*Relazione esame di*

*Neural networks and Deep Learning*

Giuseppe Porcaro N97000362

a.a. 2020/2021

Definizione delle parti del progetto:

* Parte A: Progettazione e implementazione di funzioni per simulare una rete neurale feed forward multistrato con possibilità di implementare reti con più di uno strato interno con qualunque funzione di attivazione. Implementazione dell’algoritmo di *back propagation* per qualunque funzione di attivazione e la possibilità di usare la *somma dei quadrati* o la *cross-entropy* con o senza softmax come funzione di errore.
* Parte B (traccia 3): Considerando come input il dataset mnist, si ha un problema di classificazione a C = 10 classi, per il quale si deve estrarre opportunamente un dataset di N coppie. Si fissi come algoritmo di aggiornamento la *resilient back propagation (RPROP)* e una rete neurale con un unico strato di nodi interni. Si scelgano gli iper-parametri del modello su un approccio di *k-fold*, scegliendo e tenendo fissi tutti gli altri “parametri” della rete.

Tecnologie usate

Entrambe le parti del progetto sono state realizzate nel linguaggio Python in modo tale da poter sfruttare principalmente la libreria Numpy, la quale fornisce supporto ed efficienza computazionale a tutte le principali operazioni su array e matrici di grandi dimensioni, incluse operazioni matematiche, di manipolazione, selezione, trasformazione e algebra lineare.

Per quanto riguarda l’implementazione delle funzioni per la simulazione delle reti, così come per l’algoritmo di back propagation, RPROP, funzioni di attivazione ed errore con relative derivate, questa è stata effettuata partendo da zero seguendo principalmente le metodologie viste a lezione. Altre librerie, le quali sono state usate per compiti più specifici, come ad esempio *Matplotlib* per il plot, verranno discusse nei seguenti paragrafi.

Parte A

Implementazione della rete neurale

La rete è una struttura dati per la quale, alla creazione, bisogna specificare:

* Il numero di valori in input per un singolo dato del dataset.
* Il numero di strati interni (minimo 1).
* Il numero di neuroni per ogni strato hidden (tramite array dato in input: **arrNumNeuroni**).
* La funzione di attivazione per ogni strato hidden (tramite array dato in input: **arrFunAttivazioneHidden**)
* Le derivate delle funzioni di attivazione per ogni strato hidden (sempre tramite array)
* La funzione di attivazione per lo strato di output.

Pesi e bias sono memorizzate in matrici Numpy e l’implementazione è la seguente:

* Primo layer hidden:

Dove m1 è il numero nodi dello strato, d è il numero di dati in input

* Dal secondo all’ultimo layer hidden:

Dove *i* = , *mi* è il numero di nodi dell’i-esimo strato hidden.

* Strato output:

Dove *c* è il numero di neuroni dello strato di output e *mk* è il numero di neuroni dello strato hidden precedente a quello output

Tutti i valori sono generati randomicamente nell’intervallo [-1,1]. Infine nella rete sono state memorizzate in semplici variabili tutte le informazioni base della rete quali numero di layer hidden, il numero di neuroni per ogni layer hidden, il numero di neuroni per lo strato di output e tutte le funzioni di attivazione per ogni strato.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 1. Funzione costruttrice di una rete neurale multi strato full connected.

Implementazione della simulazione in avanti

La simulazione in avanti della rete è stata implementata nelle funzioni *simulaRete* e *forwardStep* presenti nel file *backPropagation.py.* Entrambe le funzioni hanno lo stesso comportamento tranne per il fatto che la prima restituisce solo l’output della rete, mentre la seconda memorizza e restituisce in array tutti gli output di ogni strato della rete, insieme alle somme pesate dell’input. Nel dettaglio il comportamento della rete segue quello visto a lezione:

1. La prima fase del calcolo riguarda le somme pesate:

Ossia il prodotto scalare tra i pesi della j-esima connessione input dell’i-esimo neurone dell’h-esimo strato con il j-esimo output dello strato precedente h-1. Infine si somma il bias del corrispondente neurone.

1. La seconda fase è la sola applicazione della funzione di attivazione *f* dell’h-esimo strato:

Nel codice questo è tutto ciò e stato implementato verificando se la rete possiede un singolo o più di uno strato hidden. Nel primo caso si calcola direttamente in maniera hard-coded il comportamento per il singolo layer hidden e il layer output, mentre nel secondo caso si itera sul numero di strati della rete e si memorizzano, in array con dimensione pari al numero di strati hidden, tutte le componenti usate per il calcolo del comportamento. Queste serviranno per calcolare poi il gradiente della funzione di errore nella back propagation.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 2. Forward step di una rete neurale. Calcola il comportamento della rete e memorizza gli output di ogni strato della rete, insieme alle somme pesate degli input, per il calcolo del gradiente nella backPropagation.

Come già specificato, la funzione che simula soltanto la rete, restituendo l’output calcolato dalla rete, effettua lo stesso calcolo della funzione forwardStep in figura, tranne per il fatto che non memorizza output e somme pesate per usarli nella back propagation.

Algoritmo back propagation e aggiornamento pesi

Per effettuare l’aggiornamento dei pesi e bias di una rete neurale si è usato il metodo del gradiente della funzione di errore, ossia calcolare il vettore delle derivate parziali della funzione di errore usata rispetto ai pesi della rete:

Con tali derivate parziali possiamo capire se è necessario incrementare o decrementare il valore di un peso per arrivare a minimizzare la funzione di errore E.

* Se allora siamo in un punto di E decrescente, dunque per raggiungere il minimo devo incrementare il valore di
* Se allora siamo in un punto di E crescente, dunque devo decrementare il valore di

A questo punto, per poter andare a calcolare il gradiente si usa l’algoritmo di back propagation, il quale calcola solo il gradiente senza effettuare l’aggiornamento dei pesi. Per questa ultima operazione si usa l’algoritmo di discesa del gradiente.

Quindi la back propagation calcola tali gradienti:

Quindi prima di poter calcolare ci è necessario calcolare i sia per i neuroni degli strati hidden sia per quelli dello strato di output, e lo facciamo tramite la chain rule:

* strato output:

In quanto nello strato di output abbiamo che

* strato hidden:

Dove j corre sugli indici dei nodi che ricevono connessione dal nodo i.

Quindi seguendo la matematica sopra descritta l’implementazione per il calcolo delle derivate è il seguente:

Immagine che contiene testo

Descrizione generata automaticamente

Figura 3. Calcolo delle derivate per l'aggiornamento dei paramenti della rete. Se la rete ha un solo strato hidden, allora è sufficiente da solo per calcolo e salvataggio delle derivate, ovviamente dopo aver effettuato il forward step che qui è al di fuori di tale funzione.

Importante da menzionare è il fatto che prima del calcolo delle derivate controlliamo tramite flag se usiamo la funzione di *Softmax* come funzione di attivazione per lo strato di output. Se affermativo allora noi possiamo calcolare la derivata della funzione d’errore insieme al softmax direttamente. Se negativo allora calcoliamo derivata di *E* e della funzione di attivazione dello strato output in momenti diversi.

