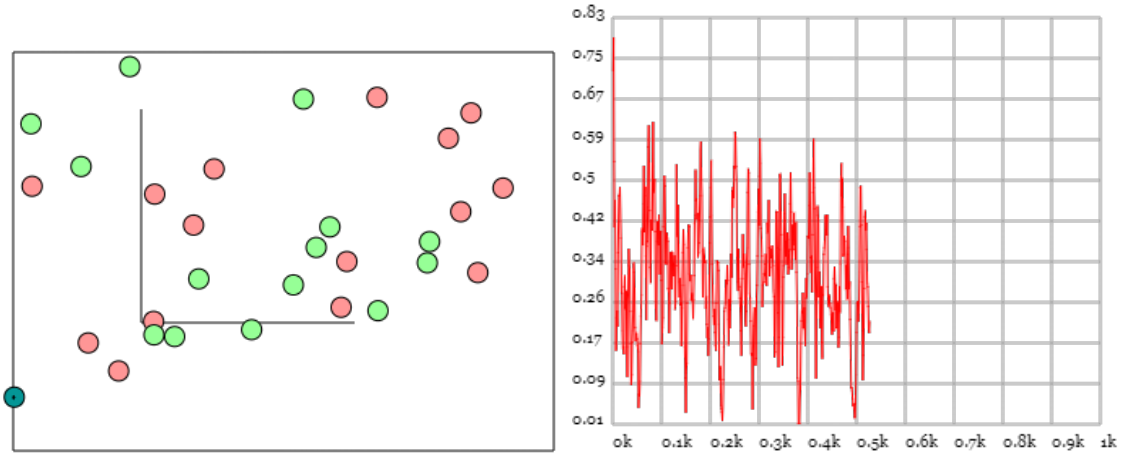
Deep Learning per principianti

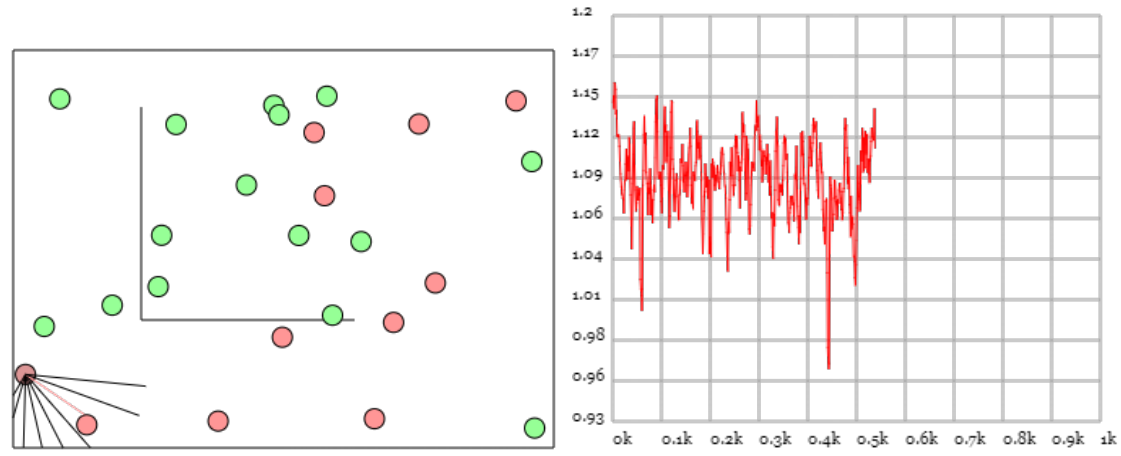
You

August 25, 2017

1 DEEP LEARNING

Che cos'è il deep learning? Quali sono i suoi principi di funzionamento?
Iniziamo con un gioco: un giocatore (pilotato dal computer) deve prendere i pallini rossi ed evitare quelli verdi per vedere il proprio punteggio aumentare (il grafico sulla destra).





