

Teoria Reti neurali

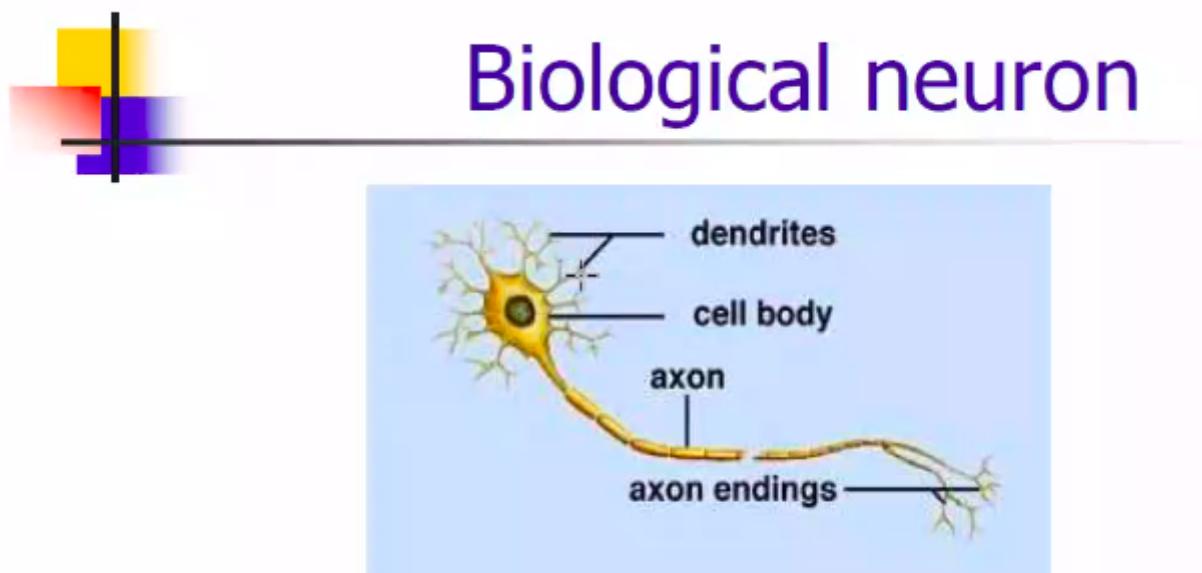
04/02/2023

Infos

- Docente: Beatrice Lazzerini
- Mail: beatrice.lazzerini@unipi.it
- per colmare il gap tra la parte teorica e pratica, è possibile scegliere di fare una presentazione sintetica su un argomento di vostro interesse, una presentazione power point, oppure appunti su un documento word, la durata della presentazione deve essere massimo di 5 minuti

Introduzione

Le reti neurali sono nate dal desiderio di riprodurre in modo artificiale le capacità del cervello umano nel risolvere problemi molto complessi. Ad esempio problemi di riconoscimento di immagini, traduzione da un linguaggio, predizione, tutti questi problemi risolti rendendo il calcolatore in grado di imparare così come fa il nostro cervello. Per ottenere una cosa di questo tipo, il passo da fare è costruire un modello della struttura a basso livello del cervello umano. I ricercatori allo scopo di costruire le reti neurali, hanno iniziato a studiare il cervello umano costruendone un **modello** per poter identificare le caratteristiche di quella realtà omettendo certi dettagli. Per capire le parti che ci interessa riprodurre, facciamo riferimento ad un neurone biologico come raffigurato qui:



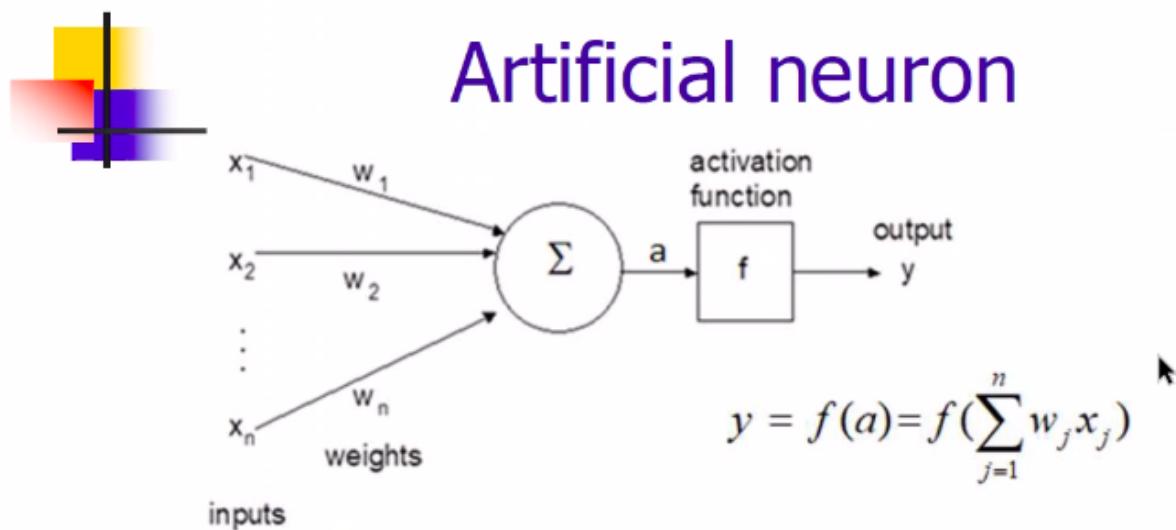
A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

e riceve input da altri neuroni attraverso i canali di ingresso che sono i **dendriti**. Questi ingressi si sommano all'interno del corpo cellulare del neurone biologico. Quando questa somma supera una certa soglia, il neurone in questione si attiva ed emette un impulso elettrico che si propaga attraverso un canale di uscita lungo che ha delle diramazioni alla fine e trasmette il segnale ad un altro neurone a valle di quello considerato, interessando altrettanti dendriti. La particolarità che va spiegata per capire cosa vogliamo riprodurre in modo artificiale. Si può pensare che ciascuna terminazione di un **assone** vada a toccare un dendrite di un neurone. I realtà non succede questo perchè esiste un piccolo gap tra la fine del ramo dell'assone e l'inizio del dendrite di uno dei neuroni a valle. Questo punto di contatto tra l'assone del neurone che emette il segnale e il dendrite dell'altro neurone è detto **sinapsi**. Come fa il segnale elettrico che viaggia lungo l'assone a passare oltre la sinapsi? C'è un processo elettrochimico, ovvero il segnale elettrico viene trasformato in un segnale chimico e poi il segnale chimico viene ri-trasformato in un segnale elettrico sul dendrita del neurone. In termini intuitivi significa che l'assone che trasmette il segnale rilascia delle sostanze chimiche ovvero i neurotrasmettitori che si legano a dei recettori nel neurone target a valle. Quindi tramite questo processo elettrochimico si può avere un influenza sul potenziale elettrico all'interno del corpo del neurone ricevente, ovvero il segnale ricevente può essere incrementato o decrementato, ovvero si può creare una differenza tra il segnale che viene trasmesso dal neurone e il neurone che lo riceve. Queste connessioni sinaptiche possono influire sul segnale che viene ricevuto e quindi sul comportamento dei neuroni a valle. Alterando la forza di queste connessioni sinaptiche, il nostro cervello impara.

Neuroni artificiali

La rete neurale è una rete di neuroni artificiali.



- An artificial neuron has many inputs but only one output.
- Each input connection has an associated adjustable *weight* (a real number). A positive (negative) weight has an excitatory (inhibitory) effect on the neuron.
- A neuron computes some function f (called *activation function*) of the weighted sum of its inputs.
- The output can be a real number, a real number restricted to some interval (e.g., $[0,1]$ or $[-1,1]$), a discrete number (e.g., $\{0, 1\}$ or $\{+1, -1\}$).

Un neurone artificiale ha molti ingressi ma un solo output. Gli ingressi nel neurone biologico sono approssimativamente sommati tra loro, quindi mi aspetto nel modello artificiale di vedere una somma di segnali di ingresso, ma i segnali di ingresso non sono ricevuti esattamente così come vengono

trasmessi in quanto vengono alterati dai neurotrasmettitori presenti a livello delle sinapsi possono variare l'intensità dei segnali, quindi si associano dei **numeri reali** (**weights**) ciascuno dei quali se è positivo incrementa il valore ricevuto in ingresso sul corrispondente ingresso prodotto da un altro neurone, se è negativo invece, il segnale che partecipa alla sommatoria sarà decrementato rispettato all'originale. $x_1 \dots x_n$ possono essere input ricevuti dal neurone a monte oppure ricevuti dall'ambiente esterno, in quanto ci vorranno uno o più neuroni che permettono di presentare un ingresso al nostro neurone. Il peso è aggiustabile, in quanto così come esiste il contributo delle sinapsi, nel neurone artificiale, i pesi saranno i parametri usati per modificare il processo di apprendimento. Il neurone artificiale quindi calcola la somma pesata degli ingressi, e quando questa somma **supera** una certa soglia (così come funziona all'interno del cervello umano) il neurone si attiva. Questo fenomeno di attivazione viene modellato con una funzione di attivazione (a soglia ad esempio) che prende come ingresso a e che emette in uscita (un numero) che dipende dal valore della somma pesata degli ingressi rispetto ad una soglia. Questa funzione di attivazione f , non sarà necessariamente una funzione a soglia come vedremo, ma è calcolata dal neurone allo scopo di produrre un'uscita da trasmettere ad altri neuroni.

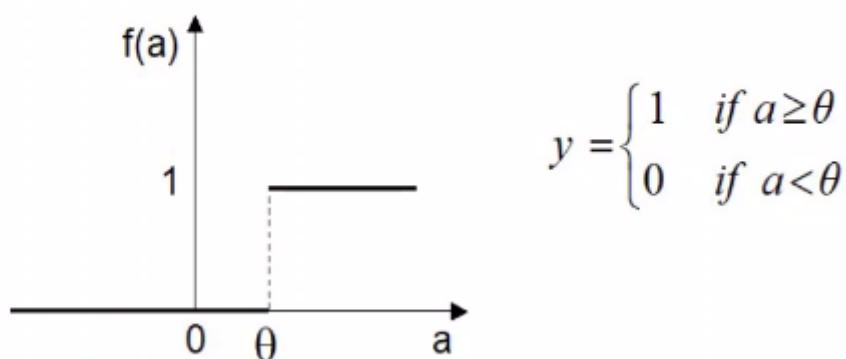
❓ Che cosa è l'uscita?

L'uscita è un numero, ma può essere un qualunque numero reale ristretto all'interno di un certo intervallo, dipende dal problema, da ciò che vogliamo realizzare, oppure può essere un numero discreto $[0,1]$ ovvero l'insieme dei numeri 0 o 1. Dipenderà dal tipo di problema che vogliamo risolvere e dal tipo di uscita che vogliamo ottenere.

❓ Come può essere fatta una funzione di attivazione?

Se pensiamo a questa funzione di attivazione come la funzione binaria a soglia:

■ Binary threshold function



ovvero emetto un 1 se a maggiore o uguale di una certa soglia θ , a sarà la media pesata degli ingressi. Pensando intuitivamente a quello che abbiamo descritto come neurone biologico, se la somma pesata degli ingressi supera una certa soglia allora viene emesso un impulso.

Però...

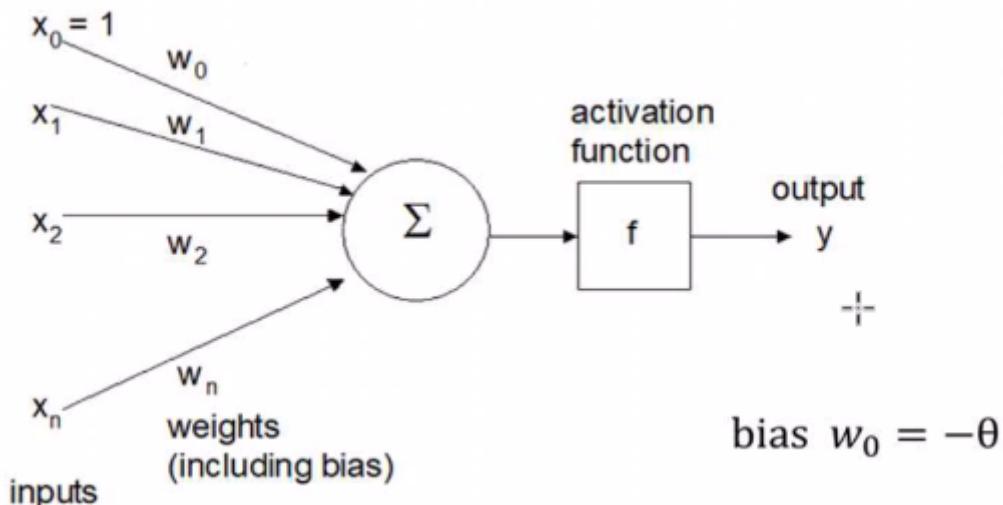
Potremmo dare 1 ad esempio se $a - \theta \geq 0$, e 0 se $a - \theta \leq 0$, portando θ al primo membro, ovvero considerare θ e sottrarla dalla somma pesata degli ingressi: ovvero l'uscita del neurone è pari al risultato dell'applicazione della funzione f alla somma che non è di n numeri, ma $n+1$. Il $-\theta$ potrei interpretarlo come un peso per un ingresso che vale sempre 1? SI

- Typically, the *threshold* is subtracted from the weighted sum of the inputs

$$y = f\left(\sum_{j=1}^n w_j x_j + \theta\right) = f\left(\sum_{j=0}^n w_j x_j\right)$$

and the negative of the threshold (called *bias*) is considered as a weight connected to a unit that always has an output of 1 ($x_0 = 1$).

$-\theta$ è come dire 1 moltiplicato per $-\theta$. 1 è il valore di x_0 , ovvero il negativo della soglia è considerato come un peso. Infatti nello schema in basso abbiamo inserito un ulteriore ingresso:



Qui al posto di n ingressi ne ho inserito uno in più, x_0 è un ingresso fittizio, nessun neurone produce x_0 perché è sempre uguale a 1, ma ho preso w_0 (detto **bias**) che è l'opposto della soglia. Risulta importante trattare le cose in questo modo, perchè avevamo anticipato che i pesi possono essere aggiustati, sono oggetti del processo di apprendimento, quindi dovremo agire proprio sui pesi, quindi se la soglia la lascio a destra del ≥ 0 la devo settare io, ma se la considero come ulteriore peso allora **la soglia la posso apprendere!** Vedremo anche da applicazioni pratiche che aggiungere questo bias in questo modo, ci permette di dare una maggiore capacità di modellizzazione di questa rete, replicando funzioni complesse.

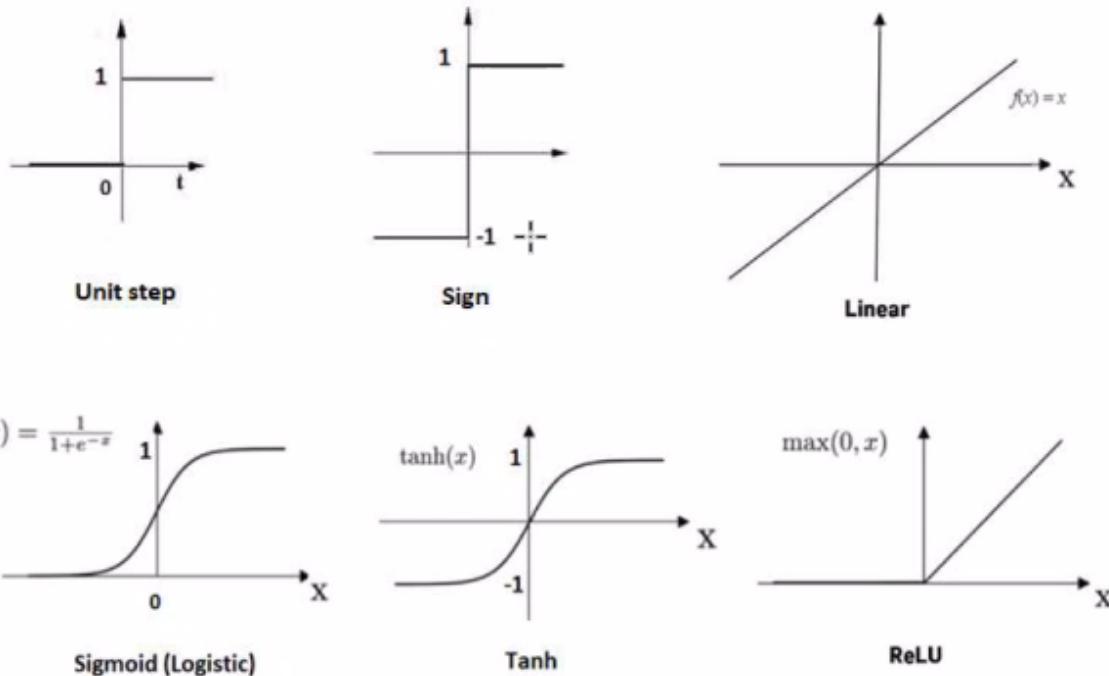
Inoltre si può interpretare $-\theta$ come $(-1)^* \theta$, quindi si impara la soglia e non il bias. Nella letteratura sulle reti neurali ci sono diverse scelte di simboli, di termini per riferirsi alle stesse cose, quindi nell'slide si è cercato non di usare un unico insieme di simboli, ma di volta in volta vari simboli per riferirsi alla

stessa cosa. La somma pesata degli ingressi infatti, indicata con net_i , si può trovare anche come net input (net_i).

Il modello di neurone visto è quello più usato, riuscendo a risolvere diversi problemi. Una possibile funzione di attivazione, quella più intuitiva, è la funzione binaria a soglia.

Funzioni di attivazione

Ne esistono tante, ed ha un significato particolare legato al problema. La funzione binaria potrebbe non bastare perchè ci serve una funzione che sia in grado di dare in uscita 1 o -1:



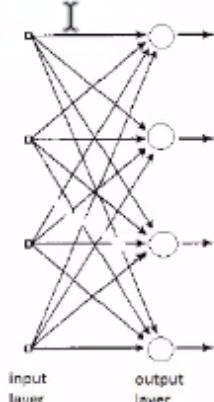
Se mi interessasse avere tutti i valori tra 0 e 1 non potrei usare questa funzione, ma usare una funzione continua. La **funzione logistica** mostrata nell'immagine ad esempio restituisce un numero tra 0 e 1. In certi casi vorremmo avere una funzione continua e derivabile pur mantenendo come risultati di uscita 0 o 1 e dunque se ho bisogno di una approssimazione abbastanza fedele della funzione a gradino, mi basta moltiplicare l'esponente della e per una costante, e faccio crescere il valore della costante, la funzione logistica si avvicina a quella a gradino. Ovvero la funzione logistica è una funzione soglia più "morbida".

La funzione logistica è tipicamente chiamata ma anche sigmoide avendo una forma ad S, se vogliamo generalizzare la funzione segno così come la funzione logistica è una versione smooth della gradino, dovrebbero usare la **Tanh** (tangente iperbolica) che è un'approssimazione smooth della segno. Un'altra funzione di attivazione è la **ReLU**, introdotta per facilitare l'addestramento di una rete neurale, rappresenta il massimo tra 0 e x .

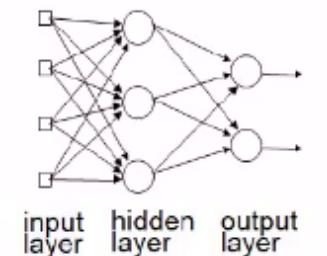
Network topology

There are many types of NNs depending on how neurons are connected to each other:

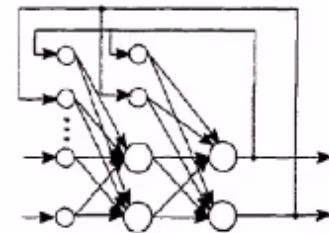
- neurons are arranged in an ordered hierarchy of layers in which connections are only allowed between neurons in immediately adjacent layers,
- connections back from later to earlier neurons are allowed,
- each neuron can connect to every other neuron,
- ...



one-layer feedforward network



two-layer feedforward network



Recurrent network

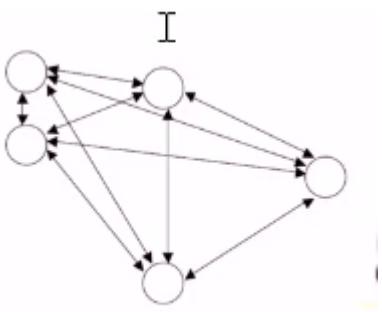
I neuroni sono organizzati in una gerarchia ordinata di livelli in cui le connessioni sono permesse solo tra neuroni immediatamente adiacenti. In questa rete, gli **input layer** sono i neuroni di ingresso, in questo caso (il primo esempio a sinistra) abbiamo una gerarchia di 2 livelli. I neuroni di uscita vengono rappresentati con dei cerchi mentre quelli di ingresso con dei quadrati. Questo perchè i neuroni di uscita calcolano qualcosa quindi calcolano il valore della funzione di attivazione che riceve come argomento la somma pesata degli ingressi. I neuroni di ingresso invece non eseguono alcuna elaborazione servono solo a far entrare nella rete i segnali provenienti dell'ambiente esterno. Per questo motivo, di solito ma non sempre, si usa un simbolo diverso per rappresentare i neuroni di input rispetto a quelli di output.

Nella rete in mezzo abbiamo i neuroni di ingresso e subito dopo un livello nascosto in quanto effettuano una loro elaborazione ma non hanno alcun collegamento né con l'ingresso né con l'uscita. Possono essere 0 più di uno.

Le 3 tipologie di architetture viste nell'immagine sono:

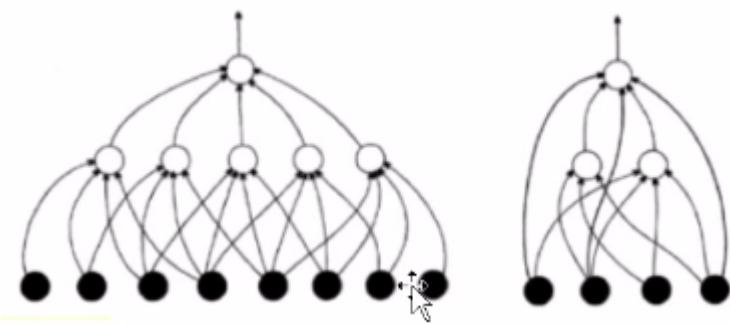
- one-layer feedforward network:** ovvero il livello di ingresso non viene considerato nel conteggio dei livelli (ma questa interpretazione varia a seconda del gruppo di ricercatori)
- two-layer feedforward network:** ha 3 livelli per chi considera l'ingresso e 2 invece per chi non considera i neuroni di ingresso
- recurrent network:** in cui il risultato della elaborazione della rete viene ridato in pasto alla rete stessa. Se contengono almeno un loop, ovvero una connessione da uno o più neuroni successivi verso quella precedente.

