

Progetto Sistemi Intelligenti - Alberto Borghese
Riconoscitore di cifre manoscritte

Francesco Bertolotti

August 11, 2018

Contents

1	Introduction	3
1.1	Scripts	3
1.2	Dependency	3
1.3	Parameters	3
2	Background	5
2.1	Gradient descent	5
2.2	MLPNN: Multi Layer Perceptron Neural Network	7
2.3	Convolution	9
2.4	CNN: Convolutional Neural Network	11
2.5	Dropout	13
2.6	RMSprop	14
3	Models	14
3.1	built MLPNN	14
3.2	built CNN	18
4	Conclusions	21
4.1	Examples	22
4.2	Improvements	24
4.3	References	24

1 Introduction

Come concordato, ho deciso di sfruttare alcuni concetti mostrati durante il corso per implementare un riconoscitore di cifre manoscritte sfruttando le reti neurali. Il progetto è stato svolto in python utilizzando diverse librerie. E' stato utilizzato il dataset del MNIST.

1.1 Scripts

Sono stati prodotti quattro script in python:

1. dataset.py: Legge e carica il dataset mnist applicando eventualmente un balancing delle diverse classi.
2. trainer.py: Sfrutta i dati caricati da dataset.py per andare ad addestrare una rete neurale. Alla fine del training salva il modello migliore e le immagini relative alla storia della loss e della accuracy durante la fase di training.
3. models.py: E' una semplice classe contenente alcune tipologie di modello neurale utilizzate da trainer.py.
4. recognizer.py: Carica uno dei modelli salvati, apre una finestra e ad ogni cifra disegnata scrive da terminale la confidenza con cui ogni cifra viene riconosciuta.

1.2 Dependency

Le librerie utilizzate in python sono elencate di seguito:

1. Keras[5]: Una nota API ad alto livello per reti neurali.
2. numpy: Un pacchetto per il calcolo scientifico.
3. optparse: Un pacchetto per la gestione di opzioni CLI.
4. tensorflow[6]: engine backend per keras, capace di sfruttare anche le GPU.
5. opencv[7]: API per la gestione di immagini.

1.3 Parameters

Per quanto riguarda trainer.py:

1. ”-d”, ”–directory”: permette di scegliere una cartella di destinazione per il salvataggio del modello.
2. ”-s”, ”–batch”: permette di scegliere la dimensione del batch con cui effettuare l’addestramento.
3. ”-e”, ”–epochs”: permette di scegliere il numero di epoche per cui si vuole fare l’addestramento.

4. ”-b”, ”-balance”: permette di specificare se si vuole effettuare un bilanciamento delle diversi classi prima di cominciare l’addestramento.
5. ”-c”, ”-committee”: permette di selezionare il numero di reti che si vogliono addestrare, con lo scopo di generare una pool di esperti che prenderà le decisioni per maggioranza.
6. ”-m”, ”-modelname”: permette di scegliere il tipo del modello, questi sono descritti nel file ”models.py”.

Ad ogni addestramento verrano generati i seguenti file:

1. ”test_loss_acc.png”, ”train_loss_acc.png”: sono il plot dell’accuratezza e della loss rispetto al training set e al test set generato epoca per epoca.
2. ”model.png”: l’immagine rappresentante il modello.
3. ”model0.best.hdf5”: il migliore modello addestrato in termini di accuratezza e loss.

Esempi di esecuzioni:

```
python3 trainer.py -s 512 -m "MLPNN_type0" -e 100 -c 20 -b
python3 trainer.py -s 128 -m "CNN2D_type0" -e 20 -c 10 -b
python3 trainer.py -s 512 -m "MLPNN_type0" -e 100 -c 20
python3 trainer.py -s 128 -m "CNN2D_type0" -e 20 -c 10
python3 trainer.py -s 512 -m "MLPNN_type0" -e 100
python3 trainer.py -s 128 -m "CNN2D_type0" -e 20
python3 trainer.py
```

Per quanto riguarda recognizer.py:

1. ”-f”, ”-modeldir”: permette di indicare la directory in cui un modello è stato salvato.
2. ”-v”, ”-verbose”: in caso sia presente, oltre a stampare la confidenze di riconoscimento per ogni cifra, stampa anche la griglia 28×28 utilizzata per il riconoscimento.

Esempi di esecuzioni:

```
python3 recognizer.py -f ./mnist_models/MLPNN_type0-batch512-balanced-committee
python3 recognizer.py -f ./mnist_models/MLPNN_type0-batch512-balanced-committee
python3 recognizer.py
```

2 Background

Alcune generalità preliminari sulle nozioni utilizzate per implementare il progetto.

2.1 Gradient descent

La discesa del gradiente è un metodo di ottimizzazione per trovare un minimo assoluto o locale di una funzione.

L'idea su cui questa tecnica è basata è estremamente semplice.

Andiamo inizialmente a scegliere dei valori iniziali per i parametri (anche casuali) della funzione obiettivo (differenziabile). A questo punto calcoliamo la derivata rispetto ai singoli parametri, troviamo quindi il gradiente. Aggiorniamo i parametri in accordo con il gradiente.

Notiamo che il gradiente rappresenta un vettore in uno spazio multidimensionale e ci dà la direzione lungo il quale l'ipercurva cresce. Se volessimo trovare un minimo anzichè un massimo sarebbe sufficiente seguirne il negativo.

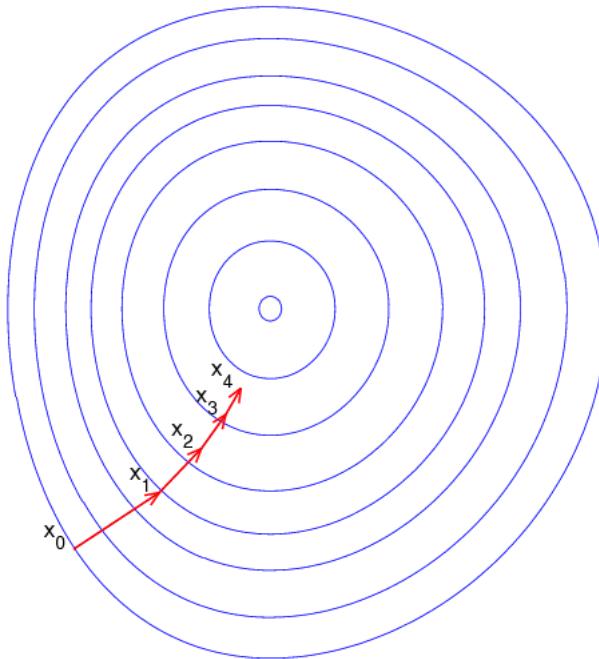


Figure 1: Discesa del gradiente

Per chiarire meglio il metodo facciamo un esempio con la seguente funzione:

$$f(x) = 4x^3 - 9x^2 \quad (1)$$

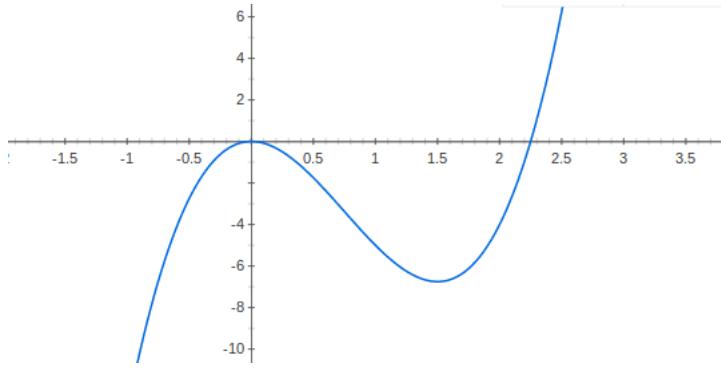


Figure 2: funzione esempio

La derivata invece è: $f(x) = 12x^2 - 18x$.

A questo punto iteriamo il procedimento:

```
inizializza i parametri;
iterazione = 0;
while iterazione < soglia:
    iterazione += 1;
    calcola gradiente;
    aggiorna parametri;
```

Seguiamo l'algoritmo per alcuni step applicando un fattore moltiplicativo di 0.01; partiamo con $x = 2.5$:

$$f(2.5) = 6.25 \quad (2)$$

$$f'(2.5) = 30 \quad (3)$$

Dovremmo spostarci di 30, quindi -0.3.

$$f(2.2) = -0.96 \quad (4)$$

$$f'(2.2) = 18.48 \quad (5)$$

Adesso di -0.1.

$$f(2.1) = -0.2646 \quad (6)$$

$$f'(2.1) = 15.12 \quad (7)$$

Ancora di -0.1.

$$f(2) = -4 \quad (8)$$

$$f'(2) = 12 \quad (9)$$

...

