Machine Learning (654aa), Anno Accademico 2019/2020

Implementazione di una rete neurale e relativa sperimentazione

Riccardo Manetti (582514) - r.manetti1@studenti.unipi.it
Gemma Martini (532769) - martini.gemma3@gmail.com

Sommario

In questa relazione si presentano la struttura e le performance di una rete neurale con flusso in avanti (più nota con il termine inglese "feed forward").

L'implementazione in Python di suddetta rete ha seguito lo stile di sviluppo "agile" e gli step raggiunti sono documentati in Sezione 2. Le capacità di allenamento ed apprendimento della rete in oggetto sono testate (1) sui 3 dataset noti in letteratura con il nome di Monk, per valutare le prestazioni su un problema di classificazione, e (2) sul dataset `ML_CUP`, con scopi di valutazione didattica su un problema di regressione. I risultati ottenuti sono descritti in Sezione 3.1 e Sezione 3.2 rispettivamente e le osservazioni a riguardo sono presenti in Sezione 4.

1 Introduzione

L'obiettivo fissato per il progetto è quello di ottenere una rete neurale in grado di apprendere velocemente e generalizzare su problemi di classificazione e di regressione. L'idea di fondo dietro allo sviluppo di tale rete neurale è la possibilità di personalizzare l'esecuzione a seconda del problema di apprendimento da risolvere. Per permettere tale libertà di scelta dei parametri si è scelto di strutturare il codice utilizzando interfacce e classi istanziate per le varie componenti necessarie alla funzionalità della rete. Sono stati successivamente eseguiti una serie di esperimenti, assegnando diversi valori a ciascun parametro, per valutare il metodo di apprendimento della rete su differenti problemi.

2 Metodo

2.1 Ambiente di sviluppo

Il linguaggio scelto per implementare la rete neurale è Python 3.6, utilizzato sfruttando la libreria `numpy` (fondamentale per effettuare semplicemente operazioni tra vettori, matrici e tensori), la libreria `matplotlib` essenziale per una chiara rappresentazione grafica delle performance dei modelli realizzati e molte altre, che si trovano citate nel codice sorgente (disponibile su GitHub).

2.2 Rete neurale

L'idea di fondo dietro allo sviluppo di questa rete neurale è la possibilità di personalizzare l'esecuzione a seconda del problema di apprendimento da risolvere. A tale proposito, lo sviluppo della rete neurale è proceduto creando un primo scheletro, in cui erano presenti classi e funzioni prototipo che rappresentavano l'astrazione di una certa funzionalità.

Più concretamente, si è partiti creando un file principale (`main`), contenente un'istanza della classe `neural_network` che eseguiva passi in avanti ("feed forward") ed indietro ("backpropagation"), implementati nel file di utilità `nn_utilities.py`.

Le interfacce `act_sigma_interface`, `optimizer` e `loss_interface` servono per fornire lo scheletro relativo alle funzioni di attivazione, agli ottimizzatori ("momentum") e funzioni di errore ("Loss"). In Tabella 1 si riportano gli scheletri delle interfacce e le classi che le implementano. Ai fini di questa trattazione si è scelto di implementare `logistic`, `relu`, `tanh` e `softmax` come funzioni di attivazione; `trivial`, `nesterov` e `adam` per i momenti; `mee` e `log_loss` come funzioni di errore.

```
class f_sigma_interface(object):
  def __init__(self):
    pass

  def compute(self, p):
    raise Exception(
      "NotImplementedException")

  def chain_rule(self, df_dp, p):
    raise Exception(
      "NotImplementedException")


class logistic(f_sigma_interface)
   {...}

class relu(f_sigma_interface)
   {...}

class tanh(f_sigma_interface)
   {...}

class softplus(f_sigma_interface)
   {...}
```

```
class optimizer_interface(object):
  def __init__(self):
    pass

  def gradient_calculation_point(
    self, weigths):
    raise Exception(
      "NotImplementedException")

  def update_weights(
    self, lr, b_size, grad, weights):
    raise Exception(
      "NotImplementedException")


class adam(optimizer_interface)
  {...}

class trivial(optimizer_interface)
  {...}

class neterov(optimizer_interface)
  {...}
```

```
class loss_interface(object):
  def __init__(self):
    pass

  def compute(self, a, b):
    raise Exception(
      "NotImplementedException")

  def partial(self, df_dp, p):
    raise Exception(
      "NotImplementedException")


class mee(loss_interface)
   {...}

class log_loss(loss_interface)
   {...}
```

Tabella 1: Estratto del codice delle interfacce implementate.

