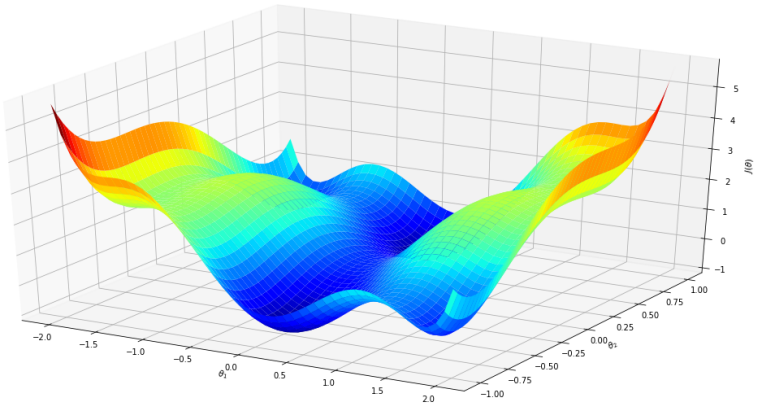
Deep learning: un problema di ottimizzazione

Bruno Neri

Dicembre 2019



Introduzione

Il concetto di ottimizzazione gioca un ruolo chiave quando si parla di machine learning e deep learning in particolare. Lo scopo principale degli algoritmi di deep learning è quello di costruire un modello di ottimizzazione che, tramite un processo iterativo, minimizzi o massimizzi una funzione obiettivo $J(\theta)$ denominata anche **loss function** o **cost function**.

I più popolari metodi di ottimizzazione possono essere suddivisi in due categorie: metodi di ottimizzazione del primo ordine, rappresentati dal metodo del gradiente, e metodi di ottimizzazione del secondo ordine o di ordine superiore fra i quali il metodo di Newton ne è un tipico esempio.

In queste note analizzeremo i metodi di ottimizzazione del primo ordine, dei quali il metodo della discesa del gradiente stocastico e tutte le sue varianti sono fra quelli ampliamente utilizzati ed in continua evoluzione.

Ci occuperemo delle loro caratteristiche con l'auspicio di stimolare nel lettore il giusto spirito critico per scelta del metodo più appropriato e renderne più

ragionevole la calibrazione dei parametri.

Gli algoritmi di ottimizzazione trattati nelle seguenti note sono disponibili nei framework di deep learning quali Keras, Tensorflow e PyTorch. Gli esempi riportati saranno implementati in python con l'utilizzo del framework Keras

Metodi di ottimizzazione del primo ordine

Nei modelli di machine learning e deep learning i metodi di ottimizzazione del primo ordine principalmente utilizzati sono basati sul concetto di discesa del gradiente. In questa sezione introdurremo gli algoritmi più rappresentativi.

La discesa del gradiente (batch)

Il metodo della discesa del gradiente è il primo e più comune metodo di ottimizzazione. L'algoritmo si basa sull'aggiornamento iterativo di un parametro θ lungo la direzione opposta a quella del gradiente della funzione obiettivo $J(\theta)$. L'aggiornamento viene sviluppato in modo da convergere gradualmente al valore ottimo della funzione obiettivo. Uno dei parametri del metodo è il learning rate η che determina l'ampiezza della variazione di θ in ciascuna iterazione e quindi influenza il numero di iterazioni necessarie a raggiungere il valore ottimo della funzione obiettivo.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta) \tag{1}$$

Il metodo è di semplice implementazione e nel caso in cui $J(\theta)$ sia una funzione convessa, la soluzione trovata sarà un punto di ottimo globale. Nel caso in cui la funzione obiettivo non sia convessa, la soluzione trovata potrebbe essere un minimo locale. Il nome deriva dal fatto che, ad ogni iterazione, l'algoritmo impiega tutti i dati del training set per calcolare il gradiente $\nabla_\theta J(\theta)$