Se alcune connessioni mancano la rete è detta parzialmente connessa. Esistono altre possibilità, ad esempio un neurone può essere connesso ad altri neuroni ma non in modo ordinato:



rete composta da 5 neuroni ciascuno connesso a tutti gli altri neuroni, si tratta di una rete fully connected ma non layered, ovvero non sono organizzate come una gerarchia di livelli. Queste reti possono essere usate come una memoria associativa. Una memoria è un insieme di celle che contengono un informazione, un elenco telefonico è una memoria associativa perché specifichiamo una parte del contenuto per poter referenziare la cella che vogliamo.

Queste invece sono fatte da un primo strato di neuroni di input, si trattano di reti stratificate, ma la prima non è fully connected, mentre la seconda rete non è layered in quanto i neuroni di ingresso sono connessi sia allo strato successivo che allo stato finale:



? Come è possibile stabilire il numero di neuroni ingresso, nei middle layers e in uscita? Dipende dal problema! I neuroni di ingresso saranno tanti quanti sono gli attributi ovvero le variabili misurate. L'oggetto in ingresso può infatti essere rappresentato con un numero variabile di attributi, ovvero una componente di quel vettore che rappresenta l'elemento in ingresso. Il numero di neuroni in uscita dipende da come vogliamo rappresentare il risultato, in base al problema da risolvere.

? Perchè introdurre i livelli nascosti?

Perchè gli strati nascosti, per mappare degli ingressi in uscite, la rete a volte, quando la funzione tra ingresso e uscita è complessa, ha bisogno di farsi delle rappresentazioni interne non visibili che le permettono di passare ad uno strato all'altro in modo graduale più facilmente.

Quando diciamo che una rete approssima una funzione ovvero mappa degli ingressi nelle corrispondenti uscite, non dico che la rete guarda l'espressione analitica, non conosce l'espressione analitica ovvero la rete ha bisogno di farsi delle rappresentazioni interne che le permettono di ricostruire con l'accuratezza voluta il mapping tra ingressi e uscite. La rete impara a fare il mapping a partire da esempi che le forniamo. Rappresentiamo la funzione non in forma analitica (**in tal caso non avremmo bisogno di una rete neurale!!!**), invece abbiamo degli esempi di corretto mapping tra ingresso e uscita e quindi addestriamo la rete con gli esempi che abbiamo.

Training

Il paradigma di apprendimento più usato è il **supervised learning** signfiica che esiste un esperto del campo applicativo che prepara degli esempi di addestramento, ogni esempio di addestramento è una coppia, dove il primo elemento è l'ingresso e il secondo elemento è l'uscita desiderata. Avremo pertanto un *training set* fatto da questi esempi, alla rete si presenta una coppia scelta dal training set, la rete darà una risposta deterministica in funzione del valore attuale dei pesi.

All'inizio dell'addestramento una rete ha diversi pesi su ciascuna connessione. All'inizio viene definita sia la topologia che la funzione di attivazione di ogni neurone, quindi usando i valori dati in ingresso ai neuroni e moltiplicando questi valori per i pesi inizialmente casuali associati alle connessioni, passiamo questa somma pesata alla funzione di attivazione e otteremo l'uscita e così via fino all'uscita dai neuroni di uscita.

⚠️ *L'algoritmo di addestramento modifica i pesi con lo scopo di minimizzare l'errore.* ⚡️

1. L'errore è la differenza tra la risposta desiderata (ovvero il secondo elemento della coppia di addestramento) e la risposta effettiva che è il valore ottenuto in base alla topologia scelta, ai pesi attuali e alle scelte fatte. Fin quando non si presenta alla rete tutto l'insieme di addestramento.
2. A quel punto si valuta il comportamento della rete, ovvero l'errore fatto su tutti gli esempi di addestramento. Può essere la media degli errori, la somma, ovvero quello che decidiamo di usare come bontà dell'errore fatto dalla rete su tutto il training set.
3. Se siamo soddisfatti signfica che la rete ha imparato. Altrimenti il training viene ripetuto.

L'ordine con cui vengono presentati gli esempi è casuale, quindi in genere gli esempi vengono presentati in ordine diverso, quindi alla fine del processo di apprendimento, possiamo dire che la rete impara da questi esempi di addestramento a costruire un mapping tra input e output.

Questa cosa vuol dire che la rete impara dall'insieme di addestramento, impara da quegli esempi quale funzione applicare, quindi vuol dire che a quel punto è *come se la rete avesse imparato la funzione che lega gli input agli output*. Ovvero presentando un input nuovo alla rete diverso da tutti quelli precedenti, la rete saprà darmi la risposta corretta.

Delta Rule

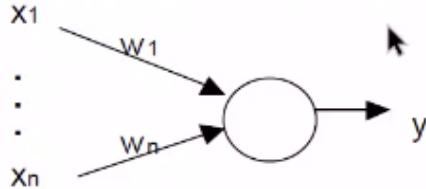
❓ In termini implementativi come si fa ad addestrare?

Un esempio di regola che usa l'errore fatto dalla rete per correggere il comportamento stesso della rete è la [Delta Rule](#):



Delta rule

- An example of an error correction rule is the *delta rule* (also called *Widrow-Hoff rule*).



- The delta rule states that the adjustment to be made is

$$\Delta w_i = \eta \delta x_i \quad \text{with} \quad \delta = t - y \quad (\text{error})$$

where t is the target output, y is the actual output and the real number η is the *learning rate*.

$$\text{Therefore } w_i(n+1) = w_i(n) + \Delta w_i(n)$$

- The original delta rule is used for one-layer neural networks with linear output neurons.

L'addestramento insegna alla rete come comportarsi al meglio ma può agire sui pesi in quanto tutto il resto ovvero la architettura della rete, le connessioni, la specifica funzione di attivazione sono state scelte e non possono essere modificate.

Stabilisce che l'aggiustamento che deve essere fatto ad ognuno dei pesi, viene modificato sommandogli una quantità Δ_{wi} pari al prodotto di questi elementi:

- η è una costante di proporzionalità reale tra 0 e 1 ed è detto **learning rate**, ovvero la velocità di apprendimento.
- δ è la differenza tra t e y . y è l'uscita effettiva del neurone mentre t è il **target**, l'uscita desiderata, ovvero il secondo elemento dell'esempio di addestramento (in quanto ogni esempio di addestramento è una coppia costituita dall'ingresso alla rete, in questo caso un vettore di n componenti, e dall'uscita desiderata t).

Quindi l'aggiustamento di ogni peso è proporzionale all'errore fatto dal neurone a cui si riferisce quel peso moltiplicato il valore dell'ingresso associato alla connessione di quel peso.

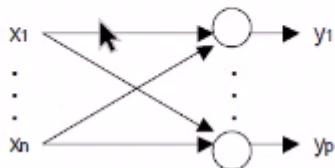
Il peso i -esimo all'istante $n+1$ ovvero quando presento l' $n+1$ esempio di addestramento è dato dal valore che quel peso aveva quando ho presentato l'esempio precedente più una modifica dettata da questa Delta Rule.

La delta rule stabilisce questa formula, potremmo prenderla per assodata, ma se riuscissimo a convincerci della validità della formula sarebbe meglio.

Se considero una rete a singolo livello, con n ingressi e un livello di uscita che non è costituito da un solo neurone di uscita ma da p neuroni di uscita, qual'è l'errore che viene commesso dalla rete quando le presentiamo un singolo esempio di addestramento?

L'errore deve esser calcolato per tutti i neuroni di uscita. Se presento il k-esimo esempio di addestramento ho un errore E_k che tiene conto della differenza tra l'uscita desiderata del neurone t_j meno l'uscita effettiva del neurone j (y_j) al quadrato. Sommando su tutti i neuroni di uscita, ottengo la somma degli errori commessi dai neuroni di uscita di questa rete, quindi l'errore sulla k-esima coppia di addestramento. 1/2 è stato inserito solo per semplificare le operazioni matematiche in quanto mi troverò a calcolare la derivata e ottenendo 2 si semplificherà con il 1/2.

- Consider the following network



- t_j = target (or desired) output
- y_j = actual output

- Error for the k -th training sample

$$E_k = \frac{1}{2} \sum_{j=1}^p (t_j - y_j)^2$$

- Overall error on m training samples

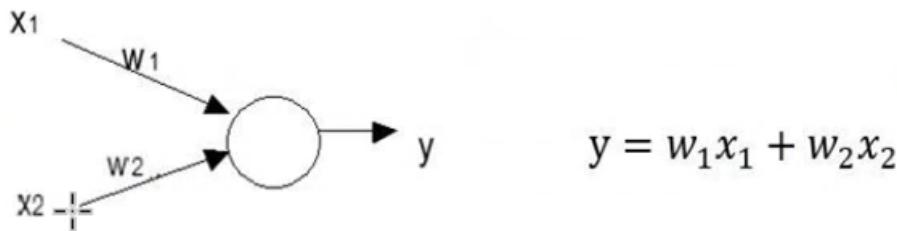
$$E = \sum_{k=1}^m E_k$$

Questo è l'errore commesso da una rete su un certo esempio di addestramento. Una rete però si deve comportare bene su tutti gli esempi di addestramento quindi quello totale sarà la somma di tutti i k-esimi errori.

Quello che vogliamo verificare in modo intuitivo è che la Delta Rule aiuta effettivamente la rete ad imparare.

Piuttosto che dimostrare quanto detto in termini intuitivi, semplifichiamo le cose, perchè sarà facile generalizzare.

Esempio:



- For a linear neuron (i.e., $f(a) = a$) and a single sample the error is

$$E = \frac{1}{2}(t-a)^2$$

- Expanding gives

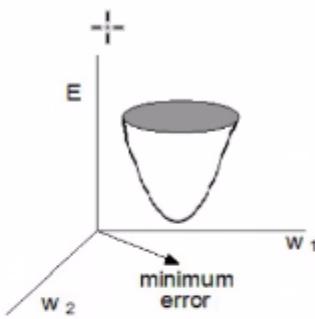
$$\begin{aligned} E &= \frac{1}{2}[t^2 - 2ta + a^2] = \\ &= \frac{1}{2}[t^2 - 2t(x_1w_1 + x_2w_2) + x_1^2w_1^2 + 2x_1x_2w_1w_2 + x_2^2w_2^2] \end{aligned}$$

Gli ingressi invece che essere n sono 2, solo 2 pesi, le uscite invece che essere p è 1. Non consideriamo il bias.

Se questo neurone è dotato di una funzione lineare $f(a) = a$ e se l'insieme dei campioni di addestramento è costituito da un solo campione, l'errore secondo la formula vista prima:

$$E_k = \frac{1}{2} \sum_{j=1}^p (t_j - y_j)^2$$

mi restituisce un paraboloide nello spazio dei pesi.



- Before training, the weights start at some random values and therefore the initial state of the network could be anywhere on the error surface.
- During training, the weights are adjusted in a direction towards a lower overall error (i.e., in the direction of the steepest descent of the error surface).

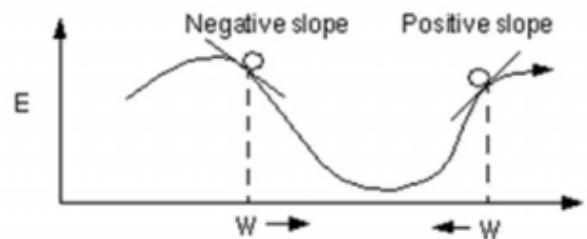
Quello che vogliamo visualizzare graficamente è la superficie dell'errore nello spazio dei pesi. Ovvero che modificando i pesi con la delta rule effettivamente lo stato della rete cambia in modo migliorativo fino a raggiungere l'errore minimo.

All'inizio dell'addestramento i pesi sono messi in modo casuale, ovvero se immagino la rete come una pallina che si trova in qualche punto del paraboloide, significa che lo stato della rete sarà in un determinato punto del paraboloide. Durante il training voglio che i pesi siano aggiustati in modo che la pallina sia spostata verso l'opposto della massima crescita dell'errore, ovvero verso il minimo errore. Se dimostrassi che la delta rule fa questo, posso dire che funziona. Per farlo dobbiamo mostrare che può essere espressa come:

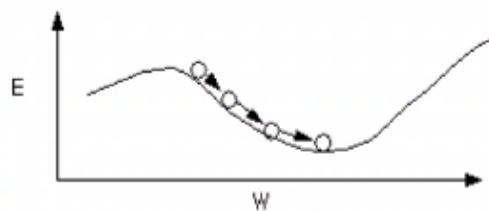
To prove the convergence of the delta rule, we just need to show that it can be expressed as

$$\Delta w_{ij} = -G = -\frac{\partial E}{\partial w_{ij}}$$

Slope of E positive => decrease W
Slope of E negative => increase W



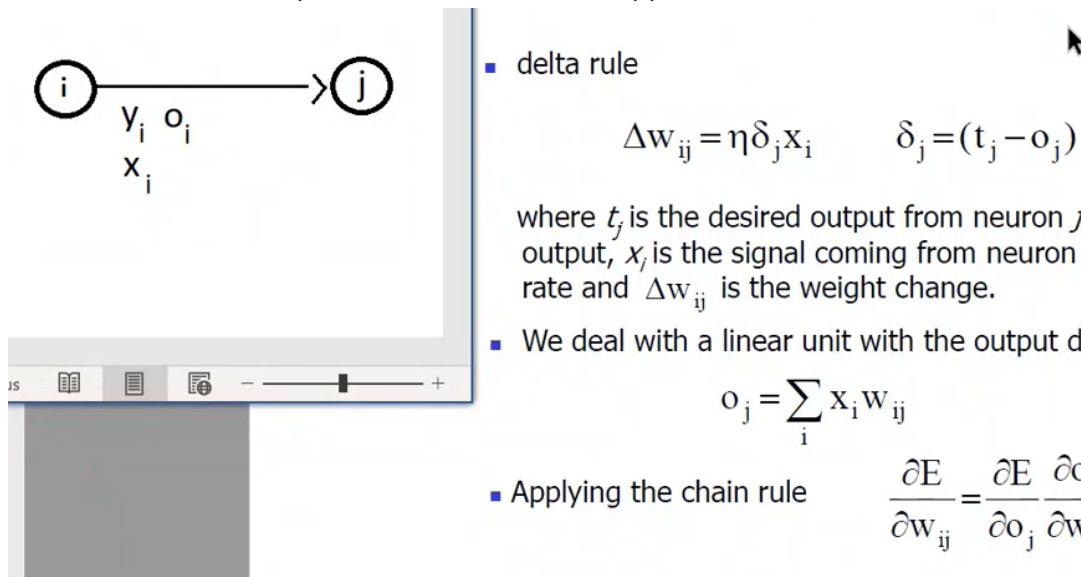
By repeating this over and over, we move "downhill" in E until we reach a minimum ($G = 0$)



ovvero che la variazione del peso è l'opposto del gradiente, ovvero l'opposto della derivata parziale dell'errore rispetto a quel peso. Infatti volendo scendere verso il minimo di una funzione bisogna andare nella direzione opposta alla derivata di quella funzione rispetto alla variabile, in quanto la derivata mi restituisce la direzione di massima crescita. Avendo più variabili uso il gradiente. Mi basta quindi dimostrare che la delta rule va nella direzione opposta al gradiente.

⚠ La notazione con Δw_{ij} è dovuta al fatto che spesso i ricercatori usano considerare coppie di neuroni di ingresso e indicare il peso associato alla loro connessione con questa simbologia:

La delta rule è stata ripetuta con la notazione appena introdotta.



Avendo a che fare con un unità lineare, l'uscita prodotta dal neurone j (o_j e y_j sono sinonimi) è la somma pesata degli ingressi. Vogliamo far vedere che la derivata parziale di E rispetto a w_{ij} è opposta rispetto alla delta rule.

La calcoliamo applicando la regola della catena,

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}} \\ \frac{\partial E}{\partial o_j} &= -\delta_j \quad \frac{\partial o_j}{\partial w_{ij}} = x_i \\ -\frac{\partial E}{\partial w_{ij}} &= \delta_j x_i \end{aligned}$$

- By inserting the learning rate, the formula for the delta rule is obtained:

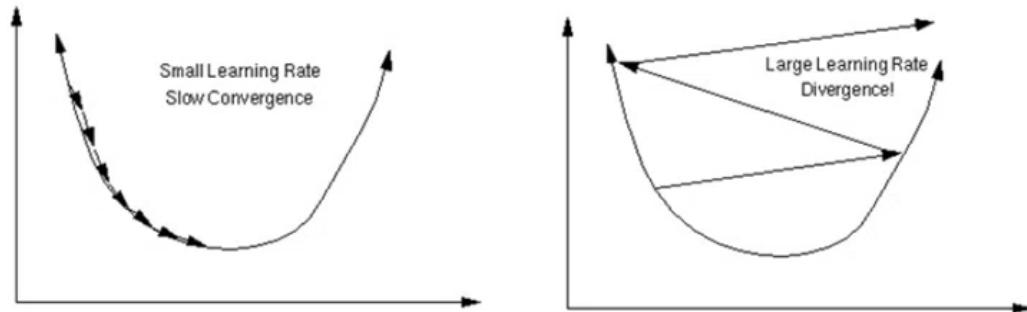
$$\Delta w_{ij} = \eta \delta_j x_i$$

Basta inserire η che è una costante (nell'esempio di calcolo è 1) per dimostrare che la delta rule modifica i pesi in modo opposto al gradiente, quindi applicando la delta rule, faccio un passo verso il basso ogni volta fino a che non raggiungo l'errore minimo.

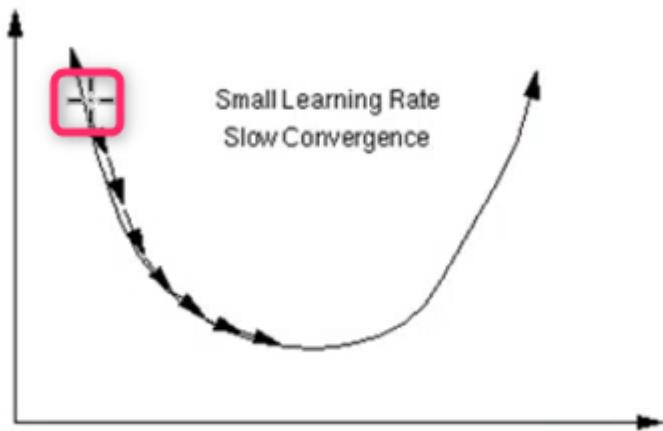
Il learning rate che è un valore tra 0 e 1 deve essere scelto in modo appropriato in quanto non può essere troppo basso:

The learning rate

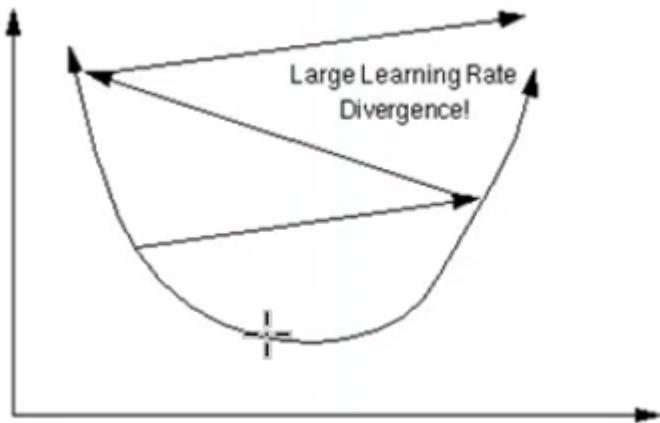
- The learning rate cannot be too low (to avoid too long training times) nor too high (to avoid oscillations around the minimum).
- Typically, the learning rate is chosen experimentally; it can also vary over time, getting smaller as the training algorithm progresses.



infatti partendo dal punto indicato in rosso:



in seguito alla generazione casuale dei pesi ci vorranno molti passi per convergere verso un errore minimo. Con un learning rate troppo alto potrei invece in modo opposto arrivare a vedere il minimo ma potrei iniziare a oscillare:



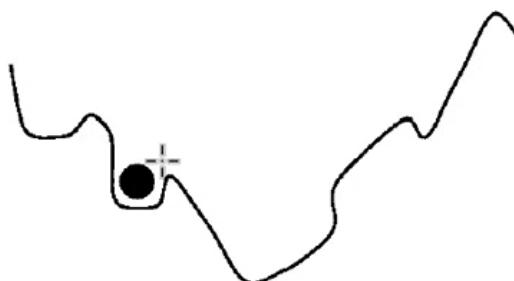
Operativamente per scegliere il valore dell' η si potrebbero fare diversi esperimento trovando il valore più adatto oppure **partire con un η alto e poi man mano che mi avvicino al fondo della valle, iniziare a diminuirlo. Ovvero variare nel tempo il valore di η diminuendolo man mano che l'algoritmo di addestramento si avvicina al minimo.**

Le figure appena viste fotografano una rete a singolo strato con neuroni di uscita a funzione lineare, ma spesso in applicazioni reali si usano reti più complesse, con più strati nascosti, con funzione di attivazione che non sono lineari, in cui la superficie dell'errore non ha un unico minimo globale con forma a paraboloida vista in precedenza ma si può rappresentare con **più minimi locali**:



Local minima

- For real-world problems, the error surfaces can have several *local minima*, and, during training, the network can get trapped in one of these minima.



ed il problema può essere che la "pallina" che rappresenta lo stato dei pesi può rimanere intrappolata in uno dei minimi locali. Lo scopo ultimo di una rete neurale è quello di addestrarla ovvero che la configurazione dei pesi è costituita alla fine del processo di addestramento. Dunque se la rete rimane intrappolata in un minimo locale vuol dire che la rete neurale ha una configurazione dei pesi che coincide con quello di un minimo locale ma non è il migliore minimo locale, ovvero la rete non ha imparato al meglio.

Per poter uscire dai minimi locali bisogna ricorrere all'utilizzo del **Momentum**:



Momentum

One technique that can help the network get out of local minima is the use of a *momentum* term. With momentum m , the weight update at a given time becomes

$$\Delta w_{ij}(n+1) = \eta \delta_j x_i + m \Delta w_{ij}(n)$$

where $0 < m < 1$ is a new global parameter that must be determined by trial and error.



Ovvero quando si aggiorna il valore di un peso w_{ij} , applichiamo una formula identica sintatticamente alla delta rule in cui si inserisce un valore m che è una costante tra 0 e 1 moltiplicato per l'aggiornamento che ho fatto allo stesso peso nel passo precedente. Se aggiorno solo applicando la generalizzazione della delta rule, faccio un passo verso il basso influenzato solo dal learning rate, ma non è detto che possa raggiungere il minimo globale. Ma se uso il momentum, aumento la velocità con cui discendo la curva in quanto sommo al learning rate un termine che ha lo stesso segno dunque che va nella stessa direzione in cui stavo andando prima, ovvero i passi che la pallina fa sono fatti in modo più veloce così da riuscire a raggiungere il minimo globale.

$$\Delta w_{ij}(n+1) = \eta \delta_j x_i + m \Delta w_{ij}(n)$$

Il delta della formula qui presentata non è la differenza solo la differenza tra l'uscita effettiva e quella desiderata, ma è più complessa. Ovvero si tratta di una **delta rule generalizzata**.