Nell'implementazione della rete sono state istanziate tre diverse classi descritte sopra, che rappresentano gli ottimizzatori, per aumentare la velocità di convergenza, molto lenta nel caso in cui il gradiente sia molto piccolo (quando ci si avvicina ad un minimo locale).
Il primo ottimizzatore implementato è stato introdotto nel 1964 da Boris T. Polyak [1] ed è formalizzato, all'iterazione t-esima, come segue

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \left(-\eta \cdot \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} + \alpha \Delta \mathbf{w}_t\right)$$

Se denotaimo con $\mathbf{v}_{t+1}$ la quantità da aggiungere a $\mathbf{w}_t$ per ottenere $\mathbf{w}_{t+1}$ ($\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1}$) si ottiene la forma equivalente

$$\mathbf{v}_{t+1} = -\eta \cdot \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} + \alpha \mathbf{v}_t$$

La seconda istanza dell'interfaccia `optimaizer` implementa il cosiddetto "Nesterov accellerated gradient" [2] che differisce dal precedente poiché tiene conto della lunghezza del passo compiuto all'iterazione precedente per selezionare il punto in cui calcolare il gradiente all'iterazione corrente. In formule,

$$\mathbf{w}' = \mathbf{w}_t + \alpha \mathbf{v}_t$$

$$\mathbf{v}_{t+1} = -\eta \cdot \frac{\partial E(\mathbf{w}')}{\partial \mathbf{w}} + \alpha \mathbf{v}_t$$

Negli ultimi anni ha preso molto piede, nell'ambito dell'ottimizzazione degli algoritmi basati su "stochastic gradient descent" l'utilizzo dell'Adam ("Adaptive Moment Estimation") [3], il quale usa una media con decadimento esponenziale dei vecchi gradienti e si formalizza come segue:

$$\mathbf{v}_{t+1} = -\frac{\eta}{\sqrt{\widehat{\mathbf{v}}_t} + \varepsilon} \cdot \widehat{\mathbf{m}}_t$$

Dove,

$$\begin{cases} \mathbf{m}_0 = \mathbf{0} \\ \mathbf{m}_t = \beta_1 \cdot \mathbf{m}_{t-1} + (1-\beta_1) \cdot \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \\ \widehat{\mathbf{m}}_t = \mathbf{m}_t / (1-\beta_1^t) \end{cases} \quad \text{e} \quad \begin{cases} \mathbf{v}_0 = \mathbf{0} \\ \mathbf{v}_t = \beta_2 \cdot \mathbf{v}_{t-1} + (1-\beta_2) \cdot \left(\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right) \\ \widehat{\mathbf{v}}_t = \mathbf{v}_t / (1-\beta_2^t) \end{cases}$$

Una volta implementati i tre tipi di ottimizzatori appena descritti, sono state sviluppate due funzioni, la prima che permettesse di effettuare la "k-fold cross validation" e la seconda che permettesse di scegliere con quale distribuzione inizializzare i pesi, come descritto in seguito.

Seguendo la filosofia della massima personalizzabilità della rete, si è consentito fin da subito di creare la rete ed allenarla utilizzando valori a scelta per la velocità di apprendimento ("learning rate", η), il numero di strati della rete, il numero di neuroni presenti in ognuno di essi, parametro della regolarizzazione di Tikhonov (λ) e la dimensione del campione esaminato prima di aggiornare i pesi ("batch size"). Quest'ultimo parametro permette di distinguere tra la versione on-line, impostandolo a 1, quella "batch", impostandolo pari alla dimensione dell'insieme di allenamento, e la cosiddetta "mini-batch", della quale viene specificata la taglia.

Nell'anno 2010 Glorot et.al. [4] evidenziano che nel semplice caso di una rete neurale formata da un solo livello, è sufficiente inizializzare i pesi con valori piccoli in modulo (ad esempio 0.01), generati casualmente con distribuzione uniforme.
Il lavoro di Glorot si è rivelato fondamentale per la scelta dei valori iniziali da assegnare alle matrici dei pesi, della rete in questione, poiché è possibile instanziarla con più di un livello nascosto. Considerando una matrice dei pesi $W \in \mathcal{M}(m, n, \mathbb{R})$, dove n è il numero di attributi (o "features") e m rappresenta il numero di neuroni contenuti nel livello nascosto, si distinguono per $i = 1, \ldots, m$ e per $j = 1, \ldots, n$:

- distribuzione gaussiana $W_{ij} \in N\left(\ \mu = 0,\ \sigma^2 = \frac{2}{n+m}\ \right)$
- distribuzione uniforme $W_{ij} \in U\left[\ -\sqrt{\frac{6}{n+m}}, +\sqrt{\frac{6}{n+m}}\ \right]$

