Appunti di Reti Neurali e Deep Learning

A.A. 2021-2022

Matteo Brunello

Contents

1 Introduzione

Le reti neurali sono utilizzate in tanti ambiti e diversi contesti. Esse si ispirano al metodo di funzionamento del cervello per cercare di acquisire la stessa duttilita' e performances. L'unita' computazionale di base nel cervello e' il *neurone*, connesso ad altri neuroni per mezzo di *sinapsi*. Le reti neurali cercano di ispirarsi a questo aspetto. Il ruolo del singolo neurone e' quello di raccogliere l'informazione che arriva da altre parti della rete neuronale e propagarla al resto della rete. E' bene notare che una rete neurale e' un architettura altamente parallela: i neuroni funzionano in parallelo con il resto dei neuroni nella rete/cervello. Inoltre, le rappresentazioni all'interno di una rete neurale sono distribuite, cioe' significa che non c'e' differenza tra memoria e ragionamento: e' tutto codificato all'interno dei pesi.

1.1 Neurone

Il *neurone artificiale* (McCulloch Pitts, 1943), cerca di catturare le caratteristiche del neurone biologico. Esso e' costituito dai seguenti componenti fondamentali:

- **Insieme di valori in input** $x_1, \ldots, x_p$, che vengono ricevuti contemporaneamente
- **Insieme di pesi** $w_1, \ldots, w_p$, collegati ai valori di input fino a raggiungere il cuore del neurone
- **Funzione di aggregazione del segnale**, che e' una somma pesata. Per ogni valore in input x_i che raggiunge il *j-esimo* neurone, il suo valore corrispondente sara' $w_{ji} \cdot x_i$
- **Funzione di attivazione** ϕ, applicata al segnale aggregato
- Un **Bias**, che indica la tendenza naturale del neurone ad eccitarsi, piu' e' grande piu' il neurone sara' attivo anche con inputs piccoli

Dal punto di vista formale, il *bias* viene integrato come un input della rete con peso corrispondente sempre pari a 1. ($x_0 = 1, w_{j0} = b$ in cui b e' il bias) In questo senso, ogni neurone avra' il suo insieme di valori di input, piu' uno (il bias). Per mezzo di questa architettura di base si possono ottenere diverse tipologie di reti, che si suddividono in due sottocategorie principali:

- **Reti ad un solo livello** (es. Percettrone)
- **Reti a piu' livelli** (es. Recurrent NN)

1.1.1 Funzioni di attivazione

Possono essere di due tipi principali:

1. *Threshold*: mandano a 1 valori ≥ 0 e 0 valori < 0
2. *Sigmoidi*: sono della forma

$$\phi(v) = \frac{1}{1 + exp(-\alpha v)}$$

dove α e' il parametro che regola la *pendenza* della curva

2 Il Percettrone

Partiamo con il trattare il primo tipo di rete: il percettrone, introdotto dallo psicologo Frank Rosenblatt nel 1958. E' un modello di rete molto semplice, in grado di classificare solo problemi linearmente separabili. Il percettrone e' basato sul concetto di neurone artificiale esposto precedentemente (*McCulloch-Pitts model*). Dal punto di vista architetturale, il percettrone e' una rete a singolo layer, cioe' e' costituita da un solo layer di output. Noi tratteremo il caso in cui questo layer sia composto da un solo neurone, ma la teoria e' facilmente estendibile a casi con piu' di un neurone. Il percettrone ha la caratteristica principale di avere la seguente funzione di attivazione (o *hard llimiter*):

$$\phi(x) = \begin{cases} 1 & if\ x > 0 \\ -1 & otherwise \end{cases}$$

Dal momento che sappiamo il funzionamento del singollo neurone, se conosciamo i pesi di ogni input possiamo calcolare molto facilmente l'output di un percettrone dato l'insieme di input. Difatti, data una certa configurazione di pesi, l'output sara':

- 1 se $w_1x_1 + w_2x_2 + \cdots + w_px_p + \underbrace{w_0}_{bias} > 0$
- -1 altrimenti (≤ 0)

Come detto, il percettrone risolve *solamente* problemi linearmente separabili. Questo e' evidente come dato un insieme di input, un singolo neurone vada ad assegnare a tale insieme una classe: positiva (1) o negativa (-1). * Se noi prendiamo per esempio il caso bidimensionale, possiamo calcolare il suo output come $w_1x_1 + w_2x_2 + w_0$ * Se noi la poniamo a 0, tale equazione ci da l'equazione della retta del *decision boundary* del percettrone: $x_2 = -(w_1x_1 + w_0)/w_2$ Si noti come l'aumentare del numero di input, aumenti di conseguenza il numero di dimensioni in cui l'iperpiano va ad agire come decision boundary. Da questo punto di vista, il bias non e' nient'altro che la distanza del decision boundary dall'origine.

2.1 Algoritmo di apprendimento

Per stabilire il valore dei *pesi* di un percettrone, bisogna costruire un *algoritmo di apprendimento*. Il setting e' di apprendimento supervisionato, per cui l'algoritmo sara' basato sulla *correzione dell'errore* per ogni esempio non classificato correttamente. Denotiamo con

- $\mathbf{x}(n) = [+1, x_1(n), \ldots, x_m(n)]^T$ l'insieme di input al passo n
- $\mathbf{w}(n) = [b, w_1(n), \ldots, w_m(n)]^T$ l'insieme di pesi al passo n
- $v(n) = \sum_{i=0}^{m} w_i(n)x_i(n)$ la funzione di attivazione al passo n, che corrisponde a $\mathbf{w}^T(n)\mathbf{x}(n)$ Siccome le istanze di train sono rappresentate con m features (corrispondenti al numero di input), possiamo indicare con
- $\mathscr{X}_1$ l'insieme dei vettori (istanze) con classe positiva
- $\mathscr{X}_2$ l'insieme dei vettori (istanze) con classe negativa

  Il processo di apprendimento richiede la regolazione del vettore dei pesi $\mathbf{w}$ in modo che **SE** le due classi sono linearmente separabili, **ALLORA** esiste un vettore $\mathbf{w}$ di pesi tale per cui

  - $\forall \mathbf{x} \in \mathscr{X}, \mathbf{w}^T\mathbf{x} > 0$
  - $\forall \mathbf{x} \in \mathscr{X}, \mathbf{w}^T\mathbf{x} \leq 0$

Il problema dell'apprendimento non e' nient'altro che trovare questo vettore $\mathbf{w}$ che soddisfi le due condizioni contemporaneamente. Possiamo quindi formulare l'algoritmo nel modo seguente:

1. Se l'esempio viene classificato correttamente, allora non modificare il peso
2. In caso contrario, il peso del percettrone viene aggiornato sottraendo o aggiungendo il *learning rate* moltiplicato per gli input

Riformulato formalmente:

1. $\mathbf{w}(n+1) = \mathbf{w}(n)$ se $\mathbf{w}^T\mathbf{x} > 0$ e $\mathbf{x} \in \mathscr{X}$ oppure $\mathbf{w}^T\mathbf{x} \leq 0$ e $\mathbf{x} \in \mathscr{X}$ (*se l'esempio e' classificato correttamente*)

2. $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n)$ se $\mathbf{w}^T\mathbf{x} > 0$ e $\mathbf{x} \in \mathscr{X}$ (*se l'esempio viene classificato* ***positivo*** *ma e'* ***negativo***) $\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{x}(n)$ se $\mathbf{w}^T\mathbf{x} \leq 0$ e $\mathbf{x} \in \mathscr{X}$ (*se l'esempio viene classificato* ***negativo*** *ma e'* ***positivo***)

Il *learning rate parameter* $\eta(n)$ controlla gli aggiustamenti che vengono fatti ad ogni missclassificazione. Se e' costante ad ogni iterazione ed e' > 0, allora parliamo di *fixed-increment adaptation rule*. Sulla base di queste osservazioni possiamo formulare il vero e proprio algoritmo:

1. Inizializza $\mathbf{w}(0)$ con valori casuali
2. Finquando ci sono esempi non classificati correttamente:
3. Prendi un esempio $\mathbf{x}$ da quelli non classificati correttamente
4. Applica l'update rule (*punto 2.*)

2.1.1 Convergenza

E' possibile provare la convergenza dell'algoritmo precedente, cioe' la sua terminazione in caso le classi siano linearmente separabili. Supponiamo di rappresentare nel piano il vettore peso dell'*n-esima* iterazione $\mathbf{w}(n)$ e il vettore stimolo $\mathbf{x}(n)$. Se il prodotto tra questi due vettori e' maggiore di 0, allora il percettrone resituera' 1 come output, contrariamente restituera' -1. Geometricamente, il prodotto $\mathbf{w}(n) \cdot \mathbf{x}(n) = \|\mathbf{x}(n)\|\|\mathbf{x}(n)\| \cos(\Theta)$, per cui possiamo dire che sara' > 0 se $-90 < \Theta < 90$. Il decision boundary ($= 0$) sara' pari all'angolo $\Theta = 90$ e 270 Ogni volta che c'e' un mismatch, il vettore $\mathbf{w}$ viene spostato verso una direzione differente, spostando di conseguenza anche il decision boundary.

> **Teorema** (*Convergenza del Percettrone*): Date le due classi C_1 e C_2, se esse sono *linearmente separabili* allora l'algoritmo di apprendimento termina in un *numero finito* di iterazioni

2.1.1.1 Dimostrazione Supponiamo che C_1 sia la classe positiva e C_2 sia negativa. Sia $C = C_1 \cup \overline{C_2}$, in cui $\overline{C_2} = \{-x \mid x \in C_2\}$. Vogliamo quindi ottenere un percettrone che assegni sempre 1 a tutte le istanze di C. Sia $\mathbf{x}(0), \ldots, \mathbf{x}(k) \in C$ la sequenza di istanze che sono state utilizzate per *correggere* il vettore di pesi fino alla *k-esima* iterazione. Dall'algoritmo, la regola per la correzione e' una somma, per cui per la generica iterazione vale $\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k)$. Secondo questa ipotesi vale $\mathbf{w}(k+1) = \mathbf{x}(0) + \cdots + \mathbf{x}(k)$ (*unwinding della funzione ricorsiva*). La dimostrazione a questo punto si basa sullo studio di come si evolve la norma del vettore dei pesi.

(Da qui in poi e' opzionale)

Possiamo stabilire due limiti: Uno inferiore e uno superiore.

2.1.1.1.1 Limite Inferiore Per ipotesi (*separabilita' lineare*) deve esistere per forza un $\mathbf{w}^*$ tale per cui ogni esempio $\curvearrowleft \in C$ viene classificato correttamente (cioe' valore > 0). Allora possiamo moltiplicare questo vettore per ambo i membri della relazione trovata in precedenza.

$$\precapprox^{*T} \precapprox (k+1) = \precapprox^{*T} \curvearrowleft (1) + \cdots + \precapprox^{*T} \curvearrowleft (k)$$

(Ho gia' moltiplicato ogni membro per la somma nella parte dx). Sia $\alpha = \min\{\precapprox^{*T}\curvearrowleft(0), \ldots, \precapprox^{*T}\curvearrowleft(k)\}$. Allora possiamo dire che vale la seguente relazione:

$$\precapprox^{*T}\precapprox(k+1) = \precapprox^{*T}\curvearrowleft(1) + \cdots + \precapprox^{*T}\curvearrowleft(k) \geq \alpha k$$

Per la diseguaglianza di *Cauchy' Schwartz* e qualche ri-arrangiamento algebrico elementare otteniamo:

$$\|\mathbf{w}(k+1)\|^2 \geq \frac{k^2\alpha^2}{\|\precapprox^*\|}$$

Abbiamo quindi dimostrato che α e' in qualche modo un *lower bound* di $\precapprox(k+1)$, cioe' che la magnitudine del vettore $\precapprox(k+1)$ cresce almento tanto velocemente quanto α. ■.

2.1.1.1.2 Limite Superiore Analogamente, possiamo dimostrare che

$$\|\precapprox(k+1)\|^2 \leq k\beta$$

- Dove $\beta = \max \|\curvearrowleft(i)\|^2$
- Cioe' che $\precapprox(k+1)$ non cresce mai piu' velocemente di $k\beta$ ■

A questo punto ci troviamo con una coesistenza di questi due limiti, per cui possono essere compatibili se e solo se

$$\frac{k^2\alpha^2}{\|\precapprox^*\|^2} \leq k\beta$$

cioe' quando

$$k \leq \frac{\beta\|\precapprox^*\|^2}{\alpha^2}$$

Questo dice che k deve essere un valore finito, per cui e' dimostrato. ■

3 Multi Layer Perceptron

Abbiamo visto col percettrone che esso classifica le classi solo nel caso di classi linearmente separabili. Esso riesce a dividere esattamente le due classi trovando un iperpiano che le separi. Nei casi in cui l'insieme non sia linearmente separabile, il percettrone non puo' fare nulla. Per ovviare a questo problema si e' introdotto il *MultiLayer Perceptron* (per cui sono stati essenzialmente introdotti i layer nascosti).