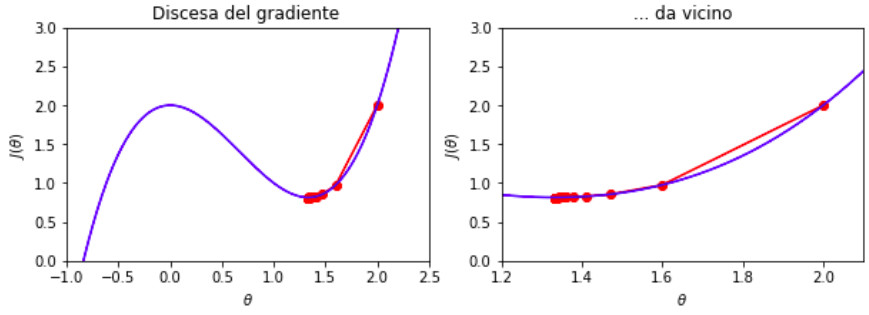


Figure 1: Discesa del gradiente

Discesa del gradiente stocastico I (SGD)

Nel metodo appena descritto il gradiente $\nabla_\theta J(\theta)$ è calcolato utilizzando, ad ogni iterazione, tutto il training set. Questo determina una elevata e ridondante complessità computazionale e non consente di effettuare gli update online di θ.

Per ovviare a questo punto debole la soluzione proposta è stata quella del metodo della discesa del gradiente stocastico. L'idea è quella di calcolare il gradiente non più mediante tutto il training set ma mediante un singolo campione selezionato in modo casuale ad ogni iterazione.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2}$$

L'utilizzo di singolo campione determina una forte diminuzione della complessità computazionale che in ciascuna iterazione adesso è dovuta esclusivamente al numero D delle feature del training set. La stocasticità nella selezione dei campioni dal training set determina anche che la ricerca della soluzione non rimanga *"intrappolata"* in un minimo locale di $J(\theta)$.

Di seguito un esempio in python dell'utilizzo di SGD fornito da Keras:

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=0.001),
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10,
                   validation_data=(x_valid, y_valid))
```

Discesa del gradiente stocastico II (mini-batch)

L'utilizzo di una selezione casuale di singoli campioni dal training set ha lo svantaggio di determinare una oscillazione della direzione del gradiente e un procedere in modo cieco del processo di ricerca della soluzione all'interno dello spazio delle soluzioni.

Un modo per diminuire la varianza del gradiente è stato quello di introdurre una variante denominata mini-batch gradient descent. Questa modalità impiega, ad ogni iterazione, un insieme di n campioni del training set e con questi calcola il gradiente $\nabla_\theta J(\theta)$. In questo modo si raggiunge il duplice obiettivo di ridurre la varianza del gradiente e rendere più stabile la convergenza.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3}$$

Il codice python seguente è un esempio di implementazione del mini batch SGD utilizzando Keras:

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=0.001),
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10,batch_size=64
          validation_data=(x_valid, y_valid))
```

Metodo del momento

Nonostante il metodo SGD sia molto popolare ed ampiamente utilizzato, il processo di aggiornamento di θ risulta spesso molto lento.

Fra i punti aperti infatti c'è una più opportuna regolazione del learning rate per velocizzare la convergenza e fare in modo che il processo di ricerca della soluzione non rimanga intrappolato in un minimo locale di $J(\theta)$

Una delle idee che si è fatta strada è quella di *"momento"* che, per come è stato pensato, gioca il ruolo di una velocità v. Il concetto è infatti derivato dalla fisica, in particolare dalla meccanica, ed è stato pensato per accellerare il processo di aggiornamento di θ, specialmente in casi di elevate curvature di $J(\theta)$ e di valori piccoli, costanti e rumorosi del gradiente $\nabla_\theta J(\theta)$. L'idea è quella di calcolare, ad ogni iterazione, una media mobile esponenziale dei gradienti storici ed utilizzare questo valore come direzione da seguire nell'aggiornamento di θ. Il parametro $\beta \in [0, 1)$ determina la velocità di decadimento dei contributi dei gradienti storici.

$$\begin{aligned} v_t &= \beta v_{t-1} - \eta \nabla_\theta J(\theta) \\ \theta_{t+1} &= \theta_t + v_t \end{aligned}$$

Di seguito il codice python per l'implementazione del metodo del momento utilizzando Keras, in cui il parametro momentum corrisponde al parametro β della funzione precedente:

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=0.001, momentum=0.9),
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10,batch_size=64,
          validation_data=(x_valid, y_valid))
```

Momento di Nesterov

Una variante del metodo del momento è quella denominata *Nesterov accelerated gradient*. L'idea alla base di questa variante è quella di valutare il gradiente non più alla posizione attuale θ bensi in una posizione $\theta + \beta v_{t-1}$ un pò più

avanzata lungo la direzione del momento v_{t-1}. Anche questo algoritmo utilizza un parametro $\beta \in [0, 1)$ che, come per il metodo standard del momento, determina la velocità di decadimento dei contributi dei gradienti storici.

$$\begin{aligned} \tilde{\theta}_t &= \theta_t + \beta v_{t-1} \\ v_t &= \beta v_{t-1} - \eta \nabla_\theta J(\tilde{\theta}) \\ \theta_{t+1} &= \theta_t + v_t \end{aligned}$$

Il codice che segue implementa il metodo di Nesterov in Keras:

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True),
              metrics=["accuracy"])

model.fit(x_train, y_train, epochs=10,batch_size=64,
          validation_data=(x_valid, y_valid))
```

Metodi di learning rate adattativo

Risulta evidente quanto una opportuna regolazione del learning rate possa avere una forte influenza sugli effetti del metodo SGD. A tal riguardo sono stati proposti diversi meccanismi adattativi per una regolazione automatica del learning rate

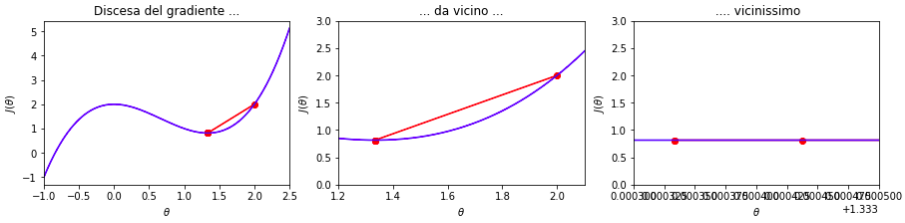


Figure 2: Learnig Rate adattivo

Un importante miglioramento all'algoritmo della discesa del gradiente stocastico è dovuto al metodo AdaGrad [2] che implementa una regolazione dinamica del learning rate basandosi sui valori storici del gradiente.

La differenza tra AdaGrad e la classica discesa del gradiente risiede nel fatto che, durante il processo di aggiornamento di θ, il learning rate non sarà più costante ma sarà continuamente ricalcolato utilizzando i valori storici del gradiente accumulati fino alla corrente iterazione. Lo svantaggio di questo procedimento deriva dal fatto che, nei casi di training time molto lungo, l'elevato valore cumulativo dei gradienti farà tendere a zero il valore del learning rate e di conseguenza θ tenderà ad un valore stazionario non corretto.

$$
\begin{aligned}
g_t &= \nabla_{\theta_t} J(\theta_t) \\
v_t &= \sqrt{\sum_{i=1}^{t} (g_i)^2 + \epsilon} \\
\theta_{t+1} &= \theta_t - \eta \frac{gt}{v_t}
\end{aligned}
$$

Il problema della tendenza di θ alla non corretta stazionarietà è stato affrontato e risolto da Geoffrey Hinton con il metodo RMSProp [2]. Questo algoritmo utilizza una finestra temporale dei gradienti storici e ne calcola, ad ogni iterazione, il momento cumulativo del secondo ordine (la varianza).

$$
v_t = \beta v_{t-1} + (1 - \beta) g_t^2
$$

E' possibile utilizzare il metodo RMSProp di Keras implementando il codice seguente. Da notare che adesso il parametro $\beta \in [0, 1)$ di decadimento dei gradienti corrisponde al parametro *rho* dell'ottimizzatore RMSProp di Keras:

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.RMSprop(lr=0.001, rho=0.9),
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10,batch_size=64,
                    validation_data=(x_valid, y_valid))
```

Il metodo Adam (Adaptive momentum estimation)[4] introduce un ulteriore miglioramento a quanto visto in precedenza. Oltre a memorizzare la media mobile esponenziale dei quadrati dei gradienti delle precedenti iterazioni, viene memorizzata anche la media mobile esponenziale dei gradienti $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$. Il nome dell'algoritmo deriva dal fatto che, ad ogni iterazione, m_t e v_t corrispondono al momento primo (la media) ed al momento secondo (la varianza) del gradiente.

$$
\begin{aligned}
g_t &= \nabla_{\theta_t} J(\theta_t) \\
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
\alpha_t &= \alpha \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} \\
\theta_{t+1} &= \theta_t - \alpha_t \frac{m_t}{\sqrt{v_t} + \epsilon}
\end{aligned}
$$

Nella seguente implementazione i parametri β_1 e $\beta_2 \in [0, 1)$ di decadimento dei momenti del primo e secondo ordine assumono i valori di default indicati dagli autori in [4].

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999),
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10,batch_size=64,
                    validation_data=(x_valid, y_valid))
```

Esempio: addestramento di una CNN

Fin qui ci siamo occupati prettamenti di aspetti teorici degli algoritmi di ottimizzazione. Vedremo adesso gli effetti di ciascuno di essi ha nell'addestramento di una convolutional neural network.

Nell'esempio proposto verrà addestrata una Deep Convolutional Neural Network pensata per la classificazione delle immagini presenti nel dataset Fashion MINST. Fashion MINST è un dataset prodotto da Zalando Research e contiene immagini di articoli di abbigliamento.

Il dataset è composto da un training set 60000 esempi e un test set di 10000 esempio. Ciascun esempio è composto da una immagine 28x28 in scala di grigio associata ad una etichetta appartenente ad un insieme di 10.

Di seguito l'implementazione della rete come modello sequenziale di Keras:

```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=64, kernel_size=2, padding='same',
                        activation='relu', input_shape=(28,28,1)),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Dropout(0.3),
    keras.layers.Conv2D(filters=32, kernel_size=2, padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Dropout(0.3),
    keras.layers.Flatten(),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])
```

Per poter iniziare l'addestramento della rete servirà prima compilare il modello associando una loss function, un metodo di ottimizzazione ed una metrica che ci consenta di valutare la bontà del training.

