

UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II



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

Elaborato finale in **Intelligenza Artificiale**

## ***Dal perceptrone di Rosenblatt alle Reti Convolutionali***

Anno Accademico 2014/2015

Candidato:

**Biagio Marco Liccardo**

**matr. N46001785**

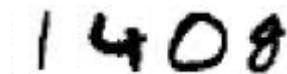
## Indice

Introduzione.....	2
Percettroni e Reti Neurali.....	3
Neuroni Sigmoidali.....	7
L'architettura di una Rete Neurale.....	11
Una semplice rete per risolvere il nostro problema....	12
La Discesa del Gradiente.....	15
Reti Convoluzionali (Deep Neural Networks)....	17

## Introduzione

Il sistema visivo dell'uomo è uno dei grandi interrogativi del mondo.

Se consideriamo una sequenza numerica scritta a mano come quella nella figura di seguito:

The image shows the number '1408' written in a casual, handwritten style. The '1' is a simple vertical stroke, the '4' has a loop at the top, the '0' is a simple oval, and the '8' has a loop at the top and a tail that curves back to the left.

ognuno di noi saprà dire in maniera abbastanza semplice che la sequenza appena vista è 1408.

Il fatto che riconoscere questa sequenza ci sembri semplice è però fallace, dato che è il nostro cervello a fare automaticamente la maggior parte del lavoro. Portiamo nella nostra testa un super computer che compie le operazioni complicate di cui abbiamo bisogno; questo è possibile perché il nostro cervello è fatto di varie cortecce, quella principale, la visiva, contiene al suo interno circa 140 milioni di neuroni, uniti tra loro tramite milioni di connessioni. La corteccia visiva tuttavia non lavora da sola, bensì è aiutata da altre cortecce che compiono operazioni più complesse (come ad esempio quelle di “image processing”) e ci permettono di riconoscere le immagini.

Riusciamo a comprendere la difficoltà di questo lavoro se proviamo a scrivere un programma che permetta ad un computer di riconoscere le cifre viste prima: noteremo che l'operazione che abbiamo svolto precedentemente in maniera molto semplice, e che ci sembrava così elementare, risulta ora più difficile e complessa. Non è facile infatti spiegare ad un computer, tramite un algoritmo, come noi siamo in grado di riconoscere i numeri e le loro forme (ad esempio il 9 che è fatto da un cerchio in alto, e da una linea dritta che scende dal lato destro) e, se provassimo a creare regole precise per questo lavoro, finiremo presto con l'incontrare eccezioni e casi particolari, che lo renderebbero troppo oneroso.

Le Reti Neurali si avvicinano al problema in maniera differente. L'idea è quella di prendere in considerazione un grande insieme di cifre note come *training example* (esempi di addestramento), come in figura:



Viene quindi sviluppato un sistema che può imparare a riconoscere le cifre a partire proprio da questi esempi di addestramento.

In sostanza le reti neurali utilizzano questi esempi per dedurre automaticamente regole atte a riconoscere le cifre scritte a mano.

Se vogliamo migliorare l'accuratezza della nostra rete possiamo aumentare significativamente il numero degli esempi forniti in modo da dare alla rete quante più informazioni possibili riguardo le cifre considerate.

## Percettroni e Reti Neurali

Dobbiamo ora spiegare cosa sia una rete neurale e per farlo possiamo utilizzare la definizione di Hecht-Nielsen:

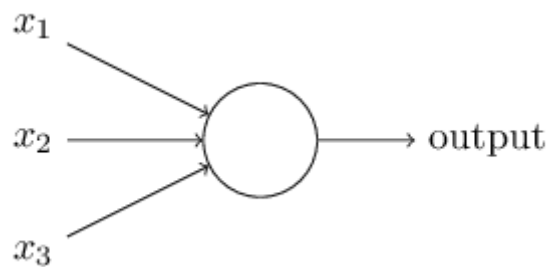
Una Rete Neurale è una struttura parallela che processa informazioni distribuite. Tale rete consta di elementi di processazione (PEs o neuroni che possono avere una memoria locale e che sono in grado di processare localmente informazioni) interconnessi tramite canali (detti connessioni) che trasmettono segnali unidirezionali. Ogni neurone ha una singola connessione di uscita che si dirama in un certo numero di

connessioni collaterali; ognuna di questa trasporta lo stesso segnale - il segnale d' uscita del neurone. Questo segnale d' uscita può essere di qualunque tipo matematico. La computazione compiuta all'interno di ciascun neurone può essere definita arbitrariamente con l'unica restrizione che deve essere completamente locale; cioè deve dipendere solo dai valori correnti dei segnali d'ingresso che arrivano al neurone tramite opportune connessioni e dai valori immagazzinati nella memoria locale del neurone.

Un tipo di neurone artificiale è il *percettrone*, fu sviluppato tra gli anni 50 e 60 dallo scienziato Frank Rosenblatt.

Un perceptrone prende degli input e produce un singolo output.

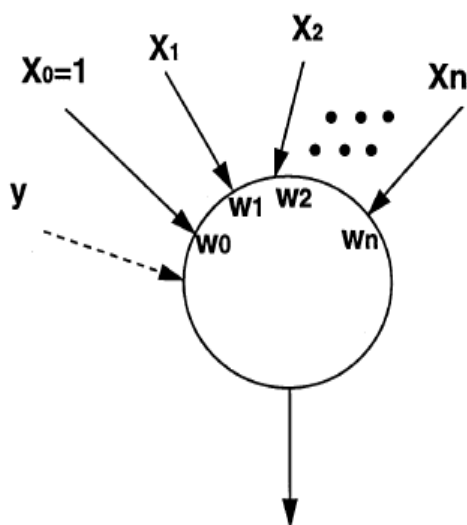
Lo schema appena descritto può essere riassunto nella seguente immagine:



Nell'esempio in figura il perceptrone prende 3 input, ma può ovviamente prenderne di più o di meno, e produce un'uscita.

Il Percettrone di Rosenblatt è leggermente diverso da quanto appena

visto: introduce infatti il concetto di *pesi* collegati agli input, che stanno ad indicare l'importanza di quell'ingresso per l'uscita finale. La figura analizzata sopra può quindi essere cambiata con quella sulla sinistra.



Vediamo che ci sono diversi input ( $X_1...X_n$ ), ognuno dei quali è collegato al proprio peso ( $w_0...w_n$ ). L'input  $X_0$  è sempre uguale ad 1.

L'output 0 o 1 è determinato dalla somma pesata come possiamo vedere nella seguente immagine:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Quindi l'uscita è 1 se la somma pesata è maggiore della soglia, 0 altrimenti.

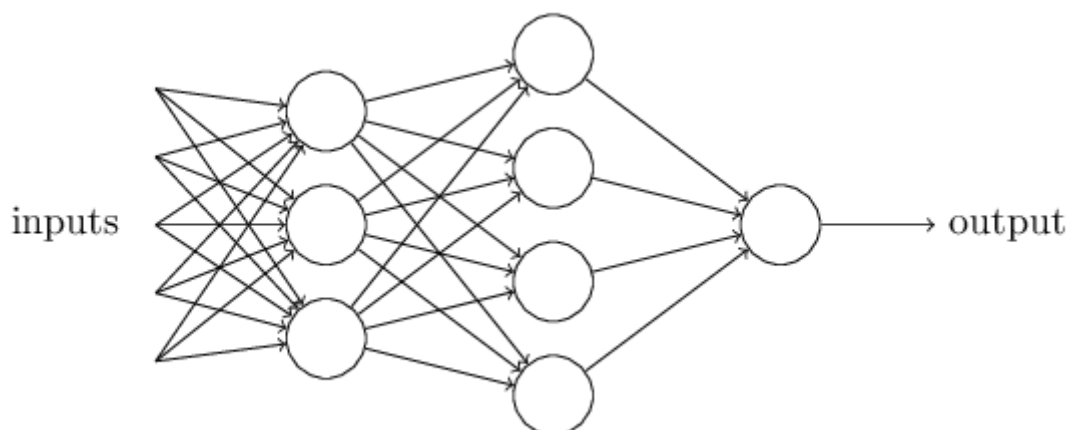
La soglia è un numero reale, un parametro del

neurone.