Per la valutazione dei modelli sono istanzate due differenti metriche, ciascuna delle quali risulta migliore a seconda del problema da risolvere. Nel caso si abbia un problema di classificazione, è preferibile l'utilizzo della cosiddetta *Log-Loss* (conosciuta anche con il nome "cross-entropy"), definita come

$$\textit{Log-Loss}(\mathbf{o}, \mathbf{y}) = -\frac{1}{l} \cdot \sum_{i=1}^{l} t_i \cdot \log(o_i) + (1 - t_i) \cdot \log(1 - o_i)$$

dove $\mathbf{y}$ è il vettore della variabile obiettivo e $\mathbf{o}$ è il vettore dell'output ottenuto dalla predizione dell'input.
Nel caso in cui invece si è difronte ad un problema di regressione, per valutare il modello si preferisce utilizzare la *Mean Euclidean Error* (MEE), la cui funzione, sempre per il vettore della variabile obiettivo $\mathbf{y}$ e il vettore dell'output $\mathbf{o}$, risulta

$$\textit{MEE}(\mathbf{o}, \mathbf{y}) = \|\mathbf{y} - \mathbf{o}\|^2 = \sum_{j=1}^{n} (y_j - o_j)^2$$

Dato che la rete neurale sviluppata esegue sia fasi di feed-forward che di backpropagation, delle rispettive funzioni di valutazione dei modelli appena descritte, e come illustrato nell'estratto di codice in Tabella 1 dell'interfaccia per le funzioni di errore, sono state implementate le funzioni per calcolare le derivate della *Log-Loss* e della *MEE*, per ogni pattern p:

$$\frac{\partial \textit{Log-Loss}}{\partial o_p} = -\frac{y_p}{o_p} + \frac{1 - y_p}{1 - o_p} \qquad \text{e} \qquad \frac{\partial \textit{MEE}}{\partial o_p} = 2 \cdot (o_p - y_p)$$

È necessario sottolineare che per allenare la rete neurale proposta su un problema di classificazione binaria si deve scegliere per l'ultimo strato della rete una funzione di attivazione che permetta di mappare i valori ottenuti nell'intervallo $[0, 1]$, in modo da rappresentare la probabilità che la classe predetta sia 0 o 1.

2.3 Pre-elaborazione e Validazione

Come descritto dallo studio di Potdar et.al. [5], utilizzare la codifica "one-hot" permette di migliorare le prestazioni di un modello di intelligenza artificiale, dunque si è scelto di pre-processare i tre dataset di classificazione (Monk's) facendo uso di tale codifica.

Come largamente usato in letteratura, si sono valutate le capacità di allenamento e predizione della rete neurale proposta sui dataset a disposizione per mezzo della "k-fold cross validation". Attraverso tale logica, il miglior modello, è stato poi valutato sull'insieme di test. A differenza dei tre Monk, per quanto riguarda il dataset ML_CUP non si ha a disposizione il valore della variabile obiettivo nell'insieme di test. Per poter ottenere una valutazione del miglior modello, si è scelto di suddividere l'insieme di allenamento ("training set") in due sottoinsiemi, 20% per il test e il restante 80% per il training (il quale poi viene ulteriormente suddiviso nella "k-fold cross validation").

La valutazione di un rete a seguito di un'unica istanziazione ed un'unico allenamento può portare a dei risultati non del tutto veritieri, considerando anche il fatto che i pesi vengono inizialmente creati secondo una distribuzione, ma comunque in modo casuale. Per questo motivo, è più corretto eseguire per un determinato numero di volte la creazione e l'allenamento di un modello sugli stessi parametri prima di esprimere una valutazione corretta.

3 Dati sperimentali

È in questa sezione che si entra nel vivo delle sperimentazioni effettuate usando la rete neurale sopra descritta, per costruire modelli in grado di risolvere problemi di classificazione e di regressione.
Prima di addentrarci nei dettagli di come i due problemi di apprendimento sono stati affrontati, si vuole premettere ciò che li accomuna, ovvero il meccanismo di allenamento/validazione basato sulla ben nota "k-fold cross validation", atta a selezionare il modello e poi valutarlo sull'insieme di test.

Durante la fase di test sono valutate circa 600 mila combinazioni per il problema di classificazione e circa 57 mila combinazioni per il problema di regressione. I calcoli per la "grid search" su tali combinazioni sono stati effettuati attraverso una macchina dotata di 64-Core Processor AMD Ryzen Threadripper 3970X (cpu MHz: 3819.793).

Si analizzano adesso più nel dettaglio gli esperimenti condotti sulle due tipologie di problemi di apprendimento (rispettivamente classificazione e regressione).

