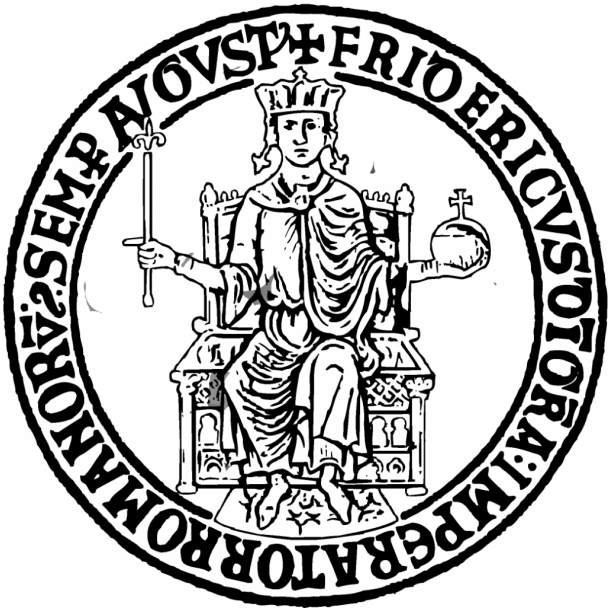
Universita' degli studi di Napoli Federico II



Corso di laurea magistrale in Informatica

Corso di Machine learning ed applicazioni

Prof. Roberto Prevete

Realizzazione di una rete neurale feed forward multistrato full connected

degli Studenti:

Marco Urbano N97000268

Ciro Brandi N97000269

Anno accademico 2017/2018

Indice

1 Introduzione al problema **2**

2 L'implementazione **3**
2.1 Le classi che descrivono la rete 4
2.1.1 Net 5
2.1.2 Layer 6
2.1.3 Error 7
2.1.4 StopCriteria 8

3 Analisi dei criteri di early stopping **10**
3.1 Introduzione alla problematica di fermata 11
3.2 Generalization Loss 12
3.3 Progressive Quotient 13
3.4 Analisi delle prestazioni 15
3.4.1 Configurazione iniziale 15
3.4.2 Risultati 16

4 Bibliografia **19**

1 Introduzione al problema

Il seguente lavoro di sperimentazione si basa sulla progettazione di una rete neurale feed-forward multistrato che sia dotata di:

- strati con un **numero di neuroni** specificato dall'utente.
- possibilità di definizione di più strati ognuno con una propria **funzione di output** (e.g. *sigmoide, heaviside step function, identità, etc*).
- funzione di learning basata sul metodo della **backpropagation** per il calcolo delle derivate parziali.
- funzione di learning basata su qualsiasi **funzione di errore** (e.g. *cross-entropy, total sum of squares, etc..*)

Inoltre la seguente rete è dotata dei seguenti metodi di learning:

- *On-line learning basato su* ***discesa del gradiente***
- *Resilient Back Propagation (RPROP)*

Il lavoro si focalizza inoltre sullo studio approfondito dei criteri di early stopping proposti dal paper "Early stopping, but when?"[1] ed in particolare saranno affrontati paragoni tra i due metodi di stop, *Generalization loss* e *Progressive Quotient*, per riuscire a decidere quale dei due criteri convenga in termini di *accuratezza/epoche necessarie*.

Il lavoro è stato effettuato con l'ausilio del dataset **MNIST**, ovvero un dataset di handwritten digits.

2 L'implementazione

Per la realizzazione di tale rete neurale è stato impiegato il linguaggio Python che tramite l'ausilio delle librerie *SciPy* e *NumPy* permette operazioni ottimizzate di prodotti matriciali e prodotti punto-punto tra matrici e vettori.