Learning Algorithms

Per completare questo insieme di concetti vediamo quali sono gli algoritmi di apprendimento. Abbiamo capito che per addestrare una rete bisogna presentarle tutti gli esempi di addestramento, si può usare da un punto di vista pratico:

- **online learning**: i pesi vengono aggiornati immediatamente dopo aver presentato ogni esempio di addestramento. Dunque effettuo una valutazione dell'errore e aggiorno i pesi in funzione del valore della valutazione ad esempio applicando la delta rule oppure quella generalizzata. I vantaggi sono quelli di poter addestrare la rete con un flusso continuo di dati di addestramento. Inoltre aggiornando subito i pesi, richiede meno spazio di memoria, in quanto non devo mantenere le variazioni dei pesi in memoria. Infine se cambio i pesi di volta in volta, a seconda dell'esempio di addestramento, l'aggiornamento che quell'esempio di addestramento posso interpretarlo come un

approssimazione del gradiente globale (dell'aggiornamento globale) e dunque potrebbe permettermi ad esempio di superare il local minima e farmi "saltare" in basso verso un minimo più importante:



- **batch (offline) learning**: in cui si presentano tutti gli esempi di addestramento alla rete, per ognuno si valuta l'errore e per ogni peso si valuta la variazione da apportare e solo alla fine dell'addestramento si sommano tutte le modifiche dei pesi. In termini tecnici la presentazione di tutto il training set è detta **epoca**. E' il più corretto matematicamente, fornisce infatti una stima accurata del gradiente e permette la parallelizzazione del processo di apprendimento, infatti dato che i pesi vengono aggiornati solo dopo aver presentato tutti gli esempi di addestramento, faccio calcolare l'aggiornamento dei pesi a ciascuno dei calcolatori differenti.



- Batch learning has the following advantages:
 - it provides an accurate estimate of the gradient vector;
 - it allows the parallelization of the learning process.
- Online learning has some advantages, in particular:
 - it can be used when there is no fixed training set;
 - it requires much less storage space than batch learning;
 - the noise in the gradient can help escape from local minima, if they are not too severe (in fact, the gradient for a single training sample can be considered a noisy approximation of the overall gradient).

- **mini-batch learning**: il training set viene diviso in mini batches, ovvero in insiemi costituiti da n , con $n > 1$ ma minore della dimensione del training set. Solo dopo aver presentato tutti i campioni di un minibatch si effettua la calibrazione dei pesi e così via presentando il secondo mini-batch, poi quello successivo... Questo è quello più usato nella pratica.

Dovremo dunque tra i vari iperparametri

Gli **iperparametri** sono quegli elementi del modello ad esempio:

- la dimensione dello strato in ingresso
- la dimensione dello strato in uscita

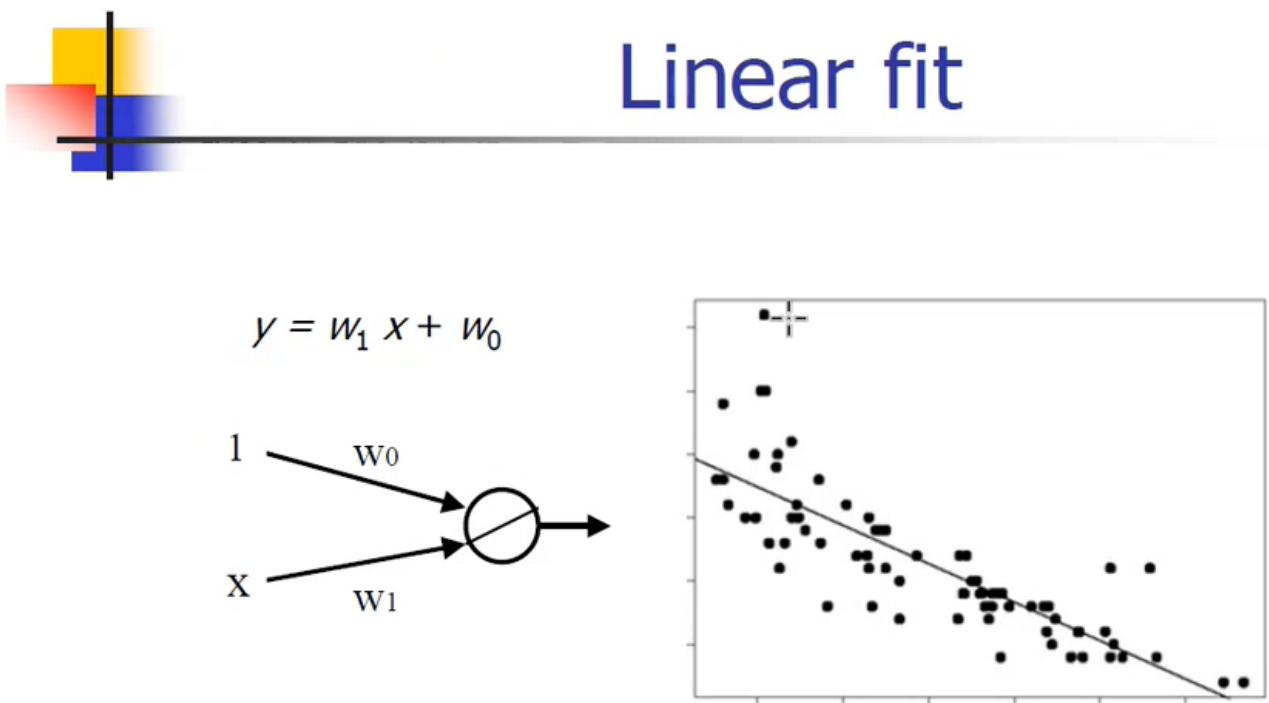
- il numero di strati nascosti
- il numero di neuroni per ogni strato nascosto
- la funzione di attivazione dei neuroni
- il valore del learning rate
- il valore del momentum
- ...

Uno tra questi è anche la dimensione del mini-batch, ovvero quanto deve essere grande il mini-batch

I parametri sono ad esempio i pesi che vengono modificati durante il processo di apprendimento

Abbiamo capito che il batch-learning è il più corretto dal punto di vista matematico ma abbiamo anche visto nelle slide precedenti che la delta rule non implementa il batch learning ma l'online learning che è meno corretto dal punto di vista matematico perché la delta rule aggiorna i pesi dopo ogni presentazione di ogni singolo esempio di addestramento. Nessun problema in quanto i matematici ci garantiscono che se il learning rate è sufficientemente basso la delta rule funziona.

Supponiamo di avere uno scatter plot che rappresenta una certa funzione. In questo esempio la x è il peso di un automobile e la y è il consumo di carburante. Dopo aver raccolto i dati, rappresentandoli è evidente che esiste una funzione che lega gli input agli output quindi abbiamo una funzione di cui non conosciamo una rappresentazione analitica ma che vogliamo descrivere usando un fitting lineare di questa funzione, al fine di poter analizzare una qualsiasi nuova macchina e capirne i consumi. Come possiamo trovare la rappresentazione lineare in un modo non matematico (ad esempio attraverso l'utilizzo dei minimi quadrati etc..) ma usando solo le reti neurali?



Questa rete neurale sarà molto semplice in quanto è un singolo neurone che ha una funzione di attivazione lineare (vedi il simbolo all'interno del neurone). Questo neurone ha come input il peso dell'automobile x e un peso che dovrà essere settato in modo appropriato alla fine dell'addestramento e poi vi è anche un bias associato al peso w_0 legato a un ingresso sempre uguale a 1. Essendo una funzione lineare si può rappresentare come:

$$y = w_1x + w_0$$

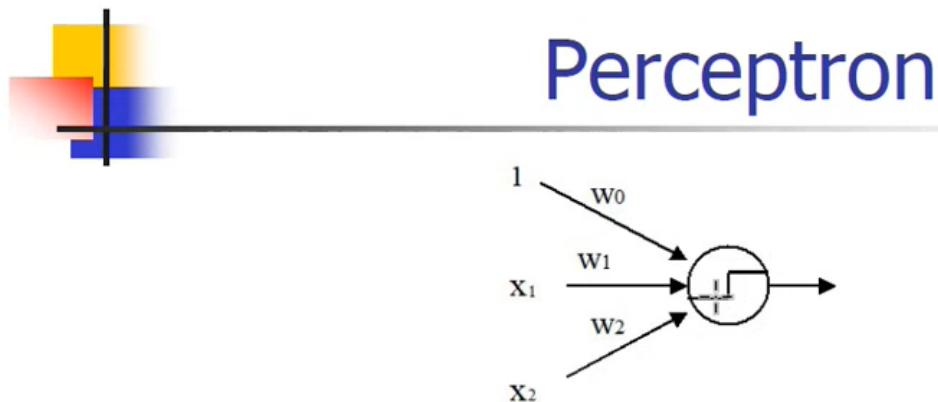
Questa è l'equazione di una retta nel piano e come possiamo addestrare il neurone? Partiamo da dei valori casuali per w_1 e w_0 e cominciamo a presentare gli esempi di addestramento ovvero i punti dello scatter plot, applichiamo la delta rule essendo il neurone lineare e modifichiamo i valori di w_1 e w_0 . Poi presentiamo un altro punto scelto casualmente e rimodifichiamo la posizione di questa retta fino a trovare una posizione molto vicina a quella che si genererebbe matematicamente. Se non usassimo il bias potremmo usare solo una parte dell'equazione ovvero:

$$y = w_1x$$

ovvero potremmo generare una retta che passa sempre per l'origine e non farebbe al caso nostro.

Ora abbiamo visto un esempio di applicazione di approssimazione di funzione con la delta rule, con neuroni lineari. Esiste un altro semplice schema di rete neurale che permette di effettuare classificazione in 2 classi, il perceptron.

Perceptron



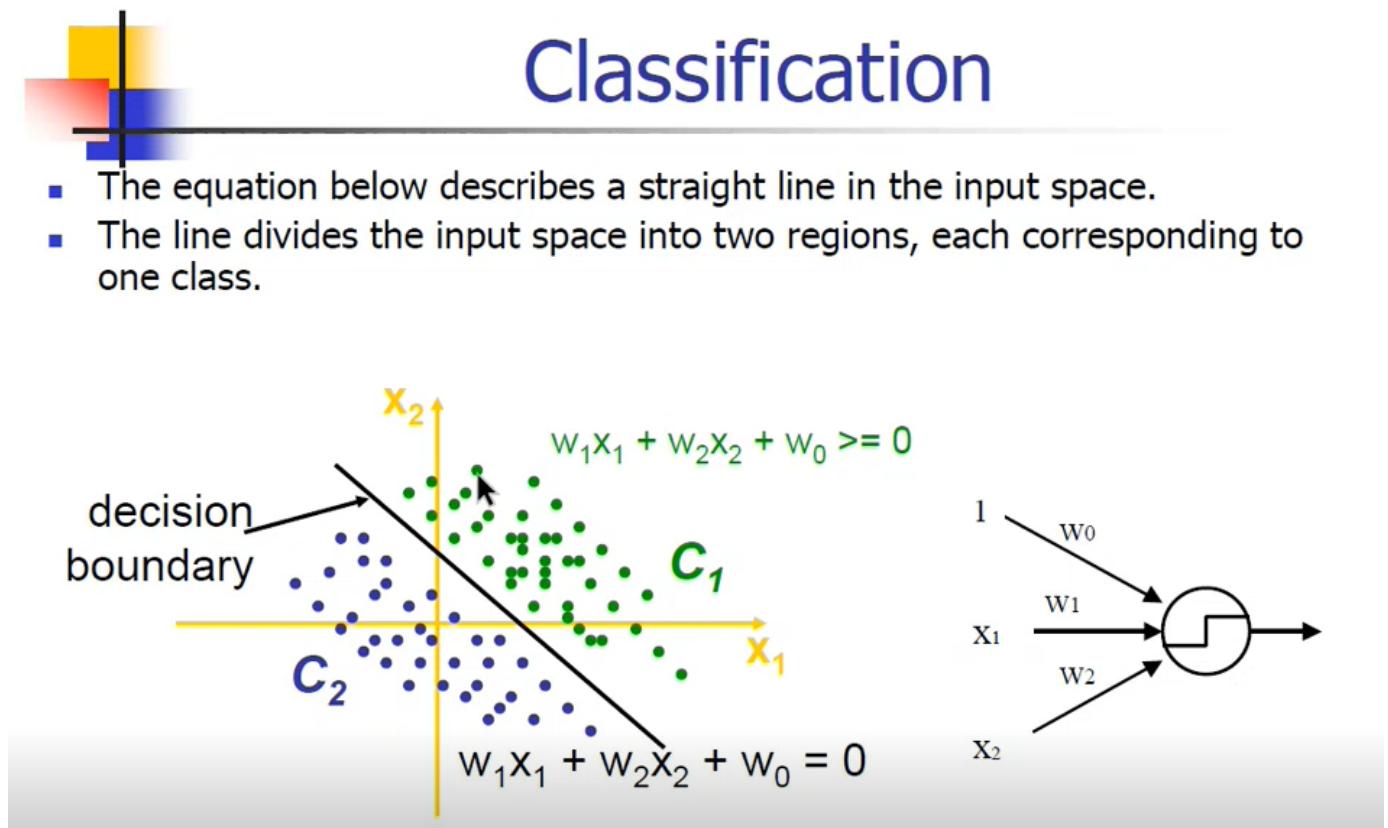
- A perceptron is a single neuron with a hard limiter as activation function (usually, the *sign function* or the *unit step function*). It is used for binary classification.
- Given training examples from classes C_1 and C_2 , the perceptron can be trained to correctly classify the training examples. E.g.,
 - if the output is +1 then the input is assigned to class C_1
 - if the output is -1 then the input is assigned to C_2

Si tratta di un singolo neurone rappresentato in questo modo per dire che la funzione di attivazione **ha**

un limite di attivazione netto come ad esempio la funzione **segno** o a .

Dunque un neurone di questo tipo se riceve in ingresso dei punti del piano che appartengono a 2 possibili classi, questo perceptron può essere addestrato per classificare correttamente gli esempi di addestramento. Ad esempio supponiamo che stiamo usando la funzione segno, questo vuol dire che il perceptron può essere addestrato per produrre +1 se l'input presentato in ingresso appartiene alla classe C_1 e -1 se appartiene alla classe C_2 .

Supponiamo di avere due classi C_1 e C_2 :



La classe verde è rappresentata sul lato destro rispetto a un confine lineare che divide le 2 classi.

L'altra classe blu è rappresentata sull'altra parte. Questo confine di decisione è una retta nel piano che viste le coordinate a cui facciamo riferimento per rappresentare i punti è rappresentata in forma implicita attraverso un equazione:

$$w_1x_1 + w_2x_2 + w_0 = 0$$

? Come facciamo a generare questa equazione?

Il perceptron è in grado di classificare i punti in 2 classi, i punti hanno infatti 2 coordinate, x_1 e x_2 , poi c'è il bias pari a 1 e l'output se uso la funzione segno è +1 o -1 a seconda che la somma pesata degli ingressi sia maggiore o uguale di 0. Se è minore di 0 infatti mi da -1.

Dunque settando a 0 l'argomento della funzione che mi da l'uscita di questo neurone ottengo l'equazione di una retta che guarda caso mi rappresenta il confine di decisione tra queste 2 classi.



Perceptron: Learning Algorithm

1. initialize the weights (to zero or to a small random value);
2. choose a learning rate η (a number between 0 and 1);
3. until the stop condition is satisfied (e.g., weights don't change):
 - for each training sample (x, t) :
 - calculate the output activation $y = f(w \cdot x)$
 - If $y = t$, do not change the weights
 - If $y \neq t$, update the weights:
 - $w^{new} = w^{old} + \eta(t - y)x$

L'algoritmo di addestramento del perceptron consiste nel:

1. inizializzare i pesi con valore casuale
2. si sceglie un learning rate
3. dato che stiamo effettuando un processo di variazione dei pesi se i pesi continuano a essere variati significa che ci troviamo ancora all'interno del processo di apprendimento. Per ciascuno esempio di addestramento ovvero per ciascuna coppia (input,target) si presenta x come ingresso al neurone, si calcola l'uscita che è funzione della somma pesata degli ingressi, se l'uscita è uguale a quella desiderata i pesi non cambiano quindi passo all' x successivo, altrimenti devo aggiornare i pesi usando la formula indicata (qui w e x sono vettori):

$$w^{new} = w^{old} + \eta(t - y)x$$

Questa formula da un punto di vista sintattico formale è identica alla delta rule ma non è la delta rule in quanto $t-y$ nel caso del perceptron, supponendo di usare la funzione binaria a soglia, il target t o è 1 o è 0.

Se t è 1 e l'uscita è 0 l'errore è 1, invece se t è 0 e l'uscita è 1 l'errore è pari a -1. Ho due soli possibili valori per l'errore fatto da un perceptron (1 o -1), se applicassi invece la delta rule per $(t-y)$ potrei avere qualsiasi numero reale in quanto la delta rule si applica a neuroni lineari.

?

Cosa accade dunque all'interno di un perceptron?

Si parte da un settaggio casuale del valore dei pesi, dunque la retta candidata per rappresentare il confine di decisione potrebbe essere ovunque nello spazio, poi considerando i punti verdi e i blu come esempi di addestramento (ogni esempio di addestramento è costituito dalle coordinate del punto e dal valore della classe pari a +1 o -1 ad esempio), modifico i pesi di volta in volta e alla fine la retta che prima era in una posizione casuale man mano si avvicina a essere quella rappresentata in precedenza.

Perceptron: decision boundary

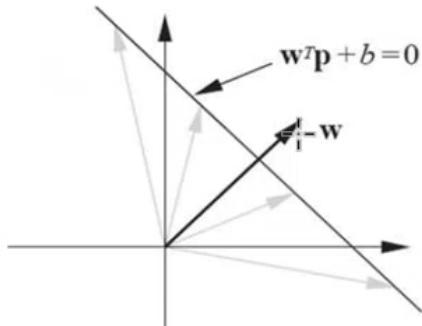
Per quanto riguarda la delta rule che abbiamo visto essere relativa ad addestramento di reti con funzione di attivazione lineare, vorremmo avere una visione intuitiva anche per il perceptron in quanto

l'algoritmo di apprendimento non coincide proprio con la delta rule.



Decision boundary

- All points on the decision boundary have the same inner product with the weight vector.
- Therefore, they have the same projection on the weight vector, so they must lie on a line orthogonal to the weight vector.

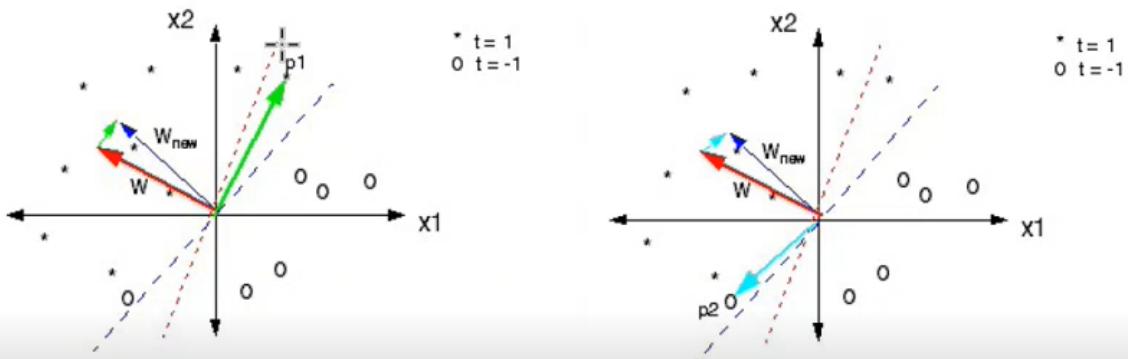


Se w è il vettore che rappresenta i pesi e la retta candidata ad essere il confine di decisione tra le due classi, il vettore dei pesi e questo confine di decisione sono perpendicolari tra loro. Dalla formula:

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

si ottiene che il prodotto scalare tra il vettore dei pesi e il vettore di ingresso x è uguale a $-w_0$ dunque tutti i punti sul confine di decisione hanno lo stesso prodotto scalare con il vettore dei pesi, dunque sono perpendicolari, ovvero dato che l'algoritmo di apprendimento cambia i pesi, ma essendo perpendicolare si porta dietro anche il confine di decisione, ovvero se in questo caso:

- The red dashed line is the decision boundary.
- We note that both p1 and p2 are classified incorrectly.
- Let us consider what happens when you choose the training sample p1 (or p2) to update the weights.



il confine di decisione è la linea tratteggiata rossa sulla sinistra e ho il punto p1 che appartiene a una classe per cui il perceptron dovrebbe produrre 1 in uscita, questo punto si trova dalla parte sbagliata del confine di decisione, dunque l'uscita del perceptron non è 1 ma se uso la funzione segno è -1 dunque significa che questo punto è in posizione sbagliata. Basta dunque far vedere che se considero p1 come esempio di addestramento e applico la regola di aggiornamento dei pesi del perceptron il confine di decisione diventa quello blu e dunque includerà p1 nel semipiano corretto. Si può fare questo in quanto il nuovo vettore dei pesi è pari a quello vecchio più eta moltiplicata per ($t-y$) dove t nel caso di p1 era 1, y mi dava -1 dunque $t-y = 2$ moltiplicato per x dunque sommo a w^{old} che è il vettore rosso perpendicolare al confine rosso tratteggiato il piccolo vettore il piccolo vettore calcolato come $2\eta x$ che è una percentuale di p1, applico la regola del parallelogramma e ottengo il nuovo vettore dei pesi perpendicolare al nuovo confine di decisione.

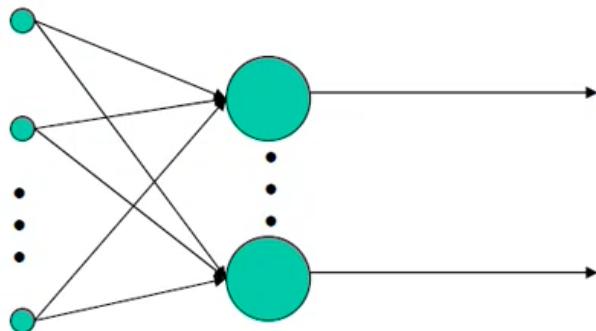
Se usassi un altro punto ad esempio p2 sulla destra potremmo avere che il nuovo confine non include p2 ma è più vicino ovvero la prossima volta che presenterò p2 le cose andranno a posto.

! La conclusione importante è che la regola del perceptron è garantita matematicamente convergere a una soluzione in un numero finito di passi (epoch) se il problema è linearmente separabile, ovvero date 2 classi linearmente separabili è possibile trovare un perceptron che le separa. E' importante in quanto possiamo usare i punti a disposizione per addestrare il perceptron ovvero trovare il confine di decisione poi da quel punto in poi potremo usare la rete addestrata dandole in ingresso dei nuovi punti per posizionarli dalla parte corretta.