Poi abbiamo l’implementazione per il calcolo e il salvataggio di tutti i gradienti per quando si usa una rete con più di un singolo strato hidden.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 4. Algoritmo per il calcolo e salvataggio dei gradienti nel caso si usi una rete con più di un singolo strato interno.

Qui inizializziamo due array *derivWhidden* e *derivBiasHidden* di dimensione 1xnStrati-1 per memorizzare le derivate parziali di tutti i strati hidden eccetto il primo. Dopo il forward step si calcolano le derivate parziali per lo strato di output e l’ultimo strato hidden. Iterando da *nStrati-2*, cioè il numero di strati hidden tranne l’ultimo e il primo, calcoliamo e salviamo le derivate parziali. Infine calcoliamo la derivata per il primo strato hidden e lo salviamo in *dhInput.* Similmente salviamo i bias della rete, ai quali però è necessario effettuare un reshape a dimensione (*dimensioneBias,1*) dato che in python i vettori non hanno una dimensione del tipo (x,1), ma vengono considerati come (x,), ossia senza una dimensione e ciò impedisce poi di effettuare tutte le operazioni tra matrici e vettori.

Ciclo di learning

A questo punto manca solo da far vedere, per quel che riguarda l’implementazione di funzioni per la simulazione di una rete neurale, il ciclo di learning. Esso è stato implementato fornendo la possibilità di poter scegliere tra l’aggiornamento in modalità batch, oppure in modalità on-line (non è stata implementata la mini-batch).

Prima di tutto quello che facciamo è inizializzare i vettori che manterranno l’errore del train-set e del validation-set per ogni epoca, poi inizializziamo l’errore minimo che ci servirà per selezionare la migliore rete ottenuta, testata poi sul validation-set.

A questo punto si itera su un numero fissato di epoche e a seconda della modalità effettuiamo l’aggiornamento. Ricordiamo che la modalità:

* On-line aggiorna i pesi dopo l’elaborazione di ogni coppia.
* Batch aggiorna i pesi dopo l’elaborazione dell’intero train-set.

Dopo l’aggiornamento, prima di iniziare l’epoca successiva, testiamo l’errore della rete con i parametri aggiornati sia sul train-set che sul validation-set e andiamo a salvarci tale rete se essa ottiene un errore minore(sul validation-set) rispetto a quella correntemente salvata. Alla fine del learning ritorniamo la migliore rete ottenuta durante il learning e i due vettori di errori.

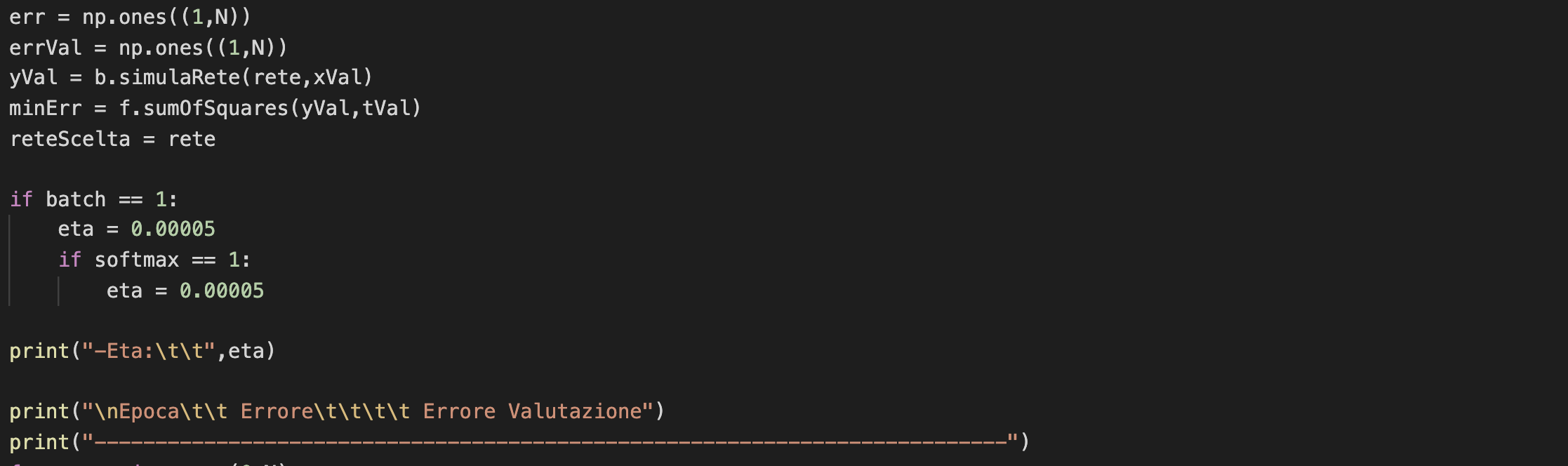


Immagine che contiene testo

Descrizione generata automaticamente

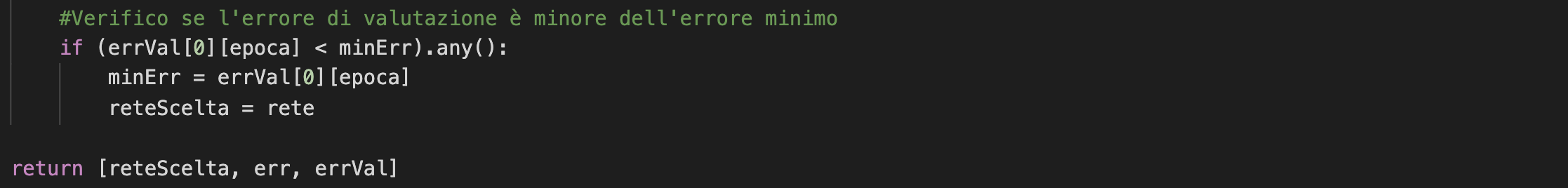


Figura 5. Funzione che implementa la fase di learning per una rete neurale feed forward

Ricordiamo anche che l’errore sul train-set durante tutte le epoche deve essere una curva discendente all’aumentare dell’epoca, in tal modo sappiamo che la rete sta “imparando”, mentre l’errore sul validation-set può sia aumentare che diminuire. Noi alla fine recupereremo la rete che minimizza l’errore sul validation-set su tutte le epoche.

Learning di una rete neurale

Adesso vediamo un esempio di learning di una rete neurale. L’esempio lo effettueremo su un problema di classificazione.