Riuscite a capire quale dei due giocatori sia il più bravo? Questa è la magia del deep learning: il secondo giocatore è stato "allenato" e quindi sa come comportarsi per aumentare il proprio punteggio. Mentre leggete continueranno a giocare e quindi potrete vedere il grafico del punteggio per ognuno dei due. Consiglio infatti di tornare a vedere i grafici più tardi, quando avranno una forma su cui si può trarre qualche conclusione.

Per capire cosa sia il deep learning bisogna intanto comprendere cosa vi sia alla base del suo processo di allenamento. L'unità base è il **neurone**, una struttura che prende in ingresso uno i più input, li processa usando una funzione matematica e restituisce quindi l'output in base ai calcoli svolti.

Esistono diversi tipi di neuroni: il primo ad essere utilizzato è stato il ***perceptron***, un neurone che prende in ingresso una serie di input *pesati* (vengono cioè moltiplicati per un *peso*), li somma, aggiunge una costante (chiamata *bias*), applica una funzione e, in base al risultato, restituisce uno 0 o un 1.
Si sono quindi messi insieme diversi *perceptron* per formare una **rete neurale** in grado di svolgere compiti più complessi di un singolo neurone da solo. Queste reti non hanno tuttavia un'architettura complessa: assumono infatti una struttura stratificata, in cui un neurone di uno strato riceve come input gli output dei neuroni dello strato precedente e fornisce il proprio output ad ognuno dei neuroni dello strato successivo, che lo accetta come input pesato.

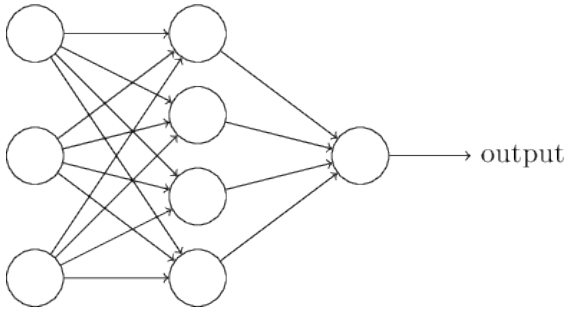


Figure 1: La struttura di una semplice rete neurale a tre strati con: 3 neuroni di input, 4 neuroni *nascosti* e un solo neurone di output.

Il primo strato viene sempre detto ***input layer*** e raccoglie tutti i neuroni che fungono da input per la rete. Tali neuroni non ricevono alcun input e il loro output è equivalente ai dati in ingresso alla rete. Servono solo come segnaposto, per indicare in modo omogeneo anche gli input della rete.
L'ultimo strato prende il nome di ***output layer***: i risultati in uscita da ciascun neurone di questo strato costituiscono l'output complessivo della rete. Infatti una rete neurale può avere anche più di un output (avrà tanti output quanti sono i neuroni nell'*output layer*).
Ognuno degli strati intermedi è denominato ***hidden layer***, chiamato in questo modo perché risulta appunto "nascosto" (non è né di input né di output). Mentre possono esservi un solo *input layer* ed un unico *output layer*, una rete può avere anche più di un solo *hidden layer*. Una rete con molti strati riuscirà quindi a compiere astrazioni sempre migliori man mano che si procede dall'ingresso all'uscita.
Ad esempio, una rete che debba riconoscere delle immagini avrà un neurone di input per ogni pixel dell'immagine (supponiamo che l'immagine sia in bianco e nero), un primo *hidden layer* che si occuperà di riconoscere figure elementari (un cerchio, un quadrato...), un secondo strato intermedio che studierà figure un po' più complesse (un cerchio con al suo interno altri cerchi...) e così via, fino all'*output layer*. Ovviamente questa è una semplificazione di come funzionano le reti neurali, anche se concettualmente il loro funzionamento è questo.

Finora abbiamo parlato di reti di *perceptron*. Ben presto però, si scoprì che questo tipo di neuroni era fin troppo limitato: un output di soli 0 e 1 non soddisfaceva le esigenze dei ricercatori. Così si passò ad utilizzare il neurone **sigmoide**. Questo tipo di neurone, a differenza del *perceptron*, non solo può accettare come input qualsiasi valore reale, ma può fornire come output un valore reale compreso tra 0 e 1. Il loro uso, tuttavia, non differisce da quelli dei *perceptron*: tutto quello che abbiamo detto finora sulle reti neurali è ancora valido anche per i neuroni sigmoidi.

