UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Informatica

Neural Networks and Deep Learning

Implementazione di una rete convoluzionale tramite l'utilizzo di una rete feed-forward classica

Progetto 8

Anno Accademico 2020/2021

Studenti
Turco Mario N97/343
Longobardi Francesco N97/344
Rauso Giuseppe. N97/357

Contents

Chapter 1

Informazioni preliminari

1.1 Traccia del progetto

Di seguito è riportata la traccia del progetto:
(Difficoltà alta) Nell'articolo "An Equivalence of Fully Connected Layer and Convolutional Layer, Wei Ma, Jun Lu, arXiv, 2017" è proposta l'implementazione di una rete feed-forward classica che si comporta come una rete convoluzionale. Studiare e descrivere gli aspetti teorici. Implementare tale rete almeno nei suoi aspetti principali e testarla sul dataset mnist, riportandone le accuratezze ottenute al variare nel numero di filtri considerati ed, eventualmente, del numero di strati convolutivi.

Chapter 2

Introduzione

In questo progetto verrà presentata una implementazione di una rete convoluzionale, convertendo i l'operazione di convoluzione in un prodotto matrice-matrice. L'implementazione è basata sull'articolo "An Equivalence of Fully Connected Layer and Convolutional Layer, Wei Ma, Jun Lu, arXiv, 2017"

2.1 Cos'è una rete convoluzionale

Le CNN si basano sull'operazione matematica detta convoluzione, così definita nel caso discreto bi-dimensionale:

$$(A \odot B)[j_1, j_2] = \sum_{k1} \sum_{k2} A(k1, k2) B(j_1 - k_1, j_2 - k_2)$$

Tuttavia, per semplicità, nelle reti convoluzionali viene spesso utilizzata una formulazione più semplice, basata sul prodotto point-wise fra matrici, definita come segue:

$$G \cdot H = \sum_{i=0}^{p_w} \sum_{j=0}^{p_h} G_{ij} \cdot H_{ij}$$

dove G ed H rappresentano rispettivamente l'input *patch* dell'immagine e il fitro, mentre p_w e p_h sono le dimensioni di una *patch* processata dal kernel.

L'operazione di convoluzione può essere anche espressa tramite prodotto tra matrici. Questo approccio risulta essere computazionalmente più efficiente a fronte di un utilizzo più elevato di memoria.

2.1.1 Funzionamento generale di una CNN

Supponiamo di avere in input un'immagine espressa sotto forma di matrice $X \in R^{N \times M}$, possiamo dividere tale matrice in zone separate e distinte (che possiamo chiamare "patch") ed associare ognuna di queste zone ad un neurone distinto.

Supponiamo che ognuna di queste zone abbia un dimensione S (s $\times$ s), idealmente

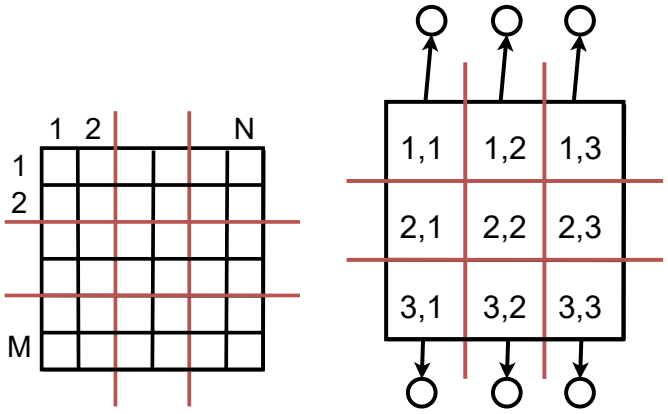


Figure 2.1: Divisione dell'immagine in regioni

vorremmo che ogni neurone sia in grado di identificare una certa caratteristica nella

sua "patch". Fissata una matrice dei pesi W (ed anche il bias):

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1s} \\ w_{21} & w_{22} & \dots & w_{2s} \\ & \vdots & & \\ w_{s1} & w_{s2} & \dots & w_{ss} \end{pmatrix}$$

essa è detta Filtro(o kernel) e resta invariata per tutti i neuroni e possiamo trovare l'output di ogni neurone applicando la funzione di attivazione al prodotto tra la matrice dei pesi e la patch di quel dato nerone.
Ad esempio per la zona (1,1) avremo:

$$\begin{aligned} z_{11} = g(w_{11}x_{11} + w_{12}x_{12} + \dots w_{1S}x_{1s} + \\ w_{21}x_{21} + w_{22}x_{22} + \dots w_{1s}x_{2s} + \\ \dots \\ w_{s1}x_{s1} + w_{s2}x_{s2} + \dots w_{ss}x_{ss} + b_{ss}) \end{aligned}$$

Il filtro rimane dunque invariato e viene applicato alle varie patch dell'immagine, come se si spostasse sull'immagine di un certo passo s detto stride.
L'insieme delle applicazioni di un kernel su tutte le regioni dell'immagine è detta feature map; è possibile avere più feature map che estraggano più caratteristiche dall'immagine.

Chapter 3

Adattamento dei dati per simulare l'operazione di convoluzione

In questo capitolo sono esposte le varie fasi del progetto che portano all'implementazione della rete convoluzionale tramite l'utilizzo di un prodotto fra matrici.

3.1 Preparazione dell'input

3.1.1 Trasformazione dell'input

In una operazione di convoluzione esiste una relazione fra lo shape dell'input e lo shape dell'output. In particolare:

$$H_{out} = \frac{H_{in} + 2 \cdot P - k_h}{s} + 1$$

$$W_{out} = \frac{W_{in} + 2 \cdot P - k_w}{s} + 1$$

$$C_{out} = f$$

Dove:

- H_{out} e W_{out} rappresentano le dimensioni dell'output della convoluzione. Essi

individuano le dimensioni delle feature map.

- H_{in} e W_{in} rappresentano le dimensioni in input alla convoluzione.
- C_{in}, C_{out} e f individuano rispettivamente il numero di canali in ingresso, in uscita e il numero di kernel.

Assumendo di avere dei filtri di dimensione $R^{k_h \times k_w \times C_{in}}$, possiamo dividere l'immagine di input in diversi vettori riga tutti di dimensione $R^{1 \times (k_h k_w C_{in})}$ ed assieme, tuttia l'immagine sarà una matrice di dimensione $R^{(H_{out} W_{out}) \times (k_h k_w C_{in})}$

Quindi ogni input di una rete CNN può essere visto come input per una rete FC come una matrice $\mathbf{M}$ di dimensioni $R^{(bH_{out} W_{out}) \times (k_h k_w C_{in})}$ dove b è il batch size o il numero di samples in ingresso.

L'input di una rete convoluzionale può essere visto come $H_{out} \cdot W_{out}$ input per la rete full connected e, l'intera matrice di input $\mathbf{M}$, sarà di dimensioni $R^{(bH_{out} W_{out}) \times (k_h k_w C_{in})}$ dove b è il batch size o il numero di samples in ingresso (quindi $b = 1$ se passiamo in input una sola immmagine, ad esempio).

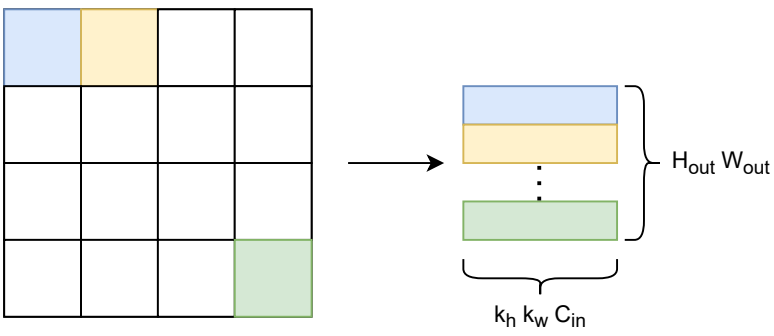


Figure 3.1: Flattening di un'immagine di input

È importante notare come, nel caso di più immagini a cui applicare la funzione *flatten_ input* ($b > 1$), il risultato sarà la matrice $\mathbf{M}$, composta da b (batch size) "sotto matrici", ognuna di dimensioni $R^{(k_h \cdot k_w \cdot C_{in}, H_{out} \cdot W_{out})}$.