3.1 Adaline

Il modello Adaline e' di fatto per struttura un Percettrone, differisce pero' da esso per l'algoritmo con cui si va ad addestrare (una variante semplificata dellla *backpropagation*) e dalla funzione di trasferimento che e' pari all'identita'. L'algoritmo di apprendimento si basa sul **Least-Mean-Square (LMS)**. Per

ogni *k-esimo* esempio etichettato $(\mathbf{x}^{(k)}, d^{(k)})$, possiamo calcolare l'errore che viene commesso sul *k-esimo* esempio nel modo seguente:

$$E^{(k)}(\mathbf{w}) = \frac{1}{2}(d^{(k)} - y^{(k)})^2 = \frac{1}{2}\left(d^{(k)} - \sum_{j=0}^{m} \mathbf{x}_j^{(k)} \mathbf{w}_j\right)^2$$

L'errore totale su tutto il dataset e' la somma degli errori al quadrato di tutti gli esempi:

$$E_{tot} = \sum_{k=1}^{N} E^{(k)}$$

E_{tot} e' una funzione quadratica dei pesi, per cui sappiamo per forza che la sua derivata esiste ovunque. L'algoritmo consiste nel applicare un algoritmo di discresa del gradiente incrementale. A grandi linee, tale algorimo cerca il valore di pesi ottimale che permette di ottenere l'output in uscita che ci si attende. Un **Gradiente** e' una quantita' che si applica a funzioni di piu' variabili. Da in output un vettore per ogni punto della funzione, che ha direzione verso i punti che risultano nella crescita maggiore del valore della funzione (*steepest ascent*). L'idea dell'algoritmo e' quella di calcolare ad ogni step l'opposto del gradiente del'errore, cioe':

$$-(\nabla E^k(\mathbf{w})) = -\left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}\right]$$

Successivamente, ci si sposta facendo un piccolo passo (di grandezza η) in tale direzione

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta(\nabla E^{(k)}(\mathbf{w}))$$

Per calcolare il gradiente, dell'errore totale, dobbiamo prima vedere come calcolare la derivata parziale dell'errore rispetto al singolo componente w_j

$$\frac{\partial E^{(k)}\mathbf{w}}{\partial \mathbf{w}_j} = \frac{\partial E^{(k)}\mathbf{w}}{\partial y^{(k)}} \frac{\partial y^{(k)}}{\partial w_j} = -(d^{(k)} - y^{(k)}) x_j^k$$

Essenzialmente si applica la *chain rule* per fare la derivata. La delta rule per l'update dei pesi sara' quindi (in forma compatta):

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta(d^{(k)} - y^{(k)})\mathbf{x}^{(k)}$$

> Si noti come tanto piu' grande sara' l'errore, tanto di piu' verra' modificato il peso.

Possiamo estendere le considerazioni fatte per Adaline (che ha la stessa architettura del Percettrone) al Multilayer Perceptron.

3.2 Multi Layer Perceptron

Le reti multi layer sono molto piu' generali delle architetture viste fin'ora, l'idea e' quella di introdurre dei livelli *nascosti*, che non interagiscono direttamente con il mondo esterno (non ricevono direttamente inputs, ne ritornano direttamente outputs). Questi livelli servono a superare le limitazioni delle reti a layer singolo, per cui riescono a rappresentare dei task che non sono linearmente separabili. Il valore di output del singolo neurone, quindi, sara' calcolato nel modo seguente:

$$v_j = \sum w_{ji} y_i$$

in cui:

- w_{ji} e' il peso dal neurone i al neurone j
- y_i e' l'output del neurone i (che viene trattato in input essenzialmente)

L'algoritmo di apprendimento classico di queste reti e' anch'esso basato sulla discesa del gradiente. Per poter utilizzare questo algoritmo, le funzioni di trasferimento devono essere derivabili ovunque. Una funzione classica di trasferimento che si utilizza e' la **Funzione Sigmoide**, definita come

$$\phi(v_j) = \frac{1}{1 + e^{-\alpha v_j}}$$

- con $\alpha > 0$

Come visto la somma pesata rappresenta l'equazione di un iperpiano nella dimensione del problema. L'inserire diversi layer intermedi, fa si che si introduca un numero diverso di questi iperpiani.

3.2.1 Algoritmo di Backpropagation

Per fare l'apprendimento di questo tipo di reti, si utilizza l'algoritmo di **backpropagation**. E' un algoritmo di discesa del gradiente che cerca i pesi che minimizzano la funzione errore obiettivo sull'intero training set. Ogni step dell'algoritmo consiste nel cambiare i valori dei pesi verso la direzione tale per cui l'errore decresce piu' rapidamente, cioe' verso l'opposto del gradiente della funzione dell'errore.

Differenzia tra due tipologie di correzioni di pesi:

- Correzioni per pesi che connettono gli hidden layers
- Correzioni per pesi che connettono gli input/output layers

L'algoritmo consiste nell'applicazione ripetuta di due passaggi principali:

1. **Passo Forward**: viene computato l'errore per ogni neurone andando a calcolare l'ouput della rete per un determinato esempio
2. **Passo Backward**: viene utilizzato l'errore per fare l'update dei pesi utilizzando la update rule

*__Osservazione:__ I nodi hidden contribuiscono all'errore attraverso i nodi dei prossimi layer.

Secondo questa osservazione, partendo dall'output layer, l'errore e' propagato al contrario attraverso la rete, calcolando ricorsivamente il **gradiente locale** (derivata dell'errore rispetto al peso) di ogni peso. La funzione dell'errore e' definita come lo scarto quadratico medio, per cui sia $e_j(n)$ l'errore commesso dal neurone j al passo n:

$$e_j(n) = d_j(n) - y_j(n)$$

Siano inoltre: * $E(n) = \frac{1}{2}\sum_{j \in Output} e_j^2$ (*errore totale output*) * $\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$ (*update rule del singolo peso*) * $v_j = \sum_{i=0}^{m} w_{ji} \cdot y_i$ (*output layer hidden*) * $\delta_j = -\frac{\partial E}{\partial v_j}$ (*gradiente layer hidden*)

Per prima cosa otteniamo la regola generica

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \tag{1}$$

$$= -\delta_j \cdot y_i \tag{2}$$

$$\tag{3}$$

per cui

$$\Delta w_{ji} = \eta \cdot \delta_j \ \cdot y_i$$

Successivamente, differenziamo tra due casi: neurone di input e output.