Multi-neuron perceptron



Multiple-neuron perceptron



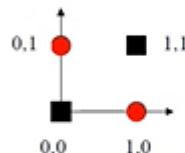
- Each perceptron has its own decision boundary for classifying the input points into 2 classes.
- A multi-neuron perceptron can perform multi-class classification.

Ovviamente in generale potremmo avere a che fare con più classi ovvero avere un multi-neuron perceptron. Se infatti gli input alimentato due o più perceptron contenuti nello strato di uscita potremo ottenere una multi-class classification.

⚠ Il perceptron può risolvere solo problemi lineari ma non quelli NON lineari come ad esempio lo XOR problem:



- The perceptron can only solve linear problems.
- The perceptron cannot solve the XOR problem:



- Possible solutions:
 - use a neuron with a specially defined activation function, e.g.,

$$y = (x_1 - x_2)^2 = \begin{cases} 1 & \text{se } x_1 \neq x_2 \\ 0 & \text{se } x_1 = x_2 \end{cases}$$

- introduce *hidden layers*.

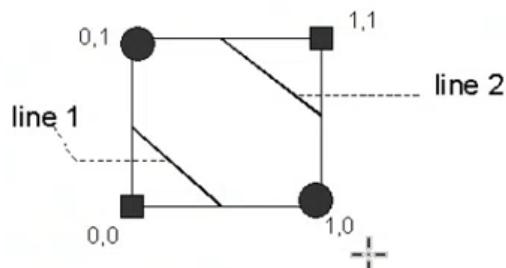
nello XOR ho quattro punti nel piano. I punti in rosso che hanno una sola componenete uguale a 1 producono in uscita 1 mentre le altre 2 coppie in nero devono produrre 0. Un perceptron non riesce a classificare queste 2 classi perchè non sono linearmente separabili. La particolare funzione di attivazione che potrei usare come mostrato nella immagine non può andarci bene in quanto in un problema non lineare ma non capibile così intuitivamente, come in questo caso, potrebbe non essere facile.

? Come si risolve allora questo tipo di problema in modo generale?

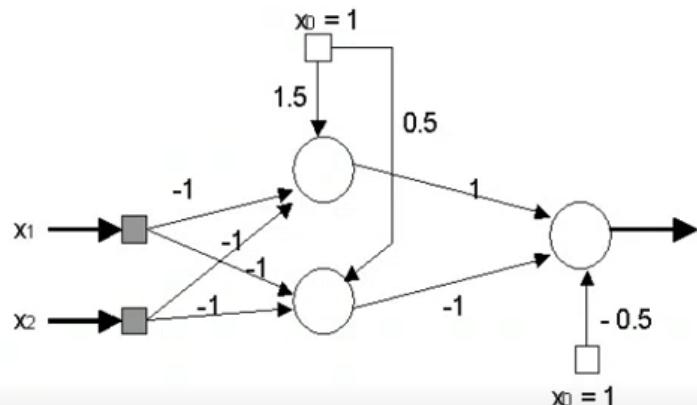
Introducendo strati nascosti.



The XOR problem solved with two separating lines



- The network needs two neurons, both fed with two inputs, to represent the two lines, and a third neuron to combine the information from these two lines.



i tondini sono una classe i quadrati un'altra classe. Prima di entrare nella descrizione formale della soluzione proviamo ad esprimere il concetto in modo intuitivo.

Possiamo sfruttare 2 linee di separazione e dire che il punto (0,0) sta a sinistra della linea 1 e tutti gli altri punti sono a destra. In modo simile per il punto (1,1) e la linea 2. Se costruisco queste 2 linee, significa che se sono dentro le 2 linee ho una classe, altrimenti ho la classe quadrato all'esterno.

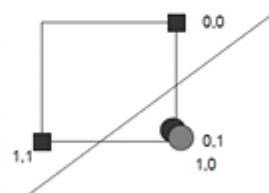
Dunque prima costruisco le linee e dopo sfrutto la presenza delle linee per ottenere la classificazione voluta. Vuol dire che la rete avrà bisogno di 2 neuroni alimentati entrambi con 2 input, le coordinate dei punti x_1 e x_2 per rappresentare le 2 linee che ho descritto in modo intuitivo. Una volta rappresentate le 2 linee, ciascun neurone in figura rappresenta "la propria area di appartenenza" ovvero il fatto di stare da una parte o dall'altra della linea si inserisce un ulteriore strato nella rete, l'ultimo neurone, che sfrutta le uscite dei precedenti neuroni.

Con x_1 e x_2 in ingresso alla rete, se supponiamo che valgano 1 e 1:

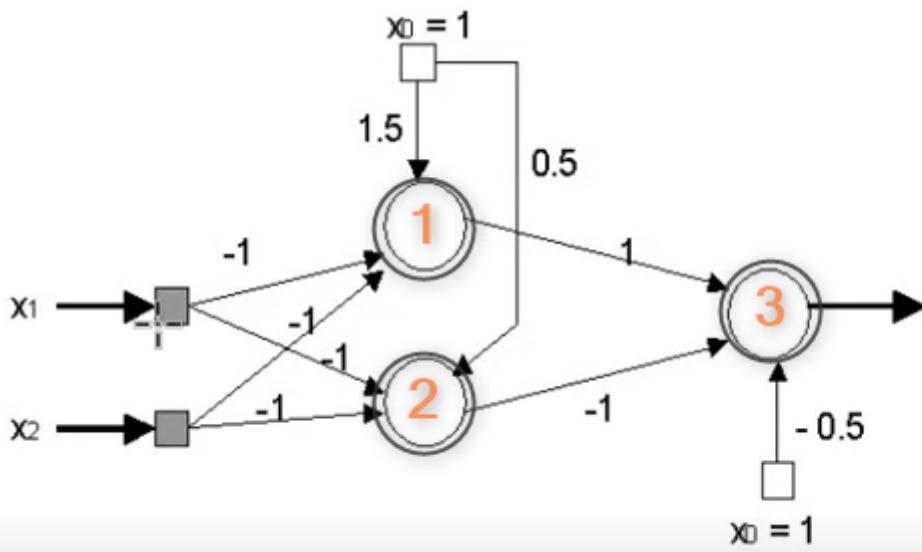


Effect of the first layer of weights

		Net input to hidden layer		Output from hidden layer	
x_1	x_2	unit 1	unit 2	unit 1	unit 2
1	1	-0.5	-1.5	0	0
1	0	0.5	-0.5	1	0
0	1	0.5	-0.5	1	0
0	0	1.5	0.5	1	1



- The inputs to the second layer are linearly separable.



l'ingresso al neurone 1 è $1^* -1 + 1^* -1 + 1.5 = -0.5$

l'ingresso al neurone 2 è $1^* -1 + 1^* -1 + 0.5 = -1.5$

sia il neurone 1 che il neurone 2 hanno come funzione di attivazione quella a soglia dunque l'uscita è 0 per il primo in quanto la somma in ingresso è negativa, anche il secondo neurone produce 0, dunque il punto (1,1) nello spazio di input della rete è stato trasformato nel punto (0,0) nello strato nascosto.

Dunque gli input al secondo livello sono linearmente separabili.

I pesi sono stati messi da noi ad arte in quanto i pesi in reti con strati nascosti dovranno essere il risultato di un algoritmo di addestramento.

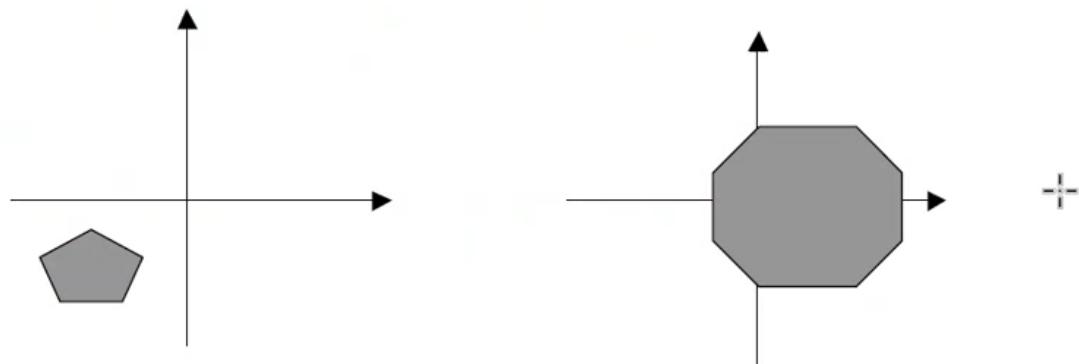
Abbiamo capito che il segreto per risolvere problemi complessi è l'inserimento di strati nascosti. Una rete con un livello nascosto riesce a modellizzare regioni con un numero di lati **al massimo uguale** al numero di neuroni nascosti.

Ogni neurone nel singolo strato nascosto riesce a modellizzare una retta se siamo nel piano, dunque ad esempio se ho 5 neuroni nello strato nascosto, avrò un neurone di uscita che mette insieme le modellizzazioni e identifica l'intersezione dei semipiani prodotti dai 5 neuroni nascosti precedenti:



One hidden layer

- A network with one hidden layer can model regions with a number of sides at most equal to (\leq) the number of hidden neurons.



Dunque in uscita da questa rete con un singolo neurone di uscita potrò dire se mi trovo all'interno della regione di decisione indicata sulla destra oppure all'esterno. In sostanza abbiamo costruito un modello di una regione di questo tipo.

Magari qualche neurone individua una retta non utile ai nostri scopi e che quindi non da un contributo alla costruzione di questa regione **ecco perchè minore o al massimo nel caso migliore uguale**.

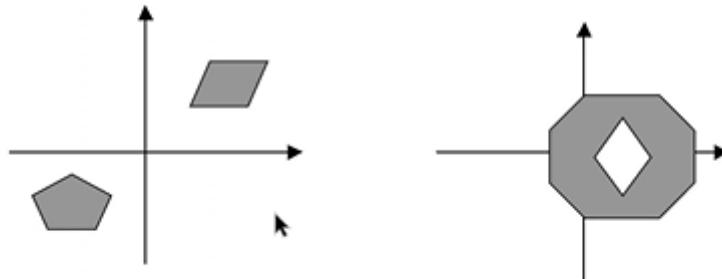
? Se usassimo 2 livelli nascosti cosa potremmo riuscire a fare?

Il neurone di uscita che prima individuava una regione del piano, ora individua un'unione delle regioni di decisioni individuate dai neuroni dello strato precedente.



Two hidden layers

- A network with two hidden layers can model arbitrarily complex decision regions.



- In practice, most problems are solved with a single hidden layer, sometimes with two.

Nella pratica la maggior parte dei problemi sono risolti con un singolo strato nascosto, altre rare volte con 2, ma in genere non occorre andare oltre i 2 strati nascosti per questo tipo di attività di

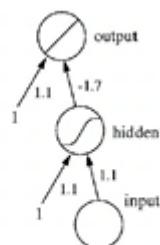
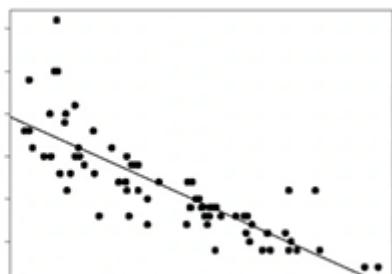
classificazione.

Questo risultato di modellizzazione di regioni nel campo della classificazione è lo stesso che da origine a considerazioni simili nel campo di **approssimazioni di funzioni**.

Tornando all'approssimazione che abbiamo visto nello scatter plot precedente:



- Consider again the following scatter plot and the best linear model found by gradient descent. The data are not evenly distributed around the line.
- We can get a better approximation by using a hidden layer, consisting of a single neuron with a tanh function.

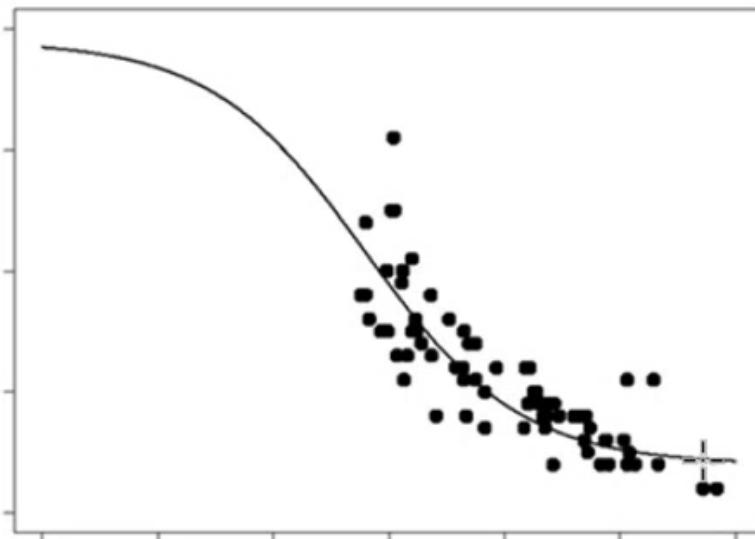


Gurdando i dati, essi non sono uniformemente distribuiti intorno alla linea, in quanto abbiamo alcuni errori notevoli (punti molto fuori). Potremmo migliorare la nostra approssimazione? Si, basta inserire strati nascosti con un unico neurone nascosto che in questo caso ha come funzione di attivazione la tangente iperbolica:

$$-1.7 * (\tanh (1.1 + 1.1 * x)) + 1.1$$

questa rete una volta addestrata approssima lo scatter plot come segue:

- Once trained, this network approximates the scatter plot as follows:



Questa approssimazione è migliore di quella precedente fatta in modo lineare.

? Ma se lo scatter plot fosse come questo potremmo ottenere una approssimazione?



Hidden layers

- What can we do if the data look like this?



- Solution: *increase the number of hidden neurons*.
- Too many neurons in a single hidden layer or too many hidden layers can have a negative effect on the network performance. In general, the best choice is to start with a network that has a reasonable minimum number of hidden neurons, then train it and only if we do not get the desired result, we start increasing the number of hidden neurons or layers.
- A network with one hidden layer and enough hidden neurons can approximate any continuous function with any degree of accuracy.

La parte a destra dell'apice dello scatter plot potrebbe essere approssimata con una tangente iperbolica quindi potrei mettere un primo neurone nello strato nascosto per approssimare questa parte di plot, poi si potrebbe inserire un secondo neurone nascosto con tangente iperbolica per approssimare

un gruppetto di punti (ad esempio quelli in prossimità dell'apice), a tendere quindi mi basterebbe aumentare il numero di neuroni nascosti.

⚠ Però attenzione... ⚠

più strati nascosti metto e più neuroni ci sono più aumenteranno i parametri del modello ovvero i pesi che andranno settati.

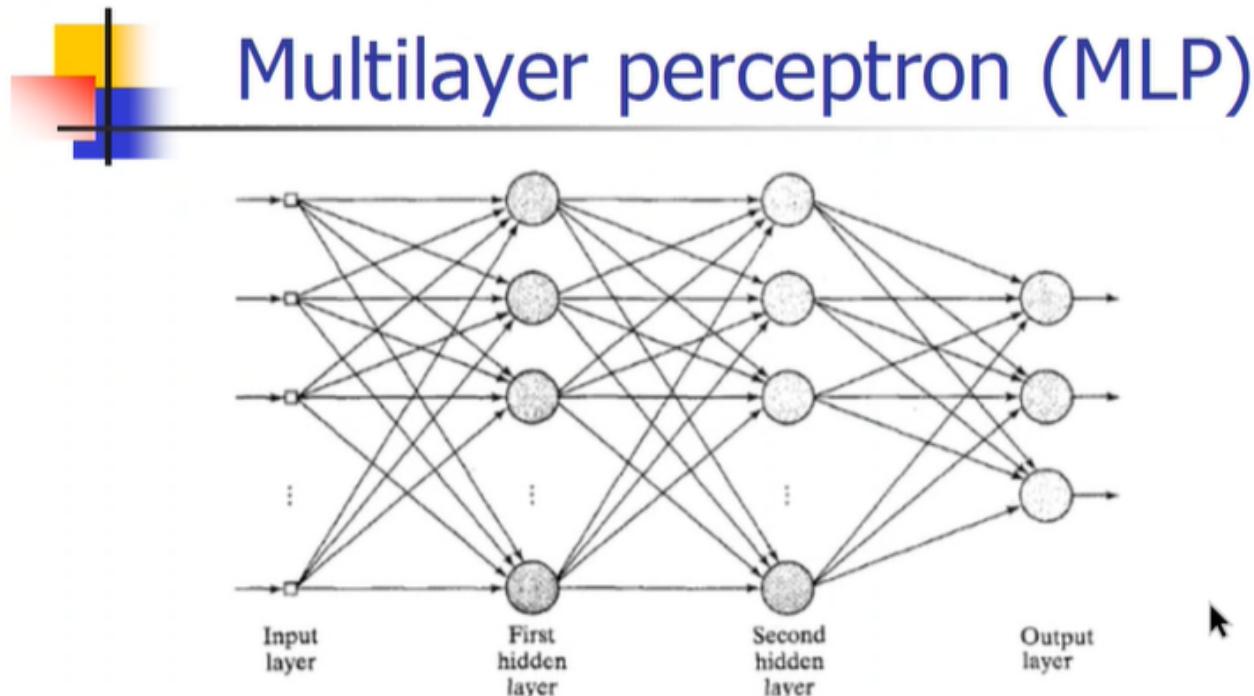
Dunque inizialmente non conviene partire con una rete molto grande, ma con una rete che dal punto di vista euristico ha un numero ragionevole di neuroni che vogliamo trattare.

Dunque qualunque sia la funzione continua che vogliamo approssimare possiamo sempre farlo usando una rete neurale addirittura con un solo livello nascosto e un numero sufficiente di neuroni in quello strato nascosto.

Avendo dunque introdotto gli strati nascosti dalla prossima volta dovremmo fare i conti con l'addestramento di una rete con strati nascosti in maniera automatica.

10/03/2023

Il multi-layer perceptron è una delle architetture più usate:



Questi in realtà non sono dei perceptron. Infatti leggendo la documentazione relativa ai perceptron, inizialmente il perceptron quando fu introdotto permetteva agli ingressi di assumere solo il valore **0** o il valore **1**. Poi nel tempo i ricercatori hanno iniziato ad usare numeri reali per l'input, ma il perceptron ha una funzione a soglia o segno dunque questo tipo di funzione vedremo che non va bene per applicare un algoritmo di addestramento generale di reti multilivello. Questo è il motivo per cui abbiamo definito noi i pesi.

Vorremmo addestrarla ovvero usare un insieme di addestramento. **Non abbiamo però i valori desiderati per i neuroni nascosti**. Negli esempi di addestramento infatti io vedo l'input e l'output desiderato, queste coppie di input e output mi permettono di fare training sulla rete. Non conosciamo cosa è desiderato dai neuroni nel primo o secondo livello nascosto. Non posso usare le formule viste fin ora in cui includevo nella formula la differenza tra il target e l'uscita effettiva. Esiste però un algoritmo molto interessante detto **Error Backpropagation** che viene usato nel caso di reti a multi livello:



Error Backpropagation

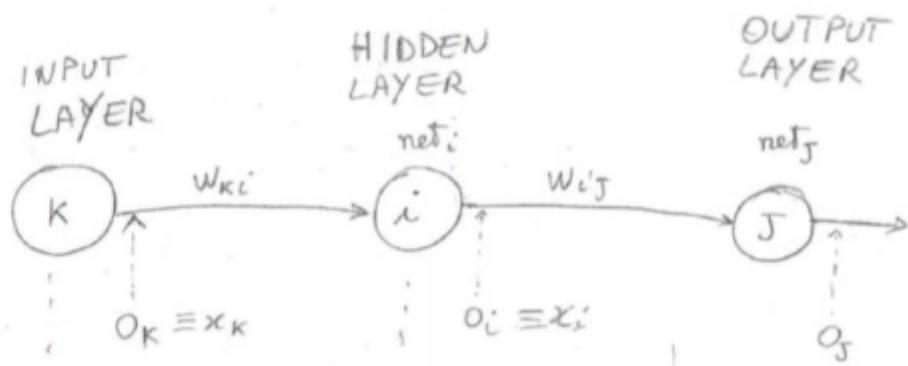
- How can we train a multilayer neural network?
- We do not have the desired values for hidden neurons.
- We use the *error backpropagation algorithm*.
 - Forward activation
 - Calculate the output error
 - Error backpropagation



Consiste in una:

- **attivazione in avanti**: presentiamo all'ingresso della rete un valore e diamo il tempo alla rete di produrre un'uscita
- **calcoliamo l'errore di uscita**: in quanto abbiamo l'uscita desiderata dallo strato di uscita, ovvero calcoliamo la differenza tra l'uscita desiderata sull'ultimo strato di neuroni e l'uscita effettiva prodotta dall'ultimo strato di neuroni
- **facciamo la retro-propagazione dell'errore**: dallo strato di uscita allo strato di ingresso.

$$E \text{ for a training sample: } \frac{1}{2} \sum_j (t_j - o_j)^2$$



L'errore per un esempio di addestramento E è la differenza tra l'uscita desiderata per il j -esimo neurone di uscita (il target t_j) e l'uscita effettiva per quel neurone (o_j) al quadrato. Inoltre introduciamo

l'1/2 in quanto sapendo che dovremmo calcolare la derivata, si semplificherà:

$$E \text{ for a training sample: } \frac{1}{2} \sum_j (t_j - o_j)^2$$

A questo punto bisogna pensare al MLP, in cui dover calcolare i pesi, ovvero il w_{ij} di ciascun arco. Per semplificare supponiamo di avere un unico livello nascosto e riferiamoci al generico neurone di input x_k , un unico livello nascosto con un generico neurone nascosto i e un generico neurone di uscita j .

L'uscita del generico neurone i sarà indicata con o_i che però in questo esempio specifico coincide con l'ingresso del neurone successivo ovvero x_j . Tale uscita è il risultato dell'applicazione della funzione di attivazione f alla somma pesata degli ingressi di quel neurone, tipicamente indicata con net_j :

$$\begin{aligned} \text{net}_j &= \sum_{i=0}^m x_i w_{ij} \\ &= \sum_{i=0}^n o_i w_{ij} \\ o_j &= f(\text{net}_j) \end{aligned}$$

Dopo aver ottenuto un'uscita dalla fase di forward activation, retropropaghiamo l'errore, ovvero dovendo andare nella direzione opposta al gradiente per ridurre l'errore, il w_{ij} lo modificherà andando nella direzione opposta al gradiente. Dunque la derivata parziale di E rispetto a w_{ij} , applicando la regola della catena, sarà pari a:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} =$$

$$= - (t_j - o_j) f'(\text{net}_j) x_i = - \delta_j x_i$$

δ_j

Noi vogliamo interpretare la formula cerchiata in rosso secondo il formato conosciuto quando abbiamo introdotto la delta rule, ovvero vogliamo tirare fuori l'errore delta di quel neurone per l'ingresso. Il δ_j è il prodotto di questi due fattori ed è una generalizzazione di quello visto nella delta rule, in quanto nella delta rule δ_j era pari solo a $(t_j - o_j)$. Nella delta rule però si considerano funzioni di attivazione lineare quindi la derivata di una funzione lineare è 1 pertanto questa formula qui generalizza la delta rule. Per modificare il peso di w_{ij} **devo sommarci un qualcosa che è meno la derivata moltiplicata per una costante di proporzionalità che è il learning rate.**

? Come modifichiamo il peso w_{ki} ?