Tale funzione è stata implementata in Matlab ed è la seguente:

Listing 3.1: flattern_input

```
function [M] = flatten_input(input, k_h, k_w, s, P)
% Flatterns the input based on kernel's size
% input - input matrix
% k_h   - kernel's heigth
% k_w   - kernel's width
% s     - stride
% P     - padding
    d = size(input);
    if length(d) == 2       %Se l'input e' una immagine
        d(3) = 1;           %imposto batchsize a 1
    end

    if length(d) == 4    % imposto il C-in
        C_in = d(4);
    elseif length(d) == 3
        C_in = 1;
    end

    H_out = (d(1) + 2*P - k_h)/s + 1;
    W_out = (d(2) + 2*P - k_w)/s + 1;
    M = zeros(d(3)*H_out*W_out, k_h*k_w*C_in);
    for j=0 : d(3)*H_out*W_out-1
        l = floor(j / (H_out*W_out));
        p = mod(j, H_out*W_out);
        m = floor(p/H_out);
        t = mod(p, W_out);
        isw = t*s;
        ish = m*s;
        if length(d) == 4
            arr = input(ish+1:ish+k_h, isw+1:isw+k_w, l+1, :);
        else
            arr = input(ish+1:ish+k_h, isw+1:isw+k_w, l+1);
```

```
33          end
34          M(j+1, :) = reshape(arr.',1,[]);
35      end
36  end
```

La funzione, a partire dalle immagini in input, le dimensioni dei kernel, lo stride ed il padding esegue il flattening delle immagini, estrae k_h, k_w e C_in che corrispondono ad altezza, larghezza e profondità del kernel.

Alle righe 9, 13 e 15 viene impostato manualmente il batchsize nel caso di una sola immmagine e viene impostato anche il C_in.

A partire dalle informazioni sul kernel vengono calcolate W_{out} ed H_{out} e viene quindi inizializzata la matrice M come matrici di zeri. Nel ciclo for successivo viene effettuato il flattening delle immagini.

Il for scorre tutte le righe di M e le riempie appropriatamente con le immagini corrispondenti ad ogni riga. In particolare viene calcolato l[1] che corrisponde all'indice di immagine (ovvero otteniamo b, riferiata alla b-esima immagine del batch), mentre p indica la "patch" su cui stiamo iterando appartenente all'immagine l-esima. Mentre m, moltiplicato allo stride (s) identifica l'indice di riga di inizio della patch su cui si sta iterando.

Invece, t moltiplicato per lo stride identifica l'indice di colonna di inizio della patch su cui si sta iterando.

Definiti i precedenti indici, è possibile recuperare la porzione di immagine d'interesse dell'iterazione corrente: difatti l'istruzione 32 utilizza ish ed isw per estrarre dall'immagine l-esima, tramite slicing, la porzione d'interesse di dimensioni $k_h \times k_w$ e le inserisce nella matrice M sotto forma di vettore riga.

[1] Si è deciso di mantenere i nomi originali delle variabili dell'algoritmo del paper di riferimento.

3.1.2 Trasformazione del kernel

Lo stesso ragionamento può essere fatto per ogni kernel che può essere "schiacciato" in un vettore colonna di dimensioni $R^{(k_h \cdot k_w \cdot C_{in}) \times 1}$, e mettendo assieme più filtri, otterremo una matrice $\mathbf{K}$ di dimensioni $R^{(k_h k_w C_{in}) \times f}$, dove f è il numero di filtri da utilizzare nel livello convoluzionale.

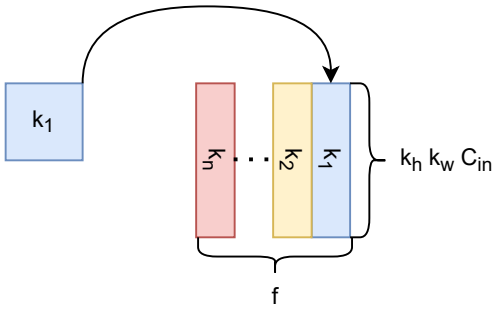


Figure 3.2: Flattening del kernel k_1 in un vettore colonna

Tale funzione è stata implementata in Matlab ed è la seguente:

Listing 3.2: flattern_kernel

```
function [L] = flatten_kernel(kernel)
% Flatterns the kernel
    d=size(kernel);
    k_h=d(1); %kernel height
    k_w=d(2); %kernel width

    if length(d)==2 %se in input abbiamo dato un solo kernel, ad esempio
         d(3)=1;     % di dimensioni (4,4,1)
                     %impostiamo manualmente la terza dimensione a 1
    end
    if length(d)==4 %se abbiamo piu' di un canale(4 dimensioni)
        C_in=d(4);  %impostiamo il numero di canali = C_in
    elseif length(d)==3 %se abbiamo un solo canale (4,4,1,1)
        C_in=1;         %impostiamo C_in ad 1
    end

```

```
17      L = zeros(k_h*k_w*C_in, d(3));
18      for i=1 :d(3) %rendo ogni filtro un vettore colonna
19          if length(d)==4
20              arr=kernel(:,:,i,:);
21          elseif length(d)==3
22              arr=kernel(:,:,i);
23          end
24          L(:,i)=reshape(arr.',1,[])';
25      end
26  end
```

All'interno del ciclo for invece effettuiamo il vero e proprio flattening della matrice estraendo ogni volta l'i-esimo kernel e facendo un reshape in un vettore colonna.

3.1.3 Output

In output avremo quindi la matrice $M \cdot K$ di dimensioni $R^{(b \cdot H_{out} \cdot W_{out}) \times f}$ che rappresenta l'operazione di convoluzione, ora convertita e adattata per essere usata nel progeto.

Tale output può essere a sua volta riconvertito, in caso di necessità, da un output di un prodotto matriciale all'output di una rete convoluzionale, di dimensioni: $R^{(b, H_{out} \cdot W_{out}, f)}$.

Chapter 4

Struttura della rete implementata

La rete è implementata come una struttura dati che mantiene in memoria le informazioni relative alla rete ad ai suoi livelli, ed in particolare per ogni livello è presente un flag, chiamato type, che identifica la natura di tale livello: di input (0), full connected(1) o convoluzionale(2).
La struttura dei livelli è mantenuta in un cell array chiamato **layers**, di lunghezza pari al numero di livelli della rete. Nel nostro caso la struttura descrive un livello di input, un livello convoluzionale ed un livello full connected.
Per ogni tipo di livello vengono mantenute diverse informazioni, ad esempio, per i layer convoluzionali vengono mantenute i metadati che specificano il numero di filtri, la dimensione dei filtri, lo stride ed il padding.

Funzioni di attivazione Per quanto riguarda le funzioni di attivazione, sono state utilizzate la funzione ReLU per il livello convoluzionale e la funzione identità per il livello full connected.

Funzione di errore Come funzione di errore è stata utilizzata la funzione Cross Entropy multi class con post-processing tramite Soft Max.