Questa forma è però molto scomoda e possiamo quindi cambiarla facendo alcuni accorgimenti. Scriveremo il seguente termine  $\sum_j w_j x_j$  come un prodotto  $w \cdot x \equiv \sum_j w_j x_j$  dove  $w$  e  $x$  sono ovviamente i vettori dei pesi e degli input. Utilizzeremo anche il termine *bias* ( $b \equiv -\text{threshold}$ ) per poi portare la soglia alla sinistra dell'equazione. Possiamo quindi riscrivere il tutto nel seguente modo:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Il bias è una misura di quanto semplice è per il percettrone produrre come uscita 1. Se infatti il bias è molto grande avremo molte chance di produrre in uscita un 1, altrimenti, se il bias è negativo e molto grande in modulo, sarà molto difficile non ottenere uno 0.



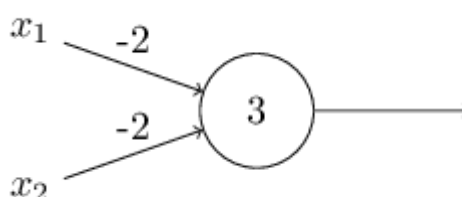
Se consideriamo l'immagine sopra, possiamo vedere che ci sono 3 strati. I percettroni del primo strato prendono decisioni dopo aver fatto la somma pesata degli input, mentre quelli del livello intermedio lo fanno

dopo aver pesato le uscite del livello ad esso precedente; così facendo è possibile prendere decisioni man mano più difficili.

Possiamo quindi asserire che andando ad aumentare il numero degli strati, la rete sarà in grado di prendere decisioni via via più complesse come vedremo in seguito ad esempio con le reti convoluzionali.

Un modo in cui possiamo utilizzare i percettroni è per realizzare funzioni logiche come una AND, una OR o una NAND.

Se abbiamo un percettrone con 2 input, entrambi con peso -2 e bias 3 come in figura:



Vediamo che il seguente schema implementa correttamente una NAND:

$$\text{input} = 00 \rightarrow (-2) * 0 + (-2) * 0 + 3 = 3 > 0 \rightarrow \text{output} = 1$$

$$\text{input} = 01 \rightarrow (-2) * 0 + (-2) * 1 + 3 = 1 > 0 \rightarrow \text{output} = 1$$

$$\text{input} = 10 \rightarrow (-2) * 1 + (-2) * 0 + 3 = 1 > 0 \rightarrow \text{output} = 1$$

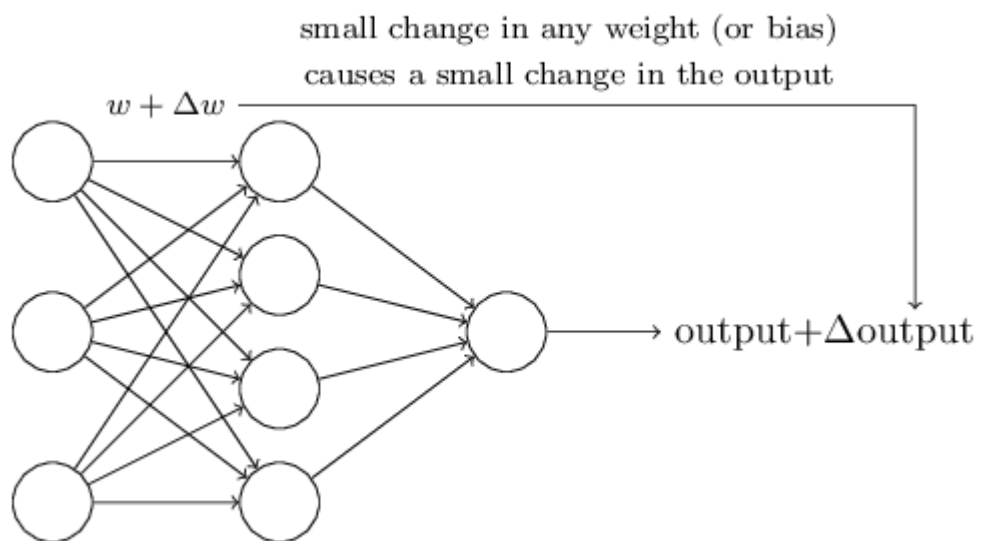
$$\text{input} = 11 \rightarrow (-2) * 1 + (-2) * 1 + 3 = -1 \leq 0 \rightarrow \text{output} = 0$$

Ci sono dei metodi che possiamo concepire, detti *learning algorithms* (algoritmi di apprendimento) che ci permettono di regolare automaticamente i pesi e i bias di una rete di neuroni artificiali. Questo *tune* avviene come risposta a stimoli esterni, e senza il diretto intervento di un programmatore.

Gli algoritmi di apprendimento ci permettono di utilizzare i neuroni in modo totalmente diverso dalle convenzionali porte logiche. A differenza delle reti fatte di NAND (o similari), la nostra rete neurale può semplicemente imparare a risolvere problemi, anche di grande difficoltà.

## Neuroni Sigmoidali

Supponiamo di avere una rete di percettroni che vorremmo utilizzare per risolvere un problema. Per capire come funziona la legge di apprendimento supponiamo di fare dei piccoli cambiamenti ai pesi o al bias nella rete. Ciò che ci piacerebbe ottenere è che questi piccoli cambiamenti corrispondano ad un piccolo cambiamento nell'output della rete. Questa proprietà rende dunque possibile l'apprendimento.



Se dunque ciò è verificato allora possiamo andare a modificare pesi e bias in modo da far comportare la rete come vogliamo.

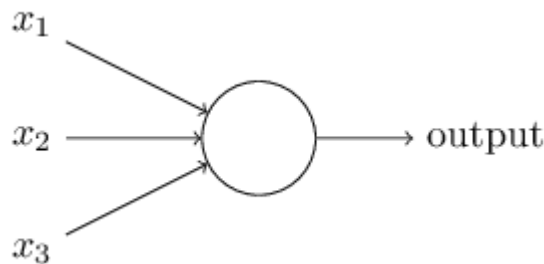
Se per esempio dobbiamo riconoscere un “9” ma viene erroneamente riconosciuto un “8”, allora possiamo andare ad applicare piccoli cambiamenti, modificando pesi e bias, in modo da garantire che la rete si avvicini sempre più a classificare l'immagine correttamente. Ripetendo questo procedimento possiamo produrre output sempre più precisi e la rete sta apprendendo.

Il problema che riscontriamo quando la nostra rete contiene percettroni è che un piccolo cambiamento ai pesi o ai bias di ogni singolo percettrone può cambiare completamente l'output, portandolo ad esempio da 0 a 1. Questo problema può causare enormi problemi nel comportamento della rete, che potrebbe per esempio ora riuscire a riconoscere un “9” correttamente, ma sbagliare tutte le altre cifre. Risulta così molto



complicato andare a cambiare pesi e bias in modo da far comportare la rete in maniera adeguata.

Possiamo aggirare questo problema introducendo un nuovo tipo di neuroni artificiali chiamati neuroni sigmoidali. Sono simili ai perceptron ma, in questo caso, cambiando leggermente i pesi e il bias, l'output subisce solo lievi variazioni. Questo è l'aspetto cruciale che permette ai neuroni sigmoidali di apprendere.