In ultimo chiariamo l'importanza del fattore moltiplicativo, chiamato in seguito "learning rate". Un fattore eccessivamente grande porta a delle modifiche (salti) altrettanto grandi quindi si ha il rischio di avere delle discese inutilmente lente in alcuni casi. Mentre un fattore piccolo necessita di più tempo per trovare un minimo già solo per il motivo che si compiono "salti ridotti".

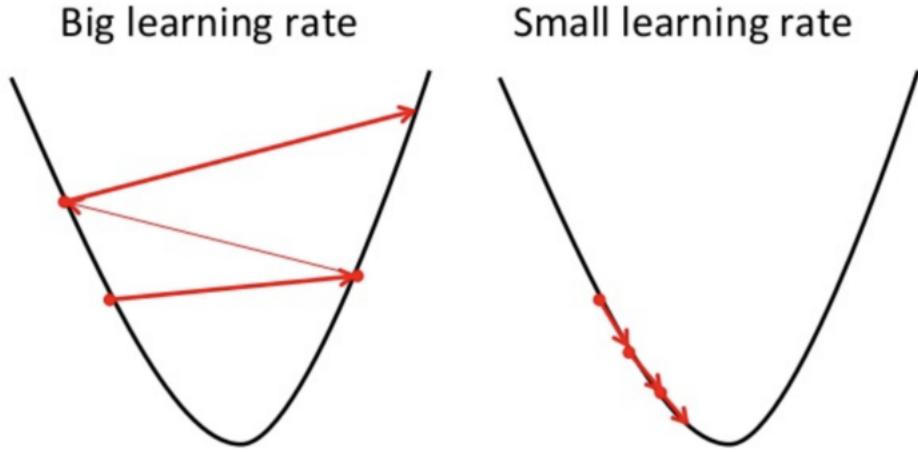


Figure 3: esempio di discesa del gradiente nel caso di un fattore moltiplicativo grande e piccolo.

2.2 MLPNN: Multi Layer Perceptron Neural Network

Un Multi Layer Perceptron è un specifico tipo di rete neurale di tipo feedforward. Un MLP è costituito da almeno 3 layer di neuroni, ognuno dei quali completamente connesso a tutti i neuroni dei livelli precedenti e successivi. Ogni neurone prende come input gli output dei neuroni dei layer precedenti, ne fa la somma pesata e applica una funzione di attivazione.

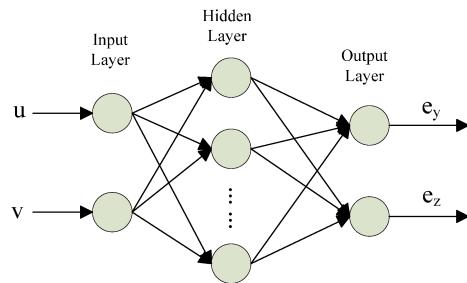


Figure 4: esempio di MLPNN

Il primo layer viene solitamente definito di input, mentre l'ultimo prende il nome di "layer di output", a tutti gli altri ci si riferisce come "hidden layers".

Un MLPNN è solitamente utilizzato in abito di classificazione, o regressione.

L'addestramento di questa rete, in ambito supervisionato, avviene minimizzando la Loss function. L'idea è quella di andare ridurre l'errore che la rete commette rispetto a quello che vorremmo. Una loss function comunemente usata è la seguente:

$$\frac{1}{n} \sum_{i=1}^n \frac{1}{2} (t_i - o_i)^2 \quad (10)$$

Dove:

1. n è il numero di casi nel dataset.
2. t_i è il risultato che vorremmo ottenere, in classificazione, la classe target.
3. o_i è l'output della rete.

Ottenerne una soluzione esatta dalla loss function è possibile solamente per casi banali. In generale siamo costretti a cercare delle soluzioni approssimate, queste vengono ottenute attraverso la minimizzazione del gradiente.

Per completezza diamo la formula per ottenere il gradiente:

$$\frac{\delta E}{\delta o_j} = \delta_j o_j \quad (11)$$

Dove:

$$\delta_j = \frac{\delta E}{\delta o_j} \frac{\delta o_j}{\delta \text{net}_j} = \begin{cases} (o_j - t_j) \sigma'(z_j) & \text{se il neurone è di output} \\ (\sum_{l \in L} w_{jl} \delta_l) \sigma'(z_j) & \text{se il neurone è interno} \end{cases} \quad (12)$$

Dove:

1. L rappresenta l'insieme di neuroni di un determinato layer.
2. w_{ij} è il peso che va dall' i-esimo neurone del layer $L - 1$ al j-esimo neurone del layer L .
3. z_j è l'output non attivato del j-esimo neurone del layer $L - 1$.
4. σ è la funzione di attivazione scelta.

L'aggiornamento di ciascun peso quindi prende la seguente forma:

$$w_{ij} = w_{ij} - \alpha \Delta w_{ij} \quad (13)$$

Dove α rappresenta il learning rate da applicare, solitamente è un numero basso compreso tra 0 e 1. Lo scopo di α è quello di rendere la discesa più lenta, in modo da raggiungere un minimo locale evitando rimbalzi.

In ultimo quindi abbiamo che l'algoritmo avrà il seguente pseudocodice:

```

inizializza i pesi in modo casuale uniforme;
iterazione = 0;
while iterazione < soglia:
    iterazione += 1
    results = mplnn();           // forward pass
    error = target-result;
    calcola il gradiente per ogni peso;
    aggiorna i pesi
return;

```

2.3 Convolution

La convoluzione è una comune operazione matematica che, date due funzioni in input, restituisce una funzione in output:

$$*: \mathbb{R}^{\mathbb{R}} \times \mathbb{R}^{\mathbb{R}} \rightarrow \mathbb{R}^{\mathbb{R}} \quad (14)$$

dove:

$$s(t) = (f * g)(t) = \int f(\tau)g(t - \tau)d\tau \quad (15)$$

mentre nella sua forma discreta:

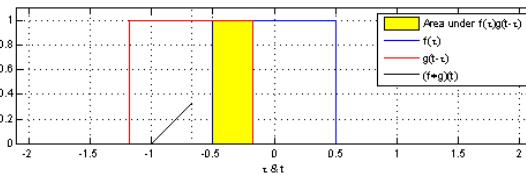
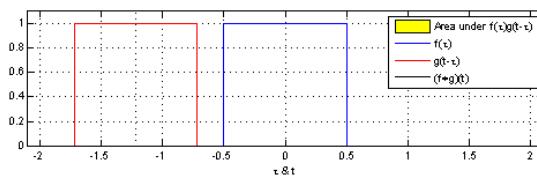
$$s(t) = (f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t - \tau) \quad (16)$$

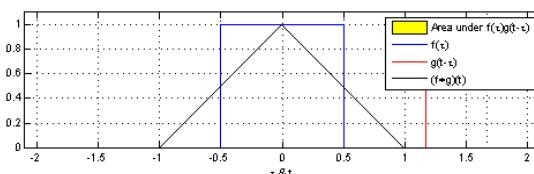
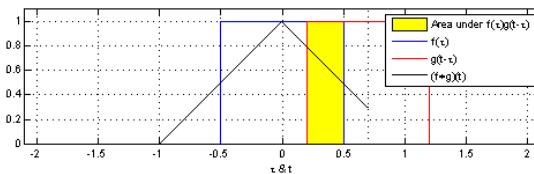
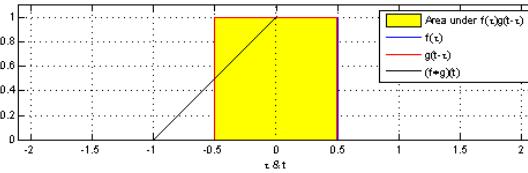
Per dare una descrizione intuitiva e semplicistica della convoluzione immaginiamo il seguente esempio:

Prendiamo due funzioni diverse da zero solamente per intervalli ristretti.

Facciamo scorrere la prima sull'asse delle ordinate, partendo da $-\infty$.

La funzione generata sarà data dalla area che passo per passo si ottiene dall'intersezione delle due figure.