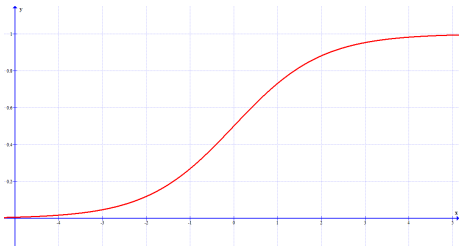


Figure 2: Il grafico della funzione *sigma*, dal cui nome deriva la denominazione dei neuroni sigmoidi. Viene utilizzata per calcolare l'output di questo tipo di neuroni.

I neuroni da soli non riescono ovviamente ad imparare. Per fare in modo che una rete neurale possa imparare si deve fare in modo di cambiare in modo appropriato i **pesi** dei collegamenti tra i neuroni e i **bias**, le costanti tipiche di ciascun neurone. Sono i valori di questi parametri che differenziano le reti neurali dei due giocatori dell'esempio iniziale, in quanto la loro struttura è identica. Ma come possiamo identificare i valori corretti? Dietro tutto il procedimento vi è ovviamente una serie di calcoli che tralascio in questa sede. Tuttavia, voglio farvi capire come funziona.

Per sistemare i valori di tutti i parametri viene usata una funzione detta **di costo**, che misura quanto la rete si sta comportando bene. Il nostro obiettivo è trovare il valore di uscita della rete che minimizza tale funzione. Tuttavia, farlo in maniera analitica, svolgendo ovvero tutti i calcoli esatti, risulterebbe troppo pesante, in quanto i parametri di cui dovremmo tenere conto sono davvero tanti (con un bias per ogni neurone e con un peso per ogni collegamento tra neuroni si arriva tranquillamente a qualche migliaia di parametri!). L'algoritmo che si segue è chiamato **algoritmo del gradiente decrescente**. Ve lo spiego ricorrendo ad un esempio.

Immaginiamo di avere una pallina da golf, che rappresenta il risultato corrente della rete neurale. La buca è il risultato corretto, quello che vorremmo che la nostra rete avesse. Quanto più la pallina è vicina alla buca, tanto più il risultato è corretto (e la funzione costo bassa). Il nostro obiettivo è ovviamente fare buca. Per prima cosa, calcoliamo il **gradiente**, che ci dice la direzione in cui dovremo tirare la pallina. Una volta calcolato è il momento di tirare. Noi siamo tipi pazienti e non badiamo al numero di tiri che dovremo fare: procediamo per tiri di piccola potenza nella direzione che abbiamo stabilito. Ci avviciniamo pian piano alla buca. Vi è un parametro che misura la potenza dei nostri tiri: è il **tasso di apprendimento** (spesso indicato con la lettera greca η). Anche per il tasso di apprendimento ci vuole tuttavia il giusto valore. Troppo basso e dovremo rimanere ore per mandare la pallina in buca. Troppo alto e la nostra pallina sorpasserà la buca, allontanandoci dal nostro obiettivo. Ripetendo quindi la sequenza calcolo del gradiente - tiro della pallina ci avvicineremo sempre di più alla buca, finché non la raggiungiamo (e avremo trovato quindi il minimo della funzione costo!). Il procedimento concettuale è quindi questo.

Nella pratica, per allenare la rete si prendono una serie di input di allenamento di cui si conosce il valore corretto di output e si danno in pasto alla rete per osservarne il risultato. Si procede quindi come descritto sopra, calcolando il gradiente del costo e aggiustando ciascuno dei parametri (pesi e bias) per muoverci verso il minimo della funzione costo. Calcolare però il gradiente su ognuno dei possibili input è tuttavia molto dispendioso. Si procede quindi a calcolare il gradiente su un gruppo di input di allenamento scelti casualmente detto ***mini-batch***. Quindi: scegliamo un *mini-batch*, alleniamo la rete con quello, scegliamo un altro *mini-batch*, la alleniamo... e così via, finché non finiamo una cosiddetta **epoca di allenamento**, ovvero finiamo di passare tutti gli input. Poi si ricomincia una nuova epoca e si ripete fino a quando non si ritenga che la rete sia pronta.