3.2.1.1 Caso Neurone Hidden Se j e' hidden, allora possiamo calcolare δ_j nel modo seguente

$$-\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$

Siccome

$$\frac{\partial E}{\partial y_j} = \sum_{k \in Next\ Layer} \frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial y_j} \tag{4}$$

$$= \sum_{k \in Next\ Layer} \delta_k \cdot w_{kj} \tag{5}$$

$$\tag{6}$$

Per cui concludiamo che

$$\delta_j = \varphi'(v_j) \cdot - \sum_{k \in Next\ Layer} \delta_k \cdot w_{kj}$$

3.3 Delta Rule Generalizzata

Se η e' basso, allora l'algoritmo convergera' molto lentamente, mentre se η e' troppo grande, causerebbe un comportamento instabile con oscillazioni sui valori dei pesi. Una soluzione e' l'introduzione di un termine di *momentum* α nella *delta rule* che tiene conto degli update precedenti.

$$\Delta w_{ji}(n) = \alpha \cdot \Delta w_{ji}(n-1) - \eta \cdot \delta_j(n) \cdot y_i(n)$$

Il *momentum* accelera la convergenza e ha un'effetto stabilizzatorio sulle direzioni che oscillano nel tempo.

3.3.1 Learning By Epochs

Nella *batch-mode* dell'algoritmo di backprop, i pesi vengono aggiornati solo dopo che tutti gli esempi sono stati processati, per cui il gradiente locale di ogni peso viene sommato con quelli delle altre iterazioni. Il processo di apprendimento procede fino ad un criterio di stop che puo' essere:

- **Total Mean Square Error Change**: Si dice che ha trovato una convergenza quando l'errore non cambia piu'
- **Generalization Based Criterion**: Dopo ogni epoca la rete viene testata su un set di validazione. Se le performance raggiunte sono adeguate allora si ferma.

  Il numero di esempi di training deve essere da 4 a 10 volte il numero di pesi della rete

  Teorema: Un singolo hidden layer e' sufficiente per calcolare qualsiasi approssimazione uniforme di un dato training set. Basta aumentare a piacimento il numero di unita' hidden.

4 Reti Neurali Multiradiali

Sono reti neurali che non hanno avuto tanto successo quanto le reti a backpropagation (siccome sono il fondamento del deep learning moderno), ma vale la pena studiarle comunque siccome sono molto simili alle *Extreme Learning Machines*. Si basano sul concetto di *funzione radiale*, cioe' delle funzioni la cui uscita dipende dalla **distanza** tra input e un determinato vettore **centro** memorizzato. L'idea e' quella di utilizzare queste funzioni come funzioni di attivazione, siccome funzionano come *attrattori locali*: piu' un vettore e' vicino al centro piu' sara' alta l'attivazione, viceversa se distante. L'architettura tipica di una RBFN consiste essenzialmente in un singolo layer hidden, i cui neuroni sono appunto le singole funzioni radiali. Il layer di output avra' invece un modello di neurone differente, che andra' essenzialmente a combinare i risultati del layer hidden per i pesi (cosi' come nel MLP).

I pesi sono solo tra layer hidden e layer di ouput

Siano:

- ϕ_i la funzione di attivazione del neurone *i-esimo* del layer hidden

- t_i il centro della funzione di attivazione *i-esima*, detto anche *centro di simmetria*
- m_1 il numero di neuroni hidden

L'output della rete per un generico esempio x sara':

$$y = w_1\phi_1(\|x - t_1\|) + \cdots + w_{m_1}\phi_{m_1}(\|x - t_{m_1}\|)$$

Con $\|x - t_i\|$ si indica la distanza Euclidea tra il centro e x

Con σ si indica un ulteriore parametro che quantifica lo *spread* della funzione, cioe' la sensibilita' del valore rispetto al centro: piu' sara' alto, meno la funzione di attivera' anche per esempi lontani dal centro.

> In generale, avere troppi neuroni hidden con spread molto basso, e' correlato ad una scarsa capacita' della rete a generalizzare: la rete si e' specializzata troppo sugli esempi del training set.

4.1 Apprendimento

Una volta formulata la rete dal punto di vista matematico, possiamo formulare anche il problema dell'apprendimento della stessa sotto lo stesso punto di vista.

Come visto, l'apprendimento richiede di trovare diversi parametri:

- I centri t_i delle funzioni radiali
- Gli spread σ di ogni funzione radiale
- I pesi w tra l'ouput layer e l'hidden layer

4.1.1 Centri

In generale, ci sono due metodologie principali per determinare i centri:

1. Inizializzare i centri delle funzioni selezionando m_1 esempi dal training set in modo casuale
2. I centri vengono calcolati mediante un algoritmo di clustering

La prima e' banale, mentre per la seconda l'algoritmo di clustering consiste nell'andare ad inizializzare i centri randomicamente, per poi iterativamente andare ad applicare una correzione al cluster piu' vicino per ogni esempio x del training set. L'algoritmo consiste quindi nei seguenti passaggi:

1. Inizializza tutti i centri scegliendoli dal training set in modo casuale
2. Estrai un esempio x dal training set
3. Determina il centro piu' vicino rispetto a x (sia t_k tale centro)
4. Applica la correzione del centro $t'_k = t_k + \eta[x - t_k]$
5. Ripeti fino a quando i centri non cambiano

La regola di correzione consiste nel *"tirare"* di una certa quantita' (proporzionata rispetto alla distanza tra il centro e l'esempio) il centro t_k verso i diversi esempi vicini ad esso.

> La metodologia di determinazione che si basa sul clustering permette anche di attenuare l'effetto dato dall'*attrattivita' locale* delle funzioni radiali

4.1.2 Spread

Lo spread viene determinato per **normalizzazione**. Siano:

- d_{max} la distanza massima tra tutte le coppie possibili dei centri che sono stati selezionati
- m_1 il numero di centri (*numero neuroni hidden*)

Allora lo spread e' definito come:

$$\sigma = \frac{d_{max}}{\sqrt{m_1}}$$

Definendo σ in questo modo, se utilizzassimo delle funzioni gaussiane come base delle RBF, otterremo che la formula della singola funzione di attivazione sara'

$$\phi_i(\|x - t_i\|^2) = exp\left(-\frac{m_1}{d_{max}^2}\|x - t_i\|^2\right)$$

4.1.3 Pesi

Veniamo ora al problema dell'apprendimento dei pesi tra l'hidden layer e l'output layer. Se noi abbiamo che $F(x)$ e' l'output della nostra rete e x_j un generico esempio del dataset, noi vogliamo che

$$F(x_j) = d_j$$

per tutti gli esempi x_j. Se noi sostituiamo a F la sua definizione data in precedenza, otteniamo

$$F(x_j) = \sum_{i=1}^{N} w_i \phi_i(\|x_j - x_i\|)$$

Se vogliamo che questo valga per tutti i $j = 1 \dots N$, avremmo un sistema a N righe e N incognite, per cui possiamo rappresentarlo in forma matriciale come

$$\Phi w = d$$

> Si noti come in questo punto stiamo ponendo che ogni neurone hidden abbia come proprio centro proprio un esempio di train, per cui ci saranno N neuroni hidden.