Le reti convoluzionali sfrutteranno questo tipo di operazione nel caso di immagini. Mostriamo il seguente esempio:

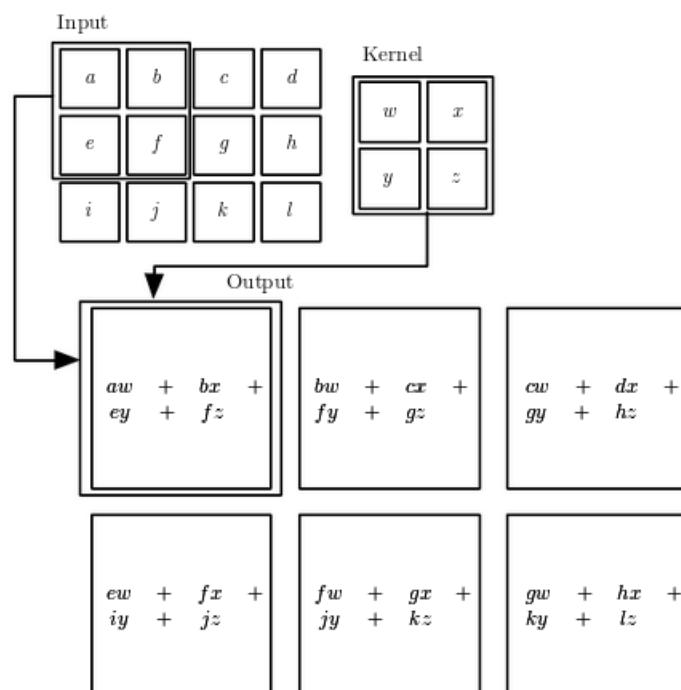


Figure 5: esempio di convoluzione 2D

2.4 CNN: Convolutional Neural Network

Sono delle reti neurali specializzate nel processamento di dati che hanno una rappresentazione a griglia (come immagini).

Utilizzano layer convoluzionali che sfruttano kernel/finestre di dimensione variabile che, scorrendo l'immagine, si attivano quando riconoscono particolari caratteristiche.

L'introduzione dei layer convoluzionali migliora l'apprendimento soprattutto grazie a 3 caratteristiche:

1. Interazioni sparse: l'utilizzo di kernel piccoli permette di avere un numero ridotto di pesi (parametri) da apprendere, rispetto ad un layer completamente connesso. Questo permette di avere un apprendimento essenzialmente più veloce.
2. Condivisione dei parametri: gli stessi parametri di un kernel vengono usati per riconoscere particolari feature all'interno dell'immagine in punti differenti. n.b. il kernel scorre su tutta l'immagine.
3. Equivarianza rispetto a traslazioni: i medesimi kernel sono in grado di riconoscere la medesima feature anche in punti differenti all'interno dell'immagine.

Dopo aver applicato un layer convoluzionale viene utilizzata anche in questo caso una funzione di attivazione.

In ultimo si utilizza un layer di Pooling che ha il compito di semplificare l'immagine. Un esempio comune è quello di max pooling, in cui ogni output del layer precedente viene aggiornato al valore massimo del suo vicinato.

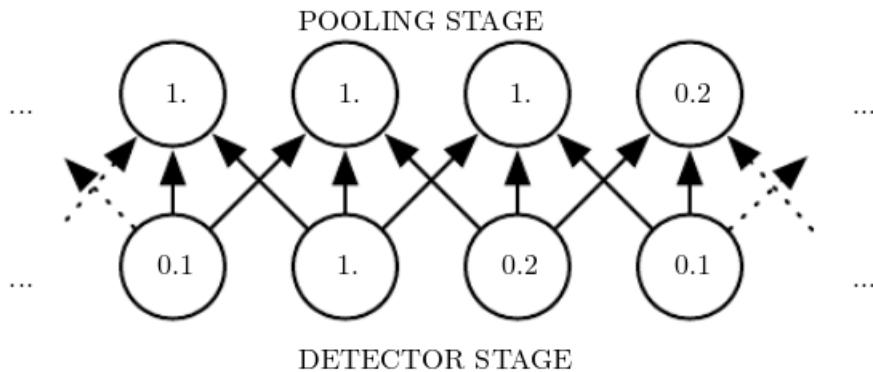


Figure 6: Max Pooling

Soltanamente questi tre layer (convoluzione, attivazione, pooling) possono essere denotati come singolo layer convoluzionale.

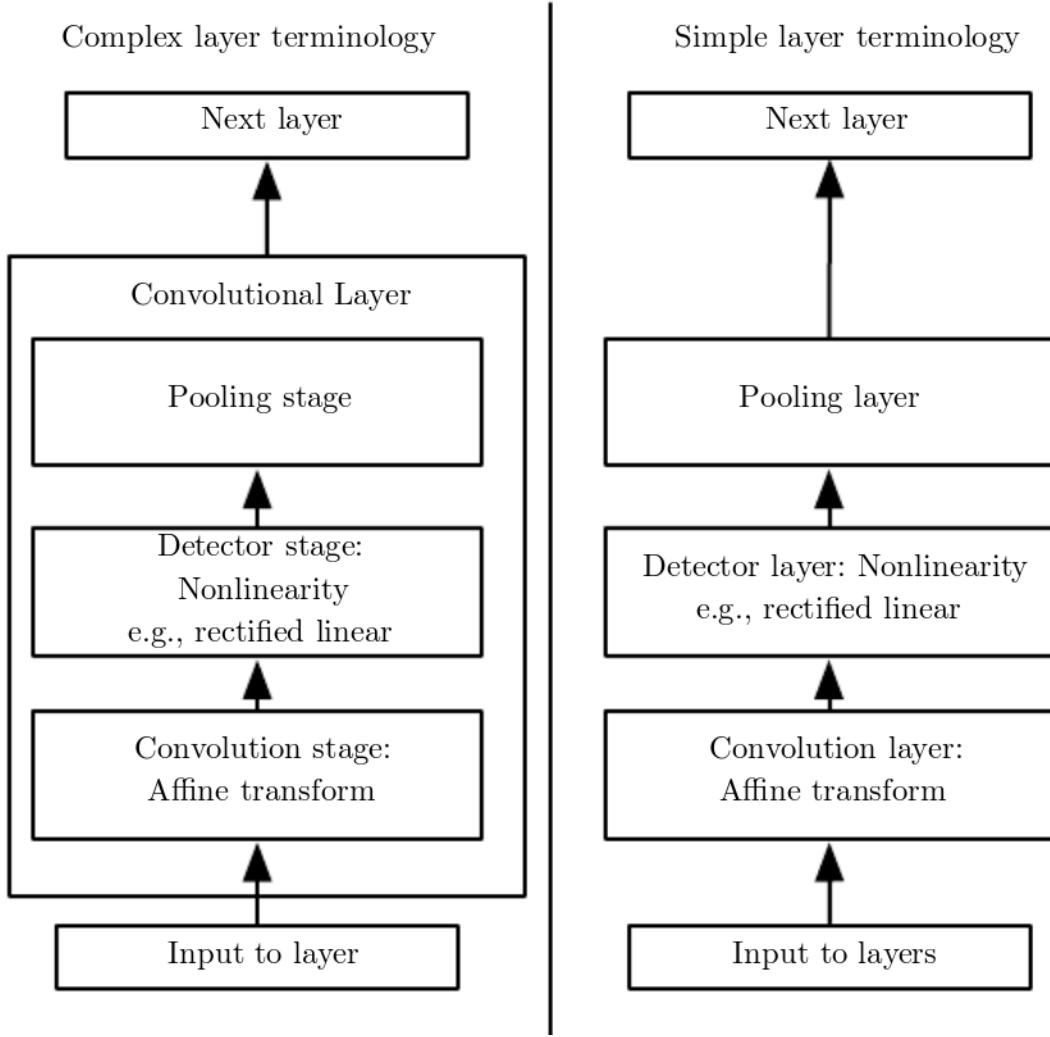


Figure 7: Convolutional Layer

L'idea di fondo è quella di utilizzare i livelli convolutivi per spezzare l'informazione contenuta nell'immagine con lo scopo di ottenere un vettore di feature che sia facile da classificare direttamente con un MLPNN. In ultimo illustriamo l'intero processo della rete neurale su un'immagine.

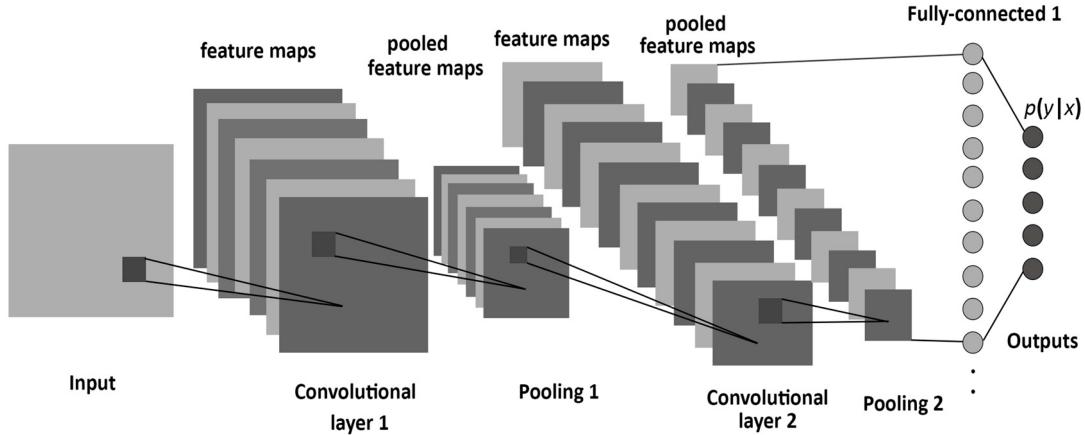


Figure 8: CNN su di una immagine

2.5 Dropout

E' un concetto tanto semplice quanto efficace, applicato esclusivamente in ambito di reti neurali.