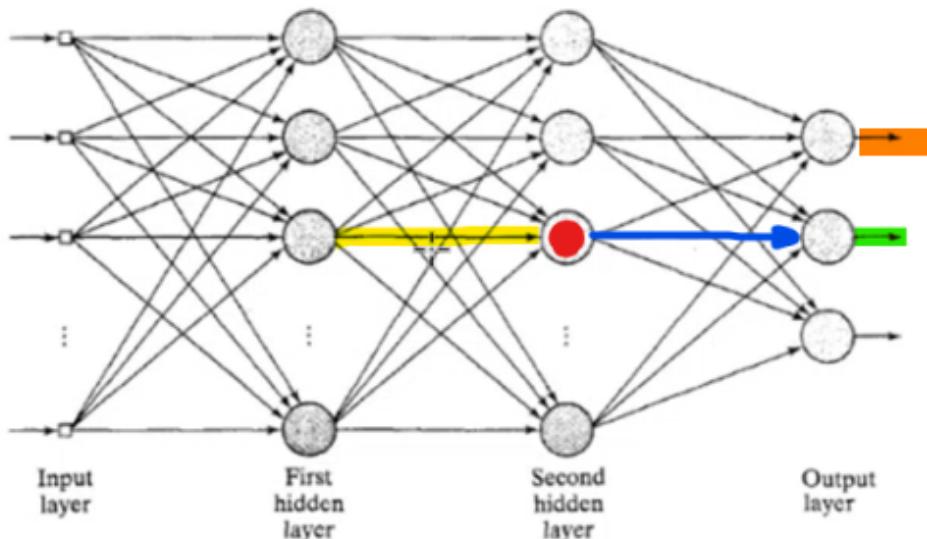
Devo calcolare la derivata parziale di E rispetto a w_{ki} .

Il tragitto essendo più lungo avremo un calcolo più lungo:

$$\begin{aligned} \frac{\partial E}{\partial w_{ki}} &= \frac{\partial E}{\partial o_j} + \frac{\partial E}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial o_i} \frac{\partial o_i}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_{ki}} = \\ &= -\delta_j w_{ij} f'(net_i) x_k = -\delta_i x_k \end{aligned}$$

In sostanza è come dire che:

Multilayer perceptron (MLP)



Multilayer perceptron with two hidden layers

il neurone in rosso lo posso addestrare cambiando il peso del canale al suo ingresso considerando il fatto che il neurone in rosso ha dato un contributo all'errore fatto dalla rete che è il prodotto dell'errore in verde per la forza del peso con cui i neuroni sono connessi (in blu). Se consideriamo il fatto però che

il neurone rosso è connesso con altri neuroni di uscita, dovrò considerare l'errore fatto dall'altro neurone di uscita a cui è connesso (in arancione).

Dunque se il neurone i è connesso a più neuroni di uscita allora il δ_i lo calcolo come $f'(net_i)$:

If, e.g., neuron i is connected to more output neurons, we have:

$$\delta_i = f'(net_i) \sum_j \delta_j w_{ij}$$

e poi dovrò sommare il prodotto tra l'errore del j -esimo neurone di uscita a cui è connesso il neurone i e il peso associato alla connessione tra il neurone i e il neurone j .

Una delle funzioni che si utilizzano come funzione di attivazione è la funzione logistica, dovendo calcolare la derivata prima:

- If we use the logistic function as activation function

$$f(net_j) = \frac{1}{1+e^{-net_j}}$$

we have

$$\begin{aligned} f'(net_j) &= \frac{e^{-net_j}}{(1+e^{-net_j})^2} \\ &= \frac{1}{1+e^{-net_j}} \left(1 - \frac{1}{1+e^{-net_j}} \right) \\ &= f(net_j)[1-f(net_j)] \end{aligned}$$

Vediamo un implementazione online (i pesi vengono aggiornati dopo aver presentato ogni singolo esempio di addestramento) dell'algoritmo di back propagation:



Backpropagation algorithm (online)

- 1. Initialize the weights to small random values
- 2. For the first input sample calculate the output of all neurons

$$o_j = \frac{1}{1 + e^{-net_j}}$$

- 3. For each output neuron calculate its error:

$$\delta_j = (t_j - o_j) f'(net_j) = (t_j - o_j) o_j (1 - o_j)$$

- 
- 4. For all hidden layers (from output to input) calculate the error for each neuron:

$$\delta_j = f'(net_j) \sum_s \delta_s w_{js} = o_j (1 - o_j) \sum_s \delta_s w_{js}$$

- 5. For all layers, update the weights for each neuron:

$$\Delta w_{ij}(n+1) = \eta \delta_j o_i + \alpha \Delta w_{ij}(n)$$

- 6. Repeat from step 2 for all training samples
- 7. Calculate the error on the training set: if the error falls within the desired tolerance, the algorithm is said to have *converged*; otherwise continue with the training

1. Inizializzazione dei pesi con dei piccoli valori casuali: si tratta di una caratteristica importante delle reti in quanto abbiamo la garanzia che qualunque sia il punto da cui partiamo sulla superficie dell'errore, l'algoritmo di addestramento ci assicura di arrivare al minimo
2. si estrae casualmente il primo esempio di addestramento dal training set, si presenta in ingresso alla rete e si calcolano le uscite di tutti i livelli di uscita. Per semplicità in foto usiamo la funzione logistica.

3. Possiamo calcolare per ciascun neurone di uscita la differenza tra l'uscita desiderata e l'uscita effettiva e l'errore lo possiamo calcolare come nella prima formula. Dato che nella slide precedente abbiamo detto che se usiamo questa funzione di attivazione, la f' è semplicemente l'uscita moltiplicata per $(1-o_j)$.
4. A questo punto iniziamo a fare la propagazione all'indietro dell'errore, ovvero per tutti i livelli nascosti partendo dall'uscita e andando verso l'ingresso, si calcola l'errore per ciascun neurone, secondo la formula che abbiamo visto essere valida per i neuroni nascosti.
5. Arrivati allo strato di ingresso, avendo adottato in questo caso l'online learning, per ogni livello si aggiornano i pesi per ciascun neurone, ovvero si usa la Δ_{wij} che poi sommiamo a ciascun peso che è data dal prodotto di una costante di proporzionalità che è il learning rate per l'uscita $o_i + \alpha\Delta_{wij}(n)$ che è il termine momentum.
6. Si presenta il secondo esempio di addestramento e così via...
7. Alla fine della presentazione di tutti gli esempi di addestramento si calcola l'errore su tutto il training set, se l'errore cade all'interno della tolleranza desiderata siamo andati in convergenza altrimenti si esegue una successiva epoca ovvero una successiva presentazione completa di tutti gli esempi di addestramento. Da un punto di vista teorico l'ordine di presentazione degli esempi di addestramento è casuale, da un punto di vista implementativo questa estrazione casuale si implementa attraverso un operazione di shuffling ovvero si mescolano inizialmente i dati e poi si prendono nell'ordine con cui sono stati mescolati.

Dunque inserendo il learning rate all'interno di questa formula:

$$\frac{\partial E}{\partial w_{ij}} = -(t_j - o_j) f'(net_j) x_i = -\delta_j x_i$$

otteniamo la stessa formula della delta rule generalizzata.

Leggendo alcuni documenti, quando si parla di delta rule ci si riferisce alla delta rule generalizzata mentre la delta rule originale viene considerata un caso particolare della delta rule generalizzata.

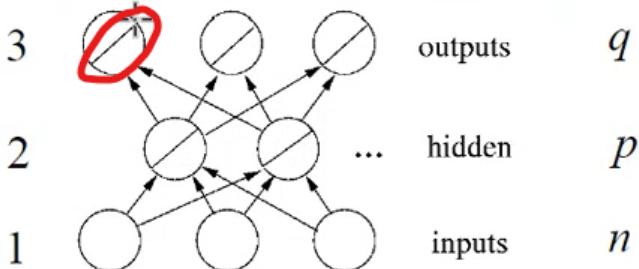
L'algoritmo di Back-propagation è in termini generali quella mostrata ma in fase di implementazione bisogna indicare lo specifico algoritmo che realizza la back propagation.

Esistono diverse implementazioni ciascuna delle quali ottimizza determinati aspetti dell'algoritmo.

Con la rete in figura potremmo dire che ci sono 2 livelli non considerando lo strato di ingresso ad esempio:

Limitation of a network with linear units

- A multilayer network with linear activation functions will only be able to solve a problem that a single-layer network can solve.



- n input neurons, p hidden neurons, q output neurons
- $W_{21} (pxn)$, $W_{32} (qxp) \Rightarrow z = W_{32}Y = W_{32}W_{21}X = WX$ with $W = W_{32}W_{21} (qxn)$
- For multilayer networks, therefore, a nonlinear activation function is required.

Una rete multilayer (con almeno uno strato nascosto) con funzioni di attivazione lineari sarà in grado di risolvere il problema risolto da una rete single layer.

Con il simbolo cerchiato in rosso si indica che ciascuno di questi neuroni ha una funzione di attivazione lineare. B

? Come possiamo descrivere al nostro calcolatore un architettura di rete neurale così che lui la possa vedere e usare per produrre un risultato?

Dovrà conoscere il numero di neuroni per ciascuno strato, dovrà conoscere la funzione di attivazione e dovrà conoscere i pesi calcolati dalla fase di addestramento.

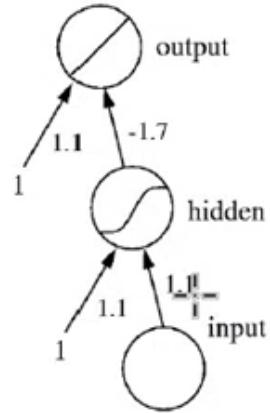
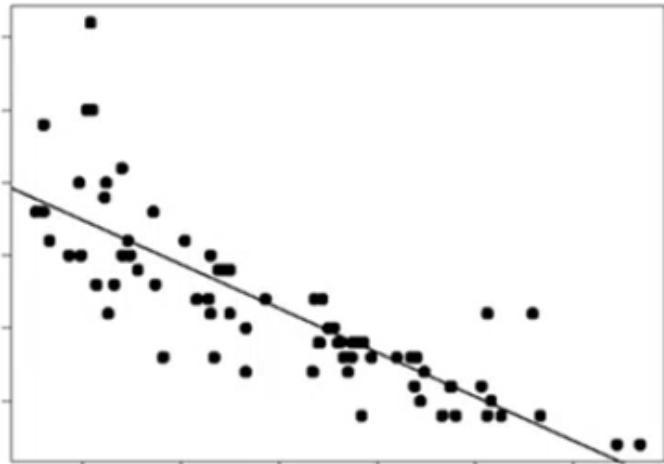
Dovremo usare tante **matrici** quante sono le coppie di strati adiacenti. In questo caso abbiamo una matrice W_{21} che rappresenta i pesi tra lo strato di ingresso e lo strato nascosto con dimensione $p \times n$.

Poi la W_{32} che è $q \times p$, è possibile indicare l'uscita di questa rete dunque come applicazione della funzione di attivazione dei neuroni di uscita agli ingressi che in questo caso è la funzione lineare dunque l'uscita z prodotta dalla rete è il prodotto della matrice W_{32} per l'uscita del livello nascosto la quale a sua volta è espressa come il prodotto di W_{21} per X con X vettore di ingresso alla rete. W_{32} e W_{21} sono compatibili in termini di prodotto (in quanto la prima è $q \times p$ e la seconda è $p \times n$) ottenendo una matrice $q \times n$ e costruendo dunque una rete con un solo livello senza livelli nascosti, dunque perchè si possa avere una rete multilayer serve una funzione di attivazione non lineare, perchè se due strati consecutivi hanno neuroni con funzione di attivazione lineari li posso collassare in uno solo.

! Per reti multilayer una funzione di attivazione non lineare è richiesta ma per quello detto poco fa, questa funzione di attivazione deve essere anche continua e derivabile per poter applicare la back propagation.

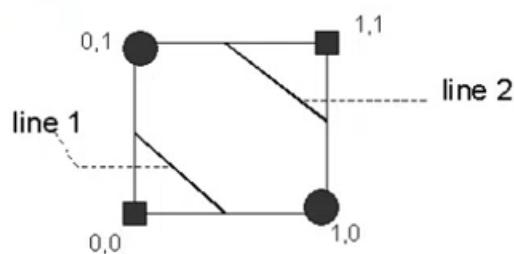
Ecco spiegato il motivo per cui nella rete indicata in basso sono mostrati i pesi come risultato del processo dell'applicazione dell'algoritmo di addestramento, in quanto qui è presente una funzione non lineare continua differenziabile...

- Consider again the following scatter plot and the best linear model found by gradient descent. The data are not evenly distributed around the line.
- We can get a better approximation by using a hidden layer, consisting of a single neuron with a tanh function.



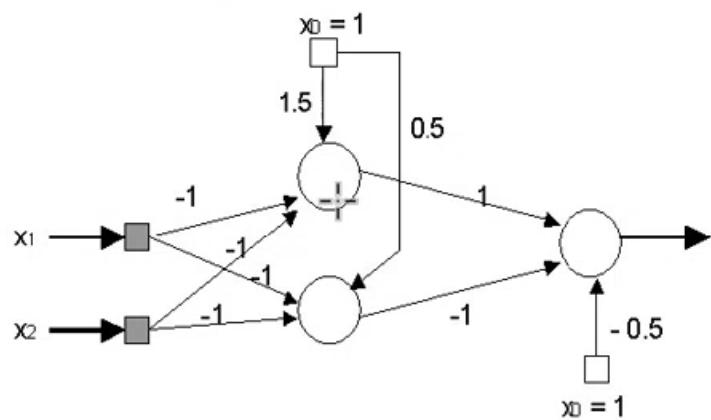
...mentre nel caso dello XOR abbiamo stabilito a mano questi pesi in quanto non abbiamo il rispetto del vincolo di differenziabilità (la funzione è a gradino!):

The XOR problem solved with two separating lines



- The network needs two neurons, both fed with two inputs, to represent the two lines, and a third neuron to combine the information from these two lines.

- We use a threshold function.



Dunque anzichè usare una funzione a soglia, dobbiamo usare al posto di una funzione soglia una funzione sigmoide o tangente iperbolica.

Creazione di un training set



Creating a training set

- Creating the training set is a crucial step.
- This includes raw data collection, data analysis using statistical techniques, variable selection, and data preprocessing (for example, normalizing inputs and targets in order to make all variables comparable to each other).
- Another important action to take for each input variable is to identify and remove the *outliers*.
- Two other typical actions are the management of *missing values* of a given variable and the management of *non-numeric data*.
- Finally, the problem of *unbalanced data* must be managed.

Creare un training set è un passo cruciale in quanto la rete riesce a imparare da quello contenuto all'interno del training set. Le operazioni incluse nella creazione di un training set includono le seguenti:

- collezione di dati raw raccolti dallo specifico dominio applicativo
- fare un analisi dei dati usando delle tecniche statistiche, ovvero avere un'idea della distribuzione dei dati
- le variabili usate per rappresentare i dati in ingresso devono essere opportune ovvero ciascuna variabile che decido di dare in ingresso deve essere correlata all'uscita in quanto lo scopo è di addestrare la rete a trovare un mapping tra ingresso e uscita. I parametri di un modello non devono essere troppi rispetto a quello che la rete deve imparare (ad esempio evitare di avere dati in ingresso simili che hanno una correlazione, oppure usare variabili normalizzate, dato che la rete calcola sulla base di prodotti e somme pesate e dunque interpreterebbe una variabile in modo maggiore rispetto alle altre).

? Quanti esempi di addestramento servono?



How many training examples?

- The training set must be a representative sample of the data the network will work on.
- Large training sets reduce the risk of undersampling the underlying function. If the training set is too small, the network can learn it perfectly but fail in the final application.
- In practice, the number of cases required for training depends on several factors, such as the size of the network and the distribution of inputs and targets. In particular, a large network usually requires more training data than a small one. In fact, there is no specific recipe. Some heuristic guidelines say there should be 5 to 10 training samples for each weight.

Grandi insiemi di addestramento sono ben accetti in quanto riducono il rischio di sottocampionamento tra la funzione di mapping e la funzione di uscita. Se il training set è troppo piccolo ho un rischio di overfitting, ovvero impara 1:1 a mappare ciascuno dei pochi ingressi nel corrispondente output desiderato.

Impara a fare una specie di associazione automatica senza però imparare il mapping !!

L'addestramento si ferma quando una qualsiasi delle seguenti condizioni si verifica:



Training stopping condition

- Training stops when
 - a certain number of epochs pass (an *epoch* is the presentation of all the training data),
 - the error reaches an acceptable level,
 - the error stops improving.

Quando si tratta con le reti non esiste solo il processo di addestramento ma vogliamo che impari a generalizzare, ovvero gli esempi di addestramento devono servire per permettere alla rete di riconoscere e classificare altri dati di test, che sono quelli tramite i quali si misura la capacità di generalizzazione della rete. Il test set consiste di coppie di ingresso-uscita desiderata che non sono mai state viste dalla rete durante l'addestramento:

Training and testing

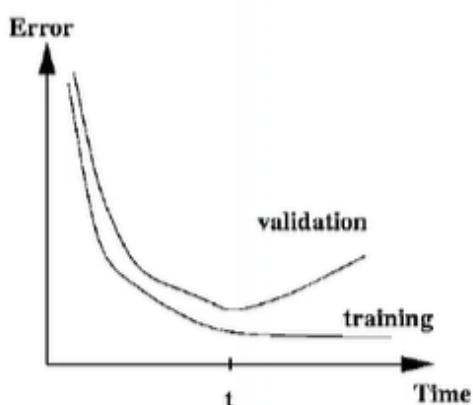
- During training, a neural network changes its weights repeatedly, epoch after epoch, to improve its performance.
- After training, we measure the *generalization* capacity of the network by testing it with an independent data set. This set, called the *test set*, consists of pairs (input, desired output) never seen by the network during training.
- During the test, the network passes the test samples forward through itself and calculates a performance index, such as the mean squared error, without changing the weights.
- Therefore, the available data are divided into two parts, for example, two thirds for training and one third for testing.

In realtà gli insiemi da considerare non sono 2 ma 3. Dobbiamo evitare che la rete riproduca questo processo di "imparare a papagallo solo gli esempi di addestramento" senza imparare alcun modo di generalizzare e per farlo abbiamo bisogno di un **terzo** data set indipendente detto **validation set**.

Early stopping (1)

In order to prevent inappropriate memorization of input data (also called *overtraining*, *overlearning* or *overfitting*), we need a third independent data set, called *validation set*, to be used during training to verify that the network is not 'overlearning'.

We can plot errors (e.g., MSE) on training and validation sets.



Both errors typically fall quickly at the start of training as the network moves its weights away from their original random positions. Over time, both curves become flatter. Typically, the training set error continues to decrease, but the validation set error eventually begins to increase.

Per capirlo in modo intuitivo proviamo a rappresentare l'errore fatto dalla rete durante l'addestramento

epoca dopo epoca e vediamo che man mano che presentiamo gli esempi, l'errore descresce e continua a decrescere. Dunque se uso gli esempi di addestramento per modificare i pesi ma uso anche un piccolo insieme di esempi tratti dalla stessa distribuzione da cui ho preso l'insieme di addestramento, alla fine di ogni epoca calcolo anche l'errore con quei pesi che ho determinato usando solo gli esempi di addestramento, dato che sia il training set che il validation set sono degli esempi rappresentativi della stessa realtà/distribuzione mi devo aspettare che l'andamento dell'errore sui due insiemi sia coerente. Ma se epoca dopo epoca arrivo a un certo punto in cui l'errore sul training continua a decrescere epoca dopo epoca ma l'errore sul validation sale, vuol dire che la rete si è intestardita ad imparare 1:1 solo le features del training set, allora bisogna fermare l'addestramento all'epoca con il valore minimo di errore per il validation set.



Early stopping (2)

- This increase shows that the network has stopped learning what training samples have in common with validation samples and has begun to learn meaningless differences. This overfitting of the training data harms the network's ability to generalize, as it merely memorizes the noise in the training data. In other words, as the training set error decreases, the test set error increases.
- For the best generalization, training stops when the validation-set error reaches its lowest point (*early stopping*).

Una rete grande può essere allenata solo in presenza di un insieme di dati di training molto grande, altrimenti se non rispettiamo la regola di massima vista in precedenza (da 5 a 10), troppi pesi sono uno svantaggio in quanto la rete li può usare per memorizzare i dati di training anzichè generalizzare, e



Network's size

- A large network has many weights that can be used to model a function. This can be an advantage for complex functions if we have sufficient training data. Otherwise, too many weights can be a disadvantage, because the neural network can use them to memorize training data. On the other hand, if the network is too small, it cannot learn the problem at all.
- The key is to find a network large enough to learn how to solve the specific problem but small enough to generalize well. The best choice is the network with the minimum number of weights needed to accurately process the test data. A good strategy is to start with a few hidden neurons and increase the number while monitoring the generalization by validating at each epoch.

18/03/2023

Per evitare l'overfitting abbiamo visto che è necessario introdurre il validation set che è un insieme di esempi usato per controllare la condotta della rete durante l'addestramento e quando si verifica l'errore minimo sul validation set durante l'addestramento dunque i valori dei pesi saranno settati uguali a valori che avevano nel punto di minimo errore. La rete addestrata viene poi testata su un terzo insieme che è il test set.



K-fold cross-validation



- leave-one out
- stratified
- iterated (repeated)

Abbiamo detto che usiamo i dati del validation set per decidere quando fermare l'addestramento ed è un discorso che va bene quando abbiamo tanti dati. Ma quando i dati sono pochi il punto in cui fermiamo l'addestramento dipende dalla scelta casuale dei campioni utilizzati per comporre il validation set. Non possiamo limitarci ad una sola ripetizione di questo fenomeno.