L'immagine con cui descriveremo i neuroni sigmoidali è la stessa che abbiamo utilizzato per descrivere i perceptron; la differenza risiede nel fatto che gli input delle reti sigmoidali possono essere anche valori compresi tra 0 e 1 (ad esempio i valori 0,148 e 0,83 sono input validi). Anche i neuroni sigmoidali hanno, come i perceptron, un peso per ogni input e un bias, ma, a differenza dei perceptron, l'uscita non è 0 o 1, bensì è:  $\sigma(w \cdot x + b)$  dove  $\sigma$  è detta **sigmoid function** (*funzione sigmoide*) ed è definita come:

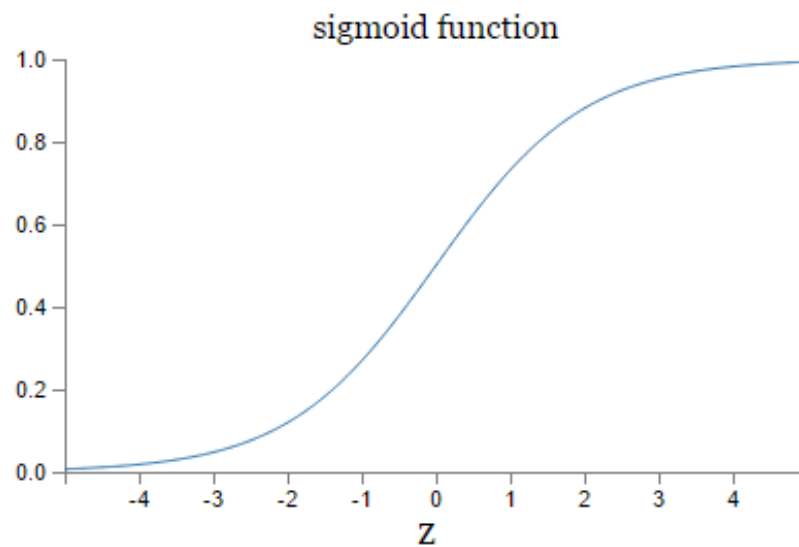
$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

oppure se consideriamo  $z \equiv w \cdot x + b$  allora possiamo scrivere

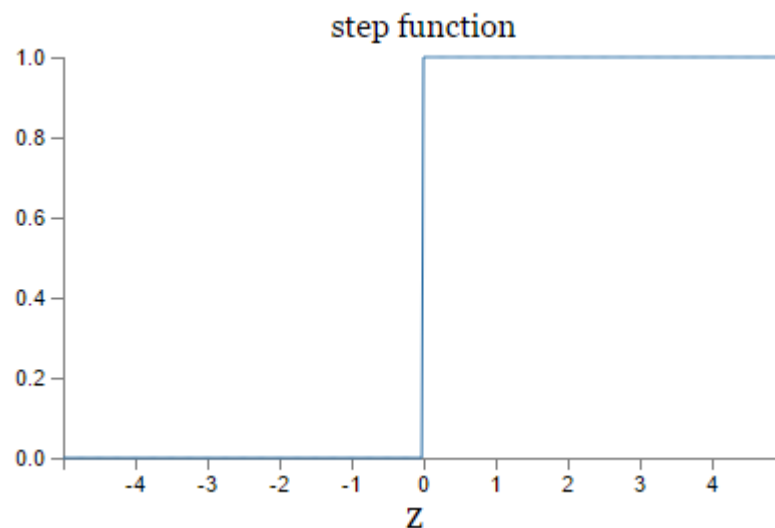
$$\frac{1}{1 + \exp(-w \cdot x + b)}$$

se  $z$  è grande e positivo allora l'output del neurone sigmoideale è approssimativamente 1, proprio come sarebbe stato per il perceptrone; dualmente se  $z$  è molto piccolo e negativo l'uscita è 0, e anche qui il comportamento è come quello del perceptrone. Solo quando  $z$  è di una misura media allora c'è grande differenza tra i comportamenti dei due neuroni.

La forma algebrica di  $\sigma$  plottata è la seguente:



Questa forma risulta essere la versione smussata di una funzione a gradino.



In effetti notiamo che, se la funzione sigmoidale  $\sigma$  fosse stata una funzione a gradino, allora avremmo avuto che il neurone sigmoidale sarebbe stato un percettrone, dato che avrebbe restituito in uscita un valore 0 o 1 a seconda del valore positivo o negativo di  $z \equiv w \cdot x + b$ . È proprio il fatto che  $\sigma$  rappresenti un percettrone smussato che garantisce che un cambiamento poco significativo dei pesi ( $\Delta w_j$ ) o del

bias ( $\Delta b$ ) produca un piccolo cambiamento ( $\Delta output$ ) nell'output del neurone.

$\Delta output$  può essere visto come:

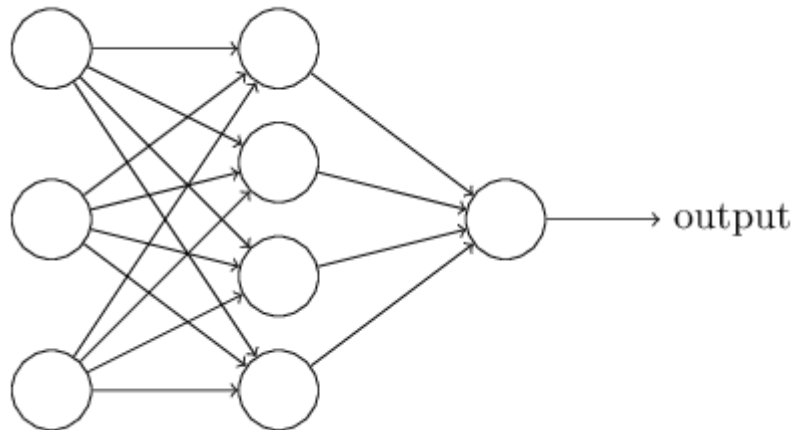
$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \sum_j \frac{\partial output}{\partial b} \Delta b$$

dove  $\frac{\partial output}{\partial w_j}$  e  $\frac{\partial output}{\partial b}$  rappresentano le derivate parziali dell'output rispetto a  $w_j$  e a  $b$ .

Il fatto che l'uscita dei neuroni sigmoidali può assumere valori compresi tra 0 e 1 può rivelarsi utile o meno a seconda dei casi: risulta utile se per esempio vogliamo usare l'output per rappresentare la media dell'intensità dei pixel in un'immagine, mentre, risulta essere meno utile, quando vogliamo ad esempio indicare se un valore di input "è un 8" oppure "non è un 8". Sarebbe un lavoro più facile da fare utilizzando il percettrone, indicando con 1 o con 0 rispettivamente "è 8" o "non è 8", possiamo però comunque utilizzare una convenzione che decida ad esempio che con un valore  $\geq 0,5$  indichiamo "è 8" e con un valore  $< 0,5$  indichiamo "non è 8".

## L'architettura di una Rete Neurale

Analizziamo la rete della figura seguente:

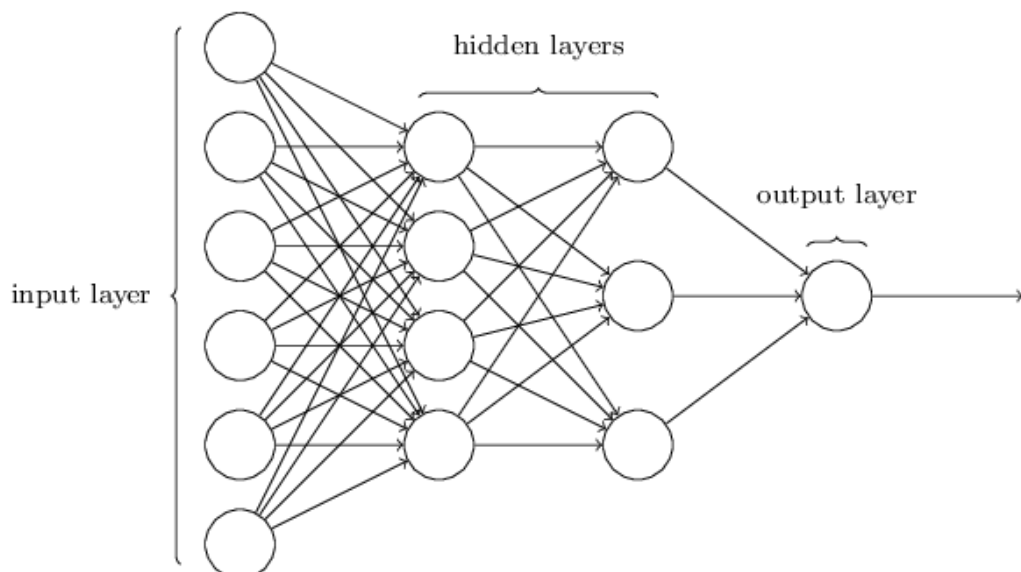


Lo strato più a sinistra è chiamato *input layer* (strato di input), dato che contiene i cosiddetti neuroni di ingresso.

