risulta ampiamente utilizzato in molte applicazioni di machine learning e deep learning

L'implementazione è effettuata tramite Python, di seguito il codice:

```
def step(self, activations, deltas):

    gradient_W = [ np.zeros(w.shape) for w in self.weights ]
    gradient_b = [ np.zeros(b.shape) for b in self.biases ]

    num_input = len(activations)

    for l in range(self.num_layers, -1, -1):

        sum_w = 0
        for k in range(num_input):
            sum_w += np.dot( activations[k][l], np.transpose( deltas[k][l] ) )

        sum_b = 0
        for k in range(num_input):
            sum_b += deltas[k][l]

        gradient_W[l] = sum_w
        gradient_b[l] = sum_b

    self.weights = [
        w - (self.eta / num_input) * dw
        for w, dw in zip( self.weights, gradient_W )
    ]

    self.biases = [
        b - (self.eta / num_input) * db
        for b, db in zip( self.biases, gradient_b )
    ]
```

Stocastich Descend Gradient:

- Inizializzazione dei gradienti a 0
- Calcolo dei gradienti per ogni Layer
- Aggiornamento dei pesi e del bias

Il seguente codice è stato implementato a lezione

```python
def step(self, activations, deltas):

    gradient_W = [ np.zeros(w.shape) for w in self.weights ]
    gradient_b = [ np.zeros(b.shape) for b in self.biases ]

    num_input = len(activations)

    for k in range(num_input):
        a = activations[k]
        d = deltas[k]

        for l in range(self.n_layers,-1,-1):
            gradient_W[l] += np.dot(a[l], d[l].T)
            gradient_b[l] += d[l]

    gradient_W = [gw / num_input for gw in gradient_W]
    gradient_b = [gb / num_input for gb in gradient_b]

    self.t += 1

    for l in range(self.n_layers):
        self.m_weights[l] = self.beta1 * self.m_weights[l] + (1 - self.beta1) * gradient_W[l]
        self.v_weights[l] = self.beta2 * self.v_weights[l] + (1 - self.beta2) * (gradient_W[l] ** 2)
        m_hat_w = self.m_weights[l] / (1 - self.beta1 ** self.t)
        v_hat_w = self.v_weights[l] / (1 - self.beta2 ** self.t)
        self.weights[l] -= self.eta * m_hat_w / (np.sqrt(v_hat_w) + self.epsilon)

        self.m_biases[l] = self.beta1 * self.m_biases[l] + (1 - self.beta1) * gradient_b[l]
        self.v_biases[l] = self.beta2 * self.v_biases[l] + (1 - self.beta2) * (gradient_b[l] ** 2)
        m_hat_b = self.m_biases[l] / (1 - self.beta1 ** self.t)
        v_hat_b = self.v_biases[l] / (1 - self.beta2 ** self.t)
        self.biases[l] -= self.eta * m_hat_b / (np.sqrt(v_hat_b) + self.epsilon)
```

MLP Ottimizzazione Adam:

- Inizializzazione dei gradienti a 0
- Calcolo dei gradienti medi
- Incremento del Timestamp
- Calcolo dei momenti di primo e secondo ordine
- Calcolo delle correzioni dei bias
- Aggiornamento dei pesi e del bias

```python
def step(self, activations, deltas):

    gradient_W = [ np.zeros(w.shape) for w in self.weights ]
    gradient_b = [ np.zeros(b.shape) for b in self.biases ]

    num_input = len(activations)

    for k in range(num_input):
        a = activations[k]
        d = deltas[k]

        for l in range(self.n_layers,-1,-1):
            gradient_W[l] += np.dot(a[l], d[l].T)
            gradient_b[l] += d[l]

    gradient_W = [gw / num_input for gw in gradient_W]
    gradient_b = [gb / num_input for gb in gradient_b]

    self.t += 1

    for l in range(self.n_layers):
        self.m_weights[l] = self.beta1 * self.m_weights[l] + (1 - self.beta1) * gradient_W[l]
        self.u_weights[l] = np.maximum(self.beta2 * self.u_weights[l], np.abs(gradient_W[l]))
        m_hat_w = self.m_weights[l] / (1 - self.beta1 ** self.t)
        self.weights[l] -= (self.eta / (self.u_weights[l] + self.epsilon)) * m_hat_w

        self.m_biases[l] = self.beta1 * self.m_biases[l] + (1 - self.beta1) * gradient_b[l]
        self.u_biases[l] = np.maximum(self.beta2 * self.u_biases[l], np.abs(gradient_b[l]))
        m_hat_b = self.m_biases[l] / (1 - self.beta1 ** self.t)
        self.biases[l] -= (self.eta / (self.u_biases[l] + self.epsilon)) * m_hat_b
```