Le basi per capire un po' come funziona il deep learning sono finite! Potete passare a vedere gli esempi disponibili, provando magari a cambiare anche i parametri di cui abbiamo parlato per vedere che risultati si ottengono!

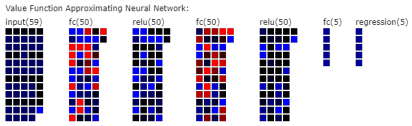
Per finire, tornando ai nostri due giocatori, si può vedere come il secondo giocatore abbia mantenuto un punteggio molto più alto e con oscillazioni meno ampie del primo. Il procedimento con cui si è allenato il secondo giocatore non è dissimile a quello che abbiamo appena visto: ha scoperto che prendere le palline rosse lo avvicinava alla buca e perciò ha sistemato i propri parametri per cercare di evitare le verdi e mangiare le rosse.

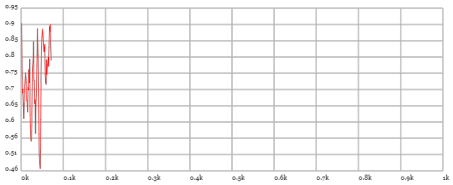
2 Reinforcement Learning

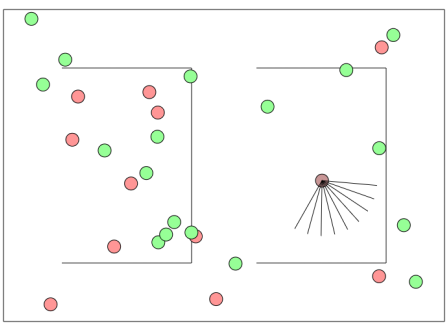
Questa è la demo dell'esempio trovato anche in presentazione del Deep Learning. In realtà, qui le cose funzionano in maniera leggermente diversa dalle altre demo. Infatti, si parla di ***Reinforcement Learning*** (o **Apprendimento per Rinforzo**).
Il *Reinforcement Learning* si differenzia dal tipico procedimento perché prevede anche il coinvolgimento di una *ricompensa*, che viene detta appunto *rinforzo*, per valutare la qualità delle prestazioni del sistema. Questa tecnica consente al sistema di adattarsi alle variazioni dell'ambiente in cui si trova. È quindi perfetto per il nostro caso, in cui l'ambiente in cui si deve muovere il nostro agente è in continuo mutamento!

Ma passiamo subito ai fatti: qui sotto trovate una rete fatta esattamente nello stesso modo di prima. Ora potrete modificare i diversi parametri: numero di *hidden layer*, numero di neuroni presenti in ognuno di essi, tasso di apprendimento, *batch size*, parametro di regolarizzazione (se non hai letto l'approfondimento relativo puoi lasciarlo così) e tasso ϵ. Quest'ultimo rappresenta la probabilità che l'agente compia un'azione casuale, invece che eseguire quella che gli darebbe una ricompensa più alta (tutti noi, ogni tanto, facciamo un'azione senza motivo).
Nella demo sono a disposizione, inoltre, diverse velocità di gioco. Infine, una volta che la rete si è allenata, si può premere il pulsante relativo per fermare l'allenamento e vedere l'agente giocare sul serio. Si noterà un cambiamento nel suo comportamento e il punteggio dovrebbe leggermente aumentare e stabilizzarsi su oscillazioni meno ampie.
Ma come sapere quando fermarsi? Dopo circa 10 minuti, la rete dovrebbe essere sufficientemente allenata (con i parametri di default), ma se avete fretta, potete cliccare sul pulsante *Carica Rete* per vedere in azione una rete già allenata.