Lo strato più a destra è chiamato invece *output layer* (strato di output), dato che contiene i neuroni di uscita.

Lo strato di mezzo è invece detto *hidden layer* (strato nascosto), dato che i neuroni facenti parte di questo strato non sono né neuroni di input né di output.

La figura appena analizzata ha un solo strato hidden, ma le reti possono avere anche più di questi strati nascosti, come ci mostra la prossima immagine:



Queste reti con più di uno strato nascosto sono chiamate *MultiLayerPerceptron (MLP)* (questo nome risulta però essere improprio, dato che gli MLP non sono fatti di percettroni, bensì di neuroni sigmoidali).

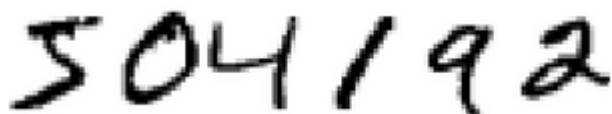
Le reti neurali di questo tipo, in cui l'output di uno strato è utilizzato come input per lo strato successivo, sono chiamate reti *feedforward* (o anche reti "in avanti"): ciò vuol dire che i neuroni dello stesso livello non possono comunicare tra loro, né con quelli di livello inferiore e nella rete non ci sono cicli. Se ci fossero cicli allora la funzione di input o dipenderebbe dall'output e ciò non avrebbe troppo senso...quindi i cicli sono banditi!

Esistono però anche reti in cui i loop sono possibili: questi modelli sono chiamati *reti neurali ricorrenti*. In queste reti l'input va ad influenzare l'output non immediatamente ma solo dopo un certo intervallo di tempo, rendendo dunque possibili, ma soprattutto sensati, i cicli di cui abbiamo discusso in precedenza.

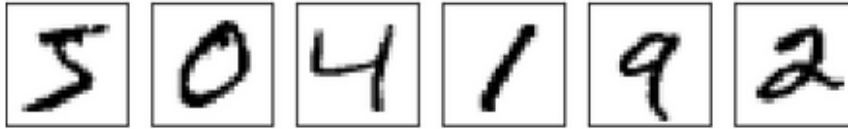
Sono comunque meno influenti delle reti feedforward dato che le loro leggi di apprendimento sono meno potenti.

## Una semplice rete per risolvere il nostro problema

Dopo aver definito le reti neurali, possiamo tornare al nostro problema originario, cioè quello di riconoscere le cifre scritte a mano. Possiamo dividerlo in due sotto-problemi: per prima cosa dobbiamo separare l'immagine da riconoscere tramite una segmentazione come mostrato nelle prossime due figure:

The image shows the handwritten number '504192' in a dark, slightly blurred font. The digits are connected and written in a cursive style.

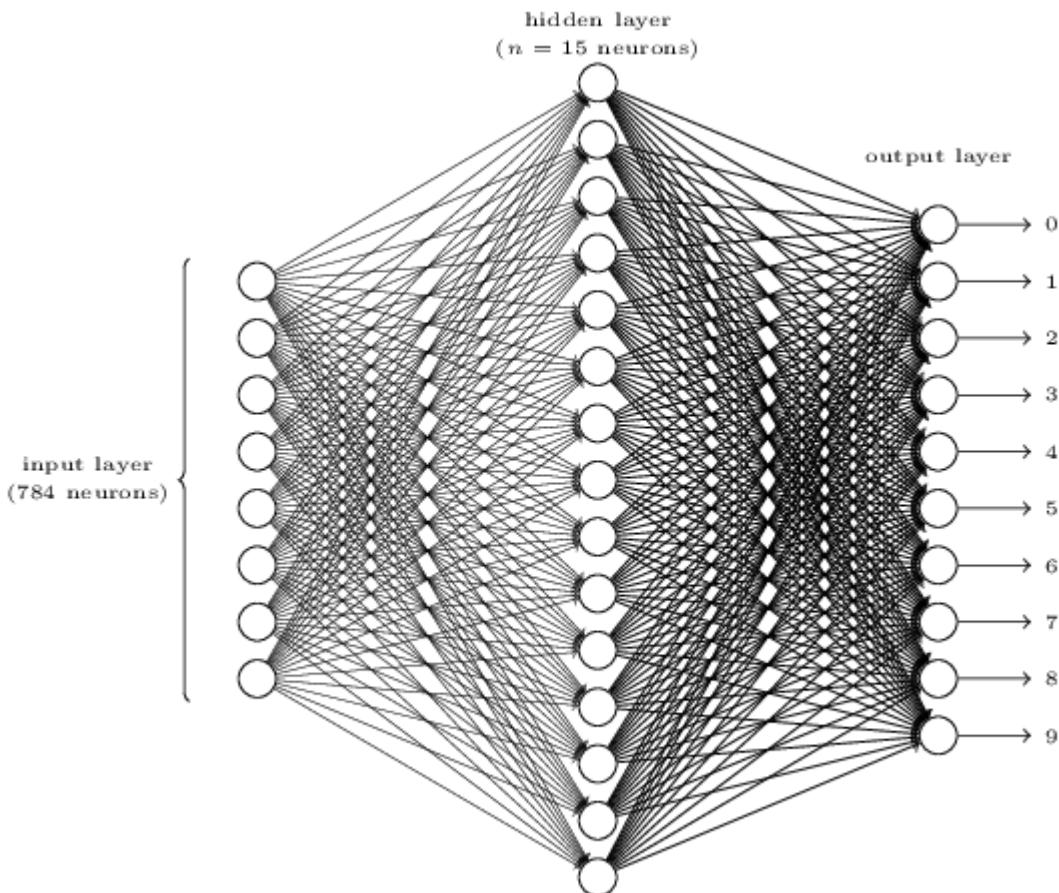
andando quindi a dividere l'immagine di sopra in sei immagini differenti.



Questo problema di segmentazione è risolto da noi umani con grossa facilità, mentre risulta di difficile risoluzione per un computer.

Una volta risolto questo primo passaggio dobbiamo affrontare il secondo sotto-problema, quello di riconoscere e classificare ogni singola cifra, andando quindi ad associare ad ogni immagine il numero corrispondente.

Ci concentreremo sul secondo problema, cioè quello di riconoscere le singole cifre, e, per farlo, utilizzeremo una rete neurale a tre strati come quella nella figura seguente:



Lo strato di input contiene 784 neuroni (la maggior parte sono esclusi dal diagramma per semplicità) dato che i dati di ingresso sono immagini fatte da una matrice di pixel di dimensione 28x28.

I pixel in input sono in scala di grigi; 0.0 rappresenta il bianco, 1.0 il nero e i valori compresi rappresentano le sfumature di grigio.

Il secondo strato è uno strato nascosto, formato da  $n=15$  neuroni (possiamo anche sperimentare diversi valori di  $n$ ).