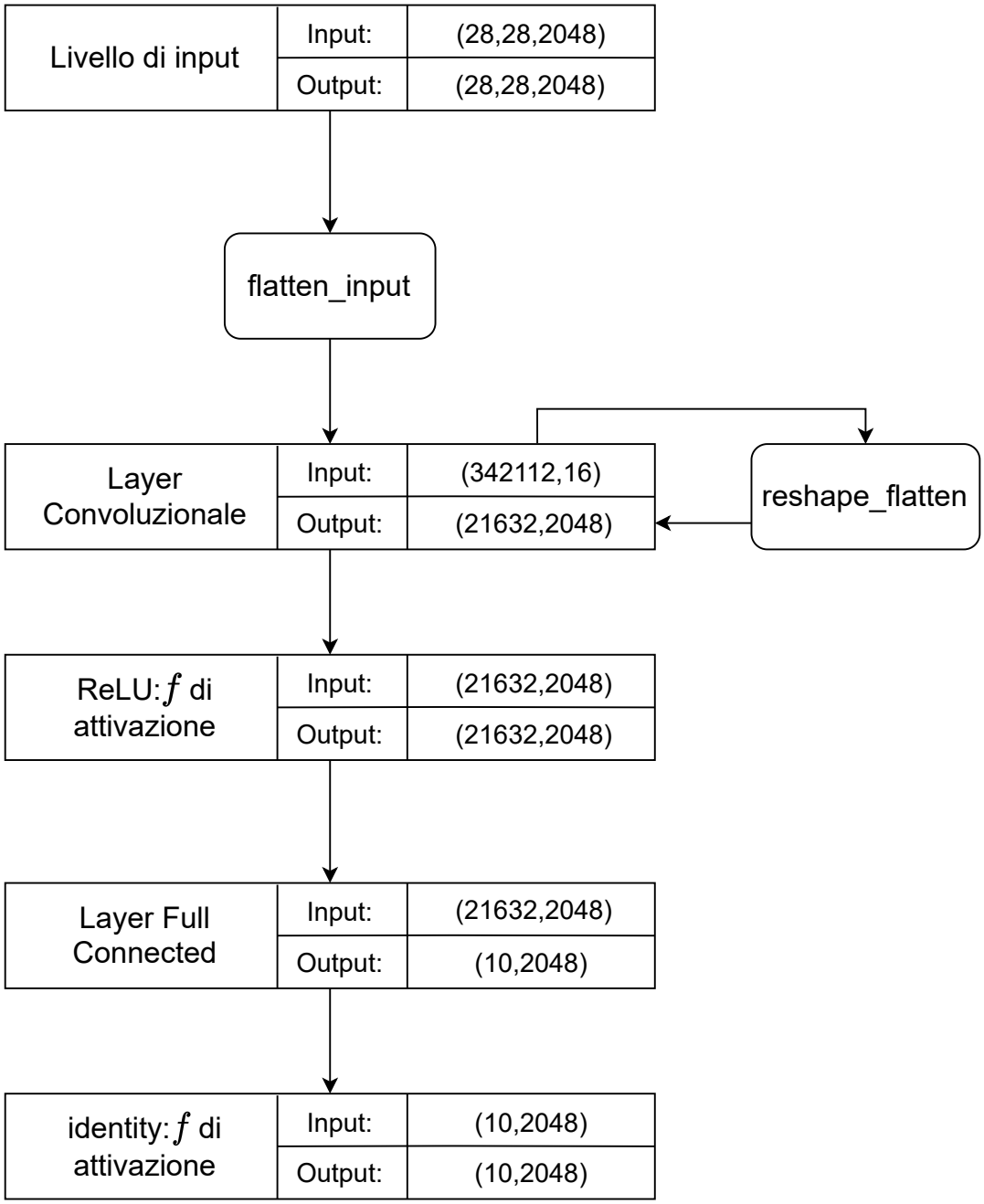


Figure 4.1: Livelli della rete con input e output shape

4.1 Fase feed-forward

La fase feed-forward della rete è implementata seguendo le regole studiate durante il corso per i livelli densi e i livelli convoluzionali. La particolarità dell'implementazione di questo progetto è l'utilizzo della funzione *reshape_flatten*

Listing 4.1: reshape_flatten.m

```
function [x_new] = reshape_flatten(x, n_filters,
   feature_map_dim)
    n_imgs = size(x, 1) / feature_map_dim;
    x_new = zeros(feature_map_dim*n_filters, n_imgs);
    c = 1;
    for i = 1 : feature_map_dim : size(x, 1)-1
        tmp = x(i:i+feature_map_dim-1, 1:n_filters);
        x_new(:, c) = tmp(:);
        c = c + 1;
    end
end
```

L'esigenza di implementazione di questa funzione viene dal fatto che l'operazione di reshape implementata già in Matlab non era compatibile con il tipo di flattening necessario per applicare la fase feed-forward come nel paper studiato. Di seguito viene mostrato un confronto tra il reshape di MATLAB (non utile per noi in questo caso) e la funzione sopra descritta.

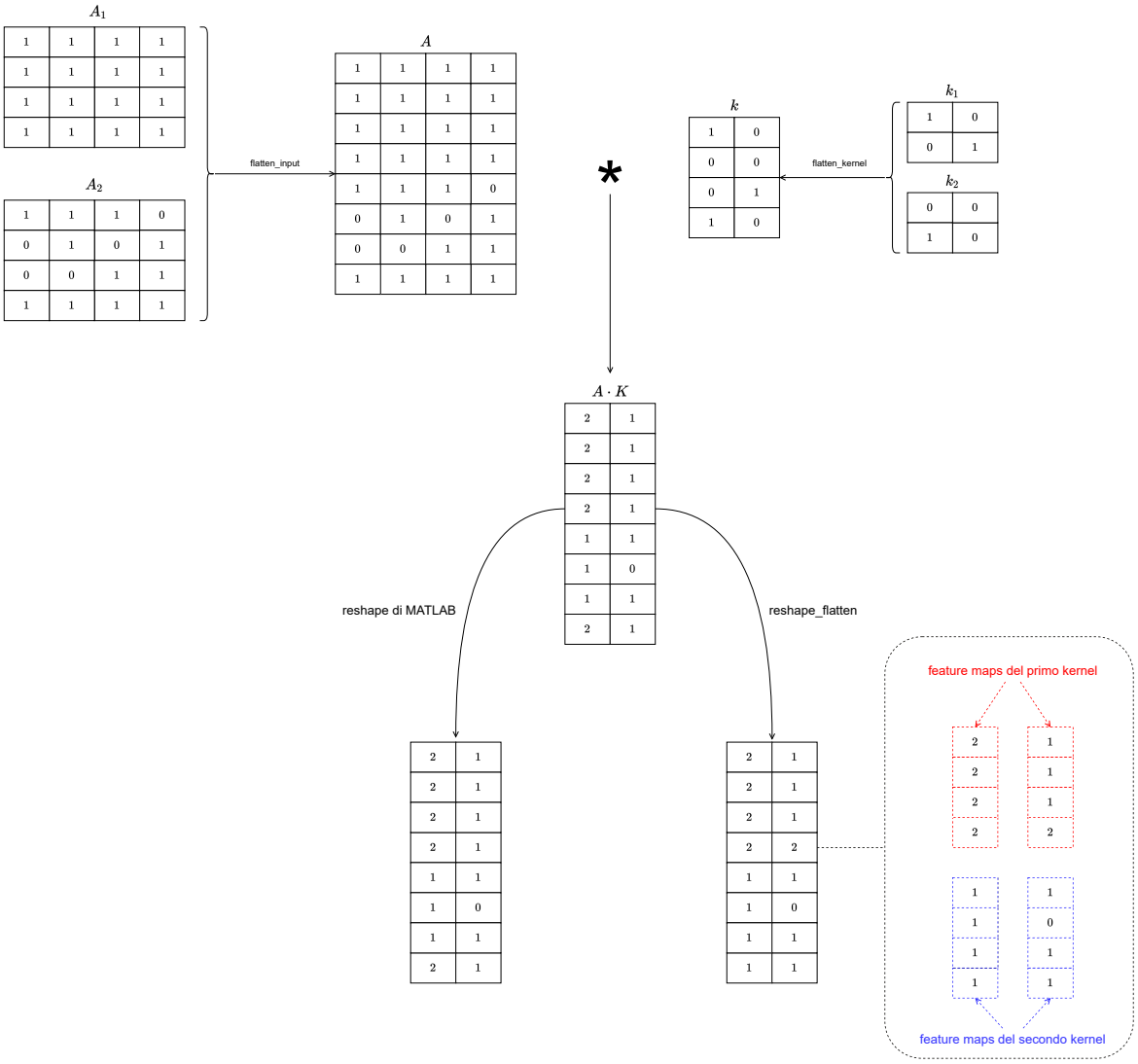


Figure 4.2: È possibile notare come i risultati siano diversi, giustificando quindi, l'implementazione della funzione sopra descritta

4.2 Backpropagation

Rispetto alla back propagation in una normale rete FC bisogna usare delle accortenze specifiche per il livello che simula una convoluzione,infatti è necessario simulare lo spostamento dei filtri sulle immagini, ricordando che tutti i neuroni di una stessa feature map fanno riferimento agli stessi filtri (ogni filtro scorre su tutta l'immagine), quindi utilizzano gli stessi pesi ed è necessario sommare tutti i delta dei neuroni della stessa feature map per calcolare $\frac{\partial E^n}{\partial w_{ij}}$. Ricordiamo che, per un punto n qualsiasi del training set, $\frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E^{(n)}}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{ij}}$. Definendo $\frac{\partial E^{(n)}}{\partial a_i} = \delta_i$, e ricordando che $\frac{\partial a_i}{\partial w_{ij}} = z_j$, allora $\frac{\partial E^{(n)}}{\partial w_{ij}} = \delta_i \cdot z_j$. In particolare,

$$\delta_i = \begin{cases} f'(a_i) \sum_h w_{hi}\delta_h & \text{Se il nodo è interno} \\ f'(a_i)\frac{\partial E^{(n)}}{\partial y_i} & \text{Se il nodo è di output} \end{cases}$$

con h indice dei nodi che ricevono connessione dal nodo i e f' derivata della funzione di attivazione del livello i-esimo. In un livello convoluzionale i kernel scorrono le immagini in ingresso generando delle *feature map* in uscita. Ogni pixel delle feature map è generato su una regione di immagine applicando la somma dei prodotti element-wise tra i pesi del kernel e i pixel della regione di immagine. Quindi i pesi del kernel sono coinvolti nel calcolo dell'intera feature map, e quindi passa su diverse regioni. Per questo motivo la derivata della funzione di errore rispetto ad un peso w_i di un kernel (indicato con un solo indice per semplicità) è pari alla somma delle derivate della funzione di errore rispetto alle regioni dell'immagine:

$$\frac{\partial E^{(n)}}{\partial w_i} = \sum_k \delta_k \cdot z_j^k$$

con k indice delle regioni dell'immagine e z_j^k k-esima regione dell'output del livello precedente (x_j^k se il livello convoluzionale è il primo livello della rete, con x immagine in input alla rete). Nel nostro caso la rete ha solo due livelli; il primo

passo della backpropagation è riportato di seguito:

$E \equiv$ *crossentropy*,

$f_1 \equiv ReLU$,

$f_2 \equiv$ *identity*

In questo contesto utilizziamo $\cdot$ per il prodotto element-wise e $\times$ per il prodotto tra matrici.