FORM INSERIMENTO PARAMETRI







PULSANTI E DATI

Mentre si allena, vediamo un po' la teoria che c'è dietro. Abbiamo già detto che si tratta di *Reinforcement Learning*, ma non quale algoritmo sia stato esattamente utilizzato: si tratta del ***Q-Learning***, uno degli algoritmi più conosciuti di apprendimento per rinforzo. Tale algoritmo consente ad un agente automatico di adattarsi all'ambiente anche nel caso in cui abbia a disposizione solo parte delle informazioni riguardante ad esso. Ad esempio, in questa demo, l'agente ha a disposizione solo l'informazione video del campo di gioco e le mosse che può eseguire, esattamente come se si trattasse di un giocatore umano.
Ma cosa avviene esattamente? Come fa a scegliere le mosse corrette? Il sistema divide il problema in un *insieme di stati* e un *insieme di azioni* per ogni stato: l'agente quindi si muove da uno stato all'altro effettuando una delle azioni disponibili. Quindi, ogni stato fornisce una ricompensa all'agente, il cui obiettivo è proprio quello di massimizzare la ricompensa totale. L'agente quindi apprende quali sono le azioni ottimali associate ad ogni stato.
Entrando nel dettaglio della demo, l'agente è rappresentato dal pallino colorato che si muove: esso ha 9 sensori, direzionati in angoli diversi, ognuno dei quali misura 3 diversi valori nella propria direzione (fino ad una distanza massima di visibilità): la distanza da un muro, la distanza da un oggetto verde e quella da un oggetto rosso. Inoltre, gli viene data la possibilità di scegliere tra 5 azioni, differenziate dall'angolo di rotazione. Il suo obiettivo è di raccogliere le *mele* (gli oggetti rossi): per ogni mela mangiata, viene data una ricompensa all'agente. Deve invece evitare il *veleno* (gli oggetti verdi), poiché essi sono associati ad una ricompensa negativa.
Ha inoltre a disposizione una memoria delle azioni passate: in questo modo, l'agente può verificare ciò che ha fatto per aiutarsi a decidere la sua mossa futura. In particolare, per evitare di scegliere azioni consecutive, che sono fortemente dipendenti l'una dall'altra, sceglie casualmente diverse azioni in diversi momenti della sua memoria, in modo da poter avere una buona varietà di situazioni su cui basarsi.
Per terminare, una piccola curiosità: questa demo è stata creata sulla base di un lavoro riguardante il *Reinforcement Learning* che voleva creare una rete neurale in grado di giocare i giochi Atari 2600. Lo scopo di questo lavoro è stato raggiunto e, su sette giochi, ben in tre il computer ha superato il punteggio di un giocatore umano esperto!

3 Overfitting

Nella sezione introduttiva abbiamo visto come funzionano (per sommi capi) le reti neurali allenate con il *deep learning*. Tuttavia, quando si fecero i primi test per verificare la correttezza del procedimento, si presentarono anche diversi problemi. Uno dei più frequenti viene detto ***ovefitting***. Quando si procede ad allenare una rete neurale, si usano due diversi gruppi di input: il gruppo di dati di allenamento, usato per sistemare i parametri della rete, e il gruppo di test, usato per verificare l'accuratezza del lavoro della rete. L'*overfitting* avviene quando la rete, invece di imparare a generalizzare sui dati di allenamento (e quindi comprendere ciò che sta facendo), si limita ad imparare a memoria tali dati, avendo quindi degli scarsi risultati quando si tratta di vedere i suoi progressi sui dati di test. La sua accuratezza smette di crescere nel tempo ed inizia ad oscillare o a diminuire. E questo è un problema, in quanto noi vogliamo massimizzare tale accuratezza.

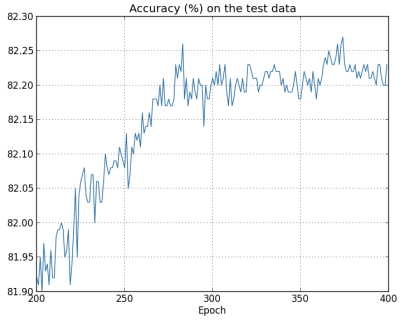


Figure 3: Il grafico dell'accuratezza nel classificare delle cifre scritte a mano da parte di una rete neurale. Si può notare come a partire dall'epoca 280, il trend in crescita si ferma e la precisione inizia ad oscillare.