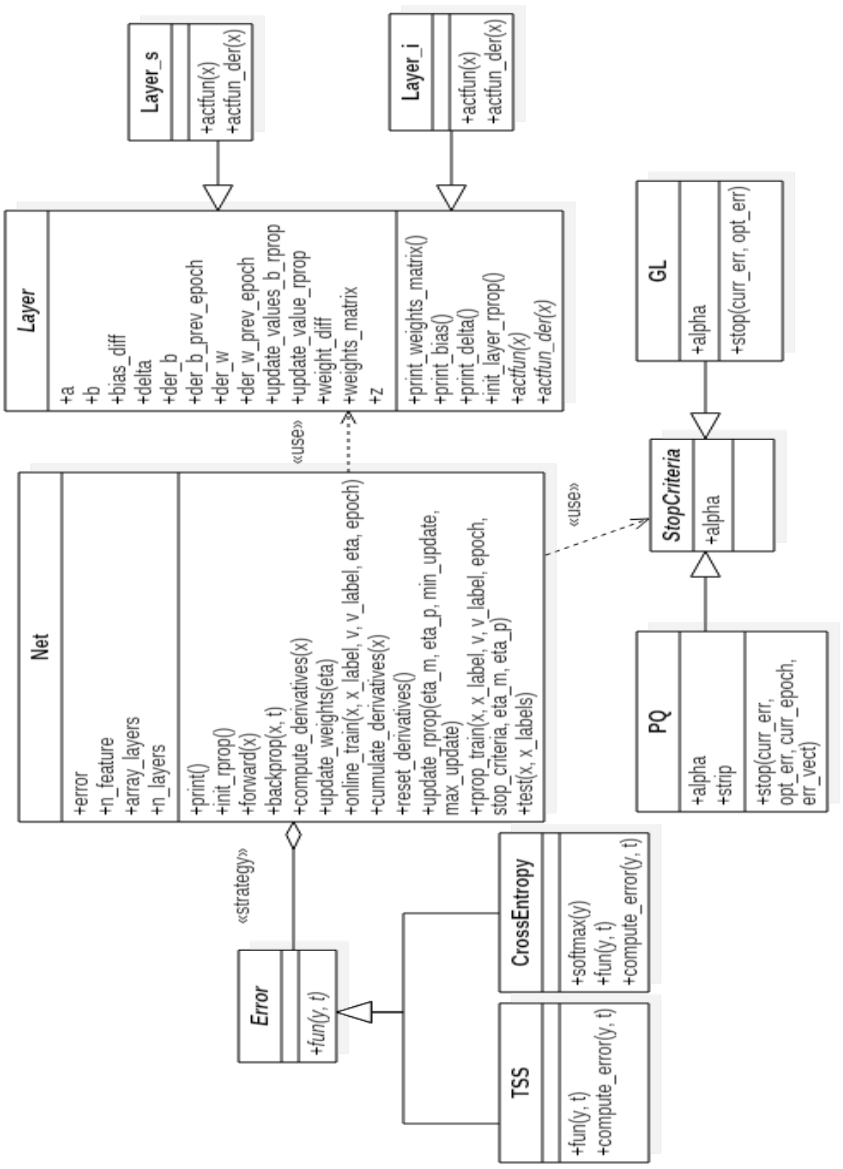
Inoltre il progetto di tale rete ha tenuto fortemente in considerazione l'idea di *modularità*, pertanto l'utente potrà scegliere di costruire la propria rete come preferisce semplicemente utilizzando le classi disponibili, le quali descrivono ogni aspetto di una rete neurale, a partire dal singolo strato, rappresentato qui dalla classe **Layer**, fino ad arrivare alle classi che servono a scegliere quale funzione di errore adottare per fare il training della rete, rappresentate dalla classe astratta **Error** che è utilizzata per applicare il *design pattern* **strategy**.

Il seguente progetto è stato sviluppato inoltre con l'ausilio dell'IDE **PyCharm**, utilizzando la versione di **Google TensorFlow 1.5.0** per ottenere il dataset **mnist**. La versione di **Python** è la **3.6**: per una corretta esecuzione delle librerie in allegato è consigliata la stessa configurazione.

Nei paragrafi successivi di questo capitolo è presente il *class diagram* descrittivo di tutte le classi che compongono la rete e la descrizione di ogni classe per fornire all'utente un modo rapido per comprendere tutte le scelte adottate in fase di progettazione.

2.1 Le classi che descrivono la rete

La rete neurale sviluppata è composta da quattro classi ognuna con un compito specifico, con la possibilità di permettere all'utente di impostare a proprio piacimento i parametri che ne descrivono il comportamento. Nei paragrafi successivi verranno dettagliate le classi principali della rete e le rispettive responsabilità.



2.1.1 Net

La classe **Net** rappresenta una rete neurale feed-forward multistrato, essa si occupa di tutte le attività principali per produrre l'output desiderato. La classe viene inizializzata con un numero arbitrario di strati, rappresentato come una lista, dove ogni valore della lista dei layer rappresenterà il numero di neuroni associato all'i-esimo layer. Per ogni strato inoltre viene scelta la funzione di attivazione da associare in base ai valori dati al costruttore all'atto della creazione della rete. Le funzioni di attivazione vengono passate alla rete anch'esse attraverso una lista nella quale ogni valore fungerà da discriminante alla tipologia di layer istanziato. Le tipologie di layer già presenti in questa versione della rete verranno dettagliate nel paragrafo successivo. Infine l'inizializzazione termina con un oggetto di tipo **Error** che rappresenta la funzione di errore utilizzata nel processo di learning.

Il seguente frammento di codice illustra l'invocazione del costruttore per la classe **Net** per l'istanziazione di una rete con *numero di **features** uguale a **10**, due livelli di cui un **hidden layer** dotato di **16** neuroni e funzione di attivazione **sigmoide** (identificata dall'intero 1) e uno **strato di uscita** con **10** neuroni e funzione di attivazione **identità** (identificata dall'intero 0). La funzione di errore scelta per questa rete è **CrossEntropy**.*

```
1 my_net = N.Net(10, [16, 10], [1, 0], E.CrossEntropy())
```

I processi di learning implementati sono quelli tra i più classici, **batch** e **online**.
Il processo di **learning online** utilizza la discesa del gradiente classica per la ricerca del minimo ed è implementato dal metodo:

```
1 def online_train(self, X, x_label, V, v_label, eta, epoch)
```