Lo scopo è quello di prevenire l'overfitting.

Essenzialmente scegliamo in modo arbitrario una probabilità tale per cui alcuni neuroni verranno tagliati fuori dal backward step e da forward step durante la fase di apprendimento. Questo forza la rete ad apprendere nuovi "percorsi" per attivare un determinato neurone.

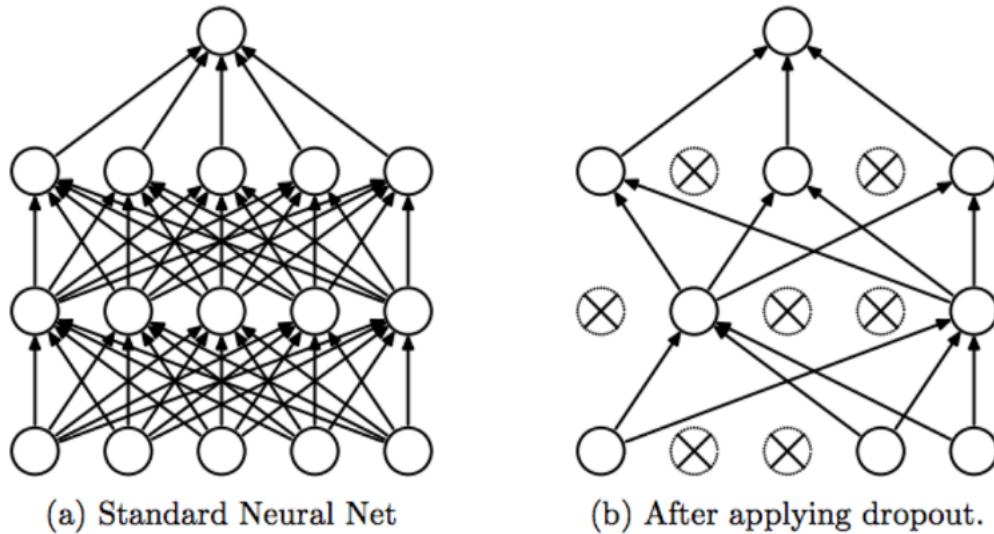


Figure 9: Esempio di Dropout

Valori di Dropout troppo alti compromettono l'apprendimento, mentre valori troppo bassi non beneficiano della mitigazione dell'overfitting. In diverse prove un valore ragionevole, che non vanifica il training, si è rivelato essere 0.2.

2.6 RMSprop

E' un metodo di ottimizzazione per la scelta del learning rate, proposto da Geoff Hinton. L'idea è quella di non seguire cecamente il gradiente ad ogni passo, ma di seguire il gradiente nuovamente ottenuto, pesato per un termine β e sommato alla media di tutti i gradienti ottenuti fino all'ultimo (in modo tale da ricordare il passato), anche questo pesato per un termine $(1 - \beta)$. Formalmente:

$$E[\Delta^2]_t = \beta E[\Delta^2]_{t-1} + (1 - \beta) \Delta_t^2 \quad (17)$$

Soltamente viene indicato come valore di $\beta = 0.9$. A questo punto l'aggiornamento di ciascun peso sarà:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{E[\Delta^2]_t + \epsilon}} \Delta_t \quad (18)$$

Dove α rappresenta il learning rate comune. Mentre $\frac{\alpha}{\sqrt{E[\Delta^2]_t}}$ rappresenta il termine di regolarizzazione.

3 Models

Per costruire un riconoscitore efficace sono stati testati diversi modelli:

1. MLPNN: Una rete a diversi livelli completamente connessa.
2. CNN[4]: Una rete convoluzionale che sfrutta kernel di dimensioni ridotte per la classificazione delle immagini.
3. Committees: Comitati di esperti che prendono decisioni basate sulla maggioranza.

3.1 built MLPNN

L'obiettivo è quello di confrontare diverse architetture a parità di Dropout, e funzioni di attivazione. Vogliamo verificare quanto la profondità della rete, la dimensione di ciascun layer e il numero di parametri totali influenza le capacità di apprendimento.

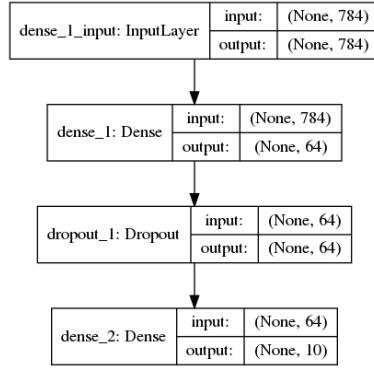


Figure 10: Modello basso con layer ristretti.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.972 e 0.107.

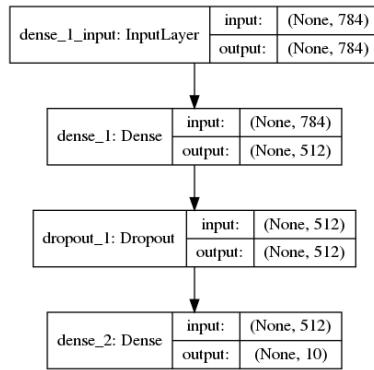


Figure 11: Modello basso con layer ampi.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.982 e 0.067.

Per ciascuno dei modelli sono stati eseguite 100 epoche di apprendimento con tecnica mini-batch di dimensione 128.

Premettendo che tutti i modelli hanno un ottimo comportamento, l'unico che spicca veramente è un quello mostrato in figura 11, con un singolo layer nascosto e una architettura "ampia".

Rispettivamente le reti profonde non ottengono dei risultati altrettanto buoni, questo dimostra che le capacità di generalizzazione di una rete si possono sviluppare anche in ampiezza.

Consideriamo ora l'andamento della loss e della accuracy per tutti e quattro i modelli.

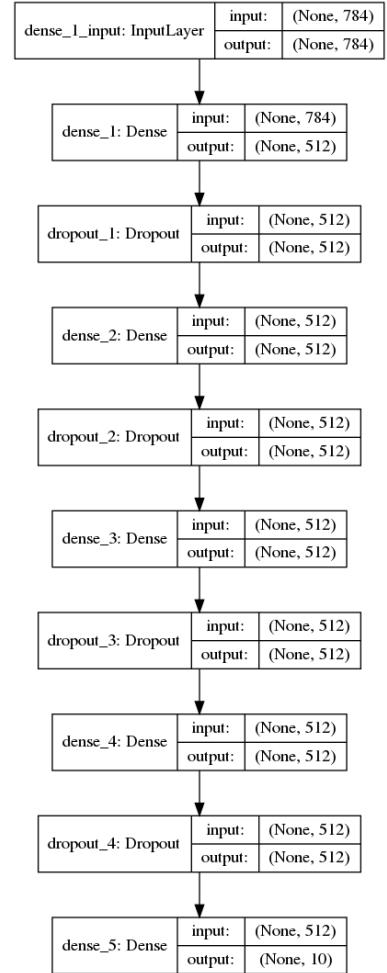


Figure 12: Modello profondo con layer ristretti.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.970 e 0.123.

Figure 13: Modello basso con layer ampi.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.980 e 0.115.

Prima i Modelli non profondi:

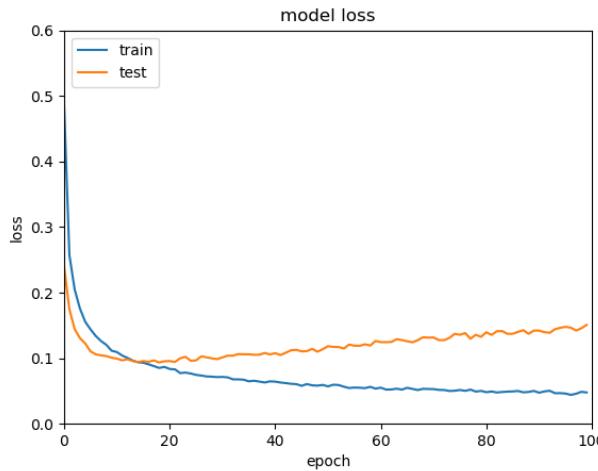


Figure 14: Loss del modello basso con layer ristretti.