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=0.001),
              metrics=["accuracy"])
```

Il nostro modello di classificazione è del tipo multiclass, pertanto utilizzeremo una loss function di tipo *categorical_crossentropy*. Il metodo di ottimizzazione utilizzato per la fase di training è SGD con un learning rate pari a 0.001. Per la valutazione della bontà della modello la metrica di riferimento utilizzata sarà l'accuratezza.

Una volta che il modello è stato compilato siamo pronti per avviare la fase di training:

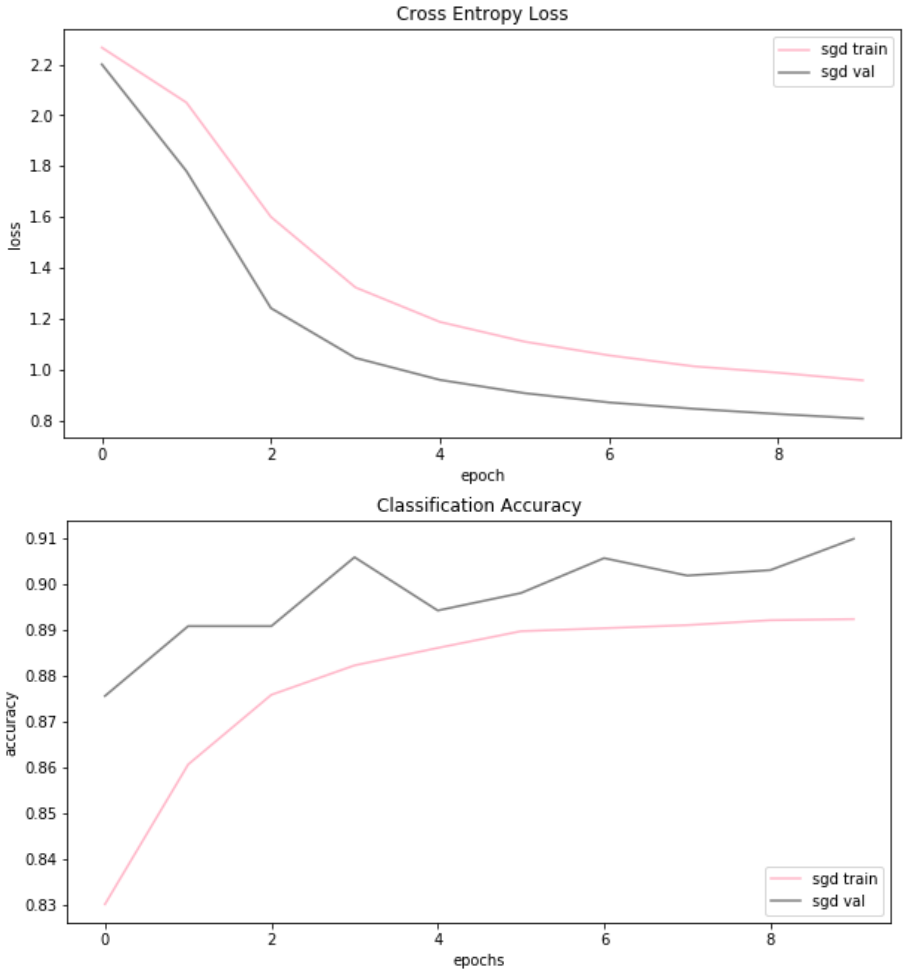
```
model.fit(x_train, y_train, epochs=10,batch_size=64,
                    validation_data=(x_valid, y_valid))
```

Il metodo fit si occuperà di avviare la fasi di training e validazione del modello utilizzando i seguenti parametri: * x_train: training set di 55000 immagini 28x28 * y_train: training set di 55500 label * epochs: numero di iterazioni, 10 nel nostro esempio * batch_size= dimensione del mini batch di esempi utilizzati per il calcolo del gradiente * validation_data: validation set per la valutazione del modello composto da 5000 immagini 28x28 e da 5000 label.

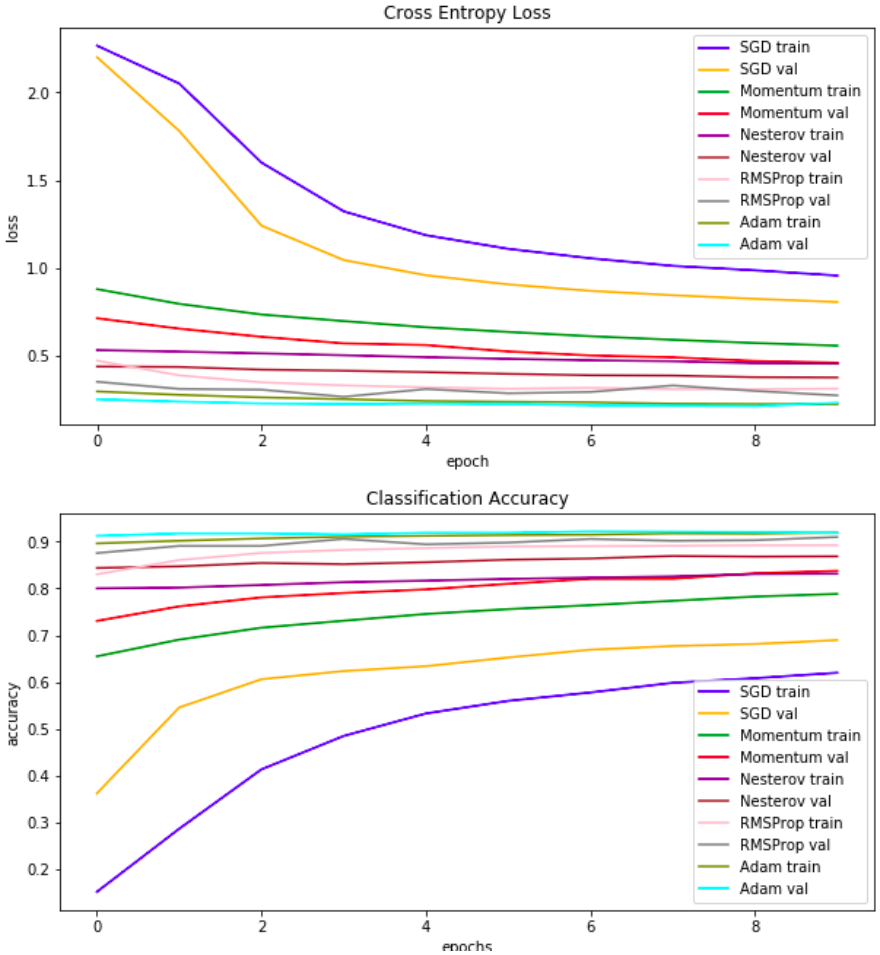
```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 80s 1ms/sample
    - loss: 2.2660 - accuracy: 0.1528 - val_loss: 2.2002 - val_accuracy: 0.3628
Epoch 2/10
55000/55000 [==============================] - 80s 1ms/sample
    - loss: 2.0501 - accuracy: 0.2872 - val_loss: 1.7805 - val_accuracy: 0.5460
Epoch 3/10
55000/55000 [==============================] - 81s 1ms/sample
    - loss: 1.6004 - accuracy: 0.4140 - val_loss: 1.2416 - val_accuracy: 0.6064
Epoch 4/10
55000/55000 [==============================] - 80s 1ms/sample
    - loss: 1.3230 - accuracy: 0.4853 - val_loss: 1.0459 - val_accuracy: 0.6238
Epoch 5/10
55000/55000 [==============================] - 79s 1ms/sample
    - loss: 1.1873 - accuracy: 0.5333 - val_loss: 0.9594 - val_accuracy: 0.6342
Epoch 6/10
55000/55000 [==============================] - 79s 1ms/sample
    - loss: 1.1098 - accuracy: 0.5602 - val_loss: 0.9073 - val_accuracy: 0.6528
Epoch 7/10
55000/55000 [==============================] - 82s 1ms/sample
    - loss: 1.0559 - accuracy: 0.5781 - val_loss: 0.8708 - val_accuracy: 0.6694
Epoch 8/10
55000/55000 [==============================] - 82s 1ms/sample
    - loss: 1.0129 - accuracy: 0.5987 - val_loss: 0.8459 - val_accuracy: 0.6772
Epoch 9/10
55000/55000 [==============================] - 81s 1ms/sample
    - loss: 0.9877 - accuracy: 0.6086 - val_loss: 0.8252 - val_accuracy: 0.6816
Epoch 10/10
```