X: dati del training set

x_label: label del training set

V: dati del validation set

eta: learning rate

epoch: numero di epoche

Il processo di **learning batch** utilizza la *resilient backpropagation*, che attraverso le informazioni locali dalla discesa del gradiente aggiorna i pesi nel modo apportuno (ovvero analizzando il segno delle derivate), è implementato dal metodo:

```
def rprop_train(self, X, x_label, V, v_label, epoch,
    stop_criteria, eta_m = 0.5, eta_p = 1.05)
```

X: dati del training set

x_label: label del training set

V: dati del validation set

epoch: numero di epoche

stop_criteria: criterio di stop

eta_m learning rate negativo

eta_p learning rate positivo

Il processo di learning termina o al raggiungimento del numero massimo di epoche o al criterio di stop scelto. Nei paragrafi successivi verranno approfonditi tali argomenti.

2.1.2 Layer

La classe astratta **layer** rappresenta un layer della rete feed-forward multistrato. Tale classe risulta essere astratta per permettere di specificare le diverse funzioni di attivazione semplicemente effettuando la definizione di una sottoclasse. Le classi definite sono **Layer_s** e **Layer_i** che rappresentano rispettivamente un layer con la funzione di attivazione sigmoide e il layer con l' identità.

Per ogni layer la matrice dei pesi e il vettore dei bias viene inizializzato con valori casuali compresi tra -0.5 e 0.5. I pesi sono stati scelti per avere un andamento smooth della funzione di errore nella fase di apprendimento.

2.1.3 Error

Invece di dotare la rete di una singola funzione di errore con cui effettuare il training è stato pensato di dare all'utente la possibilità di scegliere la propria funzione di errore o tra quelle già implementate, ovvero *somma dei quadrati* per i problemi di **regressione** e *cross entropy* per problemi di **classificazione**, oppure di scrivere la propria funzione di errore come un metodo di una sottoclasse della *classe astratta* **Error**.
Dunque, come sarà saltato all'occhio del lettore attento, questa classe serve per far si che venga implementato il ben più famoso *design pattern* **strategy**, in modo tale da fornire ogni volta una funzione di errore diversa semplicemente fornendo un oggetto della classe che la rappresenta in fase di costruzione della rete.
In particolare nel box sottostante è possibile osservare una istanza di classe **Error** fornita al costruttore di **Net** (in questo caso si tratta dell'oggetto che descrive la funzione di errore **CrossEntropy**, ovvero la stessa utilizzata per gli esperimenti effettuati).

```
my_net = N.Net(n_f, [n_neurons, label_num], [1, 0], E.
    CrossEntropy())
```

La classe astratta **Error** è dotata dei seguenti metodi astratti da implementare per poter essere sfruttata:

```
class Error(metaclass=ABCMeta):

    @abstractmethod
    def fun(self, Y, T):
        pass

    @abstractmethod
    def compute_error(self, Y, T):
        pass
```

in particolare si ha che:

fun: è la funzione usata per il calcolo dei *delta* in fase di *backpropagation*.

compute_error: è la funzione usata per il calcolo dell'errore dopo ogni *forward propagation*, ovvero utilizzata dopo ogni epoca per controllare come varia l'errore dopo il processo di training effettuato all'epoca precedente.

E per entrambi i metodi **Y** è l'i-esimo vettore di input fornito dalla forward e **T** è il vettore *target* fornito dal dataset scelto per l'i-esimo vettore.

CrossEntropy La sottoclasse **CrossEntropy** già implementata nella rete è la rappresentazione dell'omonima funzione di errore. In particolare a differenza della superclasse **Error** ha bisogno di un ulteriore metodo, ovvero la funzione di *softmax*, la quale serve per normalizzare i risultati ottenuti ed approssimarli a probabilità. Dunque per questioni progettuali, visto che questa funzione è legata strettamente alla funzione di errore in questione, è stato scelto di renderla un metodo di questa classe.

TSS La sottoclasse **TSS** già implementata nella rete è la rappresentazione della funzione di errore *total sum of squares* (somma dei quadrati) e non necessita di ulteriori metodi oltre quelli sovrascritti in fase di definizione della classe.

2.1.4 StopCriteria

La classe astratta **StopCriteria** è stata pensata come classe per rappresentare il criterio di fermata da fornire al metodo di training scelto. Come per la classe **Error**, questa classe è una implementazione del pattern **strategy**. Questa classe non è dotata di nessun metodo da sovrascrivere e dunque sarebbe da vedere più come una *interfaccia*: il problema è che ogni criterio di stop di quelli implementati ha una diversa firma e dunque non avrebbe avuto senso definire un metodo astratto da sovrascrivere. L'unico attributo di questa classe è *alpha*, che come descritto nel terzo capitolo, è la variabile che serve a decidere quando fermarsi a seconda del criterio scelto.