1. **Dataset usato e preprocessing effettuato**

Il dataset usato è il dataset mnist, ossia una collezione di immagini di cifre scritte a mano spesso usato per l’addestramento di reti neurali. Le cifre sono i numeri che vanno da 0 a 9, pertanto siamo in un problema di classificazione a 10 classi. Esempio del dataset:

Immagine che contiene testo

Descrizione generata automaticamente

Il pre-processing effettuato prevede di:

* 1. Trasformare gli array contenenti dati e etichette in Numpy array per ottenere prestazioni più efficienti su tutti i calcoli matriciali.
  2. Effettuare il reshape dei dati da dimensione (N,28,28) in dimensione (N,784) in modo da poter usare i dati nella rete neurale.
  3. Normalizzare i valori dei dati, ossia dei pixel, da range [0,255] a range [0,1].
  4. Fare shuffle delle coppie del dataset.
  5. Convertire le etichette del dataset in codifica one-hot, ossia se presa una coppia del dataset *(x,t)* con *x* appartenente alla classe 4, allora t = [0,0,0,1,0,0,0,0,0].
  6. Dividire il dataset in train-set, validation-set e test-set.
  7. Controllare che le immagini corrispondano alle relative etichette.
  8. Fare la trasposta del train-set per darlo in input alla funzione di learning.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 6. Implementazione della fase di pre-processing

1. **Valutazione errore e precisione della rete**

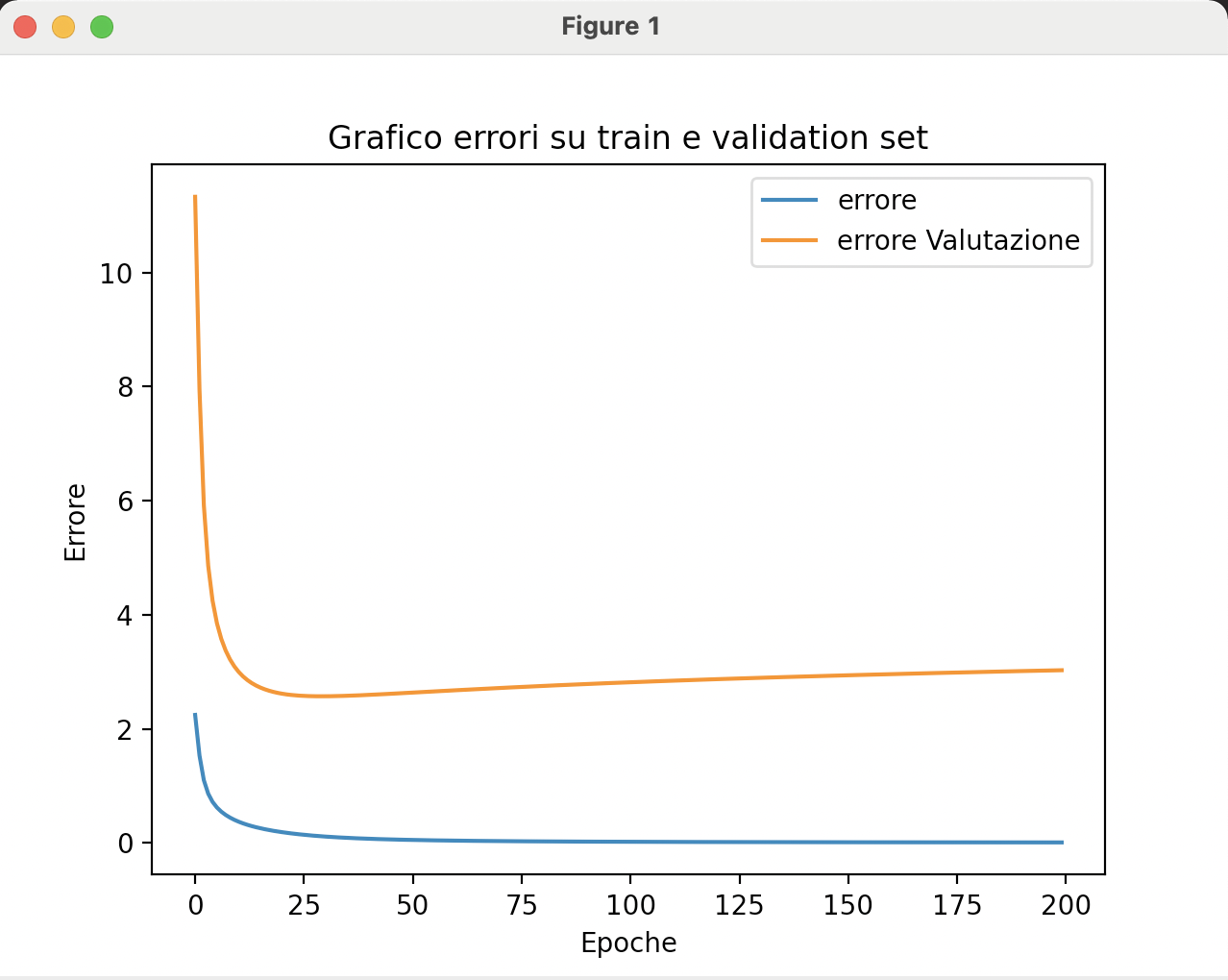
Verranno effettuate alcune valutazioni, cambiando alcuni iper-parametri, della rete usando un train-set di 5000 coppie, un validation-set di 1000 coppie e un test-set di 1000 coppie. La funzione di attivazione per lo strato di output sarà sempre la softmax per tutte le valutazioni e di conseguenza la funzione di errore sarà la cross validation con il softmax. Infine come regola di aggiornamento si userà solo la discesa del gradiente.

Per ottenere l’errore medio per su tutte le coppie, l’errore cross entropy verrà diviso del numero di coppie N usate per simulare la rete (dato che tale errore è la somma degli errori di tutte le coppie).

Indicherò tra [] le funzioni di attivazione per i vari strati hidden della rete e tra () il numero di neuroni per i vari strati hidden

* 1. **Rete:** 1 layer hidden (50) neuroni --- [sigmoide]

**eta:** 0.01 / **modalità:** online / **epoche:** 200



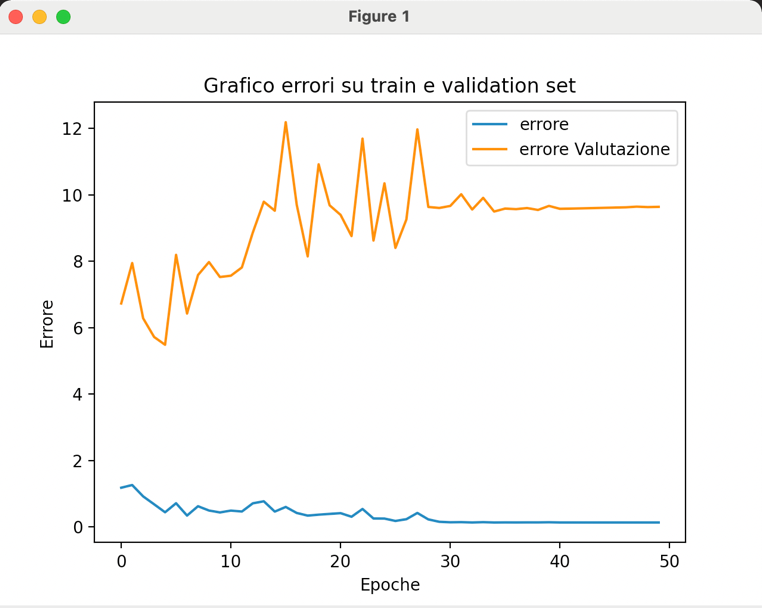
>**Errore minimo sul train-set:** 0.0073 / **Errore minimo sul validation-set:** 2.57

>**Media errore su 10 simulazioni sul test-set:** 0.4698

>**Precisione rete sulla classificazione:** 87.5 %

* 1. **Rete:** 1 layer hidden (100) neuroni --- [RELU]