```
55000/55000 [==============================] - 81s 1ms/sample
    - loss: 0.9578 - accuracy: 0.6202 - val_loss: 0.8073 - val_accuracy: 0.6898
```

Analizzando i risultati delle singole iterazioni si nota chiaramente come il valore della loss function diminuisca ed il valore dell'accuratezza aumenti, come evidenziato anche dai grafici seguenti:



Per effettuare un confronto più puntuale delle prestazioni dei metodi di ottimizzazione analizzati in queste note, il modello di rete convoluzionale è stato addestrato più volte utilizzando diversi ottimizzatori. I grafici seguenti evidenziano l'andamento delle loss function e dell'accuratezza prodotti da ciascuno dei metodi analizzati



Come si può notare chiaramente dai grafici, le prestazioni, sia in termini di velocita di convergenza sia di accuratezza della classificazione, crescono costantemente all'aumentare della complessità dell'ottimizzatore utilizzato. Risulta evidente la scarto esistente tra SGD e tutti gli altri algoritmi.

Conclusioni

Sono stati presentati i metodi di ottimizzazione del primo ordine utilizzati per l'addestramento di modelli di deep learning. Partendo dal metodo del gradiente si sono analizzati i metodi più complessi che, grazie anche ad un aggiornamento adattativo del learning rate, consentono al processo di addestramento di migliorare le prestazioni in termini di velocità di convergenza.

Un aspetto non trattato ma degno di nota è quello relativo alle effettive prestazioni dei metodi di adaptive learning rate in contesti diversi e magari più complessi. Il

lettore interessato può trovare in [5] gli spunti per un ulteriore approfondimento personale.

Il codice sorgente dell'esempio è riportato nel notebook colab [7]

Riferimenti

[1] On the importance of initialization and momentum in deep learning

[2] Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

[3] RMSProp by Geoffrey Hinton on lecture 6e Coursera class

[4] Adam: A Method for Stochastic Optimization

[5] The Marginal Value of Adaptive Gradient Methods in Machine Learning

[6] Keras optimizers

[7] Notebook Colab: Esempio di classificazione immagini Fashion MINST