GL La sottoclasse **GL** rappresenta il criterio di stop denominato *Generalization Loss* ed ha un unico metodo che restituisce un booleano per indicare se il processo di learning va interrotto o meno.

```
def stop(self, curr_err, opt_err):
  # Calcolo la generalization loss per ogni epoca
  GL_epoch = 100 * ((curr_err / opt_err) - 1.0)
  #print("GL_epoch" + str(GL_epoch) + "\n")

  if GL_epoch > self.alpha:
       return 1
  return 0
```

I parametri **curr_err** e **opt_err** rappresentano rispettivamente l'errore registrato nell'epoca corrente e quello ottimo finóra raggiunto. Questo metodo viene invocato dunque ogni volta che si raggiunge la fine di un'epoca per decidere se fermarsi o meno.

PQ La sottoclasse **PQ** rappresenta il criterio di fermata *Progressive Quotient* e come la classe descritta pocanzi restituisce un booleano per controllare se interrompere il processo di learning. Questa classe necessita oltre che del parametro *alpha* anche del parametro denominato *strip*: quest'ultimo rappresenta il numero di epoche successive su cui verificare se bisogna fermarsi o meno.
Questo valore deve essere inoltre un multiplo del numero totale di epoche scelto in fase di addestramento.

Inoltre il metodo **stop** è descritto nel seguente box.

```
1
2  def stop(self, curr_err, opt_err, curr_epoch, err_vect):
3
4    # Calcola Pk
5    numerator = np.sum(err_vect[curr_epoch - self.strip:
       curr_epoch])
6    denominator = self.strip * np.min(err_vect[curr_epoch -
       self.strip: curr_epoch])
7
8    PK_epoch = 1000 * (numerator/denominator)
9
10   # Calcola GL
11   GL_epoch = np.round(100 * ((curr_err / opt_err) - 1.0), 10)
12   curr_PQ = (GL_epoch / PK_epoch)
13
14   if curr_PQ > self.alpha:
15             return 1
16   return 0
```

I parametri richiesti sono:

curr_err: è l'errore corrente al momento dell'invocazione del metodo.

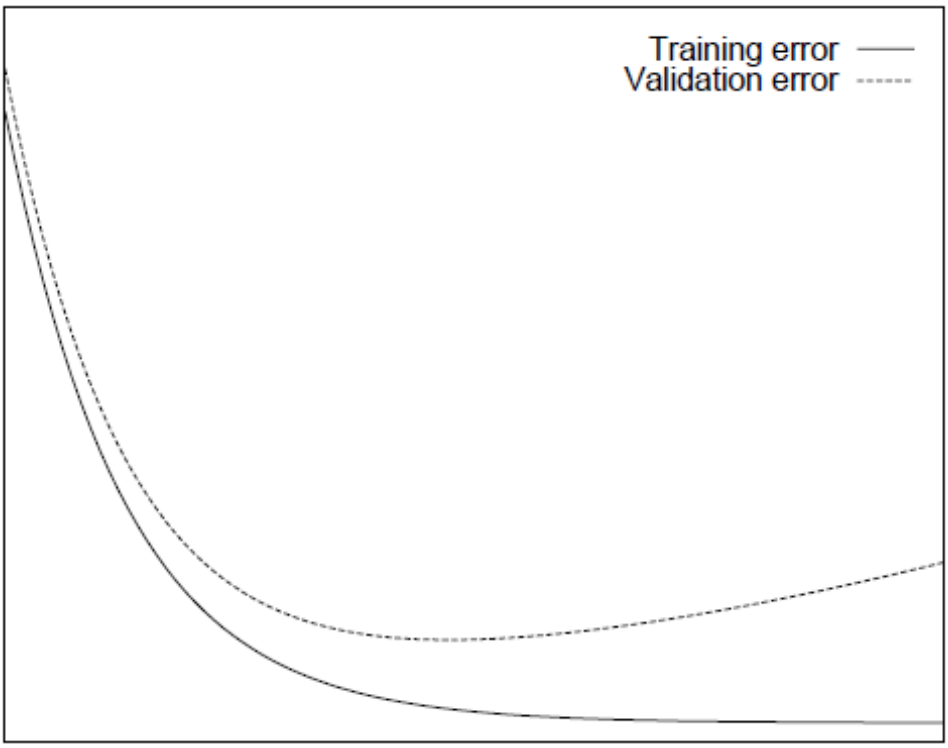
opt_err: è l'errore più basso al momento dell'invocazione del metodo.

curr_epoch: è il numero dell'epoca corrente.

err_vect: è il vettore che contiene gli errori per tutte le epoche fino a quella corrente.