Secondo il teorema di Micchelli, se una matrice di interpolazione e' nonsignolare, allora possiamo invertire la matrice e ottenere una soluzione al sistema formulato in precedenza.

> La matrice possiamo tecnicamente invertirla, ma se abbiamo un determinante molto piccolo, otterremo un'inversa che per la presenza di errori numerici al calcolatore sarebbe inutile a lato pratico

A lato pratico, pero', questo tipo di risoluzione e' infattibile per due ragioni principali:

1. La rete andrebbe a fare overfitting siccome ogni neurone viene centrato su un particolare esempio del training set
2. La matrice di interpolazione Φ contiene un numero di entrate pari a N^2 -> apprendimento pesante su datasets molto grossi

La soluzione e' andare a diminuire il numero di neuroni hidden ponendolo $m1 < N$, e a risolvere il sistema mediante il calcolo della cosiddetta *pseudoinversa* Φ^+.

Formalmente, vorremmo solo un'approssimazione per cui

$$y_i(x_i) \approx d_i$$

cioe'

$$w_1\phi_1(\|x_i - t_1\|) + \cdots + w_{m_1}\phi_{m_1}(\|x_i - t_{m_1}\|) \approx d_i$$

Ma quindi anche in questo caso possiamo riformulare il problema come un sistema lineare di equazioni in forma matriciale:

$$\Phi w \approx d$$

questa volta la matrice non e' quadrata, per cui come detto per risolvere il problema utilizzeremo la **pseudoinversa** della matrice Φ, definita nel modo seguente: $\Phi^+ = (\Phi^T\Phi)^{-1}\Phi^T$. Moltiplicando a sinistra entrambi i membri per Φ^T e facendo diversi passaggi algebrici otteniamo che:

$$w = \Phi^+ d$$

per cui abbiamo trovato una formula per calcolare tutti i pesi. Questa tecnica e' chiamata appunto della *pseudoinversa*.

4.1.4 Riassumendo

L'apprendimento di una rete RBFN richiede i seguenti passaggi:

1. Determina i centri delle funzioni radiali mediante
   - Clustering
   - Assegnamento casuale
2. Calcola σ mediante normalizzazione
3. Determina i pesi w utilizzando
   - Il metodo basato sulla pseudoinversione
   - Un algoritmo di Least Mean Square come visto sui Percettroni

4.2 Confronto con FFNN

Le RBFN presentano diverse caratteristiche che le differenziano notevolmente dalle RBFN, anche se entrambe sono degli *approssimatori universali*, cioe' che sono delle funzioni in grado di approssimare bene a piacere un qualsiasi insieme di punti m-dimensionali.

Anche le RBFN possono risolvere i problemi che sono linearmente non separabili. Questo perche' in realta' il layer hidden di queste reti non fa altro che mappare le coordinate in input in uno spazio intermedio, trasformando il problema in un problema linearmente separabile equivalente.

Nonostante siano entrambi degli approssimatori universali, possiamo identificare differenze su diversi fronti tra le FFNN e le RBFN:

Fronte Rete	RBFN	FFNN
Architettura	Singolo Hidden Layer	Piu' Hidden Layer
Modello Neurone	Non-lineare per i neuroni dell'hidden layer, lineare per i neuroni dell'output layer	Tutti i layer condividono lo stesso modello di neurone: non-lineare
Funzione di Attivazione	Prende in input la distanza Euclidea tra centro ed esempio	Prende in input il prodotto interno tra input e pesi
Approssimazione	Usano una Gaussiana per costruire approssimazioni locali di mappe non lineari	Costruiscono approssimazioni globali di mappe non lineari

5 Extreme Learning Machines

Affermazione principale: Una rete FF con un livello nascosto di funzioni sigmoidi, i pesi e i bias tra il livello input e quello nascosto, possono essere scelti randomicamente e non piu' modificati, posto che le funzioni di trasferimento del livello nascosto siano infinitamente differenziabili (le sigmoidi sono infinitamente differenziabili). Le ELM sono nate essenzialmente per risolvere i problemi di regressione, ma possono essere adattate anche a risolvere task di classificazione.

Le conseguenze di questa affermazione sono essenzialmente 2:

1. Si puo' evitare di fare l'apprendimento su una grande quantita' di pesi: quelli del primo livello.
2. Una volta fissati i pesi del primo livello, i pesi tra il livello nascosto e il livello output possono essere determinati in maniera *analitica*.

Partiamo col definire il problema dal punto di vista matematico, proprio come e' stato fatto nel caso delle Reti Neurali Multiradiali.

> Segue un cambio di notazione rispetto a quella utilizzata dalla professoressa (per rimanere in linea con il paper)

Siano:

- $g(\cdot)$ una funzione di attivazione

- $\overline{x}_i$ l'esempio *i-esimo*
- $\overline{\beta}_i$ il vettore dei pesi tra l'*i-esimo* neurone di output e l'hidden layer
- $\overline{y}_i$ l'output della rete per l'esempio *i-esimo* (ha tante componenti quanti i neuroni in output)

Possiamo espandere la notazione vettoriale in modo da riscrivere tutto in notazione matriciale:

$$H \cdot \beta = y$$

dove:

- H e' la matrice delle funzioni di trasferimento applicate al campo dei neuroni hidden (l'output dell'input layer)
- β e' il vettore dei pesi tra i neuroni hidden e i neuroni di ouput
- y e' l'output della rete

Come detto, se noi volessimo una soluzione analitica, vorremmo ottenere una soluzione di questo sistema, ma in generale sappiamo che la matrice non e' quasi mai invertibile. Vorremmo quindi ottenere

$$H \cdot \beta \approx T$$

procedendo quindi per la pseudoinversa otteniamo che e' equivalente a

$$\beta = H^{+} \cdot T$$

dove T e' il target desiderato. L'apprendimento di queste reti e' riassunto quindi dai seguenti passaggi:

> *Assegnamento pesi random del layer in input + calcolo rimanenti per pseudoinversa*

Secondo Hwang questo tipo di apprendimento permette di ottenere delle performance migliaia di volte migliori rispetto all'algoritmo di backpropagation, per di piu' ottenendo performance di generalizzazione migliori.