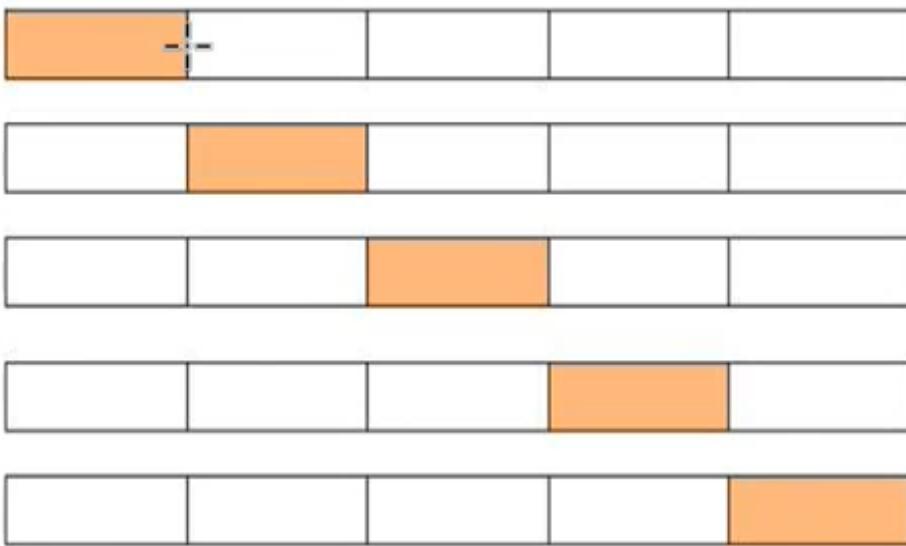
Si selezionano allora alcuni dei dati per comporre il test-set finale, i dati rimanenti vengono sottoposti alla K-fold cross validation, ovvero suddivisi in modo casuale in K gruppi detti fold che hanno la stessa dimensione e si ripete per K volte la seguente operazione:

si prende ad esempio il primo fold come test-set e i rimanenti come training set e si addestra le reti neurali su questo insieme di 4 fold lasciando fuori l'insieme di test evidenziato in arancione. Lo score dunque la valutazione del modello della rete neurale è dato dalla media dell'errore in generale sui 5 training set e la media dell'errore sui 5 test-set.

Nelle K ripetizioni della cross validation si testa **un unico modello e un unica rete** ovvero si fa riferimento alla stessa combinazione di valori degli iper-parametri (numero di strati nascosti, numero di neuroni nei vari strati nascosti, ...), ciò che cambia sono solo i 2 insiemi di addestramento e test. In genere si usa la K-fold cross validation per individuare la combinazione migliore di iperparametri, poi una volta individuata verrà riaddestrata usando tutti i K-folds e i valori alla fine di questo addestramento saranno i valori dei pesi.

In questa situazione il validation-set non viene usato in quanto è poco popolato. Quando i dati sono davvero pochi infatti esiste la metodologia **leave-one out** che suddivide i dati in fold da **1 solo elemento**.

La **stratified** parte invece da un insieme che ha una distribuzione sbilanciata permette di ottenere fold bilanciata. Infatti guardando tale figura:



ogni fold è composto estraendo casualmente dati dal dataset originale, potrei però così facendo ottenere fold sproporzionati.

Iterated (Repeated) invece è quando ho pochi dati ma devo ottenere un modello molto accurato, applico più volte la K-fold cross validation mescolando i dati all'inizio di ogni validazione.

⚠ Empiricamente si può dimostrare che con $K = 10$ si hanno dei buoni risultati.

Contrastare l'overfitting



Preventing overfitting (1)

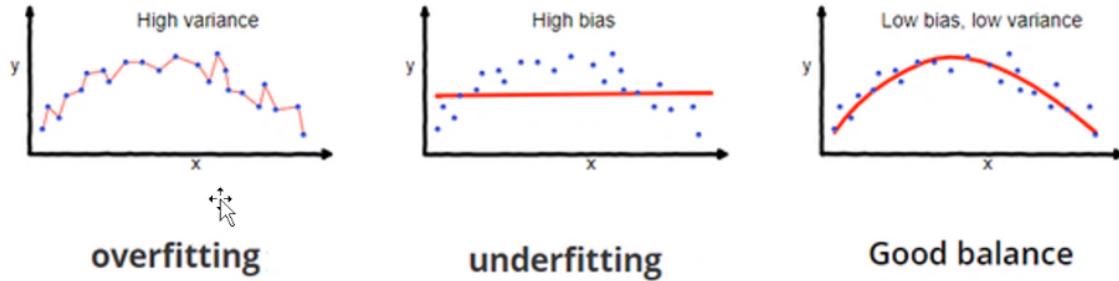
- **Early stopping**
Use a validation set.
- **Gathering more training data**
Increase the size of the training data.
- **Performing dataset augmentation**
Artificially increase the size of the dataset by introducing different types of transformations or distortions of the available data.
- **Reducing the capacity of the network**
Reduce the size of the network.

- **early stopping**: impedisce che la rete impari "a pappagallo". La rete magari si comporta in questo modo però perchè è troppo potente rispetto a quello che deve imparare, potrebbe avere ad esempio troppi pesi rispetto ai pochi dati che deve apprendere e quindi aumentando il numero di dati...
- **gathering more training data**: si può dunque aumentare il numero di dati di addestramento in modo artificiale. Ma non sempre questo è possibile pertanto...

- **performing dataset augmentation:** supponiamo ad esempio di avere come dati di addestramento delle immagini. Si possono introdurre delle traslazioni di pochi pixel, delle rotazioni, dei flipping delle immagini così da aiutare la rete a riconoscere quelle immagini modificate
- **reducing the capacity of the network:** potremmo inoltre diminuire la capacità della rete

oppure...e le seguenti 2 tecniche sono le più importanti

- **regolarizzazione dei pesi:**



in questi 3 grafici la x rappresenta l'ingresso alla rete semplificata la y l'uscita. I cerchi blu rappresentano le uscite di una funzione che una rete deve approssimare e di cui conosciamo l'uscita, dunque possiamo graficarli. Usiamo questi punti come set di addestramento e addestriamo la rete a produrre le uscite y in modo tale da diminuire l'errore ovvero la differenza tra l'uscita effettiva della rete e quella desiderata. Se esiste overfitting e se grafico l'uscita della rete addestrata, questa rete approssima bene i punti blu, commettendo un errore molto basso o addirittura nullo.

?

Come posso quindi interpretare l'andamento dell'uscita della rete?

Si introduce la **varianza** e il **bias**.

- La varianza esprime quanto sono distribuiti i valori di uscita della rete. Se ad esempio variano di molto oppure le variazioni sono minime.
- Il bias invece è definito come l'errore commesso dalla rete ed è la differenza tra l'uscita desiderata e l'uscita effettiva della rete.

Nel caso di overfitting si ha alta varianza e basso bias. Con gli stessi campioni di addestramento blu possiamo addestrare la rete e avere nel caso peggiore una linea continua che rappresenta il caso in cui la rete non ha imparato nulla in quanto è in underfitting perché non ha abbastanza pesi per costruire un modello. In questo caso abbiamo varianza nulla ma un alto bias. Per evitare queste 2 situazioni estreme di over/under-fitting dovremmo avere una curva come quella rappresentata in figura (Good balance) andando ad aumentare la capacità della rete permettendole di avere più pesi.

?

Come è possibile avere una bassa varianza e avere una rete che produce un errore nullo per tutti gli esempi di addestramento?

Preventing overfitting (2)

■ Adding weight regularization

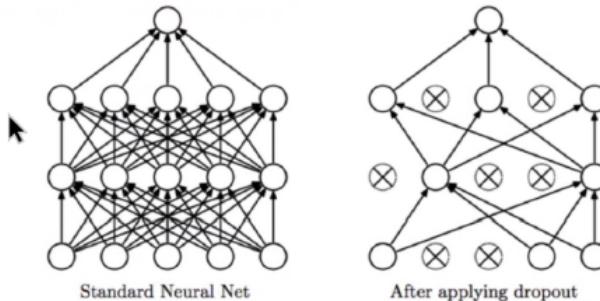
Force the weights to take small values by adding a penalty that penalizes large weights to the loss function.

$$\text{L2 regularization: } J_{Reg} = \frac{1}{2} \lambda \sum_i w_i^2$$

$$\text{L1 regularization: } J_{Reg} = \lambda \sum_i |w_i|$$

$$\text{E.g., } J_{Tot} = J + J_{Reg} \quad \frac{\partial J_{Reg}}{\partial w_k} = \frac{\partial}{\partial w_k} \left(\frac{1}{2} \lambda \sum_i w_i^2 \right) = \lambda \cdot w_k$$

■ Dropout



58

Prendendo il caso di overfitting come riferimento, modificando poco un input alla rete, la rete modifica molto l'uscita. **Bisogna forzare i pesi ad assumere valori piccoli** e lo scopo di un algoritmo di addestramento è quello di minimizzare una funzione di costo (o di loss, ovvero di errore). Questa funzione indicata con J_{Tot} viene modificata aggiungendoci un termine che penalizza pesi con valori alti (J_{Reg}). La funzione di loss da minimizzare sarà dunque quella scelta in funzione del problema da risolvere più un termine che penalizza i grandi pesi quindi espresso nella regolarizzazione L2 dalla somma dei quadrati dei pesi nella rete, $1/2$ per semplificare i calcoli e poi è presente il parametro λ che a seconda del valore che assume rappresenta l'importanza che diamo a questa penalty. In pratica il processo di addestramento consiste nella minimizzazione della somma di questi 2 termini e se diamo a λ un valore più grande rispetto ad un valore più piccolo vuole dire che diamo un importanza maggiore alla minimizzazione di J_{Reg} rispetto a J_{Tot} .

⚠ Se ci troviamo in una sola dimensione è facile vedere l'andamento in termini di varianza e bias, ma se ci troviamo in uno spazio R^N è più complesso, pertanto l'overfitting da un punto di vista pratico si riconosce guardando l'**errore sul training set** (alto) e l'**errore sul test set** (basso). Per l'underfitting bisogna invece guardare l'**errore sul training set**, se è alto allora c'è underfitting.

- **Dropout:** la rete è completamente connessa come mostrata in figura. Per ciascun livello della rete si definisce una percentuale di neuroni che durante la fase di presentazione di un esempio di addestramento vengono **temporaneamente spenti**, non prendendo parte all'esecuzione di un passo della back-propagation relativo alla presentazione di un esempio. Poi si ripristina l'architettura precedente, casualmente scelgo la stessa percentuale di neuroni a cui applicare il dropout, e per questo nuovo esempio di addestramento userò questo nuovo layout, e così via. Il drop out si usa solo sul training set, non sul validation set o test set. Il senso è che ciascun neurone è forzato ad imparare le features che sono davvero importanti che non dipendono da chi fornisce le risposte dai precedenti neuroni. Dunque esiste uno spingere i neuroni verso un comportamento più robusto.

Alla fine la rete avrà il valore dei pesi prodotto dalla "collaborazione" di ciascuna possibile rete ciascuna con un insieme diverso di neuroni che hanno preso parte alla back propagation considerando ogni presentazione di ogni singolo esempio di addestramento. Il dropout può essere specificato a tutti i livelli specificando un dropout rate. E' opportuno non applicarlo al livello di uscita perchè l'errore della rete che è poi la guida dell'algoritmo di apprendimento si basa su ciò che la rete da in output, togliendo dei neuroni, non avremmo un uscita corretta.



Multi-class classification

$$\text{Softmax function } y_j = \frac{e^{net_j}}{\sum_k e^{net_k}}$$

$$\text{Cross-entropy } H(p, q) = -\sum_x p(x) \log(q(x))$$

p(x) = true distribution,
q(x) = estimated distribution

When used as loss function

$$L = -\sum_{i=1}^C t_i \log(y_i) = -t \cdot \log(y)$$

If t is a one-hot vector:

$$t = [0 \ 0 \ \dots \overset{k}{\overbrace{1}} \ \dots \ 0] \quad L = -\log(y_k)$$

k = correct class, y_k = estimated probability for the correct class

minimum value of L : 0, maximum value: infinite

Come addestrare una rete che deve effettuare classificazione?

Nel caso di multi-class classification, lo strato di uscita della rete ha tanti neuroni quante sono le classi, ogni neurone produce un uscita rispetto alla classe assegnatagli. In questi casi la funzione di attivazione è la **Softmax**. Si calcola il rapporto per tutti i neuroni di uscita il quale produce un numero positivo minore di 1 tale che la somma sia minore di 1. Ovvero all'uscita della rete neurale riceviamo una distribuzione di probabilità, ovvero ogni singola uscita di ogni singolo neurone di uscita può essere interpretata come la stima della rete che l'ingresso presentato alla rete appartenga alla classe associata al neurone. Vorremmo che la distribuzione di probabilità della rete fosse quella desiderata. Bisogna calcolare la distanza tra 2 distribuzioni di probabilità ovvero la **Cross-entropy** che è la differenza tra due distribuzioni discrete (x è una variabile discreta), una vera $p(x)$ e una stimata $q(x)$.

I valori della variabile discreta x sono i valori delle classi, supponiamo di avere c classi dunque c neuroni di uscita

t è un one-hot vector ovvero un vettore con tutti 0 e un solo 1 in corrispondenza della classe corretta pertanto la loss, nel caso si usi come uscita desiderata un one hot vector, è data da $-\log(y_k)$.

Il valore minimo di questa loss sarà quando il neurone di uscita k -esimo mi da 1 e dunque la loss si azzera, ma più si allontana il valore y_k da 1, dunque più si avvicina a 0, più è infinita.

La classificazione binaria basta considerare quello detto fin ora con due sole classi. Nella classificazione binaria si trova spesso una rete neurale con un livello di uscita formato da due neuroni la cui funzione di attivazione è la Softmax:

Binary classification

- multi-class problem with two classes
- a single output neuron with sigmoid activation function:

$$H(t, y) = -(t \log(y) + (1 - t) \log(1 - y))$$

I

oppure si possono trovare anche singoli neuroni di uscita con la funzione Sigmoid. La loss in quel caso è la cross entropy adattata al caso, ovvero sarà il prodotto della probabilità desiderata per la prima classe t con il $\log(y)$ ovvero unica uscita effettiva di quell'unico neurone + la probabilità della seconda classe $(1-t)$ moltiplicata per il logaritmo dell'uscita effettiva della seconda classe ovvero $\log(1-y)$.

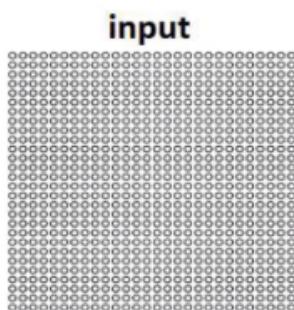
Reti Neurali Convoluzionali (CNN)

Convolutional Neural Network (CNN)

CNNs combine three architectural ideas:

- local receptive fields
- shared weights and biases
- pooling

Example



- Input: square of neurons whose values are pixel intensities.

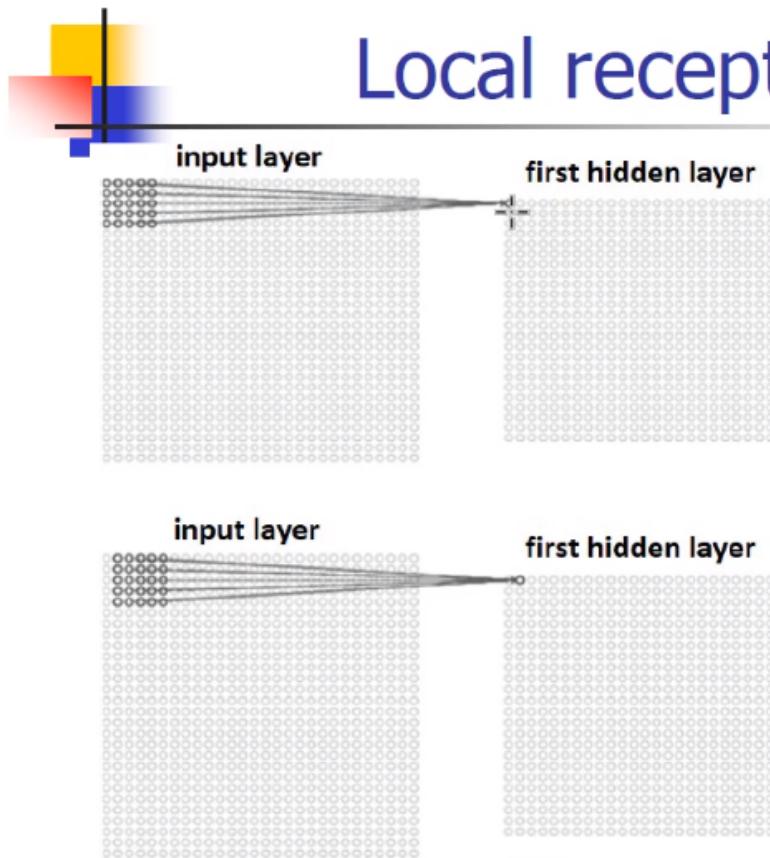
effetto che abbiamo quando noi osserviamo un'immagine. Se voglio sviluppare una rete che riconosce le lettere dell'alfabeto devo ingegnarmi ad usare delle features tramite le quali rappresento le lettere. Vorrei invece limitarmi a presentare un'immagine in ingresso alla rete e lasciare che sia la rete a trovare automaticamente le features necessarie per risolvere il problema.

Combinano 3 idee architetturali:

- campi recettivi locali
- pesi e bias condivisi
- pooling

Supponiamo che l'input non sia un vettore di elementi ma un quadrato di neuroni che rappresentano un'immagine in una scala di grigi. Dunque l'ingresso alla rete è un quadrato di neuroni che corrispondono alle intensità di grigio in ingresso alla rete.

Andiamo a introdurre i campi recettivi locali con il tentativo di costruire il primo livello nascosto:



- Each neuron of the first hidden layer is connected to a small region, called *local receptive field* (e.g., 5x5) of the input space
→ strong reduction in the number of connections.
- For each hidden neuron: 5x5 weights + 1 bias = 26 parameters.

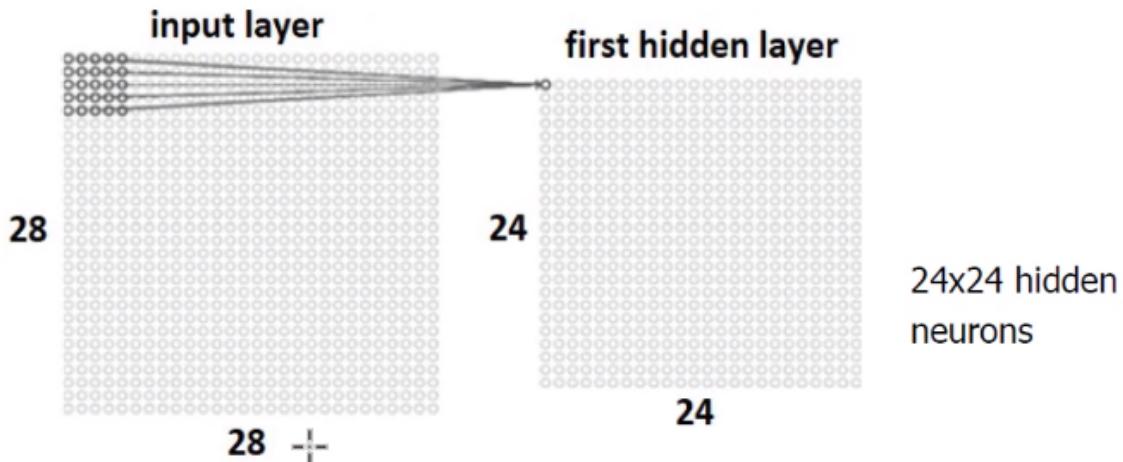
3

Il primo neurone del primo livello nascosto non lo connettiamo a tutti i neuroni del livello precedente (non stiamo parlando infatti di strati completamente connessi), ma lo connettiamo a un quadrato 5x5 e questa piccola regione è detta local receptive field, in questo modo un neurone ha un numero di connessioni molto più basso.

Il secondo neurone lo connetto ad una piccola regione ma è spostata ad esempio di un pixel verso destra, e vado avanti così. Dunque ciascun neurone di ogni strato nascosto è connesso a un local receptive field

Dunque ho una forte riduzione rispetto al numero di connessioni e ciascun neurone ha queste connessioni le quali sono associate a dei pesi:

- We slide the local receptive field (e.g., by one pixel) across the input image.



ciascun neurone è connesso a 5x5 neuroni quindi 5x5 pesi più un bias dunque i pesi ovvero i parametri del modello sono 26 per ogni neurone.

Dopo il primo neurone e il secondo neurone continuiamo su tutto il primo strato per costruire il primo strato nascosto e facciamo scorrere il local receptive field di un pixel alla volta attraverso l'immagine andando da sinistra a destra poi si scende di un pixel e così via fino ad avere come risultato che partendo da un'immagine 28x28 si produce un primo livello nascosto di 24x24 neuroni, come mostrato nell'immagine.

La seconda idea architettonale sono gli **shared weights and biases**. Ciascun neurone del primo livello nascosto in questo caso ha 26 connessioni che sono associate a dei pesi. **Si usano gli stessi pesi e lo stesso bias per tutti i neuroni nascosti.**

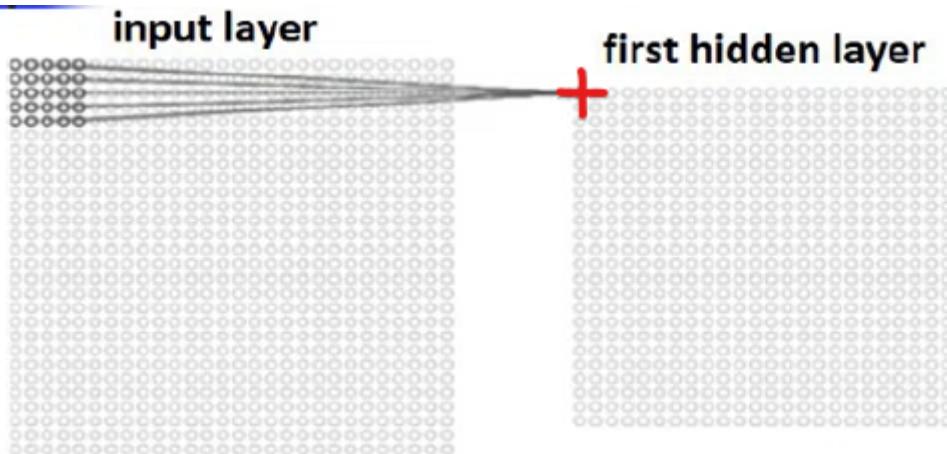
Shared weights and biases

- We use the same weights and the same bias for all hidden neurons. This means that all neurons in the first hidden layer detect the same feature, but in different positions of the input image
 - ➔ significant reduction in the number of weights.
- Shared weights and bias define a *filter* (or *kernel*).
- The filter convolves the input image and forms a *feature map*.
- The layer is called *convolutional layer*.