3 Analisi dei criteri di early stopping

Quando si allena una rete neurale, di solito si è interessati ad ottenere una rete con prestazioni ottimali di generalizzazione. Tuttavia, tutte le architetture di reti neurali standard come le reti feed-forward multistrato full connected sono soggette ad **overfitting**, mentre la rete sembra migliorare, cioè l'errore sul training set diminuisce, ad un certo punto durante il processo di training si verifica un aumento dell'errore sul *validation set*. Tipicamente per stimare l'errore di generalizzazione si fa affidamento, come precisato prima, sull'errore ottenuto in fase di learning sul *validation set*. In generale un criterio naïf per fermarsi ed evitare i peggioramenti sull'accuratezza dovuti all'overfitting è quello di **fermarsi non appena l'errore sul validation set inizia a crescere.** Il comportamento della rete in fase di training è mostrato in figura, con il fenomeno dell'*overfitting* evidenziato dalla crescita dell'errore sul validation set. Di solito dunque la prassi naïf per addestrare la rete senza



criteri di stopping avanzati è la seguente:

1. Suddivisione dei dati in **training set** e **validation set** con una proporzione 2 a 1. Il processo di learning vero e proprio e basato solo sugli esempi etichettati del **training set**, mentre l'errore

sul **validation set** è utilizzato per avere una stima dell'errore di generalizzazione.

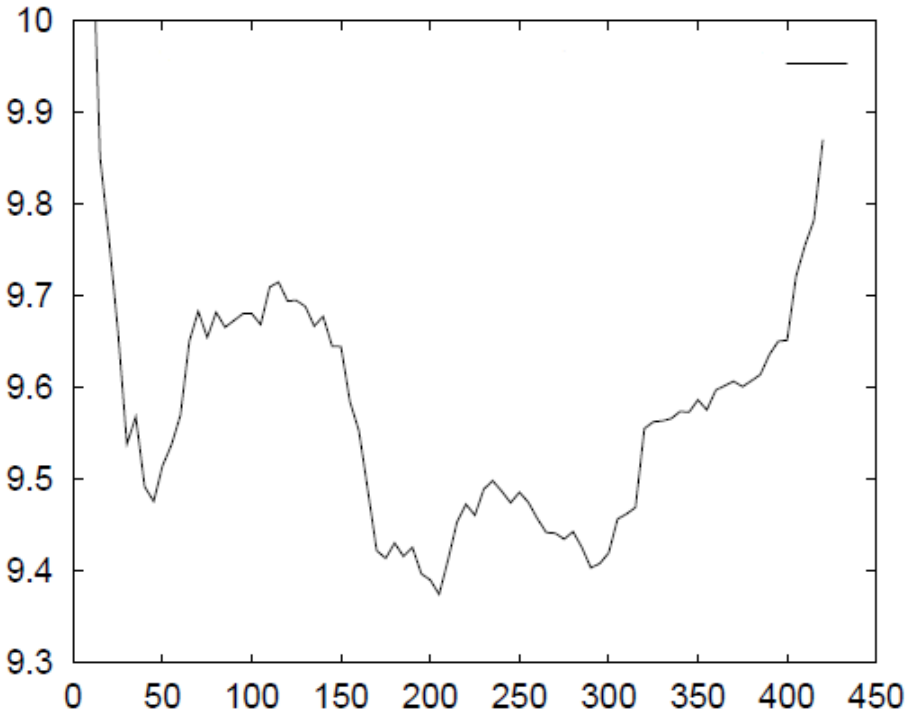
2. **Il processo di training viene interrotto non appena l'errore sul validation set inizia a crescere.**

3. La rete restituita dopo il processo di learning è quella con il miglior livello di accuratezza raggiunto al momento della fermata.

3.1 Introduzione alla problematica di fermata

Nel processo di addestramento della rete neurale, l'errore sul validation set non ha un andamento uniforme ma ha un andamento come mostrato in figura. Come si vede, l'errore sul validation set può ancora scendere

Figura 1: Andamento dell'errore sul validation set.



dopo aver iniziato ad aumentare: in un applicazione reale non si conosce mai l'esatto errore di generalizzazione ma lo si stima invece attraverso l'errore sul validation set. In generale non esiste una regola per decidere quando ci si trova in presenza del migliore errore di generalizzazione ed in pratica ci si trova in presenza di più minimi locali senza avere la certezza di aver raggiunto quello *globale*. La problematica del criterio di fermata

è basata sulla ricerca di un giusto compromesso tra l'accuratezza e il numero di epoche necessarie per ottenere tale accuratezza: nel seguito saranno illustrate due tecniche di early stopping e in conclusione saranno analizzate le prestazioni dei criteri scelti.

3.2 Generalization Loss

L'idea che sta alla base del criterio detto *generalization loss* è quella di interrompere il processo di apprendimento non appena si raggiunge un determinato livello di errore sul *validation set*. Dunque a differenza del criterio di stopping più semplice, ovvero quello in cui ci si ferma non appena l'errore sul *validation set* sale invece di continuare nella sua discesa, *generalization loss* continua l'apprendimento fino ad un determinato valore della funzione che è specificato dal parametro *alpha*. E' facile allora immaginare che, visto che in fase di apprendimento l'errore sul validation set può ancora scendere anche dopo essere aumentato, questo criterio di stopping cerca di arginare quanto più possibile il limite imposto dal più semplice criterio di stop prima illustrato.