L'*overfitting* quindi impedisce alla nostra rete di imparare correttamente: il nostro obiettivo è quindi individuare il momento in cui inizia tale fenomeno per evitare che la rete si *sovraalleni*.
Un modo per farlo è proprio tenere traccia dell'accuratezza sui dati di test mentre la rete si allena: così, quando vediamo che essa cessa di migliorare, fermiamo l'allenamento della rete. Anche se questo potrebbe non essere segno di *overfitting*, tale procedimento lo previene per certo.

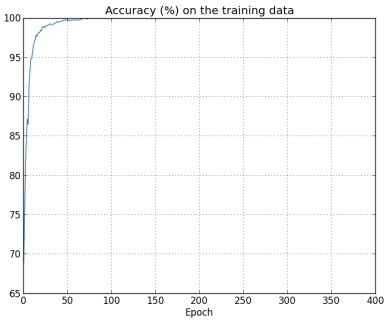


Figure 4: Il grafico dell'accuratezza nel classificare delle cifre scritte a mano da parte di una rete neurale sui dati di allenamento. Ben presto, la rete inizia a memorizzare le peculiarità di tali dati, riconoscendoli non perché ha realmente capito, ma solo perché si ricorda a memoria tutto il set di dati.

Un altro metodo per individuare l'*overfitting* è misurare l'accuratezza su un set di dati chiamato **di validazione** alla fine di ogni epoca, in modo da poter riconoscere quando la rete smette di migliorare la propria performance e fermare il processo di allenamento (tale metodo è anche detto ***early stopping***). Il set di validazione è una parte dei dati che si trova a metà tra quelli di allenamento e quelli di test: si tratta di dati presi dal set di allenamento, ma usati per testare la rete in modo veloce, con l'unico scopo di individuare l'*overfitting*, senza entrare nei dettagli delle prestazioni

della rete. Il set di validazione è anche usato, più in generale, per confrontare le prestazioni della rete per diversi valori dei parametri secondari (come, ad esempio, si può scegliere quale valore del tasso di apprendimento scegliere).
Ma perché scegliere di usare i dati di validazione al posto di quelli di test sia per l'*overfitting* che per i parametri secondari? Se usassimo il set di test, rischieremmo di causare l'*overfitting* dei nostri parametri su tali dati, poiché sarebbero validi solo per le peculiarità del set di test e la nostra rete non funzionerebbe con altri set di dati. Usando invece il set di validazione, possiamo decidere i parametri e poi valutarli con i dati di test. Questo procedimento è detto di ***held out***, in quanto i dati di validazione sono scelti tra quelli di allenamento.
Un ultimo modo per ridurre l'*overfitting* è quello di aumentare il numero di dati usati per l'allenamento della rete. Con un buon set di dati di allenamento, anche le reti più grandi hanno meno probabilità di sovraallenarsi. Tuttavia, non sempre è possibile ampliare il set di allenamento e spesso ampliarlo è un procedimento costoso, tanto da renderlo inattuabile in molte situazioni.
In realtà, esiste un altro metodo, che tuttavia non vediamo in questo paragrafo: prende il nome di regolarizzazione.

4 Regolarizzazione

Le reti neurali, durante il loro processo di apprendimento, sfruttano, come abbiamo visto, la funzione costo per decidere come sistemare i propri parametri, ovvero *pesi* e *bias*. Abbiamo visto che c'è anche un grande problema, che è quello dell'*overfitting*. Sono state sviluppate alcune tecniche che, operando sulla funzione costo, aiutano a ridurre gli effetti del sovraallenamento di una rete neurale. Tali tecniche prendono il nome di **tecniche di regolarizzazione**.
Generalmente, tali tecniche prevedono l'aggiunta di un fattore, dipendente dai pesi, dopo l'espressione della funzione costo:

$$\text{funzione costo} = \text{funzione costo originale} + \lambda * \text{funzione di regolarizzazione}$$