3.1 Un problema di classificazione: Monk's dataset

Il problema di classificazione affrontato si sviluppa nello studio dei tre set dataset noti con il nome di Monk [6]: in ognuno di questi la variabile obiettivo è di tipo binario, mentre gli attributi sono 6,

Parametro	Valori
η ("learning rate")	0.001, 0.01, 0.05, 0.1, 0.2, 0.4, 0.6, 0.8
♯ neuroni nascosti	[3], [5], [5, 5], [10], [20], [100], [150], [20, 20], [100, 100], [200, 200], [2, 2, 2]
distribuzione dei pesi	gaussiana e uniforme
dimensione batch	1, 2, 4, 8, 16, 24, 32, 64, ♯TR
funzione di attivazione	`logistic`, `relu`, `tanh` e `softplus`
λ (regolarizzazione)	0.0, 0.00001, 0.0001, 0.001
ottimizzatore	`adam` (0.9, 0.999, $1e-08$), `nesterov`(0.5), `nesterov`(0.9), `trivial`(0.0), `trivial`(0.5), `trivial`(0.9)

Tabella 2: Valori utilizzati per ciascun parametro nella grid-search nel problema di classificazione.

tutti di tipo categorico. Prima di procedere con l'allenamento della rete sui parametri descritti in Tabella 2 si è scelto di trasformare i dati di input mediante la codifica "one-of-k" che trasforma valori interi in vettori in uno spazio vettoriale a valori binari di dimensione pari al numero di classi assunte dall'attributo. Ciascuno Monk possiede il proprio insieme di allenamento e di test su cui è possibile validare e testare il modello ottenuto.

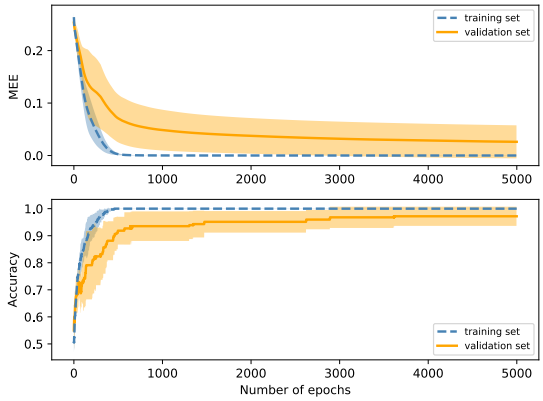
In Tabella 2 sono stati riportati tutti i valori assegnati a ciascun parametro durante l'esecuzione della "grid-search". Si può notare che per ciascuno dei parametri sono stati scelti molti valori differenti, successivamente raffinati riducendone l'intervallo durante le varie fasi di test.
Come è possibile osservare dai valori assunti dal parametro della dimensione del "batch" sono state testate la versione batch, con valore pari alla cardinalità dell'insieme di allenamento (♯TR), la versione online, con valore pari a 1, e la versione mini-batch, con una serie di valori compresi tra 1 e ♯TR (estremi esclusi).

Le due differenti implementazioni delle funzioni di errore, hanno permesso di stilare per ogni Monk due classifiche distinte riportate nella Tabella 3. Le capacità di allenamento e di generalizzazione del miglior modello per ciascuno dei tre Monk in esame è illustrata attraverso i grafici in Figura 1, Figura 2 e Figura 3 per il rispettivo Monk.

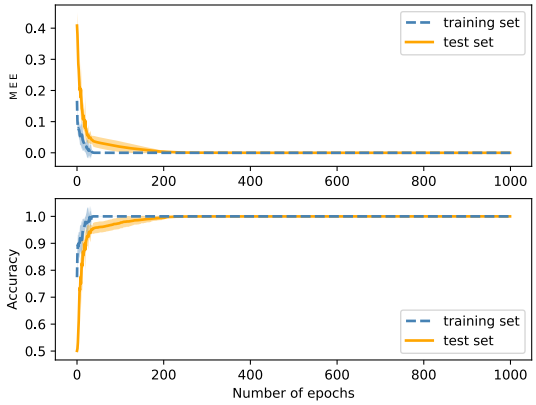
	Loss	η	♯ neuroni nascosti	distribuzione dei pesi	dimensione batch	funzione attivazione	λ	ottimizzatore	**k**	Errore
Monk 1	MEE	0.2	[150]	gaussian	32	tanh	0.0	trivial (0.5)	5	0.0326
		0.01	[150]	gaussian	8	tanh	0.0001	trivial (0.9)	5	0.0370
		0.1	[200, 200]	gaussian	8	tanh	0.0001	trivial (0.5)	5	0.0370
	Log	0.2	[100, 100]	gaussian	16	logistic	0.0	adam	3	0.4836
		0.4	[5, 5]	gaussian	4	logistic	0.0	adam	3	0.5114
		0.2	[100, 100]	gaussian	124	tanh	0.001	adam	3	0.5922
Monk 2	MEE	0.05	[20,20]	gaussian	4	tanh	0	trivial 0.5	5	0.0296
		0.01	[100,100]	gaussian	1	relu	0	trivial 0.0	8	0.0298
		0.05	[3]	gaussian	1	softplus	0.0001	adam	3	0.0317
	Log	0.06	[20]	gaussian	169	logistic	0	adam	3	0.5569
		0.001	[3]	gaussian	8	relu	0	trivial 0	3	0.5620
		0.001	[10]	gaussian	16	relu	0.001	nesterov 0.5	3	0.5774
Monk 3	MEE	0.2	[150]	gaussian	32	tanh	0	trivial 0.5	5	0.0327
		0.01	[150]	gaussian	8	tanh	0.0001	trivial 0.9	5	0.0370
		0.1	[200,200]	gaussian	8	tanh	0.0001	trivial 0.5	5	0.0370
	Log	0.01	[5]	gaussian	123	logistic	0	adam	3	0.2711
		0.001	[5,5]	gaussian	8	tanh	0	trivial 0.0	3	0.3031
		0.001	[3]	gaussian	64	tanh	0	trivial 0.5	3	0.3311