Possiamo descrivere formalmente questo criterio come segue:

" Sia E la funzione di errore scelta per l'algoritmo di training, ad esempio la Cross Entropy come nel nostro caso. Quindi si definisca con $\boldsymbol{E_{val}(t)}$ *l'errore sul validation set corrispondente all'epoca* $\boldsymbol{t}$ *e si ricordi che dato che nel caso reale l'errore di generalizzazione è sconosciuto, l'unico modo per effettuare una stima di tale errore è quello di basarsi proprio sull'errore sul validation set. Inoltre si indichi con* $\boldsymbol{E_{opt}(t)}$ *come il più basso errore ottenuto sul validation set sino all'epoca* $\boldsymbol{t}$ *e lo si definisca come:*

$$\boldsymbol{E_{opt}(t) := min_{t' \leq t}\ E_{va}(t')}$$

Date queste definizioni preliminari è possibile definire ***generalization loss*** *all'epoca t come l'incremento relativo dell'errore di validazione sul minimo fin ora raggiunto, in formule:*

$$\boldsymbol{GL(t) = 100 * (\frac{E_{\mathrm{va}}}{E_{\mathrm{opt}}} - 1)}$$

In generale un alto tasso di ***generalization loss*** *è una ovvia ragione di fermare il processo di training poichè ciò è un indicatore esplicito di* ***overfitting*** *e dunque di perdita di generalizzazione. Questa definizione dunque ci porta all'idea di questo criterio:*

Il processo di training viene interrotto non appena il valore di GL eccede una certa soglia α

Dunque si definisce GL_α *come:*

GL_α: ci si ferma dopo la prima epoca t con GL(t) $> \alpha$

Questo criterio, come è facilmente osservabile, tiene conto solamente dell'andamento del validation set e non del training set: il prossimo criterio sarà incentrato sulle informazioni che provengono sia dall'andamento dell'errore sul validation set che sul training set."

3.3 Progressive Quotient

Come anticipato nel paragrafo precedente si potrebbe voler continuare il processo di training se l'errore sul training set sta continuando a scendere **molto rapidamente**. Il ragionamento dietro questo approccio è che quando l'errore sul training sta decrescendo ancora abbastanza velocemente la perdita di generalizzazione ha ancora delle possibilità di essere *recuperata e migliorata*. Si assume perciò che spesso l'**overfitting** non inizi fino a quando l'errore non inizia a decrescere lentamente.

Possiamo definire formalmente il criterio come segue:

"Iniziamo col definire una ***training strip di lunghezza k*** *come una sequenza di* ***k*** *epoche numerate come n+ 1 ... n + k con n divisibile da k. Il* ***training progress*** *(per migliaia) misurato dopo tale* ***training strip*** *è definito come:*

$$\boldsymbol{P_k := 1000 * \left(\frac{\sum_{t'=t-k+1}^{t} E_{\mathrm{tr}}(t')}{k * \min_{t'=t-k+1}^{t} E_{\mathrm{tr}}(t')} - 1\right)}$$

Cioè ***quanto è piu grande in media l'errore sul training set rispetto all'errore minimo sul training durante la strip.*** *Dopo aver definito* $\boldsymbol{P_k}$ *è possibile definire il criterio detto* ***Progressive Quotient*** *che è definito come una combinazione di* ***Generalization loss*** *e* $\boldsymbol{P_k}$*.*

PQ_α: ci si ferma dopo la prima epoca dopo la fine della strip con $\frac{GL(t)}{P_k(t)}$

″

Dunque questo secondo criterio di stop tiene conto sia dell'andamento del training set *che di quello del* validation set.

3.4 Analisi delle prestazioni

Lo scopo di questo lavoro è quello di implementare i due criteri di stop sopra illustrati e di analizzarne le prestazioni in termini di **accuratezza** del risultato restituito e di **epoche** impiegate per restituire tale risultato. Questo capitolo si apre con una illustrazione della configurazione iniziale delle reti sulle quali sono stati raccolti tutti i dati ed infine viene chiuso dalla discussione vera e propria sui risultati ottenuti.

3.4.1 Configurazione iniziale

Approccio: per misurare le prestazioni della rete, utilizziamo un set di dati MNIST partizionato in tre parti disgiunte: training set, validaiton set e test set. Il training set viene utilizzato per regolare i pesi della rete, il validation set utilizzato per stimare le prestazioni della rete durante il training, come richiesto dai criteri di stop.

Criteri di arresto: i criteri di arresto esaminati per $\mathbf{GL}_\alpha$ e per $\mathbf{PQ}_\alpha$ sono:

- $\mathbf{GL}_{46}$,$\mathbf{GL}_{50}$,$\mathbf{GL}_{54}$,$\mathbf{GL}_{56}$,$\mathbf{GL}_{66}$,$\mathbf{GL}_{76}$,$\mathbf{GL}_{86}$,$\mathbf{GL}_{96}$,$\mathbf{GL}_{106}$,$\mathbf{GL}_{116}$
- $\mathbf{PQ}_{0.003}$,$\mathbf{PQ}_{0.005}$,$\mathbf{PQ}_{0.008}$,$\mathbf{PQ}_{0.01}$,$\mathbf{PQ}_{0.012}$,$\mathbf{PQ}_{0.014}$, $\mathbf{PQ}_{0.018}$, $\mathbf{PQ}_{0.021}$,$\mathbf{PQ}_{0.024}$,$\mathbf{PQ}_{0.03}$

Learning tasks: il problema preso in considerazione è quello di classificare una serie di immagini 28x28 presi dal data set MNIST. Ogni immagine risulta essere un digit scritto a mano. La rete ha un numero di input pari a 28x28=784, mentre 10 sono gli output della rete, pari al numero di cifre da classificare. Il numero di esempi che il dataset mette a disposizione è 50000 per il training set, 5000 per il validation set e 10000 per il test set.