**eta:** 0.1 / **modalità:** online / **epoche:** 50



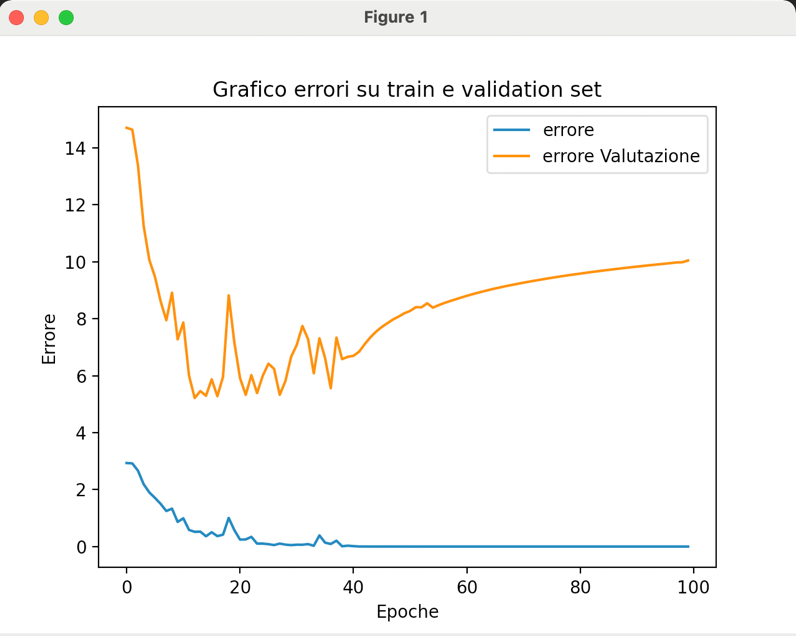
>**Errore minimo sul train-set:** 0.1378 / **Errore minimo sul validaiton-set:** 5.4875

>**Media errore su 10 simulazioni sul test-set:** 1.3544

>**Precisione rete sulla classificazione:** 87.2 %

* 1. **Rete:** 5 layer hidden (40,30,30,20,20) neuroni --- [RELU, RELU, RELU, RELU, RELU]

**eta:** 0.01 / **modalità:** online / **epoche:** 100



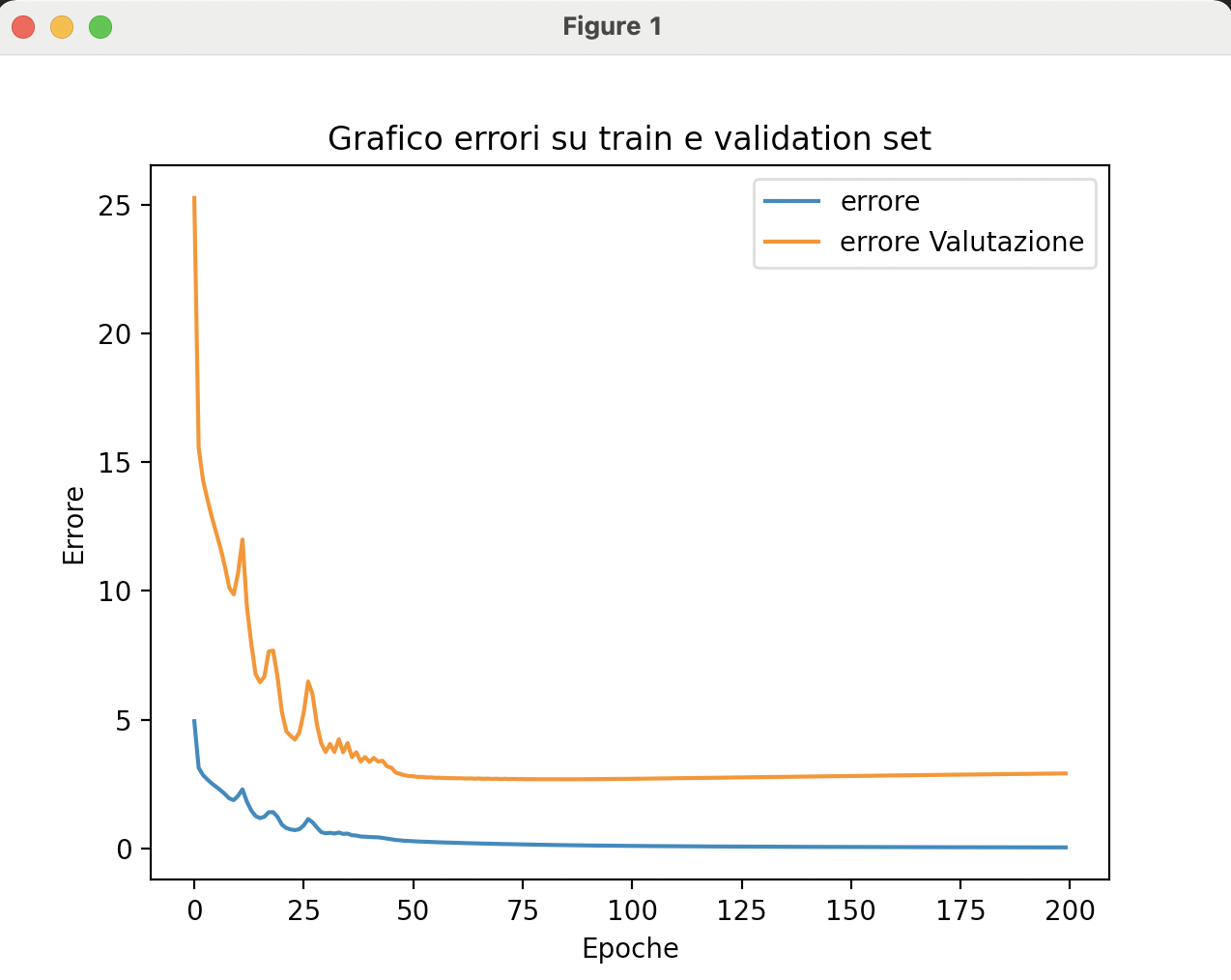
>**Errore minimo sul train-set:** 7.2\*10^-5 / **Errore minimo sul validaiton-set:** 5.21

>**Media errore su 10 simulazioni sul test-set:** 1,4877

>**Precisione rete sulla classificazione:** 87.1 %

* 1. **Rete:** 1 layer hidden (50) neuroni --- [sigmoide]

**eta:** 0.00005 / **modalità:** Batch / **epoche:** 200



>**Errore minimo sul train-set:** 0.02865 / **Errore minimo sul validaiton-set:** 2.677

>**Media errore su 10 simulazioni sul test-set:** 0.3937

>**Precisione rete sulla classificazione:** 88.7 %

Possiamo vedere come in tutti gli esempi l’errore della rete sul train-set scenda abbastanza fino ad arrivare a circa 0, mentre l’errore sul validation-set, anche se non scende sempre e non scende fino a 0, esso rimane comunque abbastanza basso. Questo ci indica che la rete sta in qualche modo sta imparando a classificare le immagini, ed infatti, effettuando la valutazione dell’errore e della precisione della rete sul test-set, questi si rivelano essere rispettivamente molto basso (tra 0 e poco più di 1) e abbastanza alto (circa il 90%).