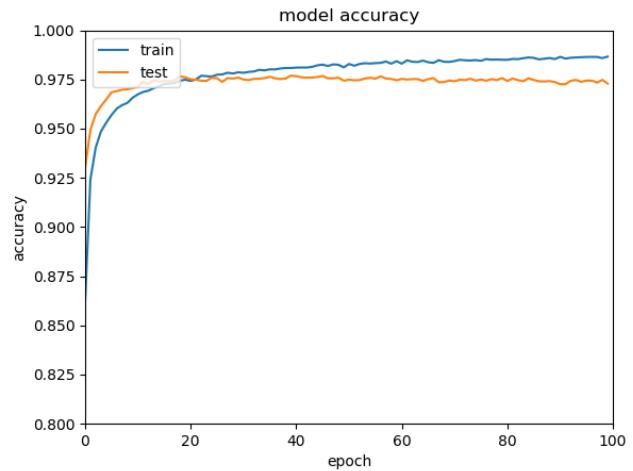


Figure 15: Accuracy del modello basso con layer ristretti.

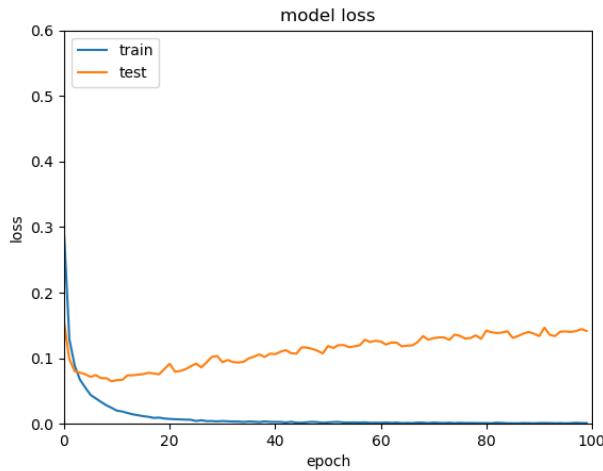


Figure 16: Loss del modello basso con layer ampi.

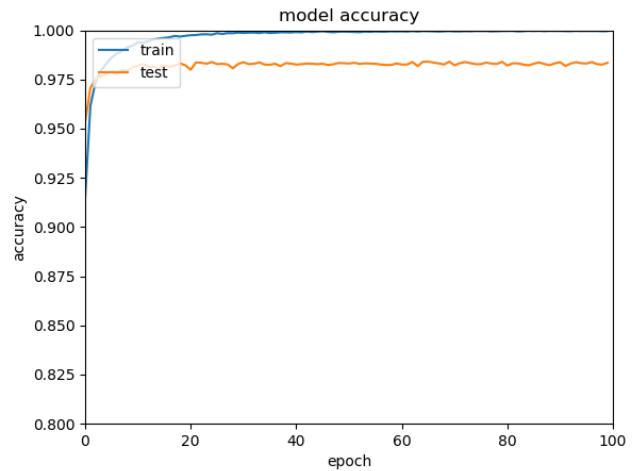


Figure 17: Accuracy del modello basso con layer ampi.

Come si può notare facilmente dalle figure, e sarà confermato anche in seguito, modelli ampi ottengono risultati migliori in tempi minori.

Infatti il modello ottimo viene ottenuto già entro le prime 20 epoche di addestramento, per quanto riguarda architetture ampie.

Vediamo ora l'andamento per le architetture profonde.

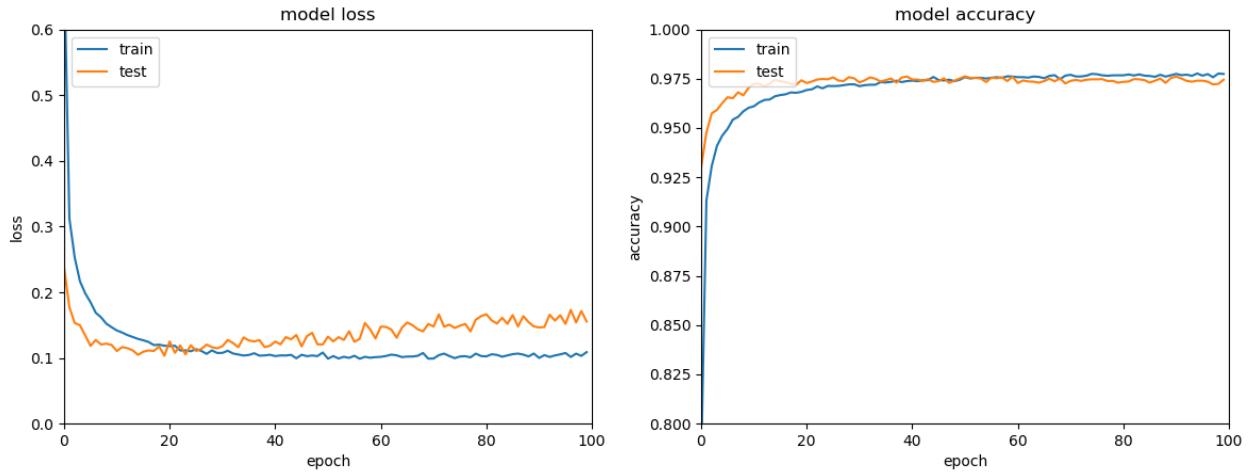


Figure 18: Loss del modello profondo con layer ristretti.

Figure 19: Accuracy del modello profondo con layer ristretti.

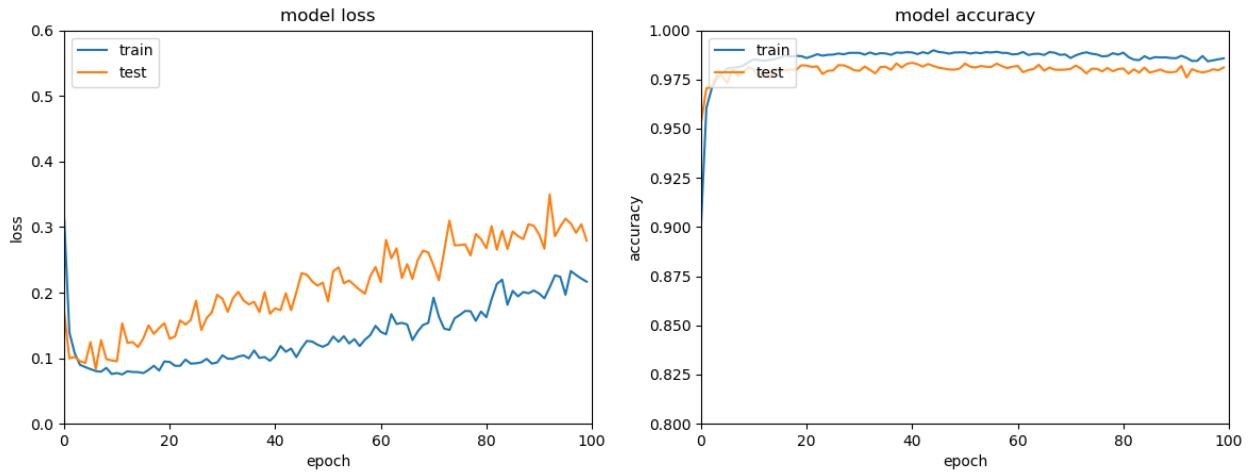


Figure 20: Loss del modello profondo con layer ampi.

Figure 21: Accuracy del modello profondo con layer ampi.

Riconfermiamo quindi ciò che è stato già detto: layer ampi favoriscono la velocità di apprendimento in termini di epoche (chiaramente ciascuna epoca ha una durata più lunga, dato il numero maggiore di parametri da aggiornare).

3.2 built CNN

Questa volta lo scopo è quello di valutare come varia l'apprendimento in funzione della dimensione dei kernel e in funzione della dimensionalità dei layer convoluzionali.

Vediamo quindi l'architettura di ciascun modello:

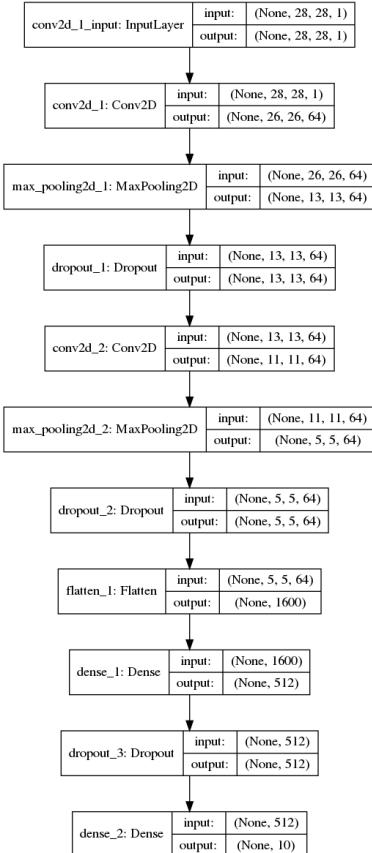


Figure 22: Modello con kernel e layer ridotti.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.993 e 0.025.