Questo perche' se ipotizziamo che la superficie dell'errore sia determinata da due coordinate w_h, w_i , per semplicita' un solo peso hidden e un solo peso in input, abbiamo che fissando il valore di una coordinata, lo spazio di ricerca delle soluzioni si vada di conseguenza a ridurre drasticamente. Cio' corrisponderebbe a ridurre la ricerca da uno spazio 3-dimensionale a uno bi-dimensionale, determinato dal piano che interseca la soluzione ottimale.

In realta' Hwang va ad ignorare un elemento fondamentale: fissare i valori in modo random, non e' detto che ci posizioni l'iperpiano in un punto che interseca un ottimo (locale).

> L'algoritmo in se dovrebbe essere iterato piu' volte in modo da posizionare il piano in punti differenti

Il metodo continua ad essere di interesse pero' per diverse ragioni:

- E' comunque competitivo dal punto di vista di performance
- Avendo necessita' di essere generalizzato, predillige le scelte dei pesi di secondo livello *piccoli*

Il fatto che i pesi siano piccoli e' spesso associato al fatto che le reti con i pesi del livello hidden piccoli siano piu' in grado di generalizzare rispetto ad altre.

> **Teorema**: Dati N esempi distinti arbitrari scelti in maniera casuale formati come coppie $(\overline{x}_i, t_i)$, e una rete con N neuroni hidde, per ogni insieme di pesi e bias $\overline{w}_i, b_i$ scelti in maniera random da un qualsiasi insieme continuo $\overline{w}_i \in \mathbb{R}^n, b_i \in \mathbb{R}$, allora con propabilita' 1, la matrice H e' invertibile, e la distanza tra $H\beta$ e T puo' essere resa nulla.

L'algoritmo di apprendimento sfrutta questi due risultati teorici per apprendere la rete.

5.1 Algoritmo di Apprendimento

Dati:

- Un training set $N = \{(x_i, t_i) | x_i \in \mathbf{R}^n, t_i \in \mathbf{R}^m, i = 1, \ldots, N\}$
- Una funzione di attivazione $g(\cdot)$
- Il numero di nodi hidden $\tilde{N}$
- Il target $T = [t_1, \ldots t_N]^T$

1. Assegna randomicamente i pesi input w e i bias b
2. Calcola la matrice di output dei neuroni hidden H
3. Calcola i pesi output β per pseudoinversione $\beta = H^+ T$

Il numero di neuroni hidden $\tilde{N} << N$.

> Tanto piu' $\tilde{N}$ si avvicina al valore N, tanto piu' migliora l'apprendimento (non significa che migliori la generalizzazione)

Si deve inoltre fare attenzione al fatto che il metodo della pseudoinversa e' numericamente molto instabile quando la matrice $H^T H$ e' *quasi-singolare* Questo problema possiamo risolverlo andando ad applicare una regolarizzazione. La regolarizzazione non e' nient'altro che la somma di un termine chiamato *termine di regolarizzazione*. Applichiamo questa tecnica all'errore

$$E = E_D + \lambda E_\beta$$

Minimizzare E non consiste piu' solo nel minimizzare l'errore di discrepanza E_D che gia' conosciamo, ma anche nel minimizzare il secondo termine. Questo porta ad un vantaggio principale:

1. Si controlla meglio l'overfitting. In questo modo posso tollerare anche dei pesi che si avvicinano al target ma non uguali. Questo perche' la regolarizzazione produce pesi che in valore assoluto sono piccoli, ed esiste una serie di teoremi dimostrati che dicono che piu' piccola e' la norma dei pesi, migliori sono le performance di generalizzazione.

2. Allontana la matrice dalla condizione di *quasi-singolarita'*, che porterebbe a soluzioni numericamente instabili. Questo perche' la soluzione diventa $\beta^* = (H^T H + \lambda I)^{-1} H^T T$, e posso quindi fare tuning del parametro λ per evitare la quasi singolarita'.

6 Self Organizing Maps

Le self organizing maps sono delle reti che il cui setting di apprendimento e' *non supervisionato*: imparano a classificare un set di input associando ogni input ad un neurone simile. In questo senso, scoprono delle regolarita' nell'insieme di input.

> *Le SOM imparano a produrre una rappresentazione (tipicamente) bidimensionale dei dati in input, preservandone l'ordine topologico.*

Preservare l'ordine topologico significa che input simili vengono mappati in neuroni che sono vicini tra did loro all'iterno della griglia.

In generale sono delle griglie bidimensionali di neuroni. Le griglie sono singole, per cui non sono delle reti neurali a multilivello. I neuroni sulla griglia sono relazionati tra loro solamente per mezzo di una relazione di *vicinanza*, per cui non esiste nessuna connessione (mediata da pesi) tra i neuroni sulla griglia.

> *E' possibile anche utilizzare una topologia* lineare* al posto di una topologia a *griglia* (o esagonale)*

Ogni neurone e' connesso mediante un insieme di pesi ad un vettore di input. Per cui, ogni neurone avra' associato:

- Un vettore di pesi (un peso per ogni feature), detto anche *prototype vector*
- Una posizione (coordinata che non cambia mai) all'interno della griglia

Best Matching Unit (BMU): E' il neurone che si attiva di piu' in tutta la rete quando gli viene presentato un esempio.

La BMU del neurone *i-esimo* viene calcolata in termini della distanza Euclidea tra il vettore di pesi w_i e il vettore di input x. La BMU e' quella con attivazione minima:

$$BMU(x) = \arg\min_i \|x - w_i\|$$

Questo perche' c'e' una relazione precisa tra il prodotto scalare e la norma:

> *Il prodotto scalare piu' grande e' quello la cui distanza e' minima.*

In questo senso determiniamo il neurone che si attiva di piu' solamente considerandone la distanza, senza dover calcolare esplicitamente l'attivazione (per mezzo del prodotto scalare).

L'apprendimento di una rete SOM si fonda quindi su due principi:

- Principio di **competizione**: ogni neurone va a "*competere*" con tutti gli altri per potersi attivare. I pesi del neurone che vince la competizione (e che quindi si attiva), vengono modificati in modo da aumentare la probabilita'

che sia di nuovo in grado di attivarsi per istanze di input simili (o per la stessa istanza di input).
- Principio di **cooperazione**: il cambiamento dei pesi non riguarda solo le BMU, ma anche i neuroni vicini ad essa

Ovviamente questi due principi vengono tenuti in conto contemporaneamente dalla regola di update dei pesi.