Immagine che contiene testo

Descrizione generata automaticamente

Figura 7. Implementazione della parte di valutazione di una rete.

Parte B

La parte B del progetto sfrutta gli stessi algoritmi e funzioni di attivazione/errore con relative derivate che abbiamo già affrontato nella parte precedente della relazione, per cui non verranno affrontati nuovamente. Verranno invece affrontate le tecniche specificatamente richieste per questa parte di progetto.

Implementazione della rete neurale

Siccome per questa parte di progetto è esplicitamente richiesta una rete neurale con un singolo layer hidden, questa è stata direttamente costruita in modo hard-coded. Quindi esattamente come la rete generica della parte A, ma senza tutte le strutture riguardanti gli strati hidden (anche qui i parametri sono inizializzati con valori randomici nel range [-1,1]):

* Layer hidden:

Dove m è il numero nodi dello strato hidden, d è il numero di dati in input

* Layer output:

Dove *c* è il numero di neuroni dello strato di output e *mk* è il numero di neuroni dello strato hidden precedente a quello output

Siccome si userà la RPROP come regola di aggiornamento, la rete possiede anche strutture per mantenere i per tutti i parametri della rete:

* Layer hidden:

DELTA

* Layer output:

DELTA

Tali sono inizializzati tutti al valore 0.1.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 8. Implementazione della rete neurale richiesta per la parte B del progetto.

Regola di aggiornamento

Come già detto all’inizio di questo capitolo, si useranno gli stessi algoritmi per la simulazione di una rete neurale, compresi il forward-step e la back propagation, mentre quello che cambia, come già abbiamo anticipato, è la regola di aggiornamento: la *Resilient Back propagation (RPROP)*

La RPROP è una regola di aggiornamento la cui idea è quella di velocizzare l’apprendimento della rete neurale basandosi sul concetto del momento. Essa per ottenere tale velocizzazione associa ad ogni parametro un ulteriore passo di aggiornamento, ossia i visti nella implementazione della rete. L’aggiornamento del parametro avverrà in tal modo:

Il valore di t indica l’epoca corrente della fase di learning in esecuzione, pertanto la RPROP per l’aggiornamento dei parametri sfrutta i gradienti sullo stesso punto di due epoche diverse: quella corrente e quella precedente.

A questo punto manca solo da vedere come aggiornare i : si prendono i gradienti e . Normalmente noi sappiamo che se allora ci stiamo muovendo nella direzione giusta e posso incrementare il parametro, mentre se significa che ho fatto un passo troppo grande quindi devo ridurlo. Andando a fare posso ottenere l’informazione che mi serve per aggiornare i . In particolare ho che:

* **IF** **THEN**
* **IF**  **THEN**
* **IF**  **THEN**

Dove e sono due nuovi iper-parametri della rete usati rispettivamente per incrementare e decrementare i .

Inoltre per evitare che diventino troppo grandi o troppo piccoli, aggiungo altri due parametri: DELTAmax e DELTAmin e così l’algoritmo diventa:

* **IF** **THEN**
* **IF**  **THEN**
* **IF**  **THEN**

Infine, siccome al passo t = 1 della RPROP non posseggo il gradiente del passo t = 0, in quanto esso non esiste, vado ad usare la classica discesa del gradiente per poi usare la RPROP dal passo t = 2 in poi.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 9. Funzioni per i calcolo dei DELTA per la implementazione della RPROP.

Algoritmo di analisi del modello: K-FOLD

L’algoritmo di K-FOLD viene usato per andare semplicemente a valutare le prestazioni di un modello di rete. L’algoritmo prevede di dividere il dataset in K fold (k insiemi disgiunti del dataset) ed effettuare k valutazioni: in ogni i-esima valutazione, con i=1,..,k, uno dei k fold viene usato come test-set mentre il restante viene usato come train-set.

Immagine che contiene testo, armadietto, tabellonesegnapunti, screenshot

Descrizione generata automaticamente

Figura 10. Esempio di K-Fold con 5 fold.

Inoltre, all’inizio della successiva valutazione, si butta via il modello addestrato in modo tale da inizializzare nuovi pesi e bias, sempre sullo stesso modello di rete, e valutarlo sui successivi train-set e test-set selezionati dal k-fold. Alla fine si avranno k valutazioni e k parametri che verranno usati per capire come si comporta la rete nel problema definito.

Nell’implementazione del k-fold si è usata una libreria per Python “KFold” della libreria “SKLearn” che automaticamente, dato il dataset, tira fuori gli indici per ottenere il train-set e il test-set ad ognuna delle k valutazioni.