MLP Ottimizzazione AdaMax:

- Inizializzazione dei gradienti a 0
- Calcolo dei gradienti medi
- Incremento del Timestamp
- Calcolo dei momenti di primo e secondo ordine
- Calcolo della correzione del bias
- Aggiornamento dei pesi e del bias

Caratteristiche Algoritmo SGD :

• **Tasso di Apprendimento**	Fisso
• **Uso dei Momenti**	No
• **Efficienza**	Alta per piccoli dataset
• **Robustezza ai Gradienti**	Bassa
• **Memoria**	Bassa
• **Convergenza**	Lenta

Caratteristiche Algoritmo Ottimizzazione Adam :

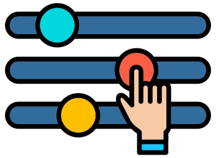
• **Tasso di Apprendimento**	Adattivo
• **Uso dei Momenti**	Si, primo e secondo ordine
• **Efficienza**	Alta per grandi dataset
• **Robustezza ai Gradienti**	Alta
• **Memoria**	Media
• **Convergenza**	Rapida

Caratteristiche Algoritmo Ottimizzazione AdaMax :

• **Tasso di Apprendimento**	Adattivo
• **Uso dei Momenti**	Si, primo e massimo del secondo ordine
• **Efficienza**	Alta per gradienti grandi
• **Robustezza ai Gradienti**	Molto Alta
• **Memoria**	Media
• **Convergenza**	Rapida e più stabile

Per il confronto delle prestazioni sul dataset MNIST abbiamo impostato gli iperparametri:

- **Numero di Epoche** : 10
- **Mini Batch Size**: 32
- **Learning Rate**: 0.1 / 0.001 / 0.002