Tabella 3: Parametri e valore di errore dei migliori 3 modelli per ciascun Monk e suddivisi secondo la funzione di errore (Loss).

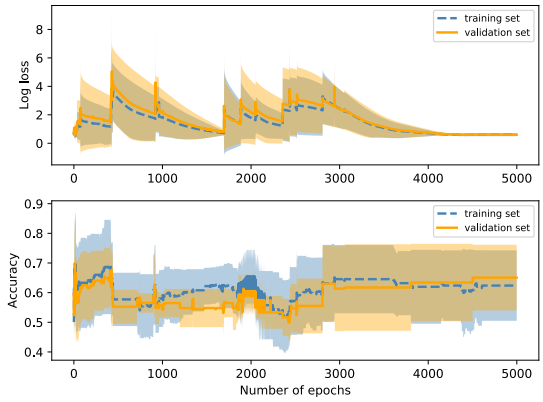
Per quanto riguarda il Monk 1, le figure 1(a) e 1(b) illustrano i risultati ottenuti rispettivamente sull'insieme di allenamento e l'insieme di test, dal miglior modello utilizzando *MEE* come metrica di valutazione. Le figure 1(c) e 1(d) illustrano invece i risultati ottenuti dal miglior modello selezionato per la *Log-Loss*. Si osserva che le capacità predittive della rete neurale per questo Monk sono ottime quando si utilizza *MEE*, mentre risultano instabili e molto dipendenti dall'inizializzazione dei pesi nel caso di *Log-Loss*.



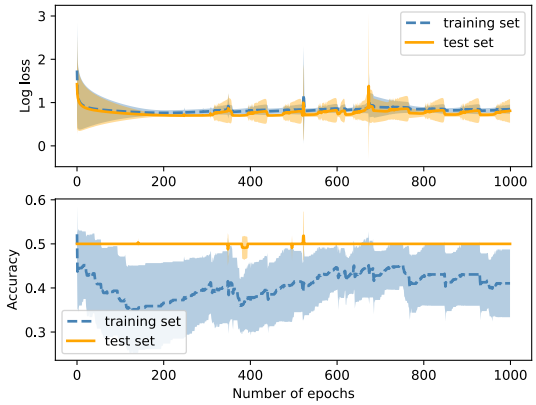
(a) Monk 1 - *MEE* - training



(b) Monk 1 - *MEE* - test



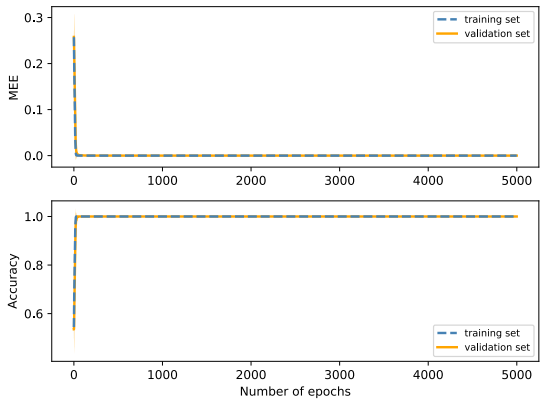
(c) Monk 1 - *Log-Loss* - training



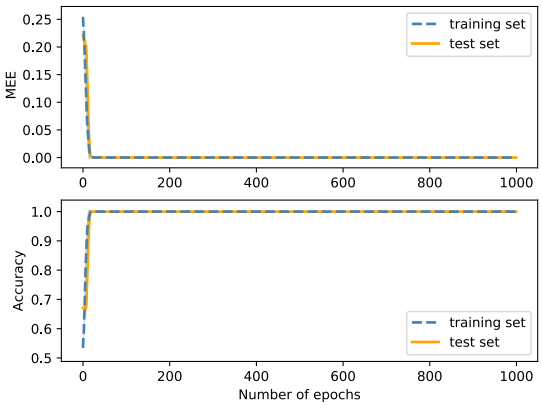
(d) Monk 1 - *Log-Loss* - test

Figura 1: Grafici della funzione di errore e dell'accuratezza ("Accuracy") per il miglior modello del Monk 1 su ciascuna metrica.