Dataset e architetture di rete: dei 50000 esempi messi a disposizione da MNIST ne sono stati utilizzati 500 per il processo di training. Su questi dati è stato eseguito un processo di training su un numero 10 reti feedforward multistrato full connected con un singolo strato di nodi interni. Il numero di nodi interni presi in considerazione è stato: **2, 4, 8, 16, 24, 32**. In totale per ciascun **alpha** scelto sono state addestrate 60 reti:

- Per alpha si ha una media dei risultati su **10 reti con 2 neuroni, 10 reti con 4 neuroni, 10 reti con 8 neuroni**, ... , **10 reti con 32 neuroni**.

E' stato adottato questo criterio di valutazione perchè valutando la media della prestazione con un diverso numero di neuroni non ci si focalizza sui neuroni presenti nello strato interno ma la valutazione è il quanto più generale possibile.

Algoritmi di training Tutte le analisi sono state eseguite utilizzando l'algoritmo di addestramento RPROP utilizzando come funzione di errore la cross entropy con parametri $\eta^+ = 1.05$, $\eta^- = 0.5$, $\Delta \in 0.05...0.2$ scelti in maniera casuale, $\Delta max = 50$, $\Delta min = e^{-6}$. RPROP richiede un apprendimento basato su epoche, vale a dire che i pesi vengono aggiornati solo una volta per epoca. **Il massimo numero di epoche scelto è 300**.

3.4.2 Risultati

E' molto interessante analizzare fin quando è utile aumentare il parametro α per ottenere prestazioni migliori: in particolare, come analizzato in questo ultimo paragrafo, è stato osservato che aumentare il parametro α non sempre porta grandi miglioramenti e richiede a volte un numero di epoche superiore per apportare miglioramenti che sono dell'ordine di pochi punti decimali. Analizziamo in dettaglio i risultati ottenuti per **Generalization Loss** e **Progressive Quotient**.

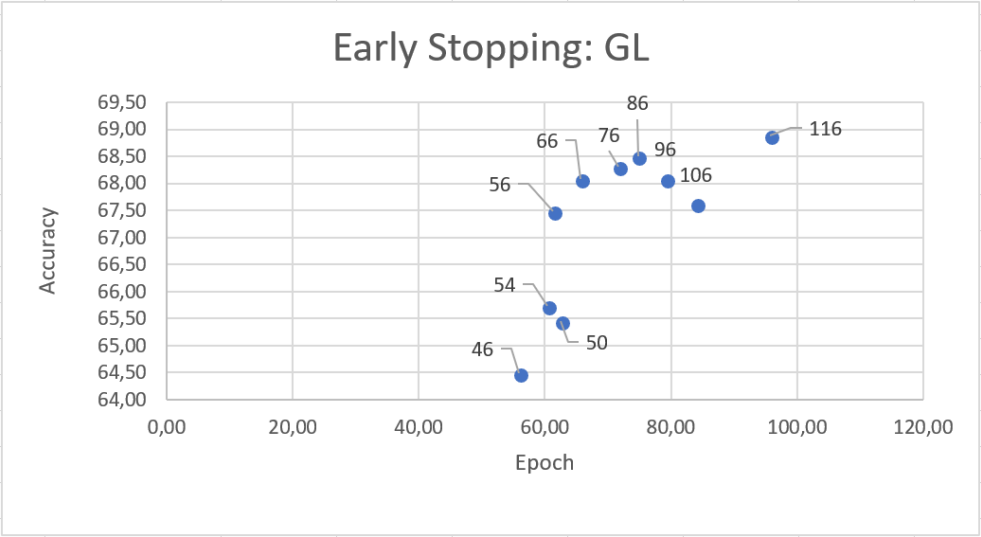
Generalization Loss

#	Epoche	Accuratezza	Alpha
1	55,85	64,48	46
2	62,53	65,46	50
3	60,42	65,73	54
4	61,23	67,49	56
5	65,70	68,09	66
6	71,70	68,31	76
7	74,73	68,50	86
8	79,10	68,09	96
9	83,95	67,62	106
10	95,67	68,89	116

Dall'analisi su **GL** si evince che l'aumento del parametro α comporta un aumento significativo dell'accuratezza registrata sulla rete: questo risultato era facilmente prevedibile perchè essendo questo criterio basato solo sul *validation set* ed essendo l'errore su quest'ultimo soggetto a crescite e cali repentini, dovuti al metodo di learning batch *resilient backpropagation*, il parametro α fa si che il processo di learning sia più *tollerante* a

queste variazioni dell'errore sul validation set e che si continui tale processo per la ricerca di un eventuale rete con accuratezze migliori.
E' interessante osservare come a partire da $\mathbf{GL_{66}}$ si raggiunge un punto in cui anche aumentando il valore α la rete non ottiene più un grande vantaggio sull'accuratezza ottenuta ed al contempo impiega un numero di epoche molto più grande: in particolare se si confronta $\mathbf{GL_{66}}$ e $\mathbf{GL_{116}}$ è osservato che si ottiene un vantaggio di soli **0,89** punti sull'accuratezza al costo di **95 epoche** contro le **65 epoche** dell'alfa più piccolo.