Immagine che contiene testo

Descrizione generata automaticamente

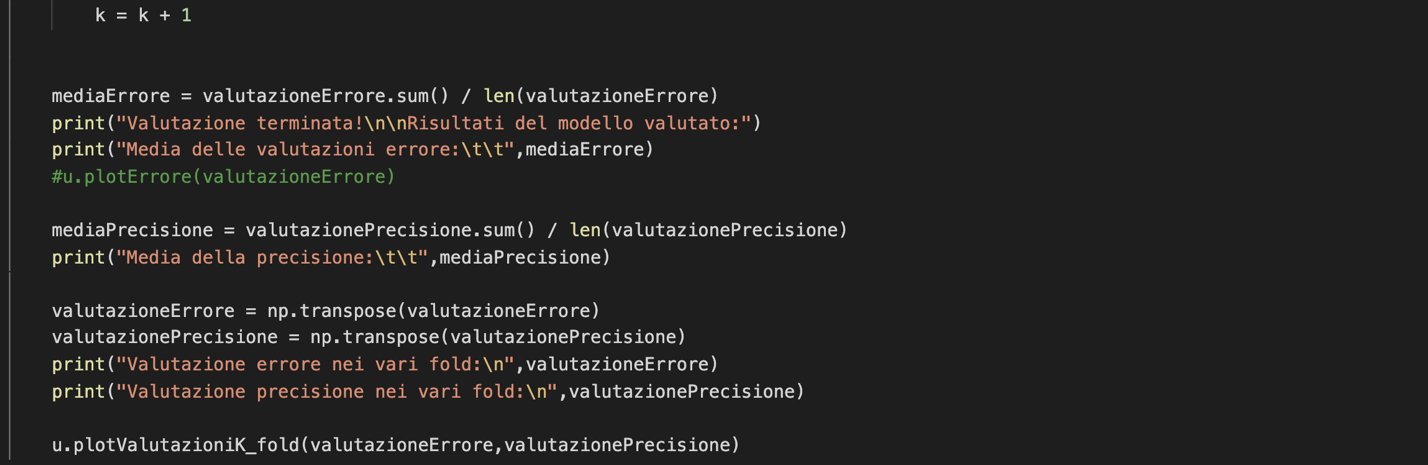


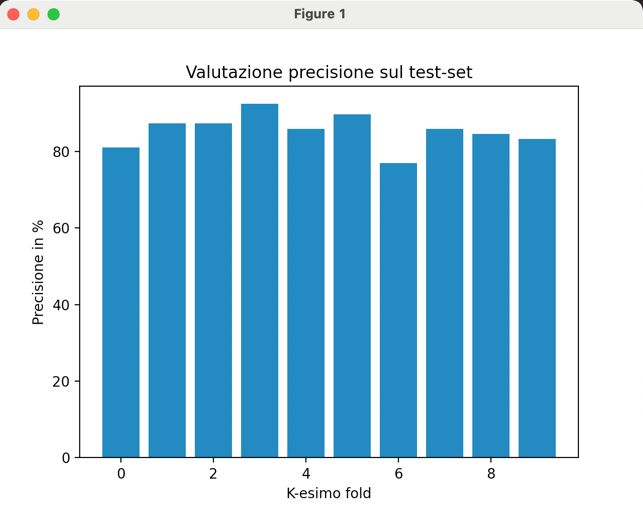
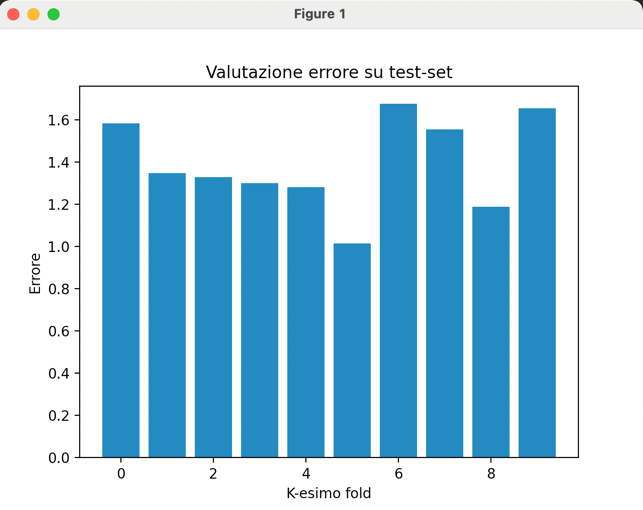
Figura 11. Implementazioen del k-fold con valutazione del modello.

Valutazione di un modello di rete neurale

Per la valutazione di un modello di rete si è usato lo stesso dataset (Mnist) e lo stesso tipo di pre-processing che abbiamo visto per la parte A del progetto ad eccezione del fatto che qui andiamo soltanto a diminuire la dimensione del dataset a 5000 coppie e non andiamo quindi a dividerlo in train-set, validation-set e test-set. Tale divisione si effettua direttamente tramite il K-FOLD in train e test set.

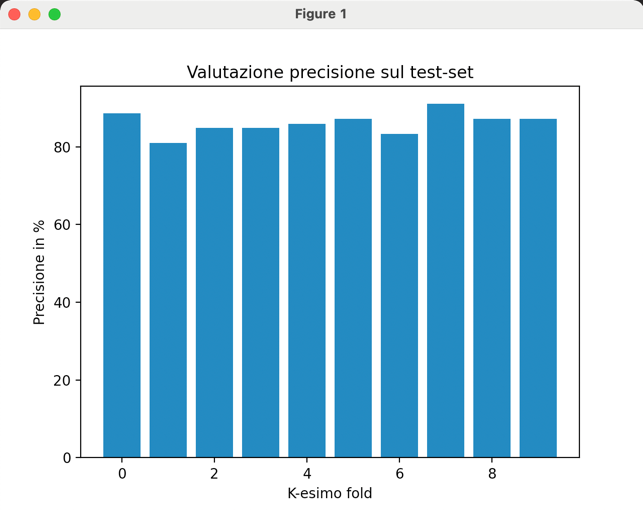
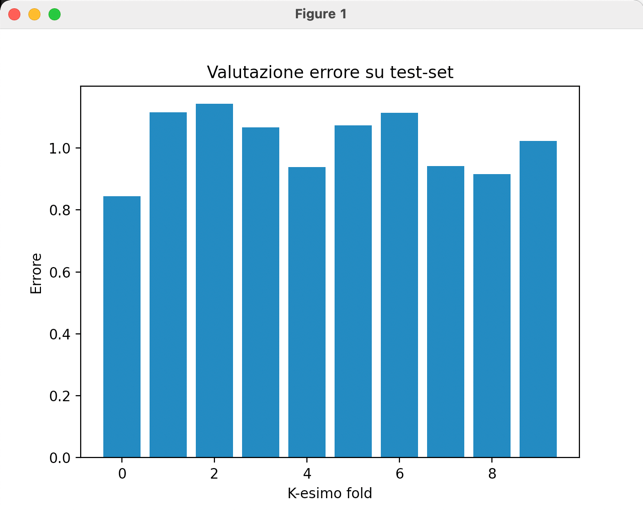
Ora andremo a valutare alcuni modelli modificando solo parametri come e della rete e il numero di nodi interni della rete. Manterremo invece fisso tutto il resto: la funzione di attivazione del layer hidden sarà la *sigmoide*, quella del layer output sarà il *softmax*, il numero di fold k = 10. Il problema su cui andremo a valutare i nostri modelli sarà ancora una volta un problema di classificazione a 10 classi.

1. **:** 1.002 / **:** 0.005 / **epoche:** 200 / **numero neuroni:** 25



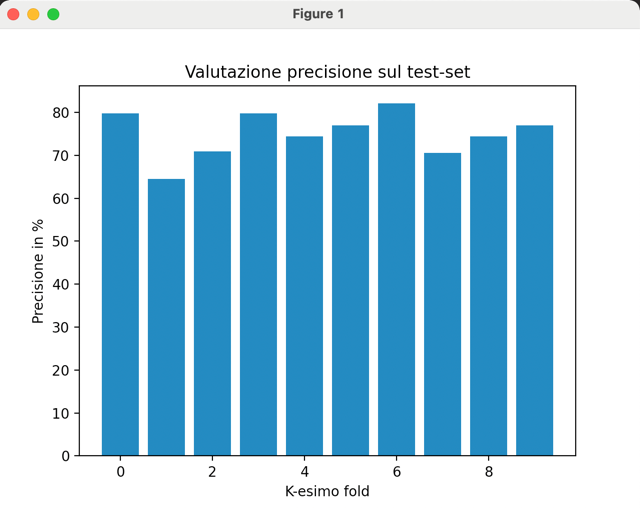
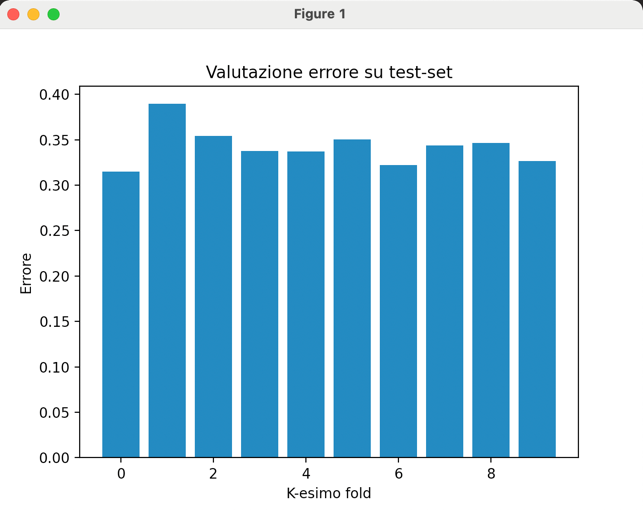
**Media valutazione errore:** 1.3935 **Media valutazione precisione:** 85.45%