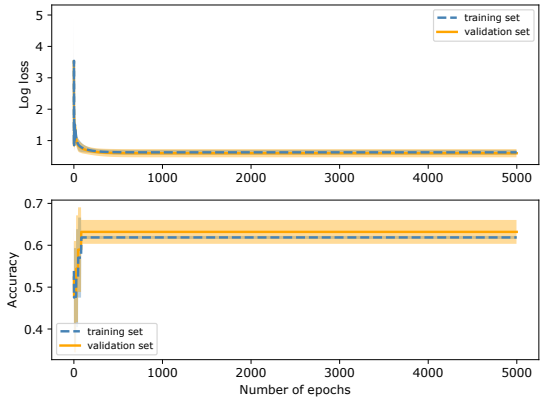
Trattando invece il Monk 2, le figure 2(a) e 2(b) illustrano i risultati ottenuti rispettivamente sull'insieme di allenamento e l'insieme di test, dal miglior modello utilizzando *MEE* come metrica di valutazione. Le figure 2(c) e 2(d) illustrano invece i risultati ottenuti dal miglior modello selezionato per la *Log-Loss*. Anche in questo caso si osserva che le capacità predittive della rete neurale sono ottime quando si utilizza *MEE*, mentre risultano leggermente peggiori e maggiormente dipendenti dall'inizializzazione dei pesi nel caso di *Log-Loss*.
Utilizzando *MEE* come misura di errore, sia sull'insieme di allenamento, che in quello di test, la rete converge molto rapidamente con ottime capacità di generalizzazione.



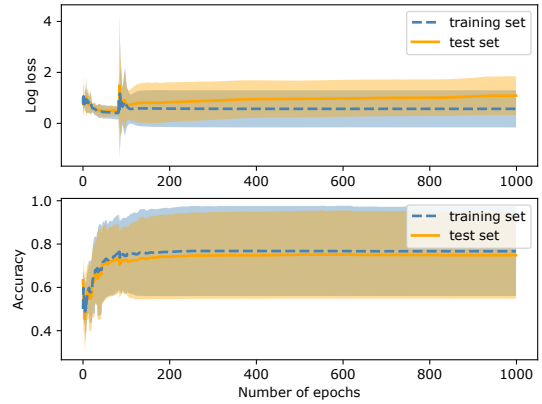
(a) Monk 2 - *MEE* - training



(b) Monk 2 - *MEE* - test



(c) Monk 2 - *Log-Loss* - training



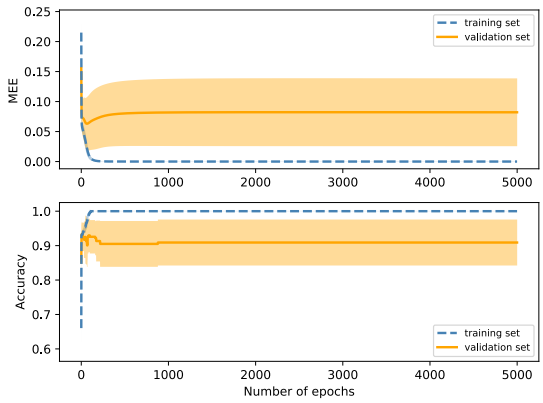
(d) Monk 2 - *Log-Loss* - test

Figura 2: Grafici della funzione di errore e dell'accuratezza ("Accuracy") per il miglior modello del Monk 2 su ciascuna metrica.