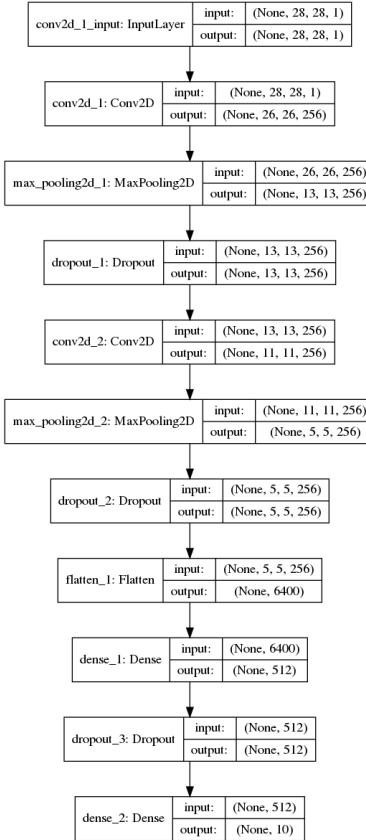


Figure 23: Modello con kernel ampi e layer ridotti.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.991 e 0.037.

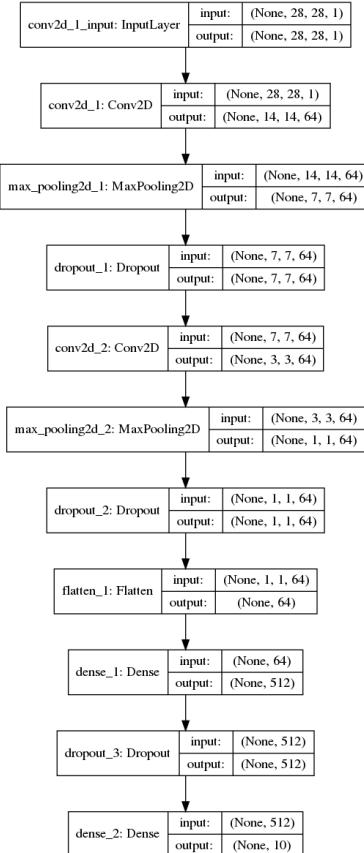


Figure 24: Modello con kernel ristretti e layer ampi.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.987 e 0.051.

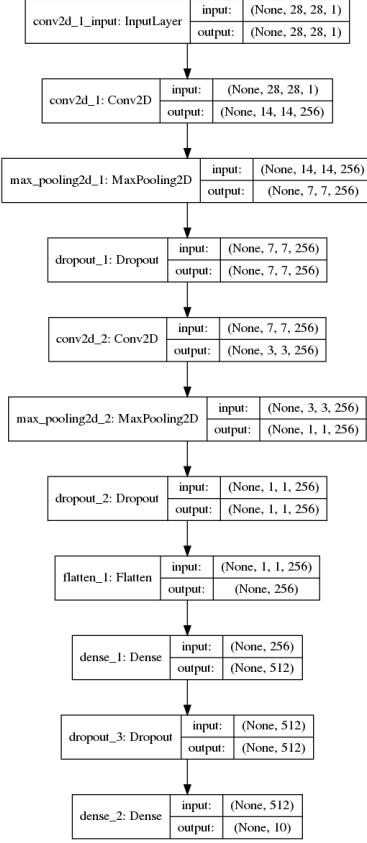


Figure 25: Modello con kernel e layer ampi.

Il miglior modello ottiene come accuracy e loss sul set di validazione i rispettivi risultati: 0.989 e 0.050.

Il modello che ottiene i risultati migliori dunque è quello che viene mostrato in figura 22. In questo caso la rete utilizza kernel 3×3 con layer di dimensione 64.

Tuttavia ancora una volta i risultati delle architetture alternative sono estremamente positivi tanto da risultare praticamente equivalenti.

Apparentemente in questo contesto la scelta di un kernel grande o piccolo non va ad influenzare l'apprendimento. Lo stesso vale per la dimensionalità dei layer.

Ciò che comunque risulta evidente fin da subito è che le reti convoluzionali hanno un impatto decisamente migliore rispetto ai percettroni multistrato, ottenendo degli score estremamente più positivi.

Andiamo ancora una volta a valutare l'andamento dell'apprendimento nei termini di loss e accuracy delle diversi reti.

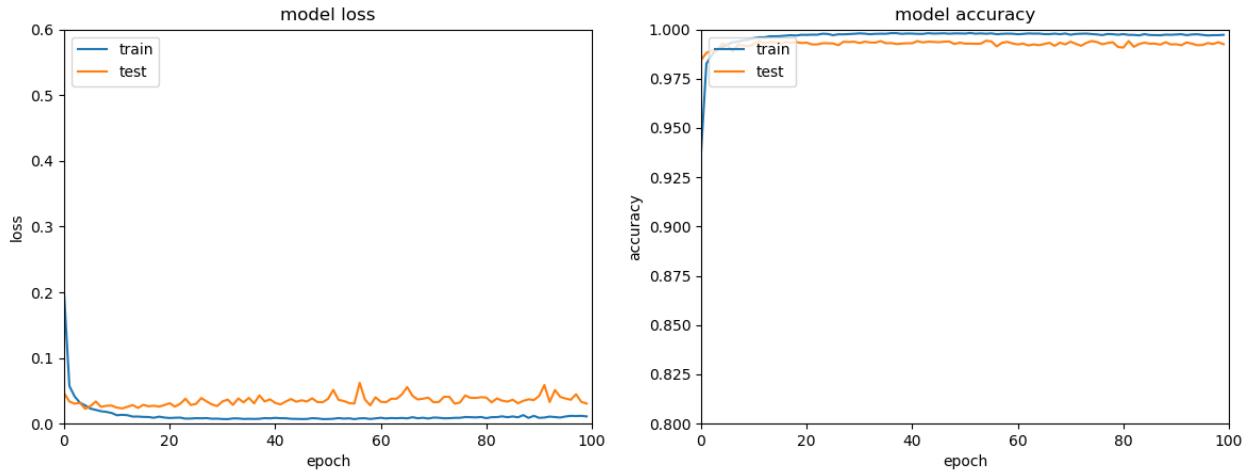


Figure 26: Loss del modello con kernel e layer ridotti.

Figure 27: Accuracy del modello con kernel e layer ridotti.

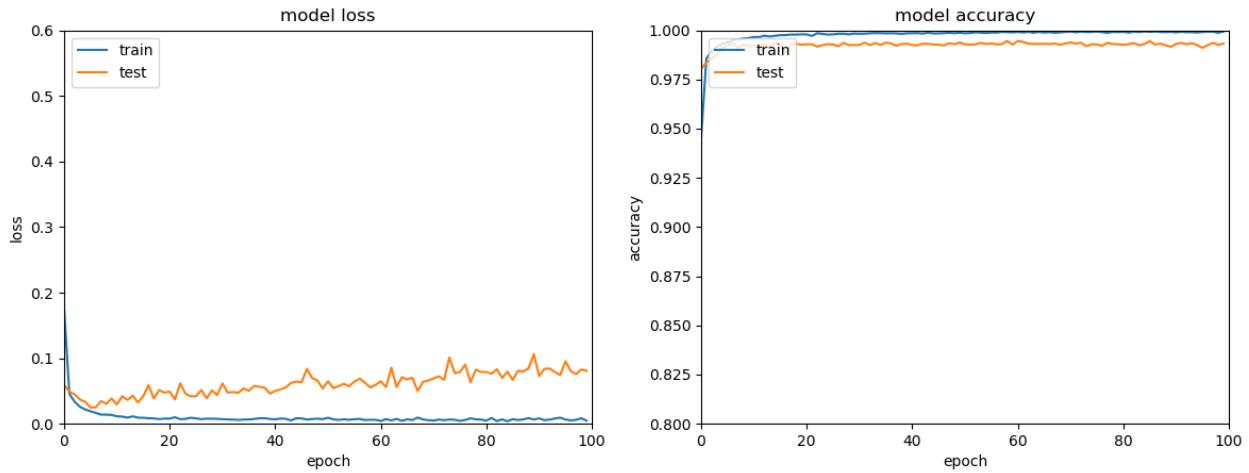


Figure 28: Loss del modello con kernel ampi e layer ridotti.