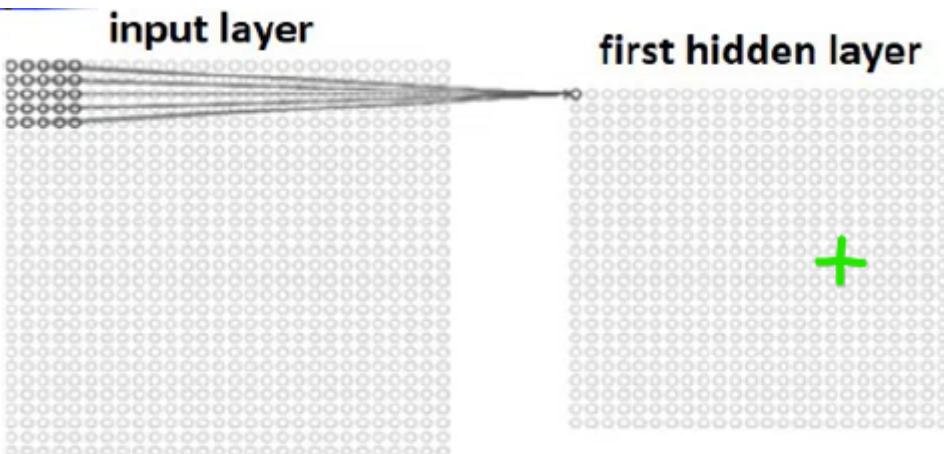
❓ Cosa vuol dire?

Ciascuno dei neuroni del livello nascosto calcola una funzione di attivazione alla somma pesata degli ingressi.

Il neurone indicato in **rosso**:

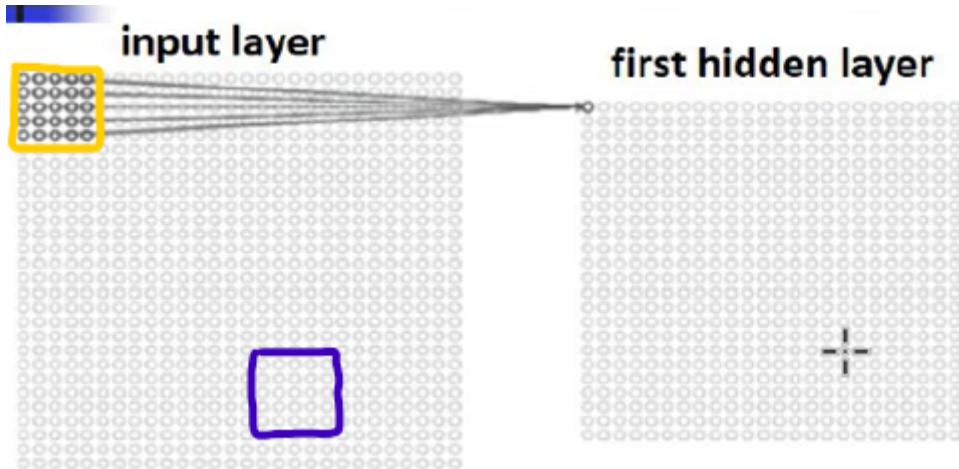


calcola la funzione di attivazione f alla somma pesata degli ingressi dunque ciascun pallino del local receptive field è moltiplicato per il peso associato alla connessione relativa. Ma anche il neurone indicato in **verde**:



applica la stessa funzione di attivazione alla somma pesata degli ingressi, ma se gli ingressi sono

diversi il risultato è diverso, ma se gli ingressi fossero uguali l'uscita del neurone sarebbe la stessa, ovvero se il pattern indicato in **giallo** è uguale a quello indicato in **viola**:



I'uscita del neurone collegato al pattern giallo sarà uguale all'uscita del neurone collegato al pattern viola! Ovvero i neuroni sarebbero in grado di rilevare esattamente la stessa feature anche se in posizioni diverse della stessa immagine!

- Questo comporta una riduzione significativa del numero di pesi, in quanto per tutti i neuroni dello stesso strato, abbiamo esattamente gli stessi pesi.

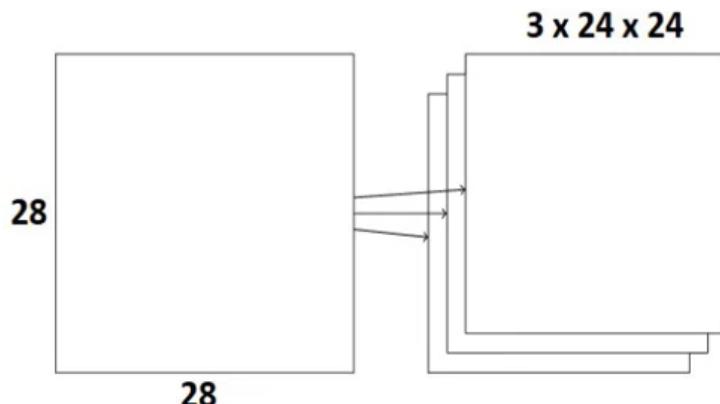
Questi pesi e bias condivisi definiscono un **filtro** o **kernel**, questo filtro viene fatto scivolare da sinistra a destra e dall'alto verso il basso e mentre facciamo muovere questo filtro attraverso l'immagine, l'operazione che viene compiuta dalla rete è la stessa indipendentemente dalla specifica posizione in cui poniamo il filtro, ovvero stiamo effettuando la convoluzione del filtro e dell'immagine di input.

Il primo neurone del primo livello nascosto produce un risultato a seguito del calcolo della funzione di attivazione sulla somma pesata degli ingressi. Poi facendo scorrere il filtro a destra di un pixel, rifaccio la computazione e produco un risultato per il secondo neurone, ovvero costruisco uno strato che è detto feature map che è una mappa bidimensionale in cui ciascun valore rappresenta la presenza o assenza di una specifica feature nel quadratino che rappresenta il local receptive field nel layer precedente. Il filtro effettua dunque questa operazione di convoluzione con l'immagine di input e forma una feature map. Questo primo strato nascosto viene infatti detto *strato convoluzionale*.

Tipicamente non ci accontentiamo di una feature map sola, nel senso che non ci interessa rilevare solo un segmento verticale nella immagine di input, magari vogliamo un segmento orizzontale, dunque in generale più features map sono considerate. Supponiamo che ci interessi considerare 3 feature map, vuol dire che il primo livello nascosto consiste di 3 feature map ciascuna delle dimensioni di 24x24:



In general, multiple feature maps are considered (e.g., 3):



Each feature map is the result of a convolution that uses a different set of weights and a different bias. The number of feature maps equals the number of filters. In this case, the network can detect 3 different features, each of which can be detected in the whole image.

Ovvero ogni feature map è il risultato dell'applicazione della convoluzione tra l'immagine di input e un filtro particolare.

Nel caso della figura la rete può rilevare 3 features differenti ciascuna delle quali può essere rilevata nell'intera immagine.

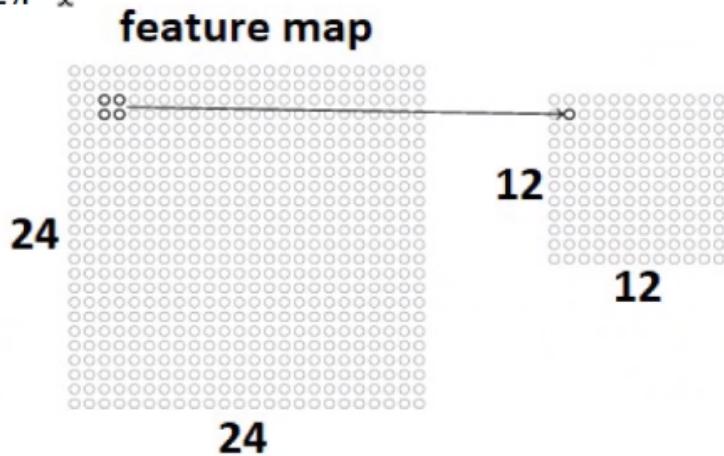
⚠ La cosa interessante è che questi filtri non li dobbiamo decidere noi ma saranno generati automaticamente dal processo di addestramento.

Abbiamo costruito un primo strato nascosto che in questo caso è fatto da 3 features map, le quali hanno tanti elementi, dunque tipicamente dopo un livello convoluzionale si introduce una feature map riassunta di **pooling**:

Pooling

A pooling layer is usually used immediately after a convolutional layer: for each feature map of the convolutional layer, the pooling layer produces a condensed feature map.

E.g., in max-pooling, a pooling unit provides the maximum activation in an input region (e.g., 2x2). $\boxed{1}$

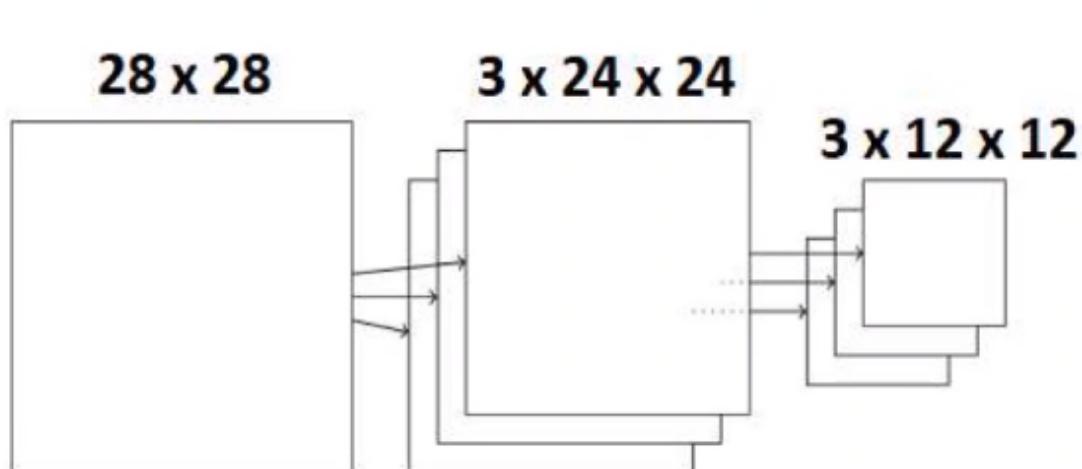


7

ad esempio in questo caso avendo una feature map di 24x24 (questo discorso vale per ciascuna delle 3 indipendentemente) questa può essere condensata applicando il pooling, ovvero un livello di neuroni successivo a quello convoluzionale che, in questo caso ad esempio, fornisce l'attivazione massima in una regione di input piccola (ad esempio 2x2 come in figura) **riducendo le dimensioni della feature map senza perdere di significatività**.

Se sono nella situazione sotto rappresentata in cui applico il max-pooling e ogni feature map la devo condensare in modo separato dalle altre, passo dal primo strato nascosto convoluzionale al primo strato nascosto di pooling che ha una dimensione di $3 \times 12 \times 12$.

In the case of 3 feature maps, we apply max-pooling to each feature map separately:

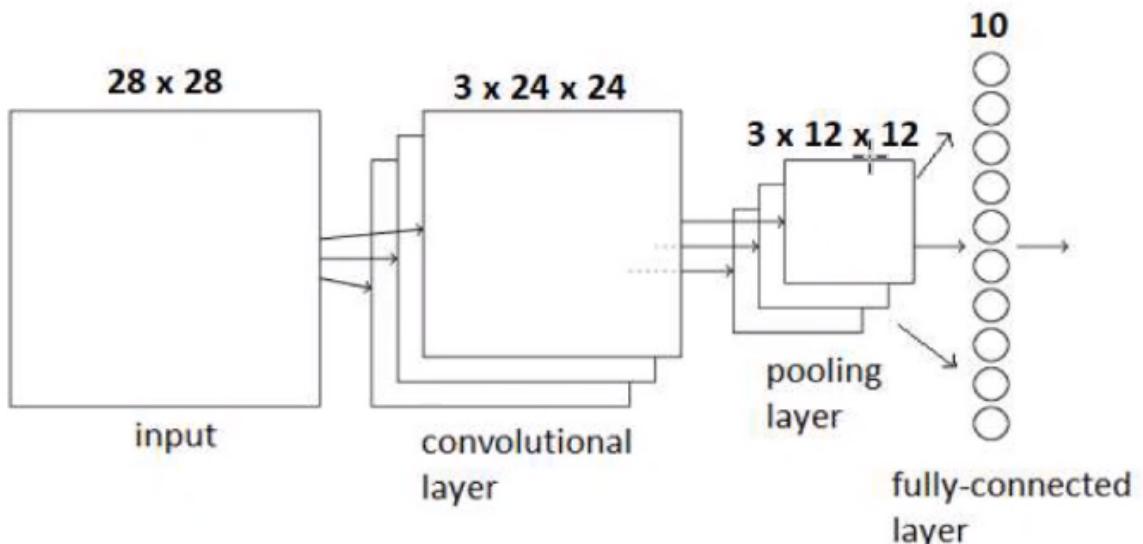


Supponiamo che voglia classificare le immagini in una di 10 classi possibili, mi basta prendere quello costruito fin ora e attaccarci uno strato di uscita di 10 neuroni, ovvero 10 classi con 10 neuroni aventi la funzione softmax dai quali leggo la distribuzione di probabilità delle 10 classi, ovvero che l'immagine di input appartenga alla n-esima classe. L'ultimo livello non può sfruttare come i due livelli precedenti delle connessioni locali, ha bisogno infatti di vedere tutte e 3 le mappe, dunque abbiamo un FULLY CONNECTED LAYER in cui ciascun neurone di uscita riceve gli ingressi da tutte e 3 le mappe, detto in termini più operativi si effettua una operazione di flattening, si trasformano queste 3 mappe in un unico piano estraendo da ciascuna mappa una singola colonna ad esempio e ciascun neurone riceve ingressi da ciascuna feature map così facendo.



The complete CNN

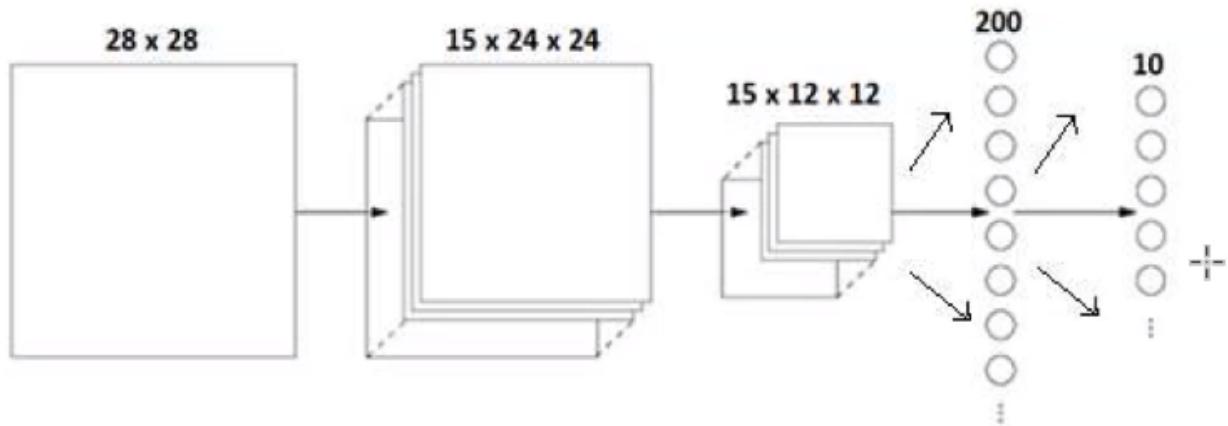
Suppose we want to develop a 10-class classifier:



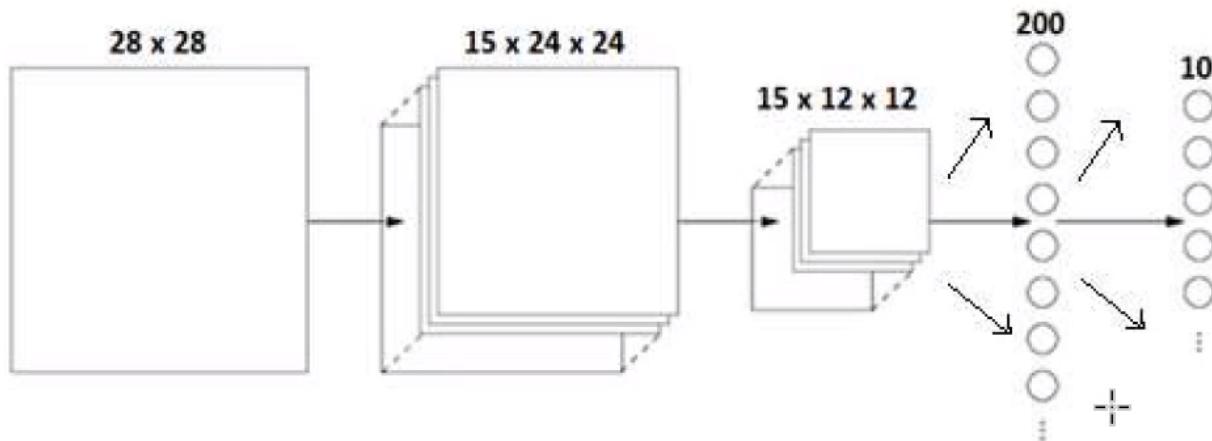
potrebbero però interessarmi ad esempio non solo 3 filtri ma 15 come nell'immagine sottostante:



Another possibility:



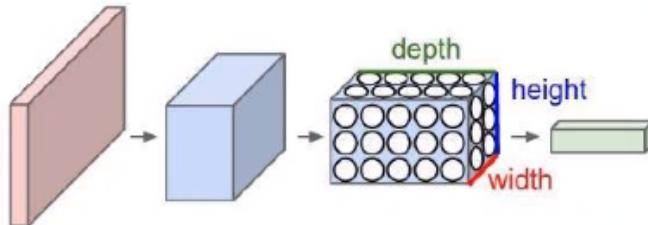
Volendo dunque possiamo aumentare il numero di livelli completamente connessi alla fine dell'architettura della rete ma possiamo anche andare oltre. Nel senso che la coppia di *livello convoluzionale* e successivo *livello di pooling*:



potremmo inserire altre coppie prima dei livelli completamente connessi ovvero l'architettura di una CNN consiste di una gerarchia di livelli dove l'input è connesso ai pixel dell'immagine e gli ultimi livelli sono generalmente completamente connessi e implementano un classificatore MLP standard:

Architecture of a CNN

A CNN consists of a hierarchy of layers. The input layer is connected to the image pixels, the last layers are generally fully connected and implement an MLP classifier, the intermediate layers use local connections and shared weights.



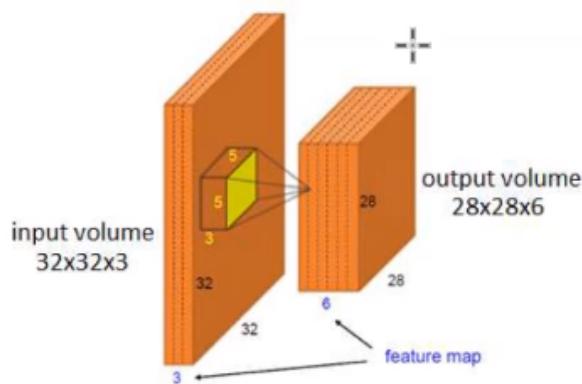
width and *height* represent the spatial organization of the feature map, *depth* identifies the different feature maps.

andando dall'ingresso verso l'uscita i livelli prima di quelli completamente connessi sono volumi con 3 dimensioni, che è vero anche per l'immagine di ingresso in quanto le immagini sono RGB dunque è come se fossero 3 mappe per ciascun pixel a differenza di un'immagine della scala di grigi che invece ha una profondità pari a 1.

Se supponiamo di avere 2 livelli consecutivi, questa coppia include un volume di input e uno di output:



For each pair of consecutive layers, there is an input volume and an output volume:



$$\# \text{ connections: } (28 \times 28 \times 6) \times (5 \times 5 \times 3) = 352,800$$

$$\# \text{ weights: } 6 \times (5 \times 5 \times 3 + 1) = 456$$

perchè la profondità è il numero di feature maps. Se fosse un'immagine RGB avrebbe una profondità pari a 3 come nell'immagine.

? Quante sono le connessioni?

Per ogni singolo neurone sono $5 \times 5 \times 3$ ovvero ogni neurone è collegato al local receptive field dello strato precedente ma il local receptive field è come se fosse composto da 3 campi paralleli una per ciascuna delle 3 feature maps. I neuroni nel volume di output sono $28 \times 28 \times 6$.

? Quanti sono i pesi?

Abbiamo detto che ogni feature map usa un solo filtro, un solo insieme di pesi, ovvero per ciascuna feature map i pesi sono tanti quante sono le connessioni di ciascun neurone, dunque $5 \times 5 \times 3 + 1$ che è il bias il tutto per 6.



- **Activation function**

rectified linear unit (ReLU) $f(x) = \max(0, x)$



- **Stride**

When a 3D filter slides over the input volume, instead of moving with a unitary step (i.e., one neuron at a time), a longer step (*stride*) can be used. This reduces the size of the feature maps, but there may be a slight penalty in accuracy.

- **Padding**

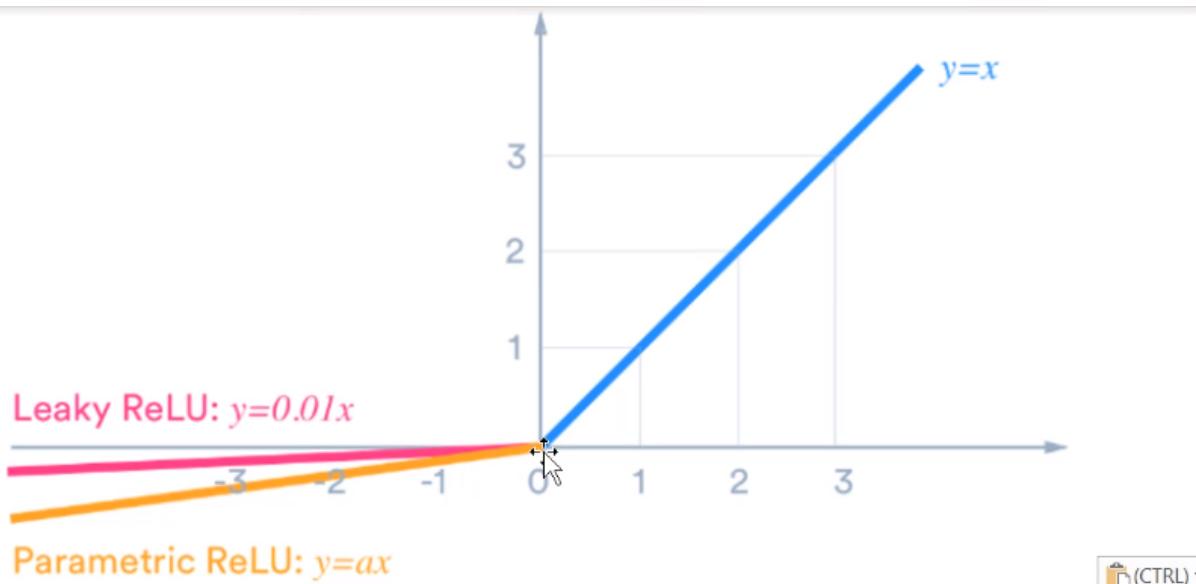
We can add a *border* (zero values) to the input volume.