Lo strato di output contiene 10 neuroni perché dobbiamo riconoscere 10 cifre (da 0 a 9). Il funzionamento tiene conto di quale neurone ha il valore di attivazione superiore e decide così quale valore in uscita verrà restituito: se ad esempio il neurone numero 6 sarà quello col valore di attivazione più elevato, allora riusciremo a dedurre che l'input era un 6.

Una volta descritta la nostra rete, ci domandiamo come possa imparare a riconoscere le cifre. La prima cosa di cui abbiamo bisogno è un insieme di dati da cui la nostra rete possa imparare il necessario per svolgere il suo compito: utilizzeremo quindi il *MNIST data set*. È un sottoinsieme di un più grande insieme di esempi collezionati dal *NIST, United States' National Institute of Standards and Technology*.



0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9

Si nota chiaramente dalla figura sovrastante che le cifre che abbiamo usato negli esempi fatti fino a questo momento, sono proprio cifre che fanno parte di quest'insieme.

Il MNIST è un database di cifre scritte a mano, che contiene un *training set* di 60,000 esempi e un *test set* di 10,000 esempi. Il training set sono cifre scritte da 250 persone, la metà delle quali sono impiegati dell'*US*



*Census Bureau*, mentre la seconda metà sono studenti del liceo. Sono immagini in scala di grigi, anch'esse di dimensione 28x28 pixel. Anche le immagini del Data Set sono di dimensione 28x28 pixel e in scala di grigi, e verranno da noi utilizzate per valutare quanto bene la nostra rete ha imparato a riconoscere le cifre scritte a mano. Il data set è stato preso da un diverso insieme di 250 persone rispetto a quelle utilizzate per il training set, tutte comunque impiegate dell' USCB o studenti del liceo. Questo cambiamento serve per vedere se il nostro sistema riconosce cifre scritte a mano con calligrafie diverse da quelle viste durante l'addestramento.

Di seguito indicheremo con  $x$  il vettore di input di dimensione 784 e con  $y = y(x)$  il corrispondente vettore di output di dimensione 10.

Esempio:

$$x = 8 \rightarrow y(x) = (0,0,0,0,0,0,0,0,1,0)^T$$

T è l'operazione di trasposizione che va a trasformare un vettore riga in un vettore colonna.

## La Discesa del Gradiente

Quello che ci piacerebbe avere è un algoritmo in grado di permetterci di trovare pesi e bias in modo che l'output della rete approssimi  $y(x)$  per ogni input  $x$ . Per quantificare la bontà del raggiungimento di questo obiettivo possiamo definire una *funzione di costo*:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

dove con  $w$  indichiamo tutti i pesi nella rete e con  $b$  tutti i bias,  $n$  è il numero totale degli input,  $a$  è il vettore degli output con  $x$  come input e la sommatoria è fatta su tutti gli input  $x$ .

$C$  è chiamata funzione di costo quadratico ed è spesso nota come **MSE** o anche *Mean Squared Error*.



Possiamo renderci conto che  $C(w,b)$  è non negativa dato che tutti i termini della somma sono non negativi. Inoltre  $C(w,b) \approx 0$  quando  $y(x) \approx a$ .

Quindi il nostro algoritmo di addestramento fa un buon lavoro se riesce a tarare i pesi e i bias in modo da ottenere  $C(w,b) \approx 0$  mentre non va bene se  $C(w,b)$  è molto grande; infatti ciò significherebbe che  $y(x)$  non è vicina all'output 'a' per un numero grande di input. Quindi il nostro algoritmo mira a trovare un set di pesi e bias che minimizzi il costo. Useremo per fare ciò un algoritmo noto come *gradient descent* (o discesa del gradiente).

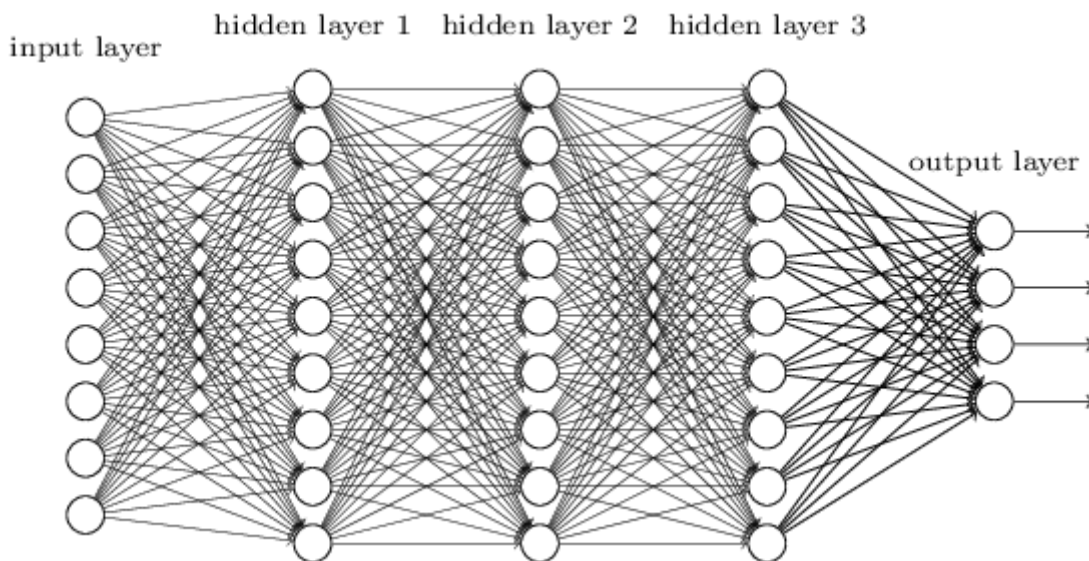
La discesa del gradiente è una tecnica di ottimizzazione che data una funzione matematica multidimensionale ci permette appunto di trovare il minimo della funzione.

Essa consiste nel valutare, in un punto scelto a caso, la funzione e il suo gradiente. Il gradiente, essendo una direzione di discesa, indica la direzione in cui la funzione tende a un minimo. Si sceglie poi un secondo punto nella direzione indicata dal gradiente. Se la funzione al secondo punto ha un valore inferiore al valore calcolato al primo punto, la discesa può continuare, seguendo adesso però il gradiente calcolato al secondo punto, che potrebbe essere molto diverso dal precedente.

La regola della discesa del gradiente è una delle regole più semplici di apprendimento delle reti neurali: è un apprendimento supervisionato e può essere applicata solo a reti neurali feedforward, cioè, come già spiegato in precedenza, reti nelle quali il flusso dei dati è unidirezionale ("in avanti"). Ci permette di calcolare la differenza tra i valori di output che la rete ottiene e quelli che invece dovrebbe ottenere. Può essere definita precursore dell'algoritmo di *backpropagation*.