Figure 29: Accuracy del modello con kernel ampi e layer ridotti.

L'unica differenza tra figure 26-27 e 28-29 è la dimensionalità dei kernel, che, nel secondo caso sono più ampi. Ancora una volta possiamo riconfermare che la variazione in dimensione non porta a cambiamenti significativi.

Mostriamo anche le ultime 4 figure:

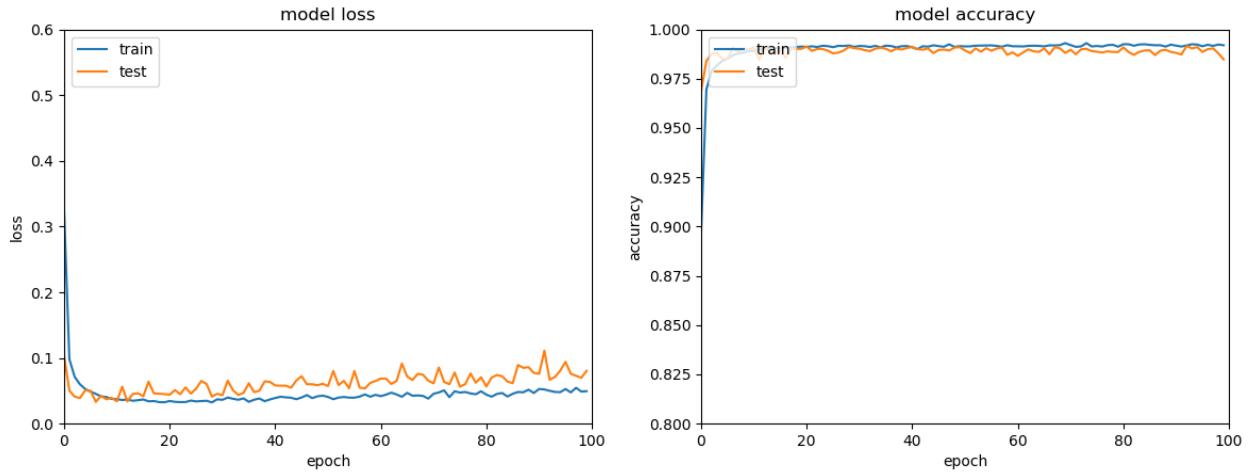


Figure 30: Loss del modello con kernel ampi e layer ristretti.

Figure 31: Accuracy del modello con kernel ampi e layer ristretti.

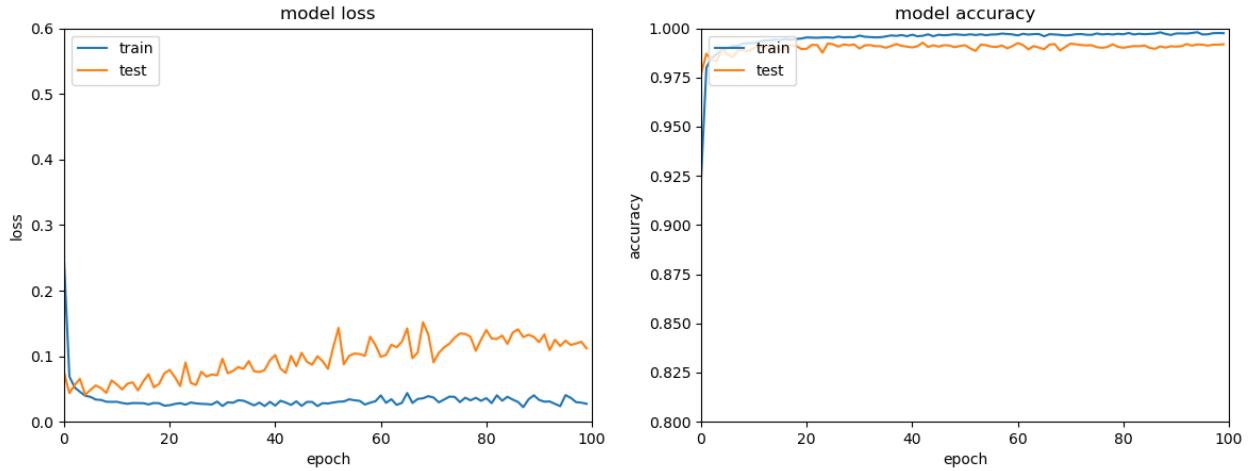


Figure 32: Loss del Modello con kernel e layer ampi.

Figure 33: Accuracy del Modello con kernel e layer ampi.

Questa volta la differenza tra le figure 30-31 e 32-33 sta nell'ampiezza dei layer, tuttavia, anche paragonando con le precedenti figure 26-27-28-29, non si riescono a vedere delle differenze significative.

4 Conclusions

Diamo ora le ultime conclusioni, anche alla luce del funzionamento del riconoscitore.

4.1 Examples

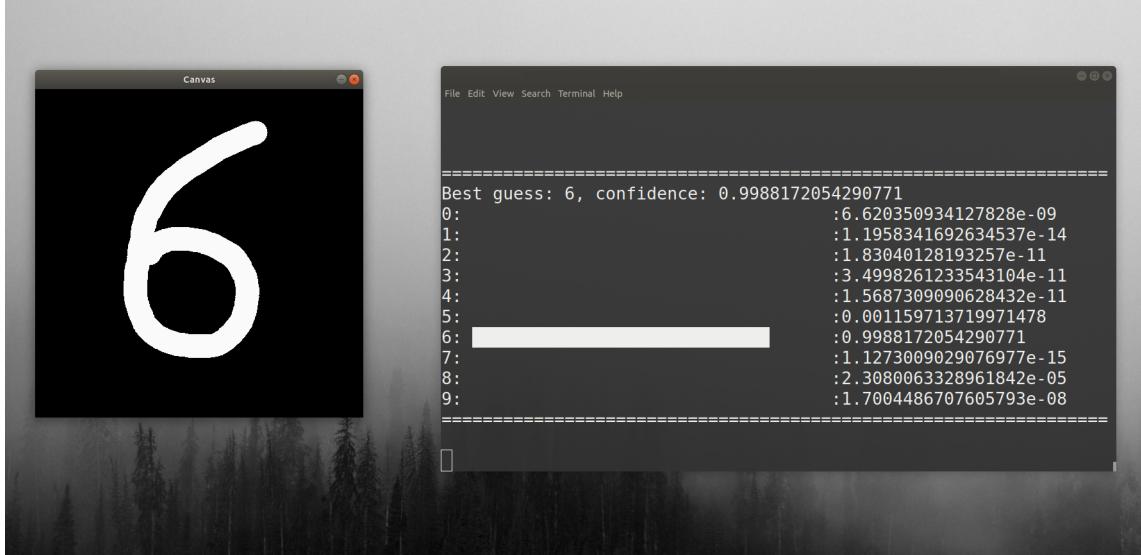


Figure 34: 6 riconosciuto come tale.

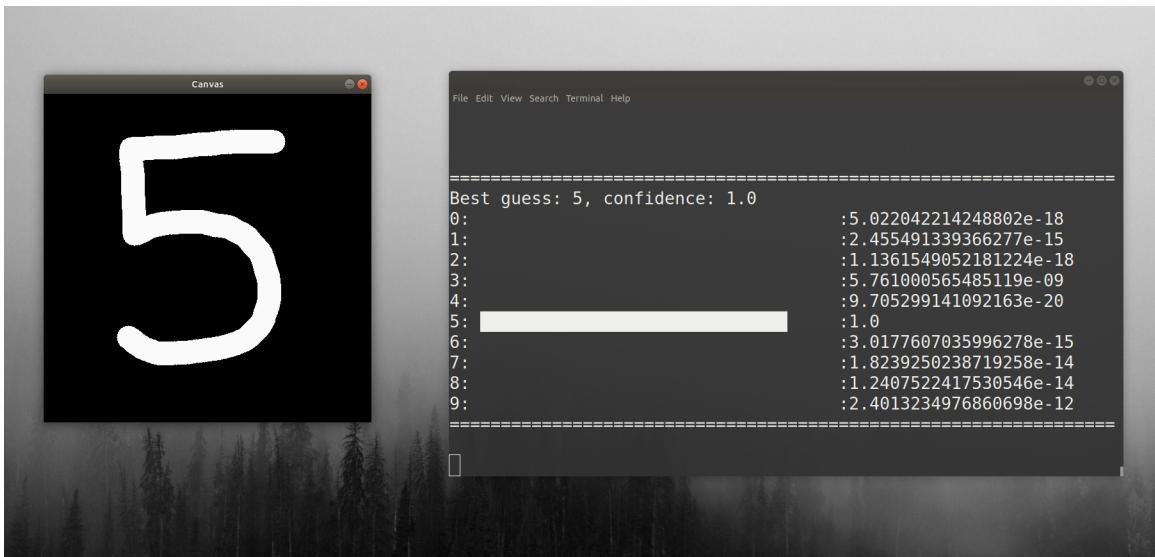


Figure 35: 5 riconosciuto come tale.