- **Pooling**

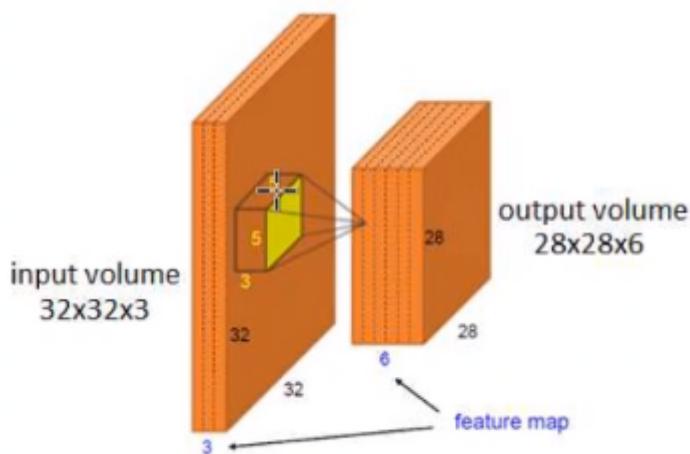
The most used aggregation operators are the maximum (*Max*) and the average (*Avg*).

13

- La funzione di attivazione più usata è la **ReLU** la quale calcola il valore massimo tra 0 e x. Quella in rosso è leggermente diversa dalla ReLU, così come quella in giallo:



- Un filtro di questo tipo (in giallo)

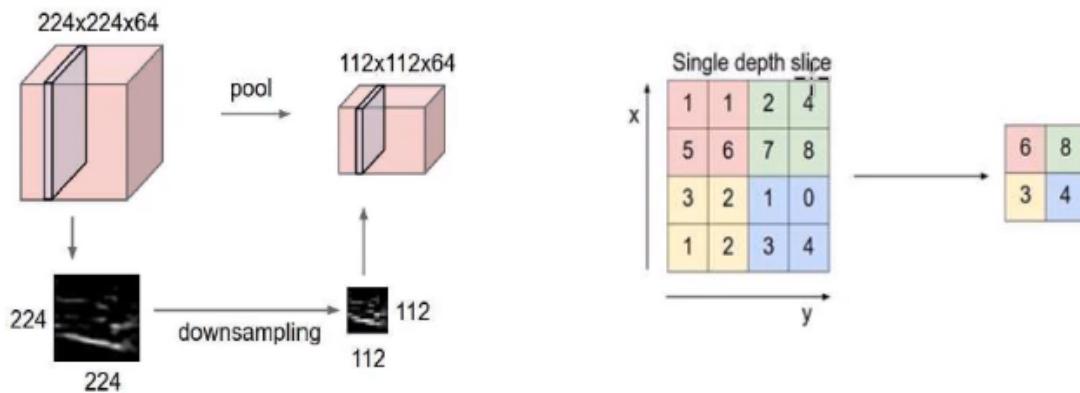


è un filtro **3D** che scorre sul volume di input potrebbe farlo con un passo (stride) maggiore di 1 (anzichè un pixel alla volta) riducendo il numero di feature map nel livello successivo ma pago in termini di accuratezza in quanto analizzo con meno intensità le zone.

- Abbiamo visto che con una feature map da 28x28 e qualunque sia la dimensione del filtro, produco una feature map di dimensione minore ad esempio 24x24 ma se mi interessa in alcuni casi mantenere la stessa dimensione della feature map di input, come posso farlo usando un filtro 5x5? Posso aggiungere tramite il **padding** delle colonne e righe di tutti 0 intorno all'input, aggiungendo un bordo ai valori di input.
- Il **pooling** può essere utilizzato da diversi operatori come il massimo e il valore medio. Un pooling che calcola il massimo con filtri 2x2 e stride 2 è uno dei pooling più usati e funziona in questo modo:



Example: max-pooling with 2x2 filters and stride=2



supponendo di avere una singola feature-map (single depth slice), quando si applica il pooling bisogna pensare a un riassunto di ciascuna feature map. Con un filtro 2x2 seleziono prima il quadrato **rosso** e poi avendo stride 2 seleziono il quadrato **verde**, poi quello **giallo** e infine quello **blu**. Per ciascun quadrato filtrato produco il massimo di quel quadrato: 6 per il **rosso**, 8 per il **verde**, 3 per il **giallo** e 4 per il **blu**.

- **Fully connected layers**

In a CNN, after a certain number of pairs (convolutional layer, pooling layer), one or more fully connected layers are used.

The size of the last of these layers, which coincides with the output layer, is equal to the number of classes in which the inputs to the CNN are classified.

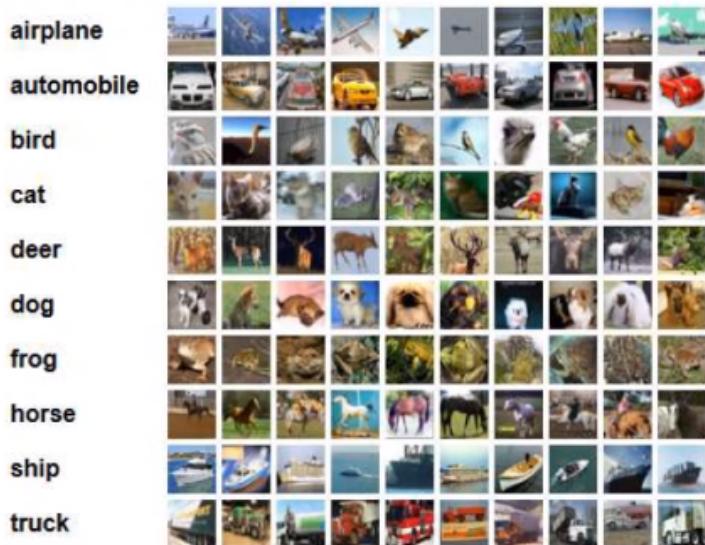
Dunque in una CNN dopo un certo numero di coppie (livello convoluzionale, livello di pooling) uno o più livelli fully connected vengono utilizzati. La dimensione di questi livelli è uguale al numero di classi in cui gli input della CNN sono classificati. Vedremo in alcuni esempi che dopo un livello convoluzionale tipicamente c'è un livello di pooling ma non è obbligatorio!

Cifar-10 è un dataset di 60000 immagini a colori appartenenti a 10 classi e la rete usa 50000 immagini di addestramento e 10000 immagini di test. Le 10 classi sono mostrate qui nell'immagine:

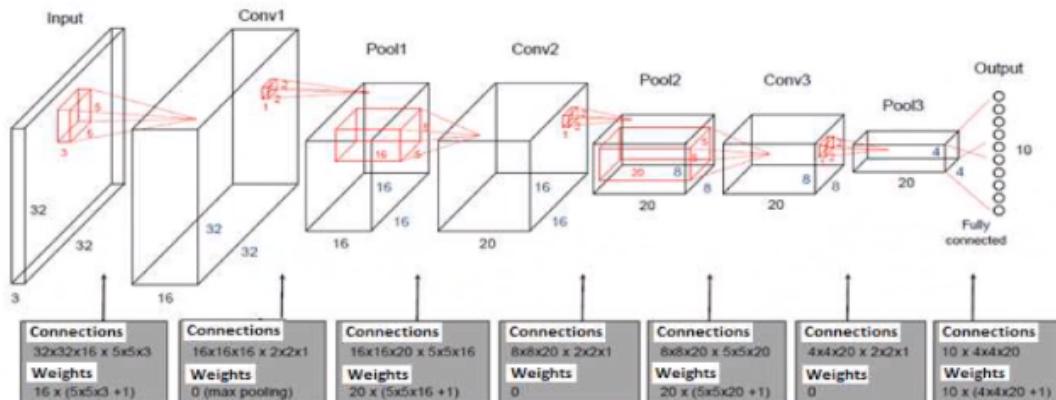
Example 1: Cifar-10

CIFAR-10 is a dataset of 60,000 32x32 color images, belonging to 10 classes (6,000 images per class). There are 50,000 training images and 10,000 test images.

Classes and 10 random images of each of them



16



Input: 32x32x3 RGB images
 Conv1: filters: 5x5, FeatureMaps: 16, stride: 1, pad: 2, activation: ReLU
 Pool1: type: Max, filters: 2x2, stride: 2
 Conv2: filters: 5x5, FeatureMaps: 20, stride: 1, pad: 2, activation: ReLU
 Pool2: type: Max, filters: 2x2, stride: 2
 Conv3: filters: 5x5, FeatureMaps: 20, stride: 1, pad: 2, activation: ReLU
 Pool3: type: Max, filters: 2x2, stride: 2
 Output: Softmax, #classes: 10

Total neurons: 31,562
 (including input layer)

Total connections:
 3,942,784

Total weights: 22,466
 (including biases)

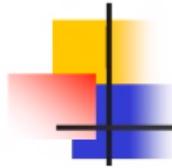
<https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html>

17

Il primo livello convoluzionale usa dei filtri 5x5(x3 che non va specificato in quanto è la profondità dell'input).

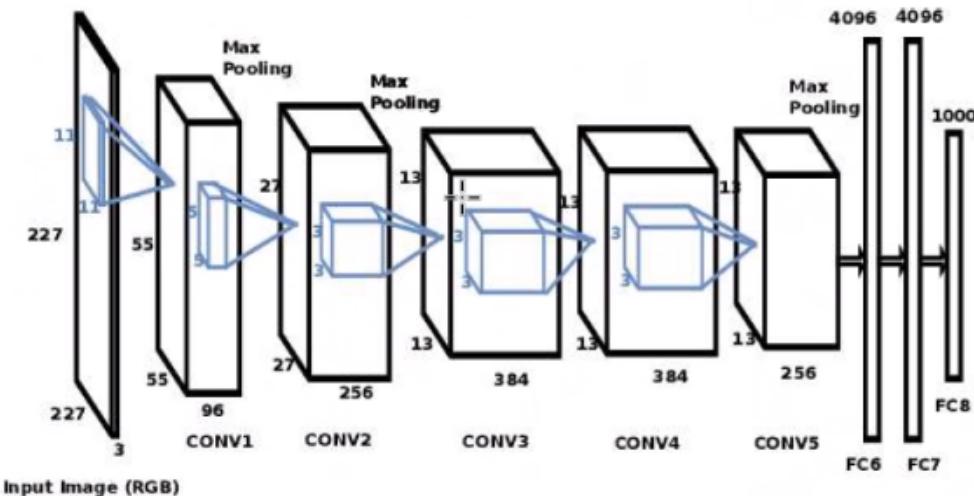
Bisogna poi specificare il numero di filtri, ad esempio 16 o il padding pari a 2.

CaffeNet è una CNN variante della **AlexNet**. Questa rete è pre-addestrata su un data set tra quelli più grandi disponibili pubblicamente che contiene oltre 15 milioni di immagini annotate a mano dalle persone con la label della classe a cui questa immagine appartiene.



Example 2: CaffeNet

CaffeNet is pre-trained on ImageNet (1000 classes and 1.2 million images).



http://faratarjome.ir/u/media/shopping_files/store-EN-1520504599-9025.pdf

18

Le classi sono addirittura più di 20 mila. Questo dataset è usato come benchmark per i modelli di classificazione delle immagini.

Questa rete consiste di 8 livelli, l'ingresso sono immagini RGB 227x227. L'ultimo livello ha 1000 neuroni in quanto la classificazione ha 1000 classi.

? Perchè la funzione più usata è la ReLU?

La backpropagation cambia i pesi andando all'indietro e applicando la regola della catena.

La regola della catena include il prodotto di tante derivate parziali, la derivata della sigmoide o della tangente iperbolica mi produce dei valori tra 0 e 1, dove il prodotto di tanti numeri **minori di 1** il risultato sarebbe portare il gradiente verso un valore pari a 0, il cosidetto **vanish gradient**, ovvero il gradiente che svanisce, ma se svanisce non posso addestrare nulla, non posso cambiare i pesi, dunque non addestro la rete dunque si usa la ReLU che per valori positivi il gradiente è 1.



In practice

Implementing a CNN (from scratch) is possible but takes a lot of development time and resources. Fortunately, there are many software (often open-source) that allow you to operate on CNNs. Among the most used there are:

Caffe, Theano, Torch, TensorFlow (Google), Keras.



Transfer Learning

Training complex CNNs on large datasets can take days/weeks even when performed on GPUs.

Additionally, training a CNN on a new problem requires a labeled training set of considerable size (often unavailable). An alternative is *Transfer Learning*:

- **Feature reuse:** an existing (pre-trained) network is used as *feature extractor*. The features generated by the network during the forward step are extracted (at intermediate layers). These features are used to train an external classifier (e.g., SVM or softmax classifier).
- **Fine-Tuning:** start with a network pretrained on a similar problem and
 - add your custom network on top of the already-trained base network;
 - freeze the base network;
 - train the newly added part;
 - unfreeze some of the top layers of the base network;
 - jointly train both these top layers and the newly added part.

20

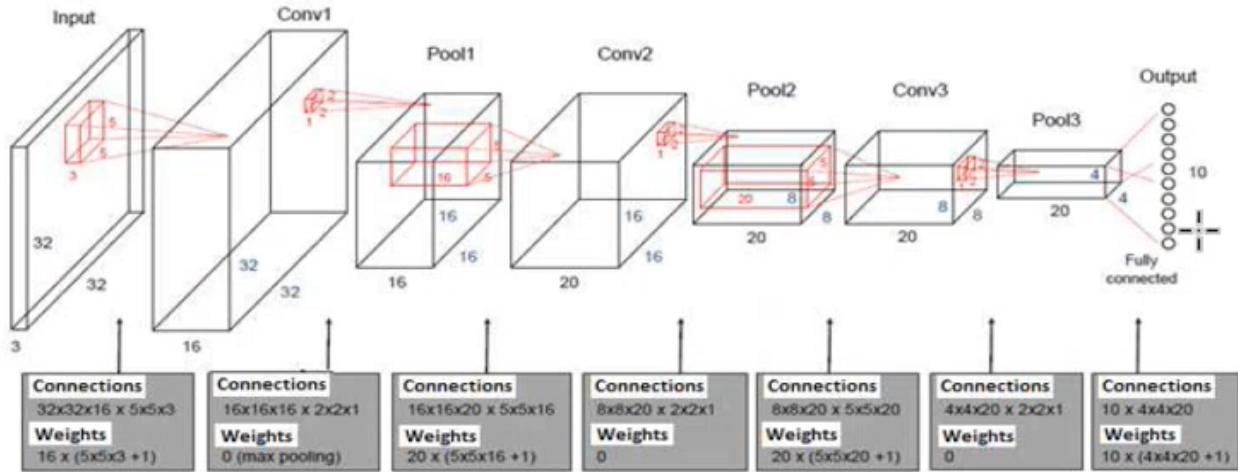
Il **transfer learning** consiste nel riutilizzare una rete preaddestrata su dati simili.

- **Features reuse:** Come possiamo descrivere il task che riesce a eseguire la CaffeNet?

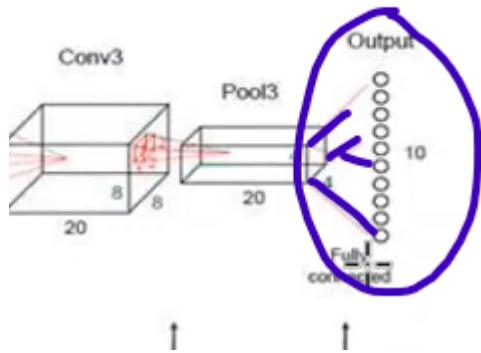
Questa rete riceve delle immagini in ingresso e riesce a classificarle imparando una gerarchia di features, le quali sono a basso livello vicine all'input e mano a mano che si va avanti sono di alto livello se sono vicino all'output. Mano a mano che si va verso l'uscita i filtri che faccio scorrere sull'immagine vedono come features quelle estratte al livello precedente, non vedono mica l'intera immagine, ad esempio vedono una feature estratta dal livello precedente come ad esempio un

nas o una bocca o un occhio, ed ecco come l'ultimo livello di uscita riesce a identificare una persona nell'immagine! Queste feature vengono usate per addestrare dei classificatori esterni.

- **Fine-Tuning:** potrei riadattare ai miei bisogni la rete preaddestrata e decidere in prima istanza di togliere l'ultimo livello perché ho magari 20 classi mentre nella CNN dell'immagine ce ne sono 10:

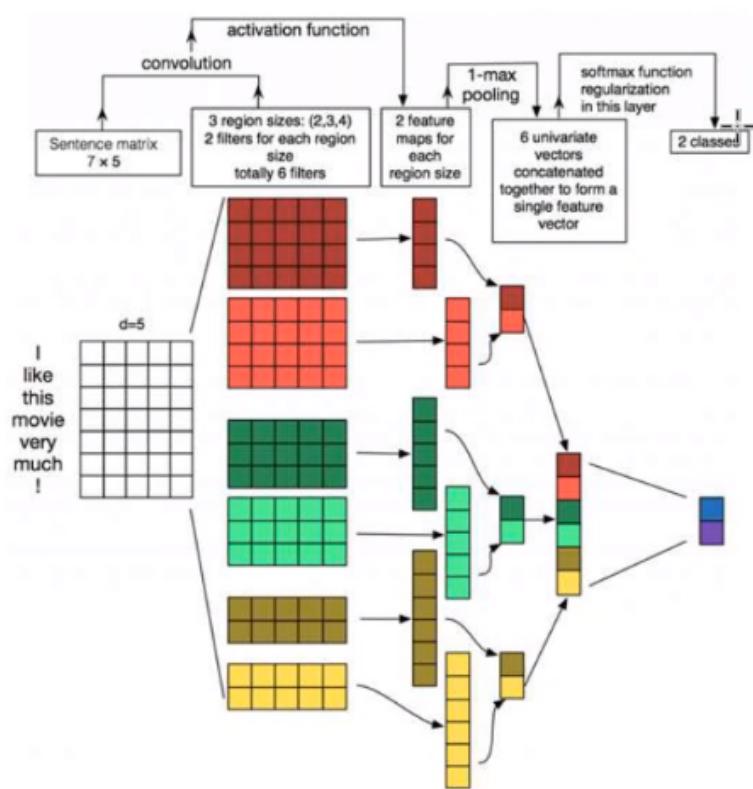


Per fare il fine tuning della rete, magari le mie 20 classi sono simili a quelle usate da chi ha addestrato la rete, congelo i pesi della parte precedente a quella che io sostituisco ed estraggo casualmente solo i valori che ci sono tra queste connessioni



addestro solo la parte nuova e poi effettuo un fine tuning finale.

CNN for sentence classification



[Y. Zhang, B.C. Wallace]

21

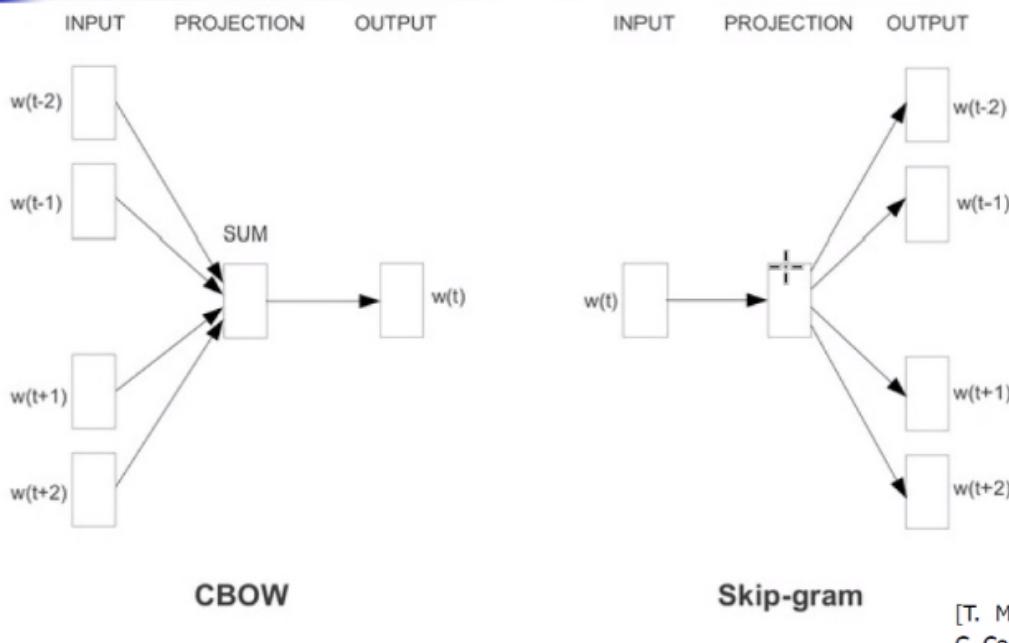
Si può applicare la CNN a del testo e dunque classificare il significato di un tweet come opinione positiva o negativa. L'immagine può rappresentare del testo, la frase si divide in parole e si vettorizza. Un modo semplice è quello di dire che si stabilisce un vocabolario di parole che mi interessano e ogni parola la rappresento con un one hot vector, dove gli elementi sono tutti uguali a 0 escluso un 1 corrispondente alla parola che mi interessa.

Il **word embedding** prevede di rappresentare le parole attraverso dei vettori nello spazio nel quale parole che hanno significato simile sono vicine tra loro. Questa rappresentazione si può ottenere da una rete la quale viene addestrata per produrla. Si effettua una convoluzione di uno o più filtri sulla matrice di ingresso che rappresenta la frase. Nella elaborazione di frasi i filtri hanno la stessa larghezza della matrice usata per rappresentare la frase e ciò che cambia è solo l'altezza la quale è la region size del filtro che facciamo scorrere. Gli autori di questa CNN hanno usato region size uguale a 2,3,4 e hanno preso 2 filtri per ciascuna dimensione.

Ogni feature map viene sottoposta a un 1-maxpooling ovvero si calcola il massimo di tutta la feature map, a questo punto gli autori concatenano e lo presentano come input ad un livello softmax che produce la classificazione in positivo o negativo.



Word2Vec



[T. Mikolov, K. Chen,
G. Corrado, J. Dean]

22

Per completare il discorso di word embedding le parole possono essere rappresentate sotto forma di 10000 elementi binari ma anche tramite word embedding il quale più utilizzato è il Word2Vecx. Questi sistemi considerano un insieme di documenti di testo dopodichè cercano di codificare ogni singola parola raccolta dai testi in un modo che ha un significato semantico. Ovvero per ogni parola individuano il contesto in cui quella parola si trova e rappresentano quella parola sfruttando le parole di contesto. Ad esempio consideriamo di prendere 5 parole dal testo, la parola al centro di quelle 5 sarebbe la $w(t)$ mentre le 2 parole precedenti ($w(t-1)$ e $w(t-2)$) e le 2 successive ($w(t+1)$ e $w(t+2)$) costituiscono il contesto che è usato per dare significato alla $w(t)$.