1. **:** 1.002 / **:** 0.005 / **epoche:** 130 / **numero neuroni:** 25



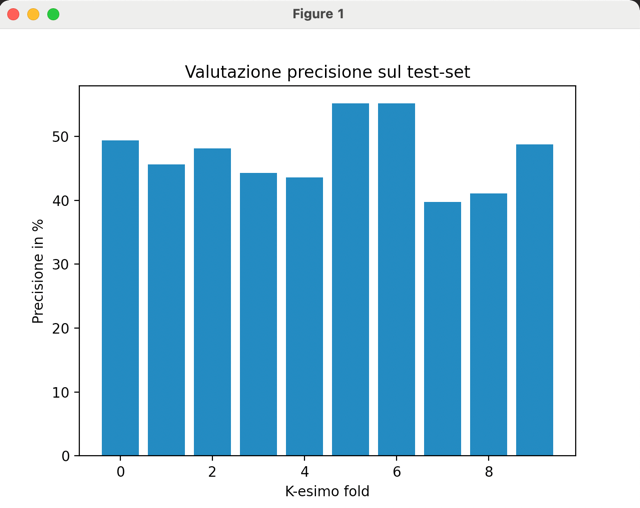
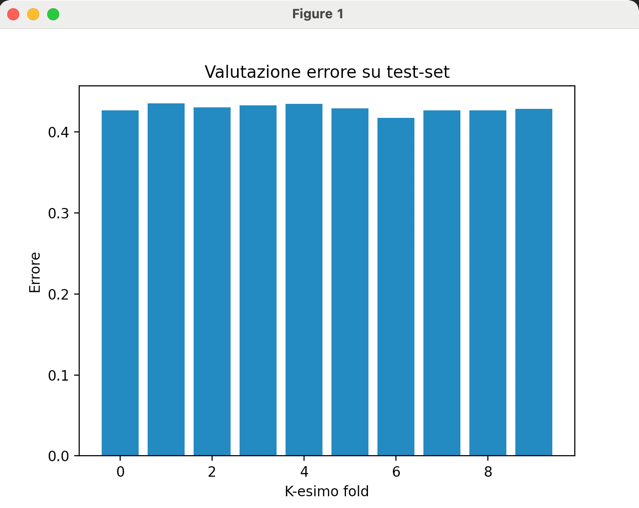
**Media valutazione errore:** 1.017 **Media valutazione precisione:** 86.10%

1. **:** 1.002 / **:** 0.005 / **epoche:** 20 / **numero neuroni:** 25



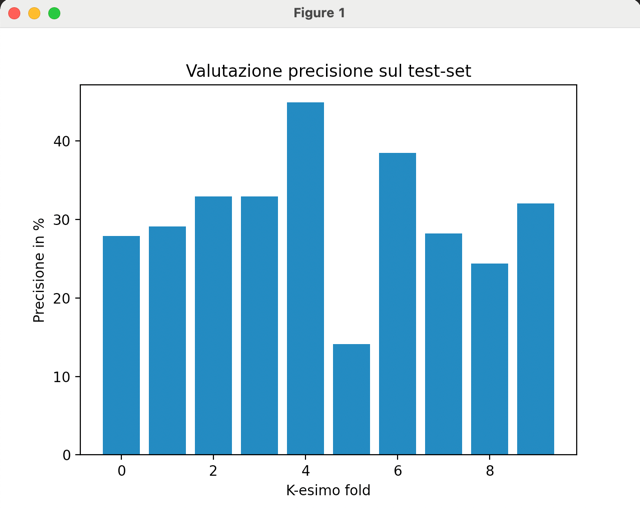
**Media valutazione errore:** 0.3422 **Media valutazione precisione:** 75.05%

1. **:** 1.002 / **:** 0.005 / **epoche:** 20 / **numero neuroni:** 5



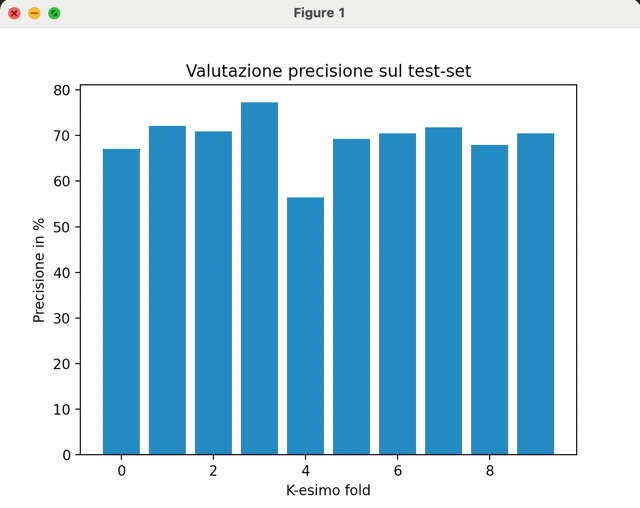
**Media valutazione errore:** 0.4782 **Media valutazione precisione:** 47.07%

1. **:** 1.2 / **:** 0.5 / **epoche:** 5 / **numero neuroni:** 5



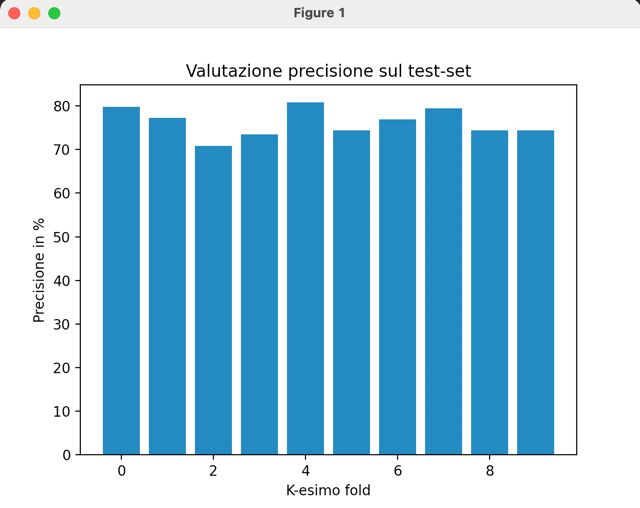
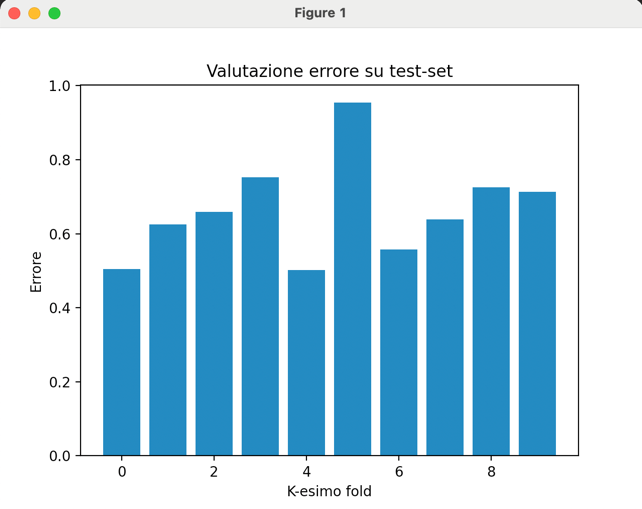
**Media valutazione errore:** 0.1123 **Media valutazione precisione:** 30.48%