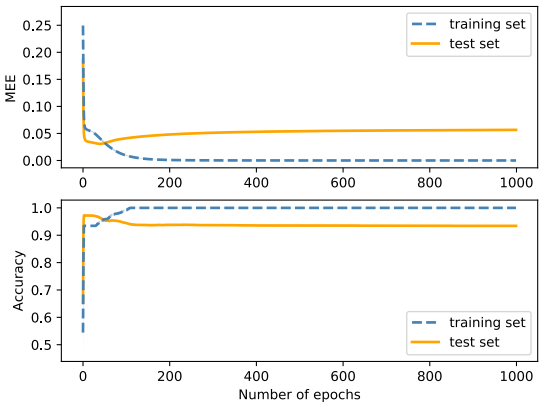
Concludendo con il Monk 3, le figure 3(a) e 3(b) illustrano i risultati ottenuti rispettivamente sull'insieme di allenamento e l'insieme di test, dal miglior modello utilizzando *MEE* come metrica di valutazione. Le figure 3(c) e 3(d) illustrano invece i risultati ottenuti dal miglior modello selezionato per la *Log-Loss*.
Le performance delle reti neurali utilizzando le due metriche differiscono sostanzialmente dai due Monk precedenti, in quanto in questo caso la *Log-Loss* risulta avere buone capacità di generalizzazione e di stabilità. Si noti però che un addestramento che supera le 700 epoche può indurre in "overfitting".
Osservando i grafici relativi alla *MEE*, si giustifica lo spazio tra la curva di allenamento e quella di validazione/test con il fatto che non vi è alcuna regolarizzazione nel miglior modello ($\lambda = 0$).

3.2 Un problema di regressione: ML_CUP

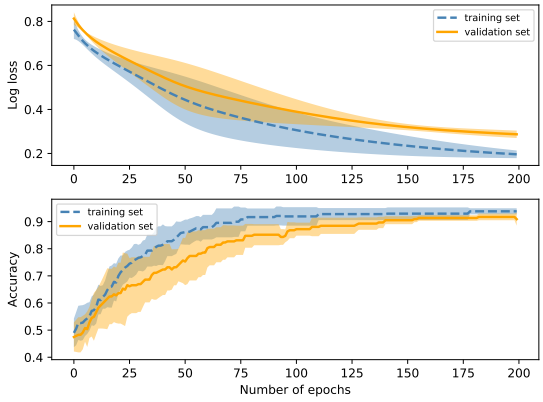
Il problema di regressione ML_CUP si presenta con una variabile obiettivo disposta come tupla di due valori reali e 20 attributi anch'essi reali. A differenza del problema di classificazione descritto sopra, nel caso della ML_CUP non si ha a disposizione un insieme di test. Questo problema è stato ovviato suddividendo l'insieme di allenamento ("training set") in due sottoinsiemi: 20% per il test e il



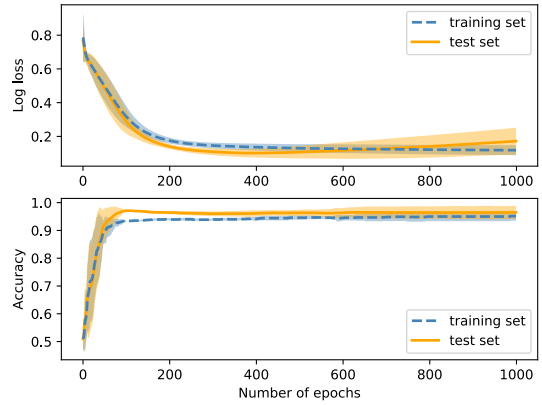
(a) Monk 3 - *MEE* - training



(b) Monk 3 - *MEE* - test



(c) Monk 3 - *Log-Loss* - training



(d) Monk 3 - *Log-Loss* - test

Figura 3: Grafici della funzione di errore e dell'accuratezza ("Accuracy") per il miglior modello del Monk 3 su ciascuna metrica.

restante 80% per il training (il quale poi viene ulteriormente suddiviso nella "k-fold cross validation"). La suddivisione nei due sottoinsiemi è stata effettuata dopo aver permutato le righe dell'insieme di allenamento dato.

La Tabella 4 riporta gli insiemi di valori assegnati agli iperparametri utilizzati nella "grid-search", mentre in Tabella 5 si osserva la classifica dei migliori modelli sul problema di regressione in esame. Si

Parametro	Valori
η ("learning rate")	0.0001, 0.0005, 0.001, 0.005, 0.01
♯ neuroni nascosti	[20], [100], [500], [20, 20], [100, 100], [150, 150], [200, 200], [300, 300]
distribuzione dei pesi	gaussiana, uniforme
dimensione batch	1, 2, 4, 8, 32, 124, 512, 1772
funzione di attivazione	`logistic`, `relu`, `tanh` e `softplus`
λ (regolarizzazione)	0.0, 0.0001, 0.001, 0.01, 0.1
ottimizzatore	`adam` $(0.9, 0.999, 1e-08)$, `nesterov`(0.5), `nesterov`(0.9), `trivial`(0.0), `trivial`(0.5), `trivial`(0.9)