> *Le BMU che si attivano vicine tra di loro rappresentano una determinata classe.*

La regola di update dei pesi e' la seguente:

$$w_i(n+1) = w_i(n) + \eta(n)(x - w_i(n))$$

In questo senso, i pesi della BMU vengono *trascinati* verso l'esempio di input (ovviamente) in modo proporzionale al learning rate η. Ovviamente, dobbiamo anche introdurre il concetto di cooperazione. Idealmente, ogni volta che viene presentato uno stimolo (esempio) alla mappa, l'update rule viene applicata alla BMU e a tutti i neuroni circostanti ad essa. La relazione di *vicinanza* dipende direttamente dalla distanza del segnale rispetto alla BMU. Piu' formalmente la *vicinanza* tra due unita' j e i e' descritta da una Gaussiana:

$$h_{j,i}(n) = exp(-\frac{d_{j,i}^2}{2\sigma(n)^2})$$

Tenendo in conto questa relazione, l'update rule **per ogni neurone** j della mappa e'

$$w_j(n+1) = w_j(n) + \eta(n) \cdot h_{j,i}(n) \cdot (x - w_j(n))$$

Nella formula, i e' l'indice del neurone BMU dell'iterazione attuale, per cui e' evidente come i neuroni che sono piu' coinvolti dal cambiamento del peso siano quelli che hanno $h_{j,i}$ grande, cioe' tali per cui la distanza tra loro e al BMU e' piccola (in questo caso, il massimo sara' quando $i = j$, cioe' quando il neurone j e' la BMU che ha distanza tra se stessa $= 0$).

> *Il coinvolgimento decresce all'aumentare della distanza, per cui i pesi saranno cambiati sempre meno per le unita' piu' lontane.*

Cosi' come il learning rate, anche l'ampiezza σ della Gaussiana decresce con l'aumentare il passare del tempo, per cui segue una decrescita esponenziale, data dalla seguente legge

$$\sigma(n) = \sigma_0 exp\left(-\frac{n}{\tau}\right)$$

per cui sempre meno neuroni verranno considerati dalla regola di update dei pesi. Tipicamente, l'apprendimento di una self organizing map e' divisa in due fasi:

1. In questa prima fase, la mappa si macro-organizza, cioe' il *"vicinato"* delle BMU ha dei valori tali per cui ogni cambiamento di peso va a influenzare in modo molto forte gli altri pesi della mappa. In questa fase il learning rate e' abbastanza alto
2. In questa fase successiva e' detta di convergenza, poiche' il *"vicinato"* si e' ristretto cosi' tanto da essere in grado di influenzare solo piu' una unita' alla volta

6.0.1 Errore

Ci due misure per quantificare l'errore delle SOM:

- **Errore di quantizzazione**: Misura l'errore medio tra ogni stimolo e il neurone che lo rappresenta meglio (la BMU). In generale, misura quanto bene la mappa rappresenta bene i dati.

$$QE = \frac{\sum_{l=1}^{L} \|x_l - w_{i(x_l)}\|}{L}$$

- **Errore Topologico**: La topologia e' preservata se punti vicini nello spazio di input sono vicini anche nel reticolo di output e viceversa. L'errore topologico misura

$$TE = \frac{1}{L} \sum_{l=1}^{L} u(x_l)$$

(dove $u(x)$ e' $= 1$ se le prime due BMU sono adiacenti tra loro nello spazio di output, 0 altrimenti)

6.0.2 Casi d'uso e applicazioni

Le Self Organizing Maps vengono utilizzate principalmente per i seguenti task:

- Clusterizzare dati
- Visualizzare dati con dimensionalita' molto alta
- Comprimere dataset molto grossi -> Creando una SOM a partire da un grosso dataset, e' possibile prendere tutte le zone adicenti delle SOM e renderle una singola istanza.

Applicazioni delle SOM:

- Riconoscere fonemi
- Descrivere modelli cognitivi

7 Hopfield Networks

Particolarmente apprezzate da scienziati cognitivi e fisici, le reti di Hopfield fanno parte del repertorio che ogni DL engineer deve sapere. Sono reti che risolvono task di tipo *pattern completion* (es. Data una porzione di frase, la rete riesce a completarla). Hanno un insieme di memorie fondamentali *"cablate"*

all'interno della rete. Data una versione corrotta di una memoria fondamentale, la rete puo' ricostruirla gradualmente.

> *Le reti di Hopfield possono essere usate per modellizzare un gran numero di fenomeni cognitivi degli esseri umani tra le quali memoria, lessico, afasia, abilita' di riconoscere facce, ecc..*

Una rete di Hopfield e' un insieme di **neuroni** completamente connessi tra di loro, per cui ogni neurone e' connesso a tutti gli altri neuroni eccetto se stesso.

> *Come regola generale, vale anche che $w_{i,j} = w_{j,i}$ (i pesi sono simmetrici)*

Ogni neurone puo' avere solo due attivazioni possibili: 1 o -1. Cosi' come ogni altra rete vista fin'ora, l'attivazione di un neurone e' calcolabile applicando la funzione di attivazione alla somma pesata di tutti gli altri neuroni della rete connessi ad esso. La funzione di attivazione tiene conto anche dell'attivazione al passo precedente. Piu' formalmente, sia $v_j(n)$ il campo di output del neurone j all'iterazine n:

$$v_j(n) = \sum_{i=0}^{N} w_{ij} \ \cdot \ y_i(n)$$

Allora l'output del neurone j all'iterazione n e' calcolabile nel modo seguente:

$$\varphi(v_j)(n) = \begin{cases} 1 & \text{if } v_j(n) > 0 \\ -1 & \text{if } v_j(n) < 0 \\ \varphi(v_j)(n-1) & \text{if } v_j(n) = 0 \end{cases}$$

Ovviamente:

$$y_j(n) = \varphi(v_j(n))$$

Intuitivamente, se l'attivazione attuale del neurone e' 0, allora sara' uguale al valore che aveva precedentemente. L'attivazione potrebbe quindi cambiare o non cambiare.