La regolarizzazione può essere vista come un compromesso tra trovare pesi piccoli e minimizzare la funzione costo. Il parametro λ viene detto **tasso di regolarizzazione** e serve per determinare il bilanciamento di tale compromesso: quando λ è piccolo, allora preferiamo minimizzare la funzione costo, mentre quando è grande, cerchiamo di trovare pesi piccoli.
Una delle tecniche più utilizzate è chiamata **regolarizzazione L2** o **decadimento dei pesi**: utilizza la somma dei quadrati dei pesi come funzione di regolarizzazione. Ma questi sono dettagli. Sappiate solo che il nome *L2* deriva dal fatto che utilizza dei quadrati.
Esistono anche numerose altre tecniche di regolarizzazione. Di seguito ve ne riporto tre delle più usate.

- La **regolarizzazione L1** è analoga alla L2, solo che usa la somma dei pesi, invece della somma dei loro quadrati.
- La tecnica di ***dropout*** invece funziona diversamente, in quanto modifica non la funzione di costo della rete, ma la rete stessa. Abbiamo visto il principio di funzionamento di una rete neurale e come essa riesca ad allenarsi. Ecco, questa tecnica prevede di applicare il solito procedimento dimezzando prima i neuroni degli *hidden layer*! Per ogni epoca di allenamento si scelgono (casualmente) la metà dei neuroni da tenere e quella dei neuroni da scartare e si allena la rete così ottenuta. Si ripete quindi il procedimento, tenendo e scartando neuroni diversi ad ogni epoca: una volta che si ritiene che la rete sia pronta, si prende la rete originale e si dimezzano i pesi uscenti dai neuroni nascosti: abbiamo ottenuto una rete pronta a svolgere il proprio compito. In poche parole, è come se usassimo tante reti diverse e poi prendessimo come risultato la media di tutti i risultati di queste reti.

  FIGURE DI CONFRONTO TRA RETE COMPLETA E RETE PRESA IN ESAME
  La figura qui sopra mostra il procedimento svolto durante il dropout. Ad ogni epoca, poi, faremo in modo di avere una rete diversa ogni volta, considerando neuroni diversi.

- Si può anche pensare di **ampliare artificialmente i dati di allenamento**, in quanto ottenere nuovi dati per allenare la rete è sempre una buona idea. Il problema è che non sempre è possibile, oppure è troppo costoso ottenerne di nuovi. Quindi se ne generano di nuovi a partire da quelli che abbiamo già a disposizione. Ad esempio, se la nostra rete dovesse riconoscere delle cifre scritte a mano, potremmo applicare delle piccole rotazioni o delle lievi dilatazioni o restrizioni ai dati che abbiamo già in possesso, creando delle immagini nuove da fornire alla nostra rete neurale. In generale, si cerca di espandere il set di allenamento cercando di riprodurre quelle che sono le variazioni che di solito hanno nella pratica.

  FIGURE DI CONFRONTO TRA DUE 5
  Nonostante la differenza sia minima, per l'analisi svolta dalla rete sono due immagini completamente diverse.

Potreste chiedervi come tutto questo può aiutare a ridurre l'*overfitting*. Ebbene, la regolarizzazione aiuta le reti neurali a generalizzare meglio, in quanto una rete con pesi piccoli non varia il proprio comportamento se cambiano alcuni dei dati di input. Questo le rende particolarmente difficile memorizzare le peculiarità dei dati, mentre la aiuta ad apprendere meglio quelli che sono i modelli e gli schemi dei dati di allenamento.

Il principale problema della regolarizzazione è che non si è ancora capito esattamente il perché essa aiuti a migliorare le prestazioni di una rete neurale, ma abbiamo a disposizione solo evidenze pratiche di questo fatto. Nonostante questo, la regolarizzazione è ampiamente utilizzata e ci aiuta a migliorare le prestazioni delle nostre reti neurali.