## Reti Convoluzionali (Deep Neural Networks)

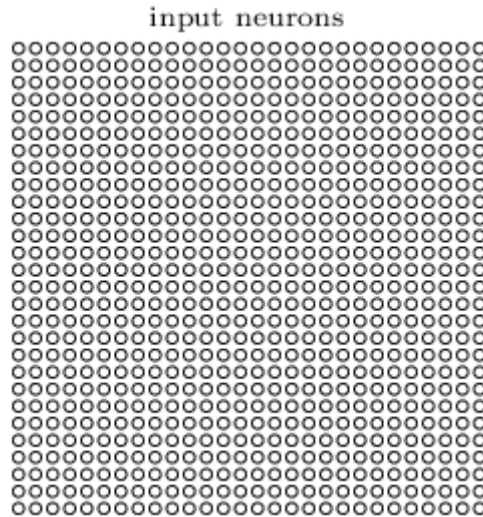
Abbiamo fino ad ora utilizzato reti in cui strati adiacenti sono totalmente collegati l'un l'altro: infatti ogni neurone nella rete è connesso ai neuroni negli strati adiacenti come in figura:



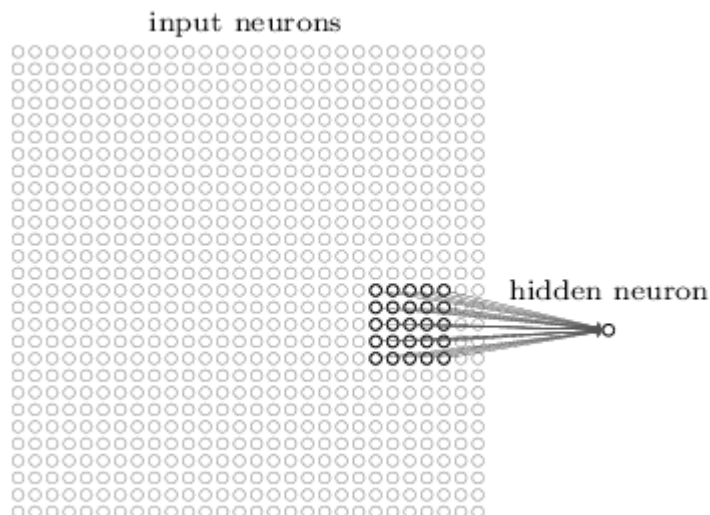
Andando a configurare la rete in modo che identifichi correttamente le cifre vediamo che, dopo vari tentativi, otteniamo un'accuratezza del 98% utilizzando training e test data dal MNIST data set. Questa rete non tiene conto della struttura spaziale delle immagini: potremmo invece utilizzare una struttura che possa trarre vantaggio proprio da questa componente spaziale, ed è qui che introduciamo le **reti neurali convoluzionali**. Esse utilizzano architetture che sono particolarmente adatte alla classificazione delle immagini, rendendo molto più veloce il loro addestramento.

Le reti neurali convoluzionali, anche dette **deep neural networks**, usano tre idee di base: *local receptive fields*, *shared weights and biases*, *pooling*.

*local receptive fields*: nelle reti convoluzionali consideriamo come input un qualcosa simile a quanto mostrato nella seguente figura:

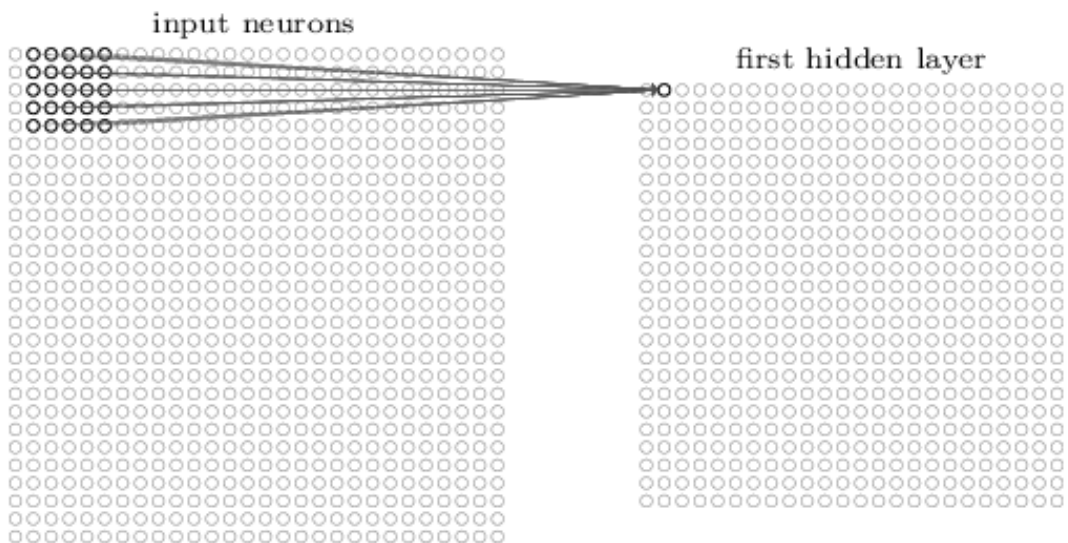
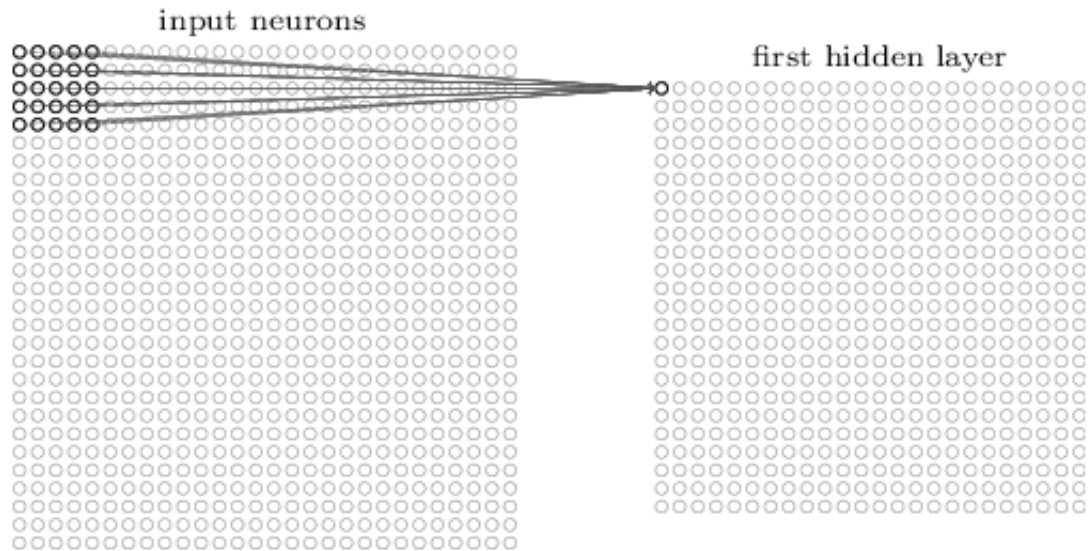


Faremo connessioni tra i pixel dello strato di input e quelli dello strato hidden, ma non li conatteremo tutti, bensì solo alcuni di questi. Ogni neurone del primo strato nascosto sarà connesso ad una piccola regione dello strato di ingresso, una regione 5x5 ad esempio, ottenendo la seguente figura:



Questa regione dell'immagine dello strato di input è chiamata *local receptive field*. Ogni connessione apprende un peso (ne otterremo quindi  $5 \times 5 = 25$ ), invece il neurone nascosto a cui è associata la connessione apprende un bias totale.

Andremo quindi a connettere le regioni ai singoli neuroni effettuando di volta in volta uno shift come nelle seguenti figure:

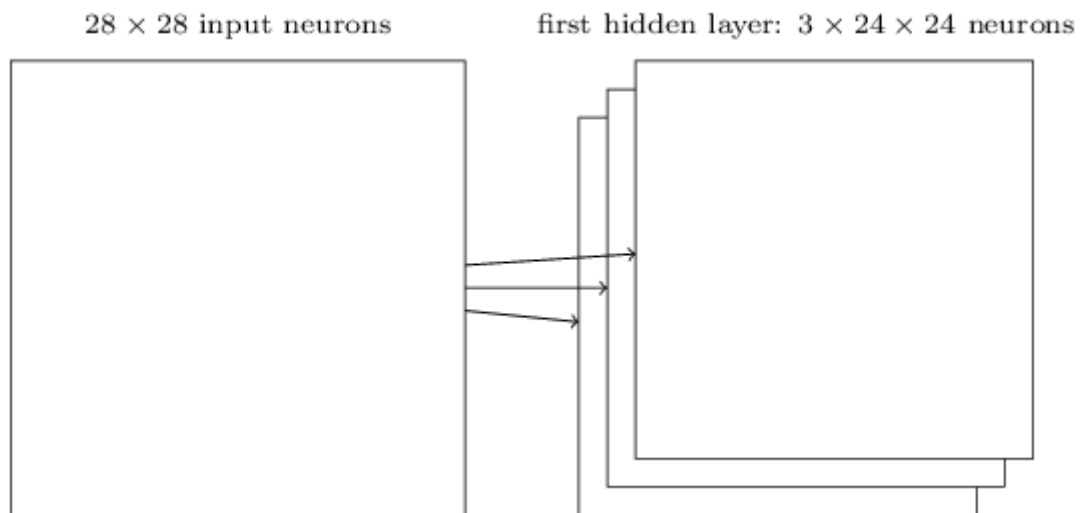


Così facendo se abbiamo un'immagine di input 28x28 e le regioni 5x5 otterremo 24x24 neuroni nello strato nascosto.