1. $\delta_{out} = f_2'(a_2) \cdot \frac{\partial E}{\partial y}$
2. $\nabla_{w^{(1,2)}} E = \delta_{out} \times z_2^T$ con $\nabla_{w^{(1,2)}} E = [\frac{\partial E}{\partial w_1^{(1,2)}}, \ldots, \frac{\partial E}{\partial w_{p_1}^{(1,2)}}]$ avendo p_1 pesi tra il livello convoluzionale e quello full-connected di output. Nell'implementazione questo gradiente è in realtà una matrice, tenendo conto della struttura matriciale dei pesi che abbiamo utilizzato dall'inizio.

```
1    W_deriv = {};
2    deltas = {};
3    bias_deriv = {};
4
5    [a_, z_] = forward_step_convFC(net, x);
6    z_ = [{flatten_input(x, net.layers{2}.dim(1), net.layers
        {2}.dim(2), net.layers{2}.stride, net.layers{2}.padding)
        } z_];
7
8    delta_out = net.deriv_func{end}(a_{end});
9    delta_out = delta_out .* derivFunErr(z_{end}, t);
10
11   deltas{net.n_layers} = delta_out;
12
13   W_deriv{net.n_layers-1} = delta_out * z_{end-1}';
```

Abbiamo usato le strutture *W_ deriv* e *bias_ deriv* per salvare le derivate parziali

della funzione di errore rispetto ai pesi e bias. La struttura *deltas* è utilizzata per salvare i δ utilizzati nel calcolo della backpropagation.
L'implementazione dell'operazione di convoluzione come prodotto tra matrici richiede un "flattening" dell'input (come visto nel paragrafo 3.1), e quindi le zone dell'immagine sono le righe della matrice prodotta dall'operazione descritta nel paragrafo 3.1. Di conseguenza il secondo passo della backpropagation è il seguente:

1. $\delta_1^{(k)} = \sum_{i=1}^{\text{n\_img}} \delta_i$ con n_img numero di immagini, k indice di zona dell'immagine utilizzata per fare una convoluzione e $\delta_i = w^{(k)T} \times \delta_{\text{out}}^{(i)} \cdot f_1'(a_1^{(k)})$. La sommatoria interna somma i delta relativi alla zona k di tutte le immagini. δ_i infatti rappresenta il delta relativo all'immagine i calcolato con i pesi $w^{(k)}$ relativi alla zona k dell'immagine, il $\delta_{\text{out}}^{(i)}$ dell'immagine i-esima e $f_1'(a_1^{(k)})$ con $a_1^{(k)}$ la parte di a_1 (output del livello prima dell'applicazione della funzione di attivazione) relativa alla zona k dell'immagine

2. $\nabla_{w^k} E = \sum_k \delta_1^{(k)} \times x^{(k)T}$ con $\nabla_{w^k} E$ gradiente di E rispetto ai kernel

```
w_d = size(net.weights{end-w});
dim_delta = size(deltas{i+1});
dim_flatten = size(z_{end-z});
deltas{i} = zeros(net.layers{i}.n_neurons, dim_flatten(1));
c = 1;
batch_size = dim_flatten(1) / (net.layers{i}.H_out*net.layers{i
    }.W_out);
delta = zeros(net.layers{i}.n_neurons);
for k=1 : net.layers{i}.n_neurons : w_d(2)
    for img=1 : batch_size
        delta = net.weights{end-w}(:, k:k+net.layers{i}.
            n_neurons-1)' * deltas{i+1}(:, img);
        delta = delta .* net.deriv_func{end-a}(a_{end-a}(k:k+
            net.layers{i}.n_neurons-1, img));
```

```
12
13          deltas{i}(:, c) = deltas{i}(:, c) + delta;
14      end
15      c = c + 1;
16  end
17
18  W_deriv{i-1} = zeros(size(net.weights{i-1}));
19  dim_flatten = size(z_{end-z});
20
21  for k = 1 : dim_flatten(1)
22      W_deriv{i-1} = W_deriv{i-1} + (deltas{i}(:, k) * z_{end-z}(
            k, :))';
23  end
```

Di seguito la funzione completa per il calcolo della backpropagation

Listing 4.2: Backpropagation

```
1   function [W_deriv, bias_deriv] = backpropagation_convFC(net, x,
        t, derivFunErr)
2
3       W_deriv = {};
4       deltas = {};
5       bias_deriv = {};
6       %% FASE FORWARD-PROPAGATION
7       [a_, z_] = forward_step_convFC(net, x);
8       z_ = [{flatten_input(x, net.layers{2}.dim(1), net.layers
            {2}.dim(2), net.layers{2}.stride, net.layers{2}.padding)
            } z_];
9
10      delta_out = net.deriv_func{end}(a_{end});
```

```
11      delta_out = delta_out .* derivFunErr(z_{end}, t);
12      deltas{net.n_layers} = delta_out;
13      W_deriv{net.n_layers-1} = delta_out * z_{end-1}';
14      w = 0; %Questo indice serve per iterare sulle matrici W
           della rete
15      a = 1; %Questo indice serve per gli input a dei neuroni nei
            livelli
16      z = 2;
17      bias_deriv{net.n_layers-1} = sum(delta_out, 2);
18      for i=net.n_layers-1 : -1: 2
19          if net.layers{i}.type == 2
20              w_d = size(net.weights{end-w});
21              dim_delta = size(deltas{i+1});
22              dim_flatten = size(z_{end-z});
23              deltas{i} = zeros(net.layers{i}.n_neurons,
                   dim_flatten(1));
24              c = 1;
25              batch_size = dim_flatten(1) / (net.layers{i}.H_out*
                   net.layers{i}.W_out);
26              delta = zeros(net.layers{i}.n_neurons);
27              for k=1 : net.layers{i}.n_neurons : w_d(2)
28                  for img=1 : batch_size
29                      delta = net.weights{end-w}(:, k:k+net.
                           layers{i}.n_neurons-1)' * deltas{i+1}(:,
                            img);
30                      delta = delta .* net.deriv_func{end-a}(a_{
                           end-a}(k:k+net.layers{i}.n_neurons-1,
                           img));
31
```

```
32                      deltas{i}(:, c) = deltas{i}(:, c) + delta;
33                  end
34                      c = c + 1;
35              end
36
37              W_deriv{i-1} = zeros(size(net.weights{i-1}));
38              dim_flatten = size(z_{end-z});
39
40              for k = 1 : dim_flatten(1)
41                  W_deriv{i-1} = W_deriv{i-1} + (deltas{i}(:, k)
                       * z_{end-z}(k, :))';
42              end
43
44          else     %Livello denso
45              deltas{i} = net.weights{end-w}' * deltas{i+1};
46              deltas{i} = deltas{i} .* net.deriv_func{end-a}(a_{
                   end-a});
47              W_deriv{i-1} = deltas{i} * z_{end-z}';
48              if net.layers{i}.use_bias == 1
49                  bias_deriv{i-1} = deltas{i};
50              end
51          end
52          w = w + 1;
53          a = a + 1;
54          z = z + 1;
55      end
56  end
```