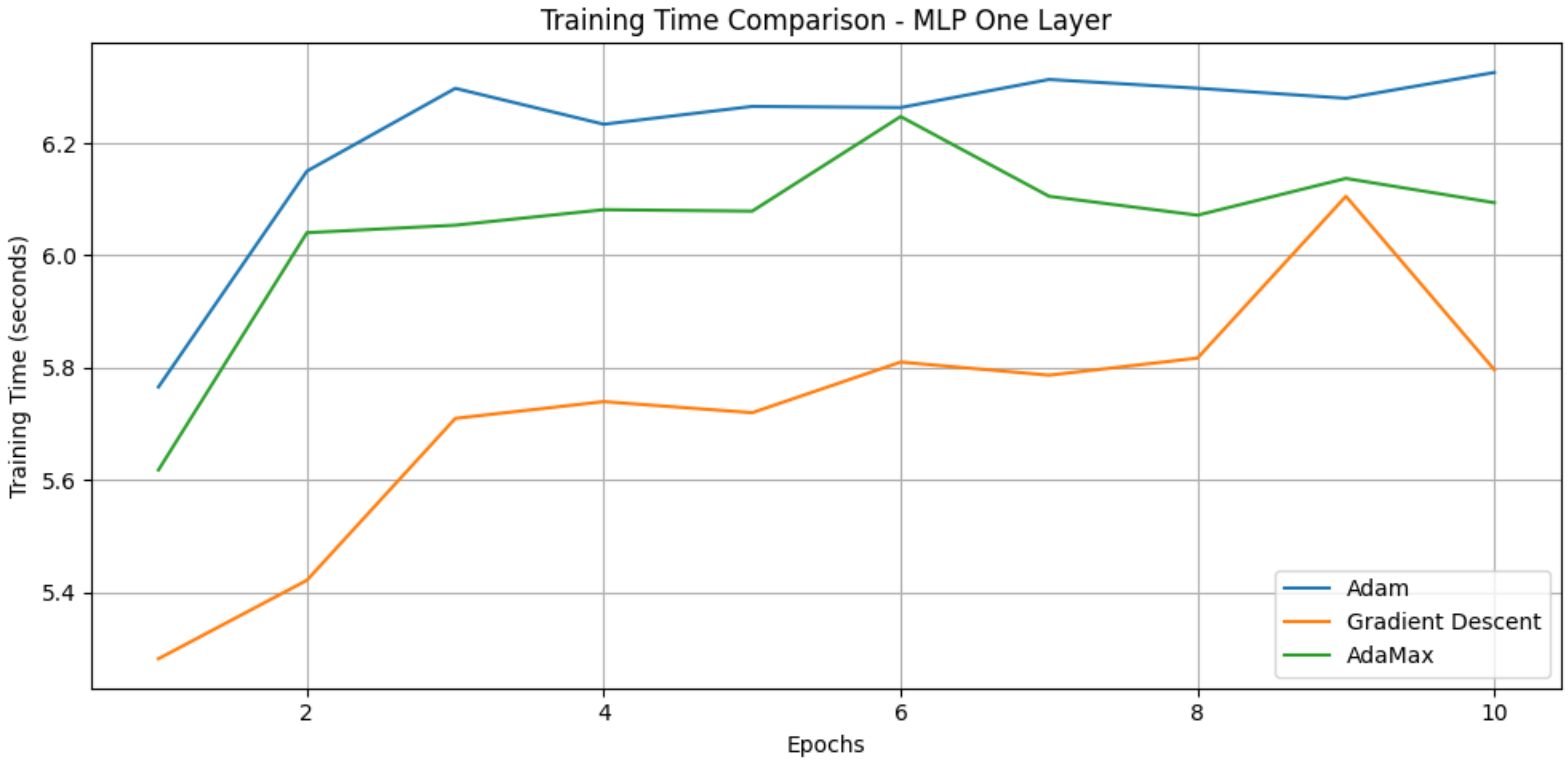


Per le prestazioni ci siamo soffermati in particolare su:

- **Tempo di Addestramento del modello per ogni Epoch**
- **Accuratezza del modello per ogni Epoch**