> *In questo senso, ogni neurone funziona come se fosse un automa a stati finiti, la cui funzione di transizione e' appunto φ*

7.0.1 Funzionamento Generale

L'idea principale di queste reti e' quella di avere un neurone per ogni input, per cui gli input sono rappresentati come vettori di features che hanno solo due valori: -1 e 1. Una volta presentato l'input alla rete, ogni neurone corrispondente alla determinata feature verra' attivato o meno a seconda del valore dell'input. Successivamente, si applica la funzione di attivazione a neuroni scelti in modo casuale fino a quando nessun neurone cambia piu' il suo valore (*stato stabile*). In questo senso, l'**output** della rete non e' nient'altro che lo stato stabile raggiunto dalla rete.

*Qualsiasi sia la configurazione di input, si raggiungera' **sempre** uno stato stabile in un numero finito di passi.*

Una rete di Hopfield non ha un solo stato stabile, ma ce ne possono essere piu' di uno. Come regola generale, vale che se ho uno stato stabile, avro' che anche il suo **negato** e' uno stato stabile.

7.0.2 Apprendimento (Storage Phase)

Come detto in precedenza, la rete ha diverse memorie fondamentali salvate. Tali memorie le possiamo vedere come degli *attrattori*: stati stabili verso la quale la rete converge con l'iterazione ripetuta della funzione di attivazione. L'apprendimento (la fase di storage) non e' altro che aggiustare i pesi in modo tale da rendere una particolare memoria un *attrattore* (cioe' uno stato stabile).

Intuitivamente, uno stato e' stabile quando:

- Tutte le unita' con la stessa attivazione sono connesse da pesi positivi
- Tutte le unita' con le attivazioni opposte sono connesse da pesi negativi

Questa principio intuitivo e' una generalizzazione del Principio di Hebb.

Principio di Hebb: *"Neurons that fire together, wire together"*

Per cui se due neuroni sono entrambi attivi (1), la sinapsi che li colleghera' dovra' avere un valore alto (1). In questo modo l'attivazione di uno rafforza l'attivazione dell'altro. Come detto la nostra regola e' una generalizzazione, per cui anche due neuroni non attivi (-1) verranno lo stesso connessi da pesi positivi. Al contrario, i neuroni discordi devono avere pesi sviluppati negativi, in modo da indebolirsi.

Vediamo piu' nel dettaglio la regola di update dei pesi. Prendiamo un caso specifico e ipotizziamo di voler memorizzare solamente una memoria fondamentale f_1. Come gia' detto, non e' nient'altro che un vettore con tante quante componenti sono i neuroni N. Apprendere questa memoria corrispondera' quindi ad applicare la seguente regola per tutti i pesi i, j

$$w_{j,i} = f_1(i) \cdot f_1(j)$$

dove $f_1(i)$ e' l'*i-esima* componente di f_1 (vale lo stesso per j). Questo perche' se i e j hanno valori discordi, allora anche il peso sara' negativo, mentre se hanno segno concorde avranno peso positivo. Non ci resta che generalizzare ad un numero arbitrario di memorie fondamentali. Ipotizziamo quindi di avere M memorie fondamentali $f_1, \dots f_M$. La regola generalizzata non e' nient'altro che la *media* della singola regola di update applicata ad ogni memoria:

$$w_{j,i} = \frac{1}{M} \sum_{k=1}^{M} f_k(i) \cdot f_k(j) \quad \text{if } j \neq i$$

*Questa regola viene applicata una sola volta, per cui in questo senso si parla di **one-shot learning**.*

7.0.3 Inferenza (Information Retrieval)

Come gia' detto, una volta memorizzate le memorie fondamentali, possiamo utilizzare la rete per fare information retrieval. L'algoritmo per far cio' ha una garanzia teorica di convergenza ed e' particolarmente semplice.

1. Dato il segnale x^*, inizializza tutti i neuroni con i valori corrispondenti al segnale
2. Itera fino alla convergenza applicando la regola di update φ a neuroni scelti in modo casuale
3. Se per ogni neurone l'applicazione della regola non genera un nuovo valore, allora abbiamo raggiunto uno stato stabile e si puo' terminare ritornando tale stato

7.0.3.1 Teorema di convergenza Il teorema di convergenza si basa su una misura di **Energia**. L'energia misura la propensione della rete a cambiare stato. Piu' l'energia e' alta, piu' sara' possibile che la rete cambi stato. Ogni stato ha un valore associato di energia, in generale ci sono 2^N stati possibili in una rete con N neuroni. Conseguentemente avremo lo stesso numero finito di valori di energia.

Intuitivamente il teorema di convergenza dice che ogni volta che si applica l'update rule, l'energia totale della rete diminuira' ad ogni stato successivo. Ma siccome il numero di valori di energia possibili e' effettivamente finito, si arrivera' prima o poi ad un punto in cui l'energia associata ad uno stato sara' 0. Questo stato e' appunto uno *stato stabile*.

Formalmente, l'energia di un determinato stato e' calcolata nel modo seguente:

$$E(n) = -\frac{1}{2}\sum_{i}^{N}\sum_{j}^{N} w_{i,j}(n) \cdot y_i(n) \cdot y_j(n)$$

L'energia misura quanto e' propensa la rete a cambiare stato, per cui e' una misura che vorremmo vada a 0 dopo ogni iterazione. Ogni prodotto $w_{i,j}(n) \cdot y_i(n) \cdot y_j(n)$ e' positivo quando:

- $y_i(n)$ e $y_j(n)$ hanno lo *stesso segno* e $w_{i,j}(n)$ e' *positivo*
- $y_i(n)$ e $y_j(n)$ hanno *segno opposto* e $w_{i,j}(n)$ e' *negativo*

Iniziamo con lo scomporre dalla somma un generico y_k, per cui otteniamo

$$E = -\frac{1}{2}\sum_{i\neq k}^{N}\sum_{j\neq k}^{N} w_{i,j} \cdot y_i \cdot y_j + 2\sum_{j\neq k}^{N} w_{kj} \cdot y_k \cdot y_j$$

Facciamo la stessa cosa, ma consideriamo invece E', cioe' l'energia dopo che y_k e' cambiato di stato ed e' diventato y'_k

$$E' = -\frac{1}{2}\sum_{i\neq k}^{N}\sum_{j\neq k}^{N} w_{i,j} \cdot y_i \cdot y_j + 2\sum_{j\neq k}^{N} w_{kj} \cdot y'_k \cdot y_j$$

Allora, la differenza tra l'energia al passo attuale E e quella al prossimo passo E' e'

$$E - E' = \left(-\sum_{j \neq k} w_{kj} \cdot y_j \right) \cdot (y_k - y'_k)$$

Per cui possiamo distinguere due casi:

- Se y_k passa da $+1$ a -1, allora $(y_k - y'_k) > 0$ e $\sum_j w_{kj} y