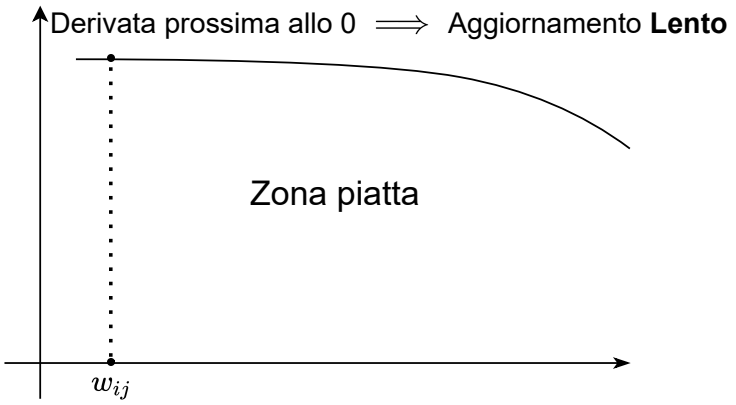


Figure 4.3: Esempio di zona piatta della derivata

4.3 Aggiornamento pesi

Il metodo di ottimizzazione utilizzato per il calcolo delle derivate è la discesa del gradiente con momento, il quale permette di attraversare più velocemente le zone piatte della derivata, ovvero quando essa risulta essere prossima allo zero. Il momento è rappresentato proprio dall'introduzione dell'iper-parametro μ moltiplicato per $\Delta w_{ij}^{(t-1)}$

$$\Delta w_{ij}^{(t)} = \eta \frac{\partial E}{\partial w_{ij}} + \underbrace{\mu \cdot \Delta w_{ij}^{(t-1)}}_{\text{Momento}} \quad t.c.\ 0 < \mu < 1$$

La regola di aggiornamento dei pesi diventa quindi

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} - \Delta w_{ij}^{(t)}$$

4.4 Mini-batch

È stato utilizzato un approccio basato sul paradigma **mini-batch learning**, usando un *batch-size* pari a 32 immagini.

Tale approccio è una via di mezzo tra l'approccio del batch learning, che prevede il calcolo delle derivate su tutto il dataset, e l'online learning, in cui vengono calcolate le singole derivate considerando un elemento del dataset alla volta senza calcolare una "derivata totale".

Nel mini-batch learning, il dataset è diviso in tanti piccoli batch di dimensioni pari a *batch-size* e, per ognuno di essi, viene calcolato il gradiente e aggiornati i parametri.
È stato inoltre implementato un sistema di permutazione casuale dei batch, al fine di ridurre il rischio di **overfitting**.

Chapter 5

Training

Il training è stato eseguito su di un training set composto 2048 immagini MNIST, mentre il validation error è stato calcolato su un insieme con le stesse dimensioni. Le prestazioni della rete sono state valutate su di un test set di 1000 elementi estratto da MNIST. I tre insiemi risultano essere, intuitivamente, disgiunti. Infine, a rete è stata addestrata per 100 epoche.
I parametri utilizzati sono i seguenti:

- **Learning Rate**=0.0001
- **Momento**=0.8
- **Stride**=2
- **Padding**=0
- **Dimensione kernel**=4×4

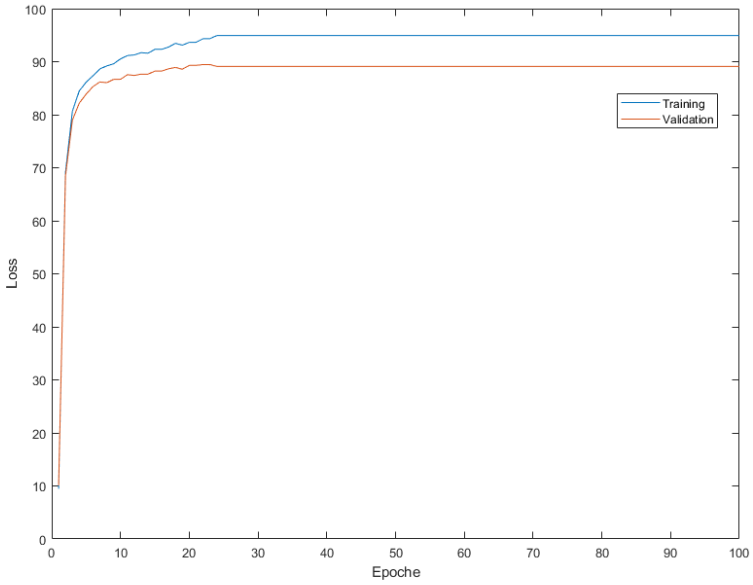
I valori migliori per il learning rate ed il momento sono stati trovati empiricamente tramite un grid search.
È possibile far partire un training e scegliere tutti gli iper-parametri utilizzando lo script **StartTraining.m** presente nella consegna.

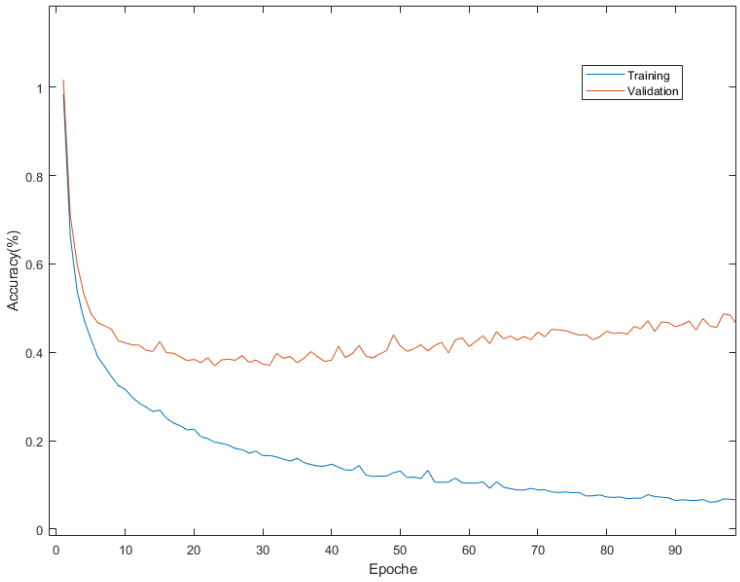
5.1 Prestazioni riscontrate

5.1.1 256 kernel

Con un numero di filtri pari a 256 è stata riscontrata un'accuratezza pari al 94.9% sul **training set**, dell'89.1% sul **validation set** e dell'85.9% sul **test set**.

Grafici





5.1.2 128 kernel

Con un numero di filtri pari a 128 è stata riscontrata un'accuratezza pari al 94.6% sul **training set**, dell'89.9% sul **validation set** e dell'86.2% sul **test set**.

Grafici

È possibile verificare come l'andamento delle curve sia abbastanza regolare e non mostri segni di overfitting o underfitting.
La loss sul validation set diminuisce molto rapidamente nelle prime 20 epoche per poi iniziare a scendere lentamente, ma stabilmente.
Allo stesso modo, l'accuracy aumenta molto velocemente nelle prime 10 epoche per poi diventare stabile attorno al 90%.

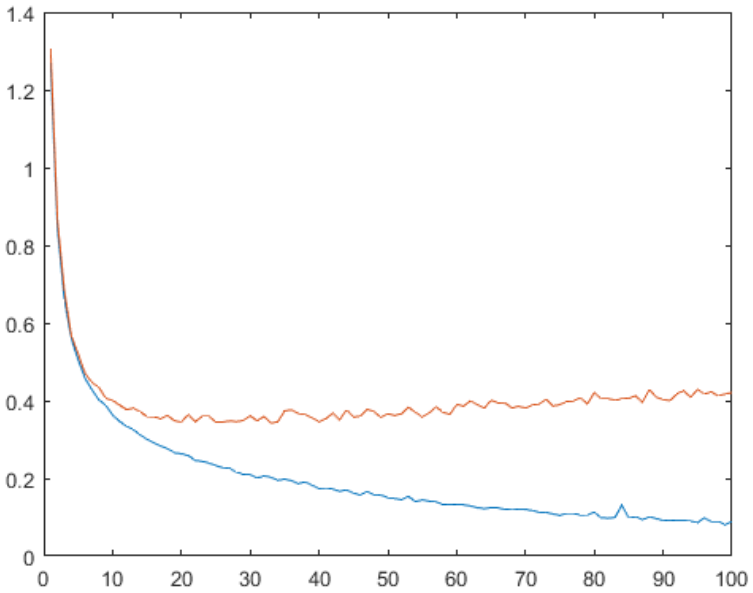


Figure 5.1: Loss sul training set (linea blu) e sul validation set (linea arancione) al crescere delle epoche