***shared weights and biases:*** abbiamo detto che ogni neurone ha un bias e 5x5 pesi connessi alla regione: utilizzeremo questi pesi e bias per tutti i 24x24 neuroni. Questo significa che tutti i neuroni nel primo strato nascosto riconosceranno la stessa feature, solo collocata diversamente nell'immagine di input. Questo rende le reti convoluzionali molto adattabili all'invarianza di un'immagine ad una traslazione.

Per questa ragione chiamiamo la mappa di connessioni dallo strato di input a quello nascosto *feature map*, i pesi *shared weights* e i bias *shared bias* dato che appunto sono condivisi.

Ovviamente per riconoscere un'immagine abbiamo bisogno di più di una mappa di feature, quindi uno strato convoluzionale completo è fatto da più feature maps:



Nell'immagine vediamo 3 feature maps: ovviamente il numero delle feature maps può aumentare nella pratica, e si può arrivare ad utilizzare strati convoluzionali con addirittura 20 o 40 feature maps.

Un grande vantaggio nella condivisione di pesi e bias è la riduzione consistente dei parametri coinvolti in una rete convoluzionale. Considerando il nostro esempio, per ogni feature map abbiamo bisogno di 25 pesi (5x5) e un bias (condivisi), quindi 26 parametri. Supponendo di avere 20 feature maps avremo 520 parametri da definire.

Con una rete totalmente collegata con 784 neuroni di input e ad esempio 30 neuroni dello strato hidden, abbiamo bisogno di 784x30 pesi più 30 bias, arrivando ad un totale di 23,550 parametri.

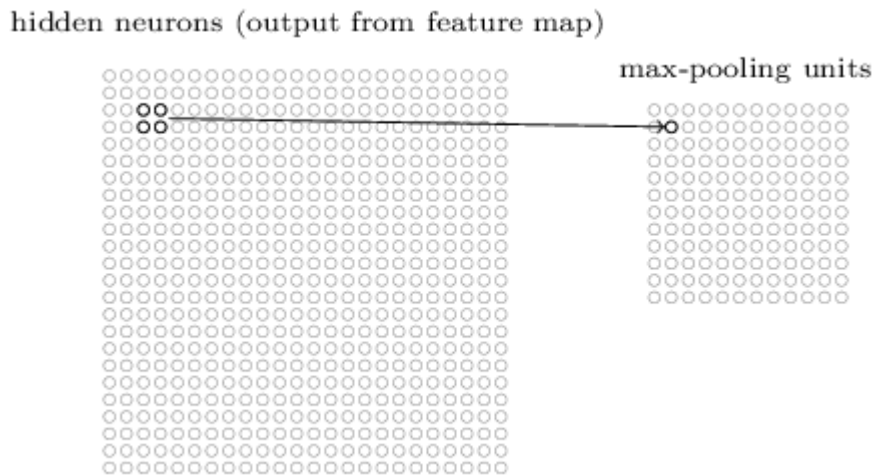
La differenza è evidente.

***pooling layers:*** le reti convoluzionali usano anche strati pooling posizionati subito dopo gli strati convoluzionali; questi semplificano l'informazione di output dello strato precedente ad esso (quello convoluzionale).

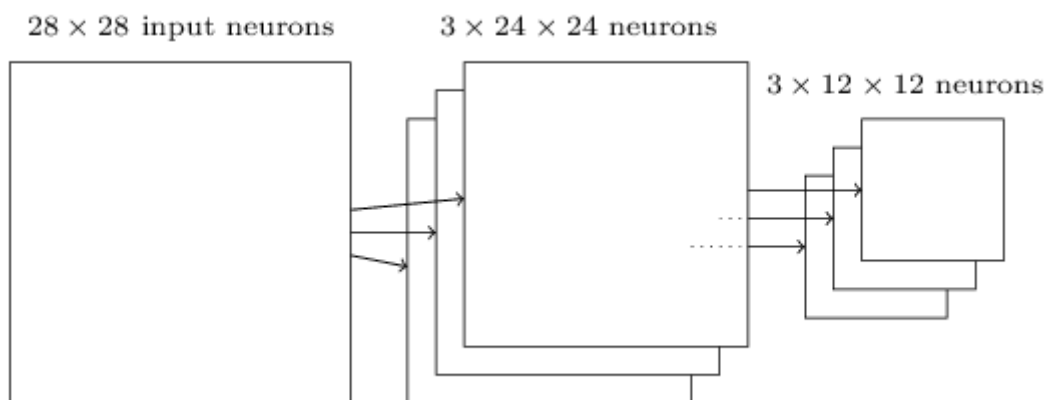


Prende in ingresso le feature maps che escono dallo strato convoluzionale e preparano una *condensed feature map*.

Per esempio possiamo dire che lo strato pooling potrebbe riassumere, in ogni sua unità, una regione  $2 \times 2$  di neuroni dello strato precedente: questa tecnica è chiamata *max pooling* e può essere riassunta col seguente schema:

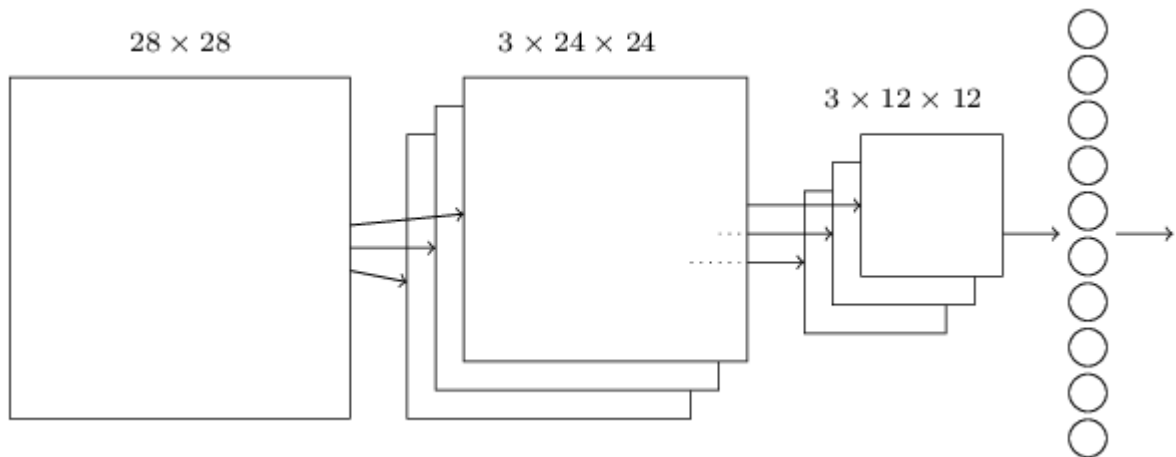


Ovviamente di solito abbiamo più feature maps e applicheremo il max pooling ad ognuna di esse separatamente:



Quindi se abbiamo, come nell'esempio, 3 feature maps di dimensione  $24 \times 24$ , ne otterremo altrettante di dimensione  $12 \times 12$ , dato che stiamo assumendo che per ogni unità riassumiamo una regione  $2 \times 2$ .