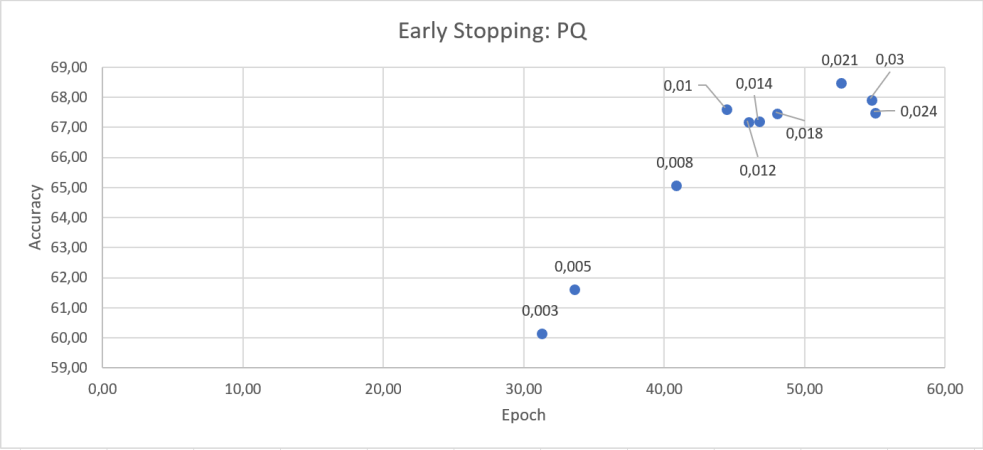
In conclusione è possibile affermare che rispetto al dataset da noi utilizzato per questo esperimento è consigliato l'utilizzo di criteri di stopping quali $\mathbf{GL_{30}}$ fino a $\mathbf{GL_{66}}$, in quanto si evince un importante miglioramento dell'accuratezza entro quest'intervallo. Non conviene utilizzare criteri di stopping quali $\mathbf{GL_{67}}$ o superiori poichè, in quanto riscontrato su questo dataset e con la configurazione di rete illustrata, non si ottiene più un vantaggio significativo. E' inoltre possibile osservare l'andamento dei vari processi di addestramento con GL nel seguente grafico.



Progressive Quotient

#	Epoca	Accuratezza	Alpha
1	31.17	60.19	0.003
2	33.50	61.65	0.005
3	40.75	65.11	0.008
4	44.33	67.65	0.010
5	45.92	67.23	0.012
6	46.67	67.25	0.014
7	47.92	67.50	0.018
8	52.50	68.52	0.021
9	54.92	67.53	0.024
10	57.67	67.96	0.030

Per il criterio si stop PQ abbiamo sì un andamento crescente dell'accuratezza al crescere del parametro α, ma tale aumento non è così sostanziale rispetto al numero di epoche richieste. In particolar modo si evince che a partire da $\mathbf{PQ_{0.01}}$ l'accuratezza continua a crescere di pochi decimi ma richiede un numero di epoche maggiore. Salta subito all'occhio che tra $\mathbf{PQ_{0.01}}$ e $\mathbf{PQ_{0.03}}$ c'è un guadagno di soli 0.31 sull'accuratezza ed un numero di epoche richiesto per $\mathbf{PQ_{0.03}}$ che è di **57** contro le **44** di $\mathbf{PQ_{0.01}}$.



In conclusione è possibile affermare che rispetto al dataset da noi utilizzato per questo esperimento è consigliato l'utilizzo di criteri di stopping quali $\mathbf{PQ_{0.001}}$ fino a $\mathbf{PQ_{0.01}}$, in quanto si evince un importante miglioramento dell'accuratezza entro quest'intervallo. Non conviene utilizzare criteri di stopping quali

$PQ_{0.012}$ o superiori poichè, in quanto riscontrato su questo dataset e con la configurazione di rete illustrata, non si ottiene più un vantaggio significativo. Inoltre si osserva un numero di epoche minore per raggiungere accuratezze superiori rispetto al criterio di stop precedentemente osservato. Il parametro k, ovvero la strip di epoche, scelto per questo esperimento è 5.

4 Bibliografia

Riferimenti bibliografici

[1] "Early stopping, but when?", Lutz Prechelt, Fakultät für Informatik, Universität Karlsruhe D-76128 Karlsruhe, Germany, prechelt@ira.uka.de, http://www.ipd.ira.uka.de/prechelt/