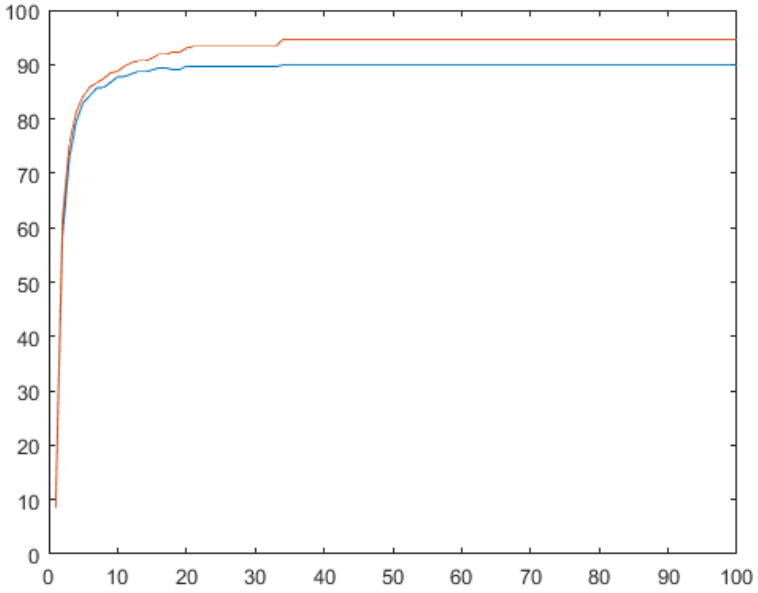
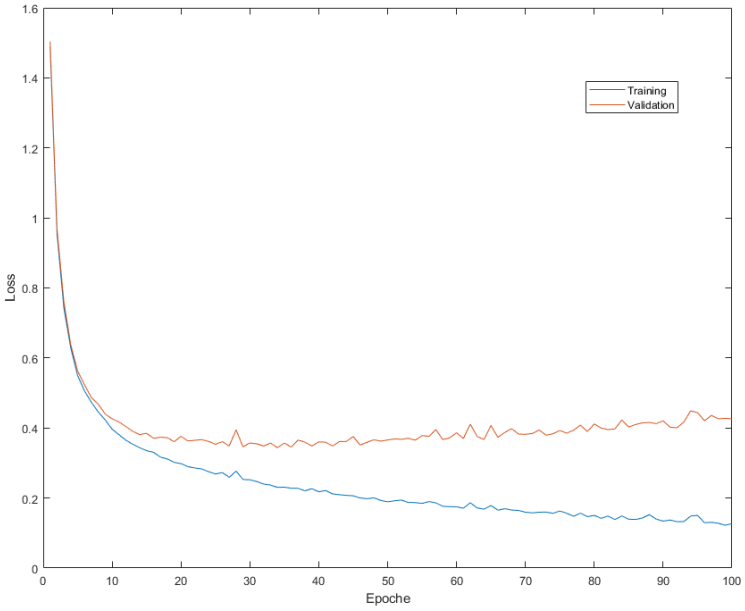


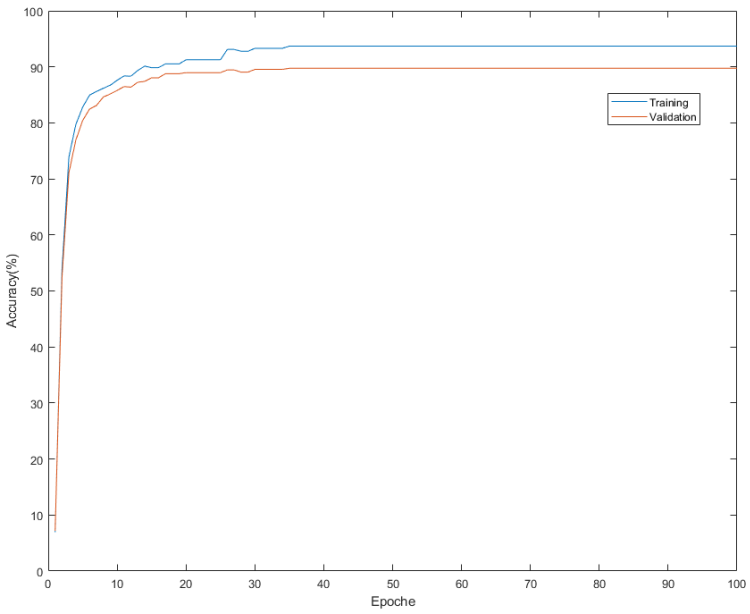
Figure 5.2: Accuratezza sul training set (linea arancione) e sul validation set (linea blu) al crescere delle epoche

5.1.3 64 kernel

Con un numero di filtri pari a 64 è stata riscontrata un'accuratezza pari al 93.7% sul **training set**, dell'89.7% sul **validation set** e dell'87.7% sul **test set**.

Grafici

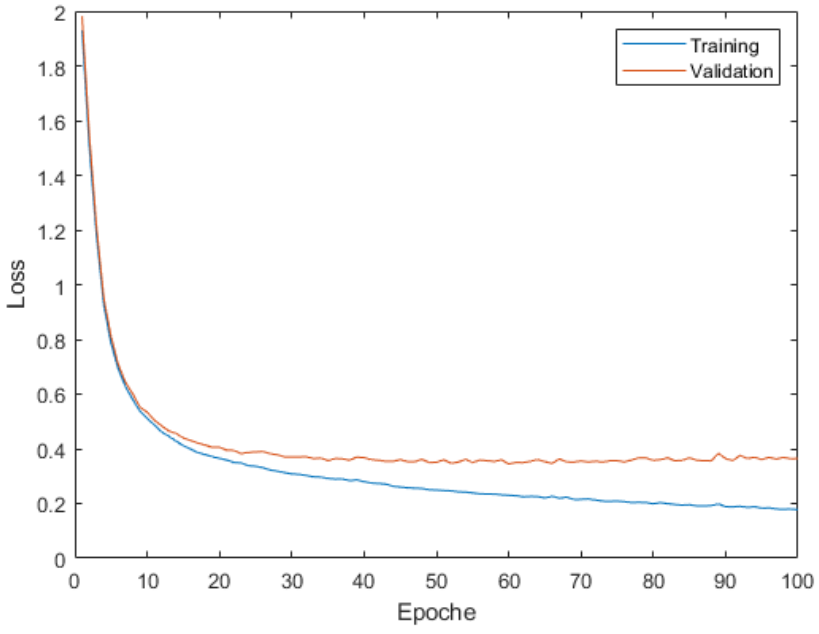


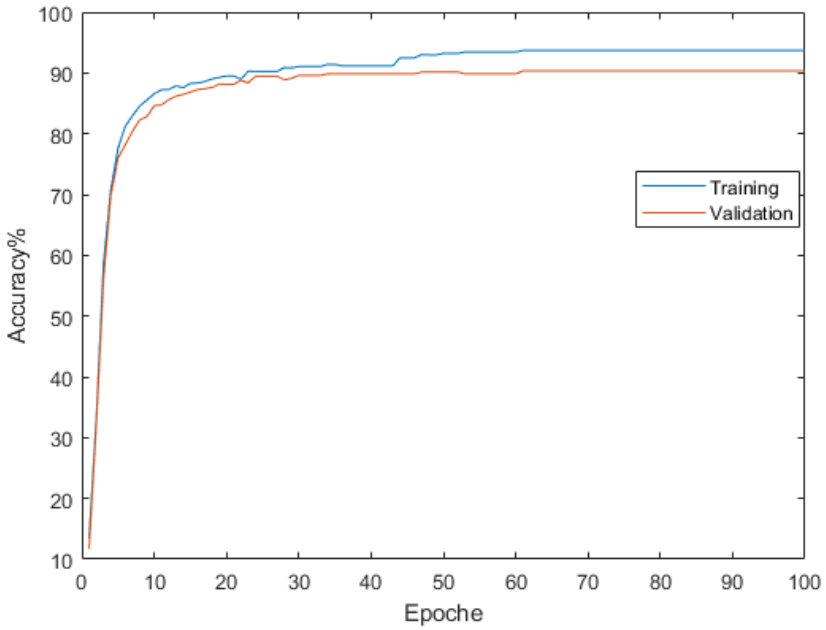


5.1.4 32 kernel

Con un numero di filtri pari a 32 è stata riscontrata un'accuratezza pari al 93.7% sul **training set**, dell'90.3% sul **validation set** e dell'86.5% sul **test set**.