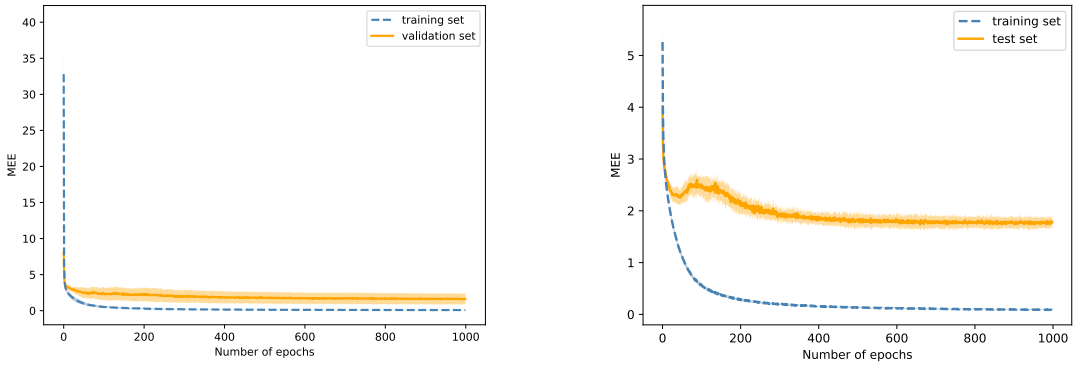
Tabella 4: Valori utilizzati per ciascun parametro nella grid-search nel problema di regressione.

noti che i primi cinque modelli differiscono tra loro solo per i valori di "learning-rate", taglia della batch e l'iperparametro di regolarizzazione (λ), mantenendo fissi l'ottimizzatore, la funzione di attivazione, nonché la distribuzione iniziale dei pesi e la struttura (due livelli nascosti da 200 neuroni l'uno).

	Loss	η	♯ neuroni nascosti	distribuzione dei pesi	dimensione batch	funzione attivazione	λ	ottimizzatore	**k**	Errore
ML_CUP	MEE	0.0005	[200, 200]	gaussian	1	relu	0.0001	adam	5	1.3764
		0.001	[200, 200]	gaussian	4	relu	0.0	adam	5	1.4191
		0.0005	[200, 200]	gaussian	1	relu	0.0	adam	5	1.4321
		0.001	[200, 200]	gaussian	8	relu	0.0001	adam	5	1.4609
		0.001	[200, 200]	gaussian	8	relu	0.0	adam	5	1.4632

Tabella 5: Parametri e valore di errore dei migliori 5 modelli per il problema di regressione ML_CUP.

Come si evince dalla Figura 4, i risultati sperimentali su velocità di apprendimento e capacità di generalizzazione del miglior modello non sono ottimali; infatti il modello ottiene un punteggio alto sull'insieme di test, con scarsa stabilità (si può anche notare un picco tra le 100 e le 200 epoche di allenamento).
Tale comportamento si è riscontrato anche nei quattro modelli riportati in Tabella 5, anch'essi ben posizionati in classifica con errore dell'ordine di grandezza di 1.4.



(a) ML_CUP - training

(b) ML_CUP - test

Figura 4: Grafici della funzione di errore *MEE* ottenuti per il miglior modello del problema di regressione ML_CUP. Le figure (a) e (b) illustrano rispettivamente i risultati ottenuti sull'insieme di allenamento e sull'insieme di test.

4 Conclusioni

A seguito delle sperimentazioni della rete neurale su differenti valori degli iperparametri e quattro diversi task siamo in grado di formulare alcune considerazioni generali.
Prima di tutto, si è appresa l'importanza della complessità computazionale degli esperimenti effettuati, infatti alcuni esperimenti che si sono rivelati molto poco incisivi in termini di performance hanno richiesto per la loro esecuzioni moltissime ore di tempo macchina su più core.
Un'altra importante conclusione è che del codice ben strutturato e modularizzato è essenziale per permettere modifiche e migliorie in corso d'opera ed una semplice e veloce esecuzione degli esperimenti.
Non da meno è l'importanza della resa grafica dei risultati sperimentali, in quanto, metodi esclusivamente algebrici, sebbene tengano in considerazione alcune caratteristiche non riescono a fornire una

chiara idea dell'andamento generale (ad esempio un basso errore ad un'epoca fissata non afferma nulla sulla stabilità della convergenza).

Il nickname del gruppo è *The disimparati*.
Si autorizza la divulgazione e la pubblicazione dei nomi, dei risultati preliminari e della classifica finale.

Riferimenti bibliografici

[1] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[2] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[3] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics.*

[5] Kedar Potdar, Taher Pardawala, and Chinmay Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International Journal of Computer Applications*, 175:7–9, 10 2017.

[6] S. B. Thrun. *The MONK's Problems A Performance Comparison of Different Learning Algorithms.* CMU-CS. Carnegie Mellon Univ., School of Computer Science, 1991.