Le precedenti cifre sono state disegnate in modo chiaro, e infatti il riconoscitore funziona in modo corretto.

Mostriamo ora un caso limite:

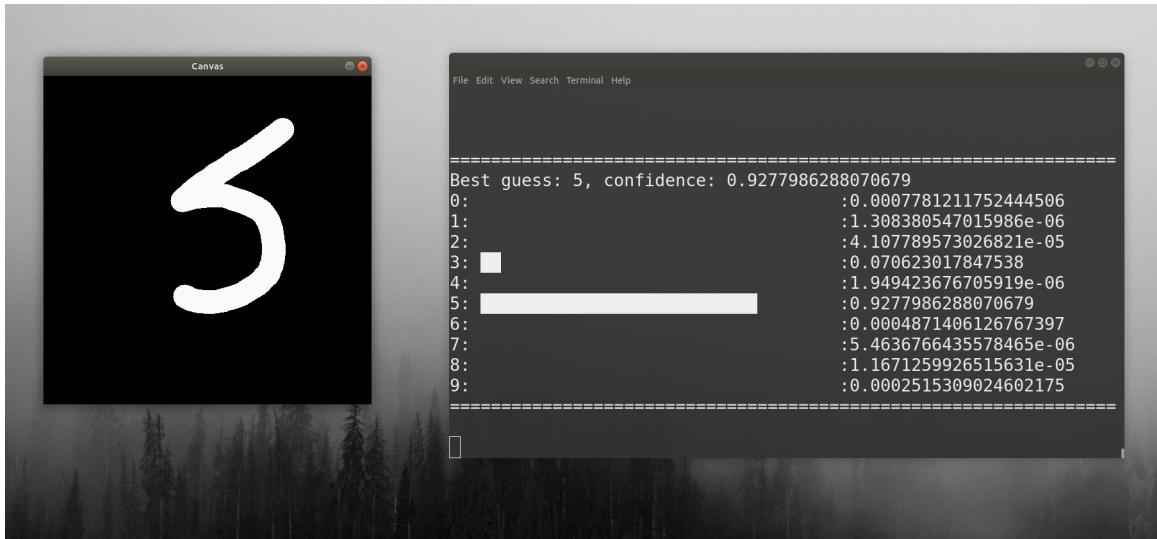


Figure 36: 5 limite riconosciuto come tale.

Apparente è abbastanza comune all'interno del dataset mnist la precedente versione del 5, questo porta a qualche confusione soprattutto nei confronti del 6.

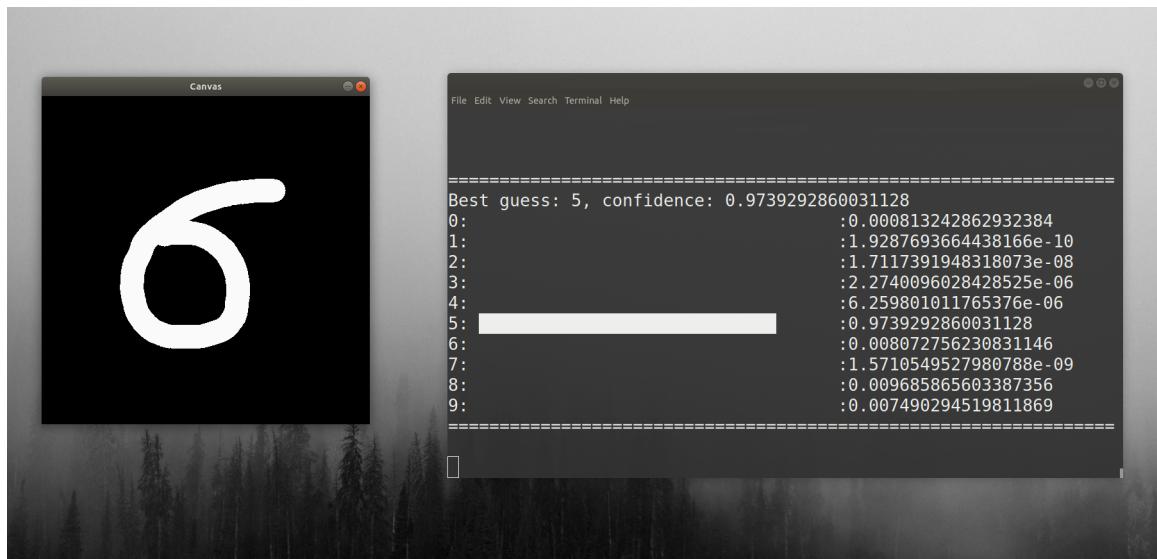


Figure 37: 6 riconosciuto come 5.

Lo stesso avviene anche per il 7 e l'1 che sono intrisecamente simili.

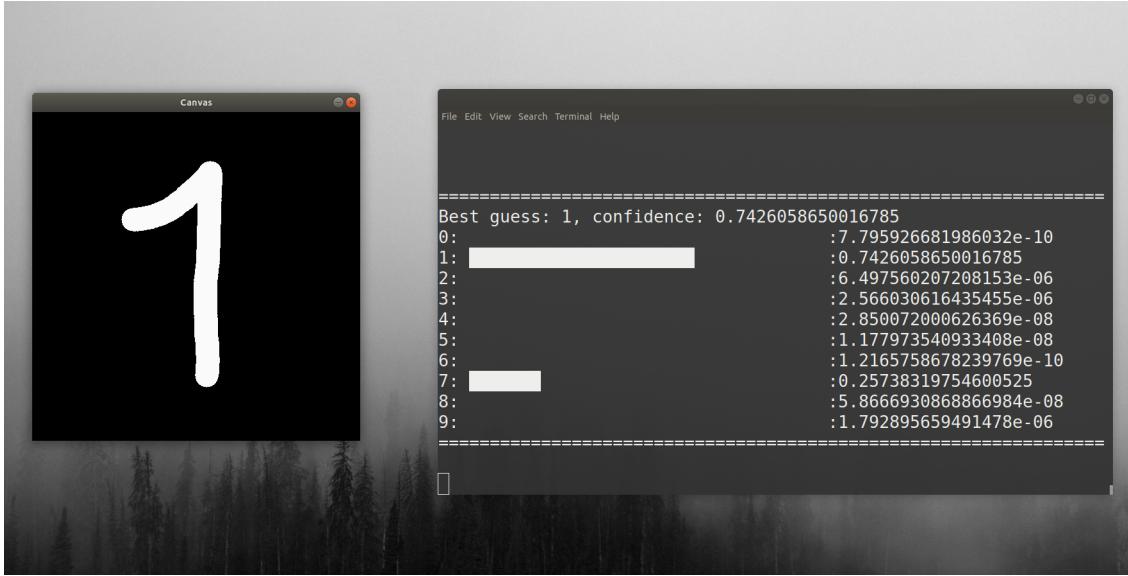


Figure 38: 1 riconosciuto come tale.

Ma in questo caso, anche un operatore umano potrebbe avere qualche dubbio.

4.2 Improvements

Il riconoscitore funziona complessivamente bene, anche se in certi casi commette errori.

Sicuramente è possibile avere un riconoscitore più preciso andando ad ampliare il dataset con casi più diversificati e più numerosi, o eventualmente, facendo un oversampling delle immagini più critiche.

Un'altra fonte di problemi sta nel fatto che le immagine del dataset sono state scritte a mano su carta e digitalizzate in seguito. Mentre quelle disegnate sulla finestra sono già digitali e vengono semplicemente ridotte in dimensione fino a 28×28 , in modo da essere compatibili con le immagini nel mnist.

Nonostante la somiglianza comune, le nuove immagini non sono prodotte nello stesso modo e questo può essere una causa contributiva alla misclassificazione di alcuni esempi.

4.3 References

References

- [1] Goodfellow-et-al-2016, title=Deep Learning, author=Ian Goodfellow and Yoshua Ben-gio and Aaron Courville, publisher=MIT Press, note=<http://www.deeplearningbook.org>, year=2016
- [2] Claude Lemaréchal,
https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf, 2010.

- [3] Geoffrey E. Hinton,
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
- [4] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [5] <https://github.com/keras-team/keras>
- [6] <https://www.tensorflow.org/>
- [7] <https://opencv.org/>