Grafici



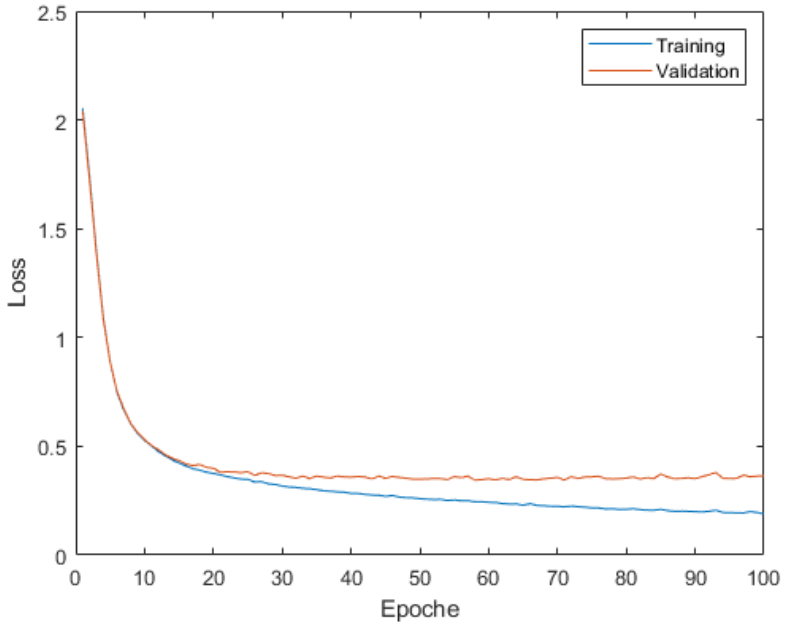


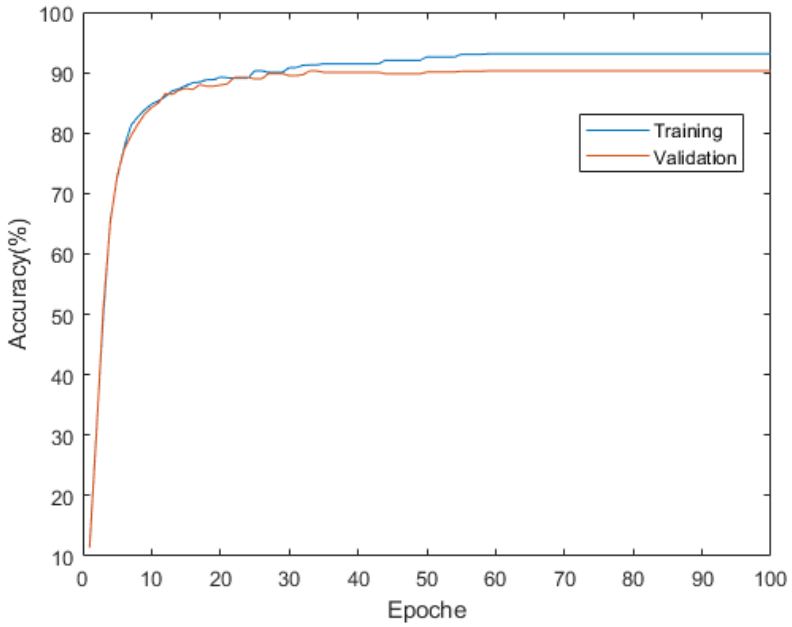
È possibile notare come con 32 kernel si sia ottenuto un risultato leggermente migliore rispetto a 128 kernel e che, a confronto, l'andamento della loss sia molto più stabile con 32 kernel.

5.1.5 16 kernel

Con un numero di filtri pari a 16 è stata riscontrata un'accuratezza pari al 93.1% sul **training set**, dell'90.2% sul **validation set** e dell'87% sul **test set**.

Grafici

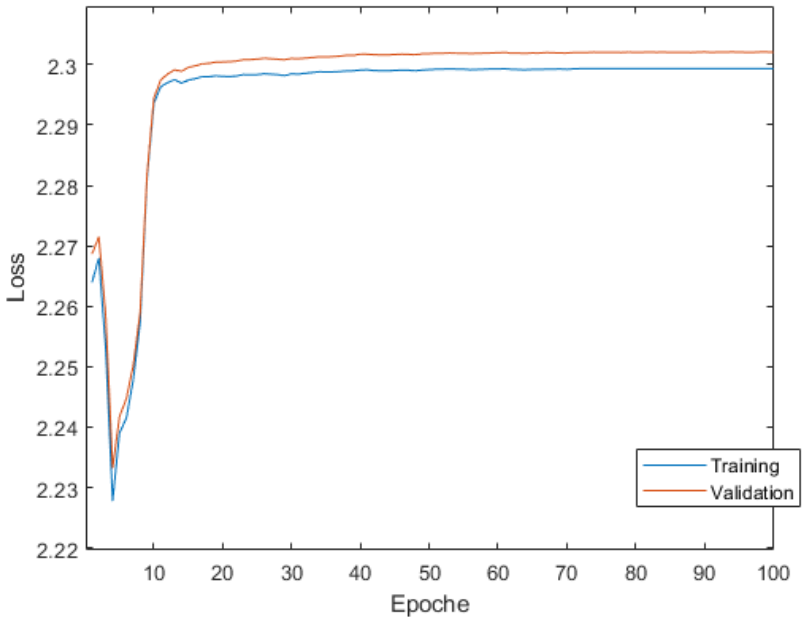


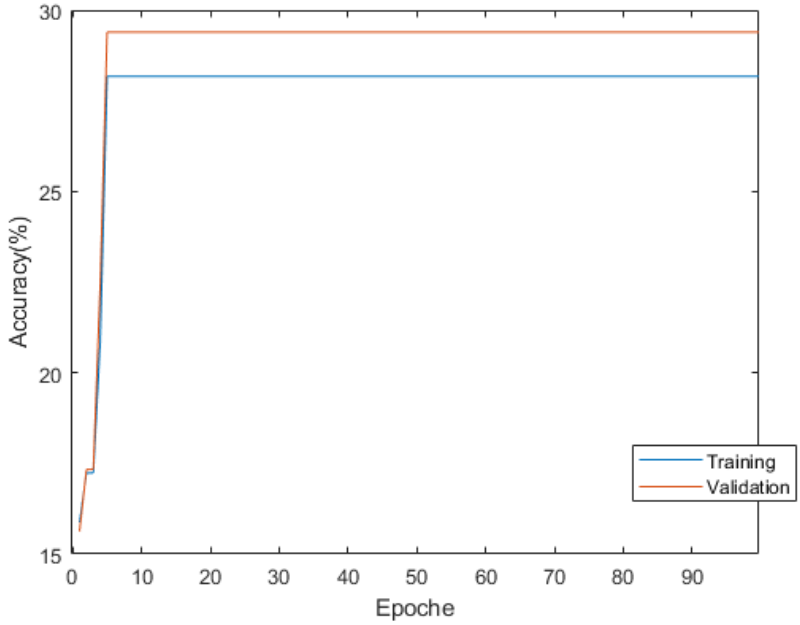


5.1.6 8 kernel

Con un numero di filtri pari a 8 è stata riscontrata un'accuratezza pari al 28.1% sul **training set**, dell'29.4% sul **validation set** e dell'30.5% sul **test set**.

Grafici





5.2 Esempi di classificazione

Di seguito sono riportate delle immagini create utilizzando lo strumento pennello di Photoshop su un'immagine 28x28 con sfondo nero.
Tali immagini sono state caricate su matlab e convertite in grayscale tramite la funzione **convertImage.m** presente nella consegna.

5.2.1 Tre

Si può osservare come, nonostante il rumore, la rete[1] sia piuttosto convinta verso la giusta classificazione.
Si precisa che il rumore non è stato introdotto volontariamente ma è probabilmente dovuto al processo di esportazione di photoshop.