Andando a combinare insieme queste tre idee formeremo una rete convoluzionale completa: la sua architettura può essere vista nella seguente immagine:



Ci sono i  $28 \times 28$  neuroni di input seguiti da uno strato convoluzionale con un local receptive field  $5 \times 5$  e 3 feature maps. Otterremo come risultato uno strato nascosto di  $3 \times 24 \times 24$  neuroni. C'è poi il max-pooling applicato a regioni  $2 \times 2$  sulle 3 mappe di feature ottenendo uno strato nascosto  $3 \times 12 \times 12$ .

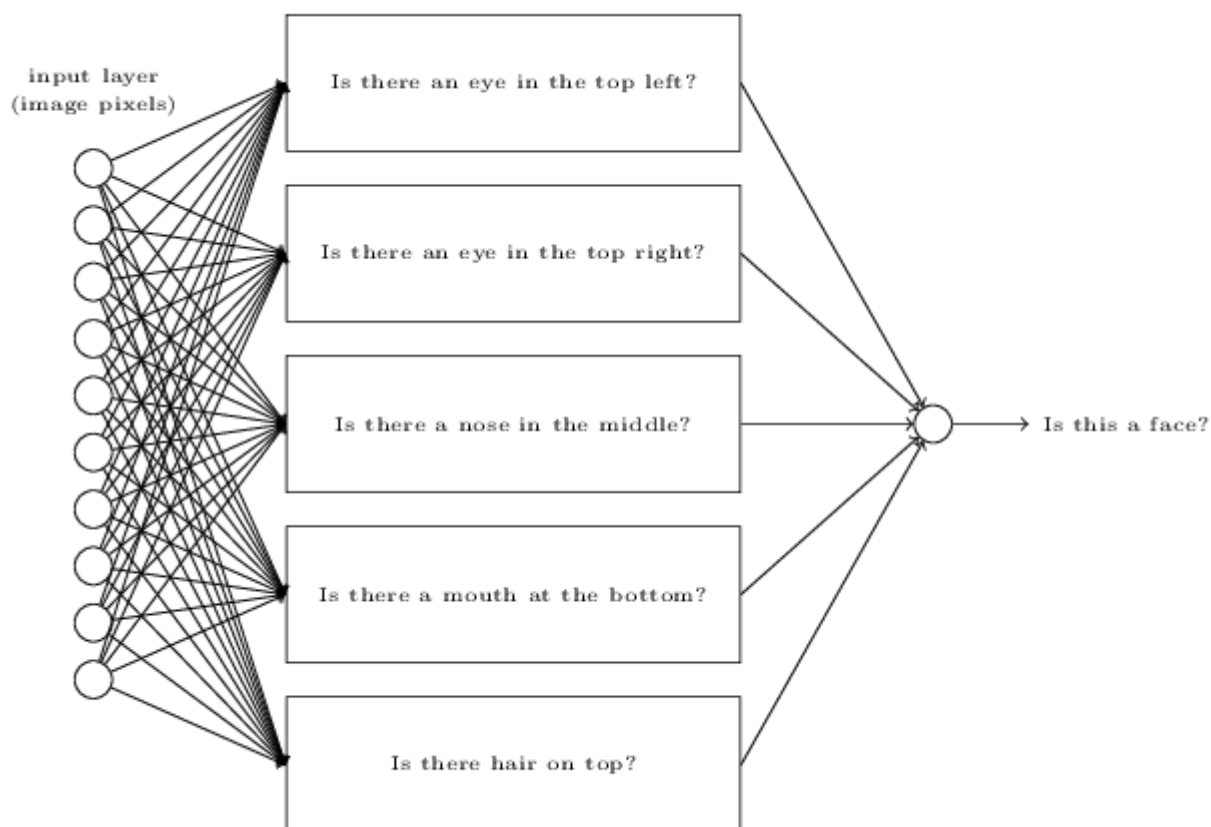
L'ultimo strato è totalmente connesso: collega quindi tutti i neuroni dallo strato di max-pooling a tutti i 10 neuroni di uscita, utili a riconoscere l'output corrispondente.

Questa rete sarà addestrata tramite la discesa del gradiente e l'algoritmo di backpropagation.

Proviamo ora a risolvere, con le reti convoluzionali un problema leggermente diverso rispetto a quello affrontato fino ad ora, cioè quello di riconoscere la presenza o meno di un volto in una foto. Possiamo provare a decomporre il problema in piccoli sotto problemi come ad esempio: c'è un occhio in alto a destra? C'è un occhio in alto a sinistra? Ci sono dei capelli in alto? ...e tante altre domande simili. Ottenendo quindi risposte come "sì" o "probabilmente sì" riusciremo a dedurre se c'è una faccia oppure no. Ovviamente ci sono dei deficit in questo sistema

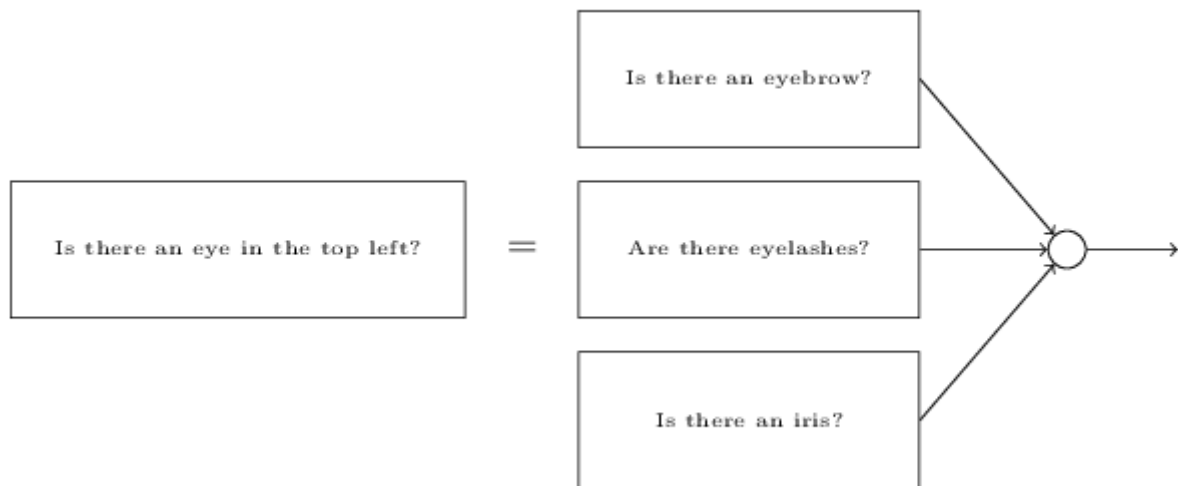
dato che la persona potrebbe essere calva oppure potrebbe essere girata, permettendoci di vedere solo una parte del suo volto.

Possiamo risolvere questi sotto problemi utilizzando delle reti neurali, con un'architettura simile a quella nella figura seguente:





A loro volta i sotto problemi ora analizzati possono essere scissi in altri sotto problemi più piccoli: la domanda “c'è un occhio in alto a sinistra?” può essere decomposta in: “ci sono ciglia?”, “ci sono sopracciglia?”, “c'è un'iride?” andando quindi a formare uno schema simile:



Anche queste domande possono essere divise in altre piccole domande, andando quindi a formare sempre più strati: arriveremo al punto in cui i vari substrati della rete risponderanno a domande molto semplici, che possono essere verificate a livello dei singoli pixel.

Il risultato finale sarà dunque una rete che risponde ad una domanda molto complicata, andando però a farlo tramite domande molto più semplici, e utilizzando quindi reti formate da più strati nascosti (tra 5 e 10).

L'utilizzo di queste reti porta a performance migliori se paragonate alle reti con meno hidden layers, dato che queste reti profonde riescono a creare una complessa gerarchia di concetti.