Confronto

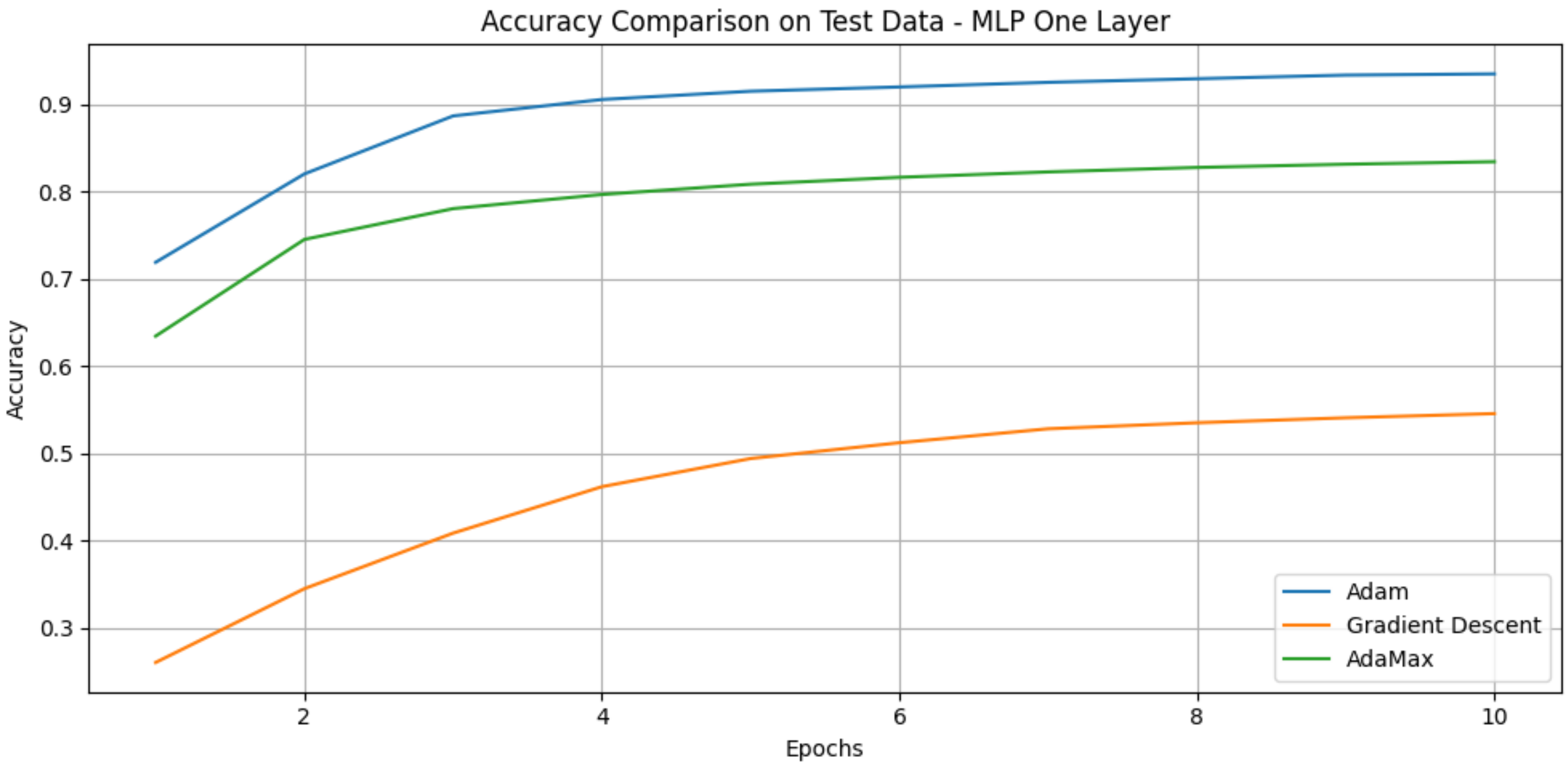
Grafico Tempo di Addestramento effettuato su MLP con uno strato nascosto:



- **Hidden Layer :** 60 neuroni

- Tempo Training Totale SGD : 57.19 secondi
- Tempo Training Totale Adam : 62.19 secondi
- Tempo Training Totale AdaMax : 60.53 secondi

Grafico Accuratezza per Epoca effettuato su MLP con uno strato nascosto:

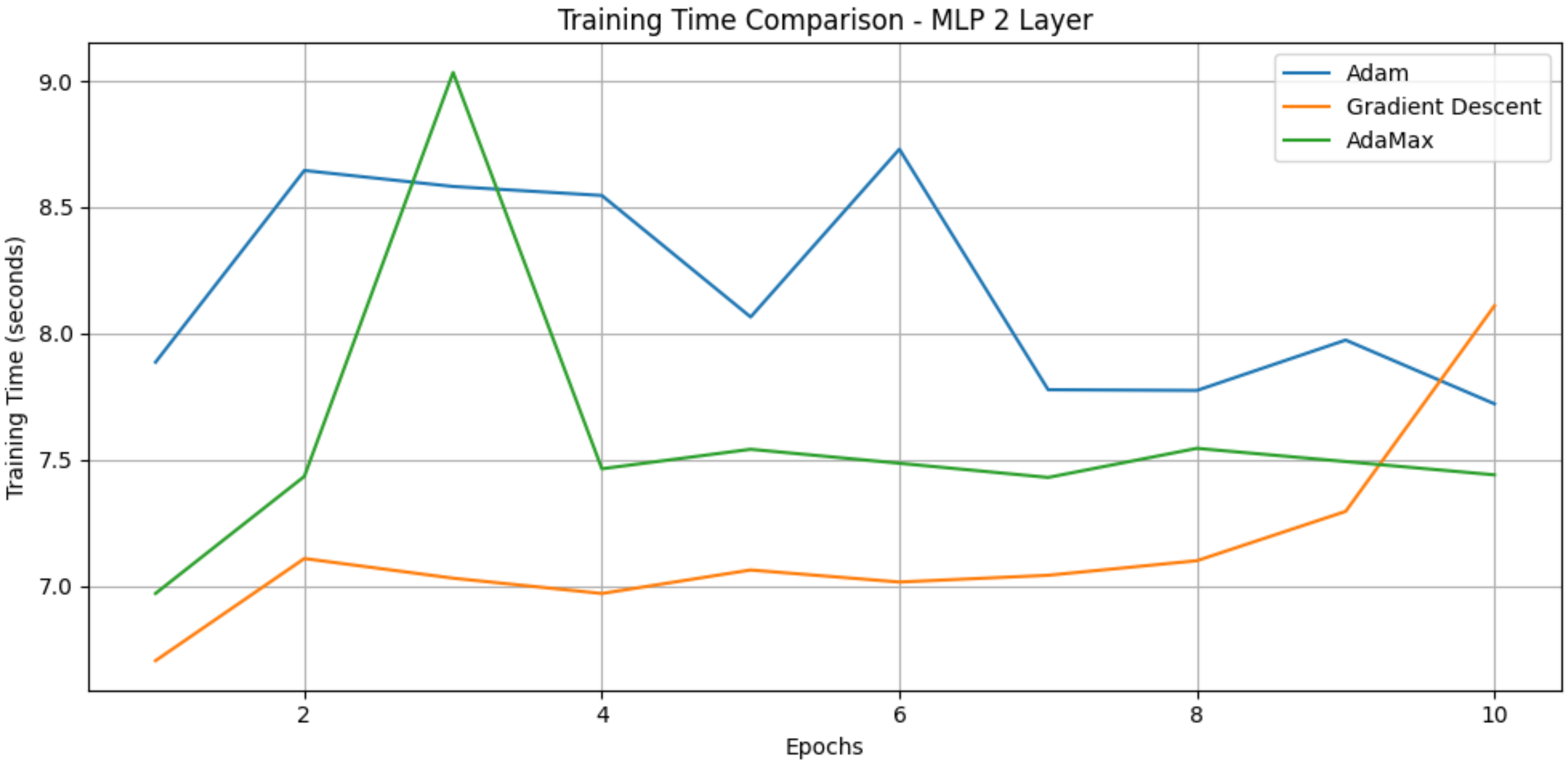


- **Hidden Layer :** 60 neuroni

- Accuracy SGD : 46.31%
- Accuracy Adam : 88.90%
- Accuracy AdaMax : 78.98%

Confronto

Grafico Tempo di Addestramento effettuato su MLP con due strati nascosti:

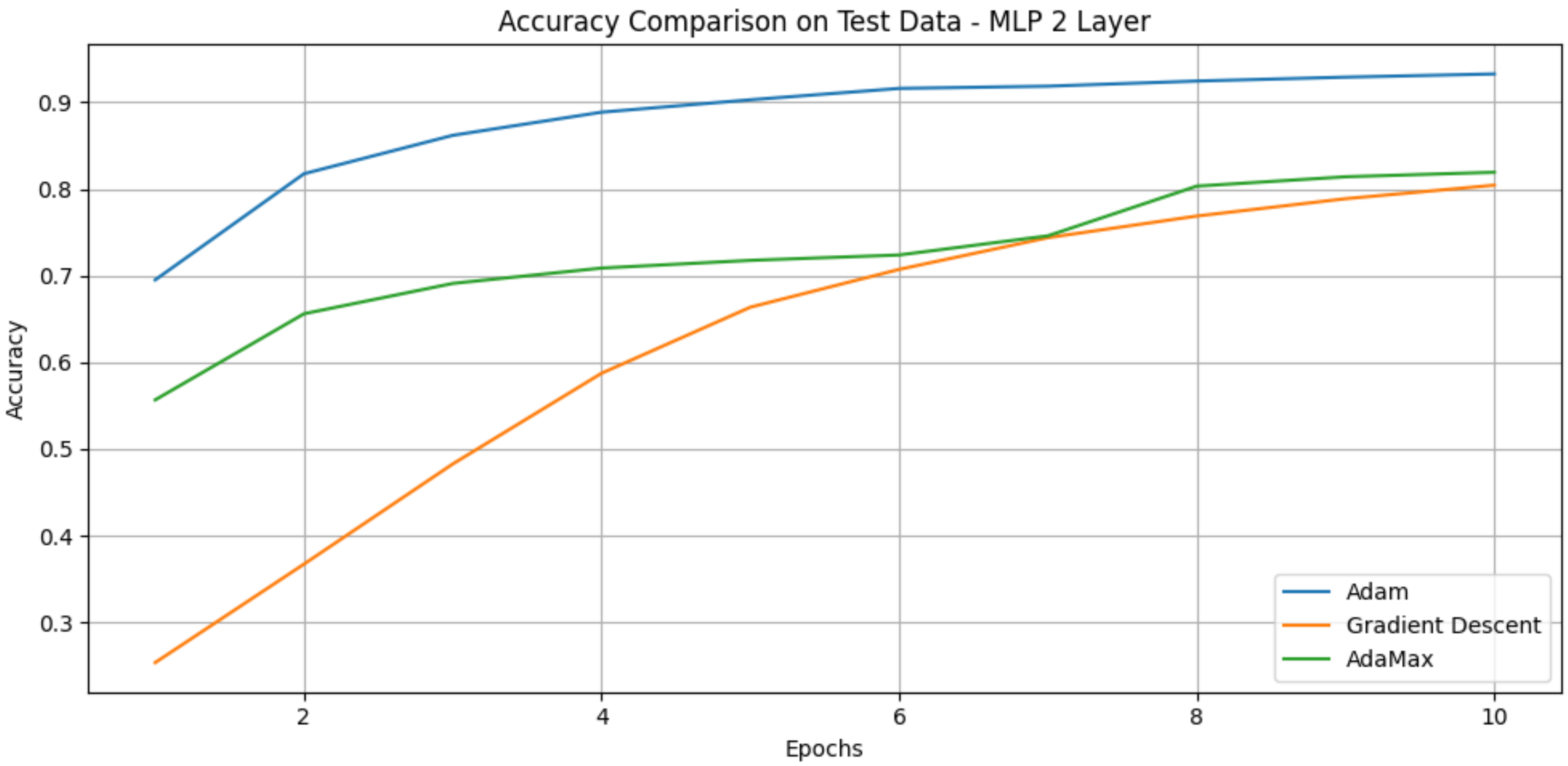


- **I Hidden Layer :** 60 neuroni
- **I I Hidden Layer:** 30 neuroni

- Tempo Training Totale SGD : 71.45 secondi
- Tempo Training Totale Adam : 81.71 secondi
- Tempo Training Totale AdaMax : 75.85 secondi

Confronto

Grafico Accuratezza per Epoca effettuato su MLP con uno strato nascosto:



- **I Hidden Layer :** 60 neuroni
- **I I Hidden Layer:** 30 neuroni

- Accuracy SGD : 61.68%
- Accuracy Adam : 87.86%
- Accuracy AdaMax : 72.36%

Confronto

In **conclusione**, osservando i grafici e i risultati ottenuti, possiamo dedurre che:

- ✓ I **tempi di training** risultano più lunghi per gli algoritmi Adam e AdaMax che per SGD

- ✓ L'**accuratezza** più elevata è ottenuta dall'algoritmo Adam con rispettivamente l'87% e l'88%

- ✓ Pertanto l'algoritmo Adam ha mostrato prestazioni superiori rispetto agli altri due algoritmi in termini di accuratezza, mentre SGD si è confermato più veloce confermando la sua semplicità e alla minore complessità computazionale

Grazie per la vostra attenzione