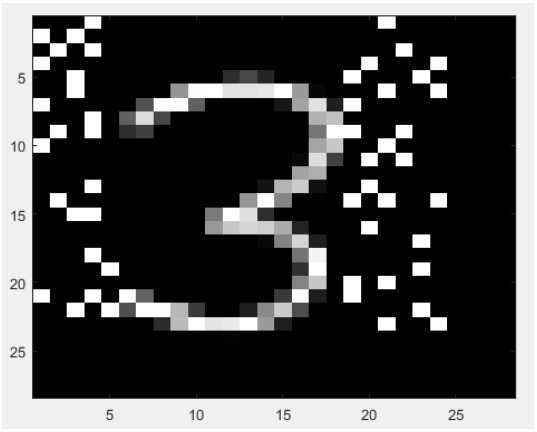


Figure 5.3: Esempio di "3" scritto a mano

```
Result:3
Probability:94.1642%
Second best match:2
Probability:5.7392%
```

[1]Questo esempio e quelli successivi sono tutti basati sulla rete addestrata con 128 kernel

5.2.2 Sei

Un altro esempio interessante da analizzare e peggio classificato dalla rete rispetto all'immagine mostrata precedentemente, è dato dal seguente sei, generato alla stessa maniera. Si può notare come, nonostante il forte rumore la rete riesca a classificarlo in maniera corretta ma con un certo grado di incertezza.

```
Result:6
Probability:57.9637%
Second best match:4
Probability:21.3095%
```

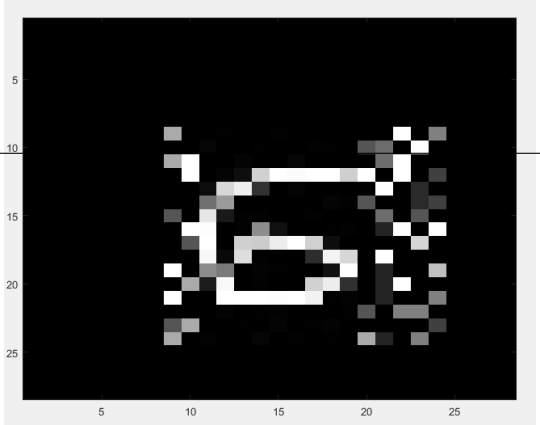


Figure 5.4: Esempio di "6" scritto a mano

5.2.3 Otto

In questo terzo esempio viene presentato un "8" generato alla usuale maniera, in questo caso la prediction della rete è un po' meno incerta rispetto al caso precedente, ma comunque piuttosto migliorabile.

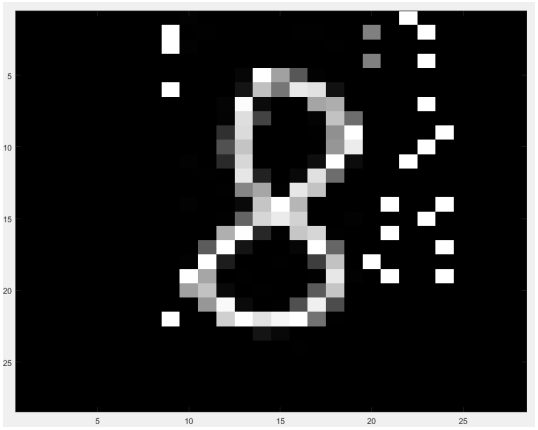
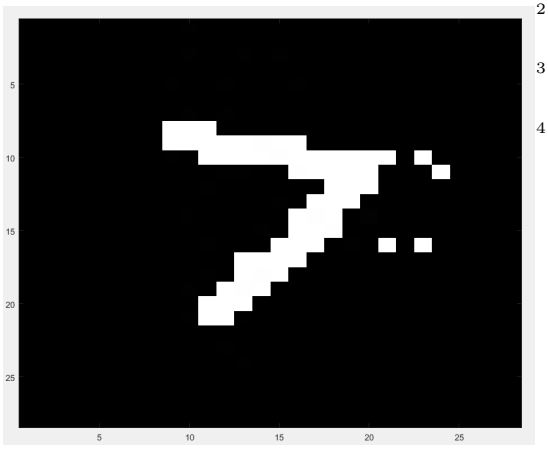


Figure 5.5: Esempio di "8" scritto a mano

```
Result:8
Probability:65.9619%
Second best match:2
Probability:16.6924%
```

5.2.4 Sette

In questo ultimo esempio si può notare come la predizione sia riuscita correttamente e con un forte grado di sicurezza da parte della rete.




```
Result:7
Probability:97.3677%
Second best match:9
Probability:0.80367%
```

Figure 5.6: Esempio di "7" scritto a mano

5.3 Tabella riassuntiva delle performance

Di seguito è riportata la tabella riassuntiva dell'accuratezza al variare del numero di kernel, tenendo costante il valore degli altri iper-parametri.

# Kernel	Accuracy Train	Accuracy Validation	Accuracy Test
8	28.1%	29.4%	30.5%
16	93.1%	90.2%	87%
32	93.7%	90.3%	86.5%
64	93.7%	89.7%	87.7%
128	94.6%	89.9%	86.2%
256	94.9%	89.1%	85.9%

Chapter 6

Tutorial - utilizzo della rete

Lo script per la creazione della rete e del training è *StartTraining.m*. La definizione della rete richiede 3 *cell array*:

- Uno per i livelli della rete definito nel seguente modo:

  $\{layer_1, \ldots, layer_n\}$ con

  $$layer_i = \begin{cases} \{type = 0, dim\} & \text{se } layer_i \text{ è un livello di input} \\ \{type = 1, n\_neurons\} & \text{se } layer_i \text{ è full-connected} \\ \{type = 2, n\_neurons, dim, padding, stride\} & \text{se } layer_i \text{ è convoluzionale} \end{cases}$$

  Per il livello convoluzionale, $n\_neurons$ indica, seppur con un errore di nomenclatura, il numero di kernel.

  Nel nostro caso, per i test presentati in questo documento, abbiamo usato il seguente *cell array* (salvato come *layers.mat*:

  {{type=0, dim = [28 28 1]},
  {type=2, n_neurons=128, dim=[4 4], padding=0, stride=2},
  {type=1, n_neurons=10}
  }

- Uno per le funzioni di attivazione dei livelli definito nel seguente modo:

  $\{actv_1, \ldots, actv_n\}$

Nel nostro caso (salvato come *actvFunc.mat*):
{@relu, @identity}

- Uno per le derivate delle funzioni di attivazione definito nel seguente modo:
  $\{deriv\_actv_1, \ldots, deriv\_actv_n\}$
  Nel nostro caso (salvato come *actvFuncDeriv.m*):
  {@reluDeriv, @identityDeriv}

A questo punto è possibile creare la rete con la funzione *net_conv_FC* che prende in ingresso i 3 cell array definiti sopra e il numero di livelli della rete che, nel nostro caso, è 3 (input, convoluzionale, denso):

```
1 net = net_conv_FC(layers, actvFunc, actvFuncDeriv, 3);
```

La rete restituita è una struct con i seguenti campi:

- *weights*: cell array che contiene le matrici dei pesi delle connessioni tra i livelli. In particolare, contiene *n_layers*-1 matrici di pesi.
- *activations*: contiene il cell array delle funzioni di attivazione passato in ingresso
- *deriv_func*: contiene il cell array delle derivate delle funzioni di attivazione passato in ingresso
- *layers*: cell array che contiene le informazioni sui livelli. Ogni elemento ha le informazioni prese dal cell array passato in ingresso (nel nostro caso *layers*) più delle informazioni che vengono aggiunte durante la costruzione della rete. In particolare, le informazioni aggiuntive sono *output_shape*, *use_bias*, *bias* per tutti i livelli (tranne di input), *feature_map_dim*, *H_out*, *W_out* per i livelli convoluzionali

- *n_ layers*: numero di livelli (compreso quello di input)

Il training della rete viene fatto con la funzione *learningPhase_ convFC* specificando anche la funzione di errore e la sua derivata:

```
[err, final_net, err_val, acc_tr, acc_val] =
   learningPhase_convFC(net, EPOCHE, XT, YT, XV, YV,
   @softMaxCrossEntropy, @softMaxCrossEntropyDeriv,
   TRAINING_TYPE, ETA, MOMENTUM, BATCH_SIZE);
```

Abbiamo incluso uno script, *StartTraining.m*, per eseguire il training con gli stessi parametri utilizzati nel progetto. Quindi è sufficiente cambiare i parametri nello stesso script per eseguire esperimenti diversi. Abbiamo implementato una funzione *predict* che prende in ingresso la rete e gli input (nel nostro caso immagini) e stampa la classificazione della rete con le prime due probabilità più alte nell'array restituito. È possibile quindi testare la rete dopo l'addestramento utilizzando questa funzione.

Chapter 7

Conclusione

In conclusione, la rete da noi implementata non risulta essere generalizzata: non è possibile, allo stato delle cose, inserire ulteriori livelli convoluzionali nella rete, tuttavia è possibile aggiungere livelli densi. Di conseguenza, un punto di partenza per implementazioni e migliorie future potrebbe essere quello di generalizzare la rete per accogliere più livelli convoluzionali.