1. **:** 1.2 / **:** 0.5 / **epoche:** 50 / **numero neuroni:** 5



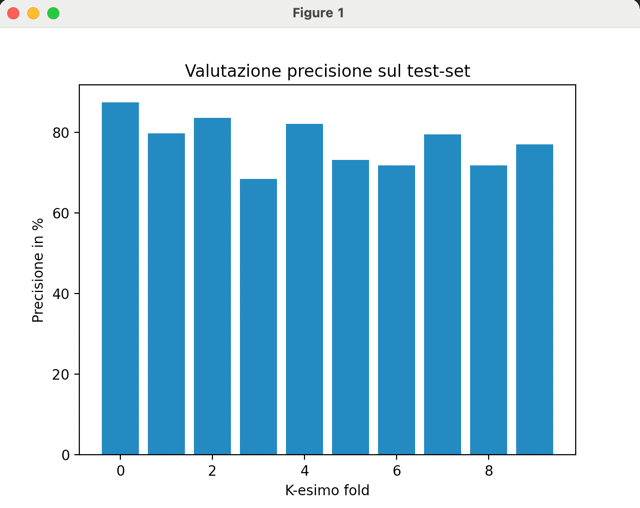
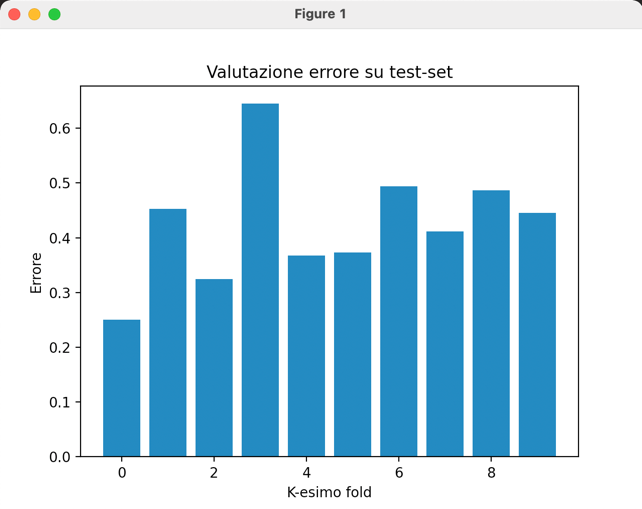
**Media valutazione errore:** 0.8122 **Media valutazione precisione:** 69.38%

1. **:** 1.2 / **:** 0.5 / **epoche:** 50 / **numero neuroni:** 50



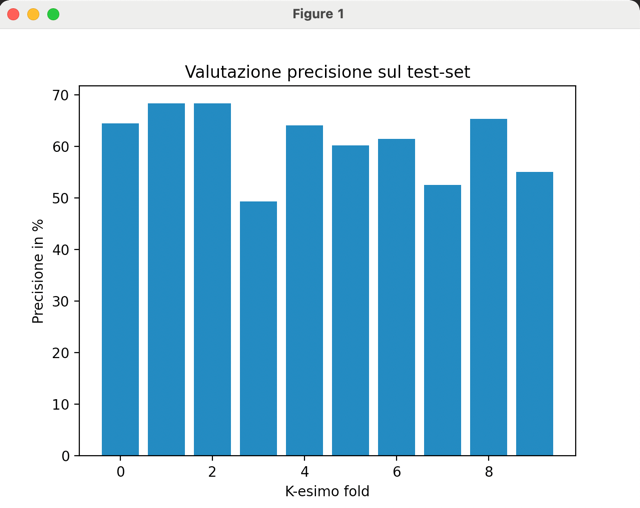
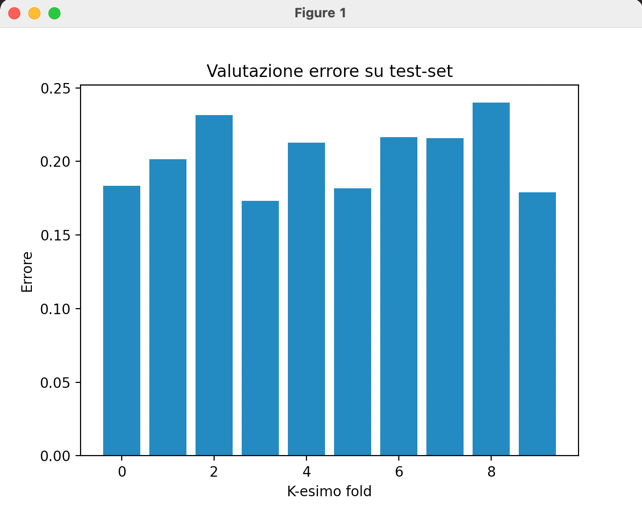
**Media valutazione errore:** 0.6632 **Media valutazione precisione:** 76.15%

1. **:** 1.7 / **:** 0.8 / **epoche:** 30 / **numero neuroni:** 50



**Media valutazione errore:** 0.4251 **Media valutazione precisione:** 77.41%

1. **:** 1.7 / **:** 0.8 / **epoche:** 10 / **numero neuroni:** 100



**Media valutazione errore:** 0.2035 **Media valutazione precisione:** 60.96%

In queste valutazioni di differenti modelli possiamo vedere come l’errore sul test-set sia generalmente basso, sempre tra lo 0 e poco più di 1. La cosa interessante, da come si può vedere dai grafici, è che con un buon bilanciamento, tra gli eta, il numero di neuroni e il numero di epoche, sia possibile andare ad ottenere un buon compromesso tra precisione e tempo di esecuzione. Infatti confrontando ad esempio la d) e la f) possiamo vedere come, aumentando di molto gli eta, ma solo di un poco il numero di epoche, riusciamo ad ottenere una precisione percentuale della rete di più del doppio con circa lo stesso tempo di esecuzione(non memorizzato). Comunque alla fine il miglior modello da usare per il test, basandoci sulla percentuale di precisione è il modello b), il quale con 70 epoche in meno rispetto ad a) riesce comunque ad ottenere una precisione maggiore, indicando che è possibile ottenere circa gli stessi risultati, ma con un numero di epoche inferiore in quanto, probabilmente, con questi eta e questo numero di neuroni il learning sul test-set non migliora più di tanto dopo una cerca epoca. In questi casi si potrebbe anche andare ad implementare un approccio di *early stopping* per evitare di eseguire epoche che non portano a grossi aggiornamenti.

Principali funzioni usate



Nella libreria delle funzioni sono state implementate anche altre funzioni viste a lezione, ma che non sono state usate semplicemente perché non esplicitamente richiesto, come ad esempio:

Con relative derivate.

Riferimenti usati per realizzare il progetto

1. Appunti e registrazioni delle lezioni
2. Bishop - Neural Networks for Pattern Recognition – Per la Back Propagation e Softmax,
3. http://mathesaurus.sourceforge.net/matlab-numpy.html - Per le operazioni Matlab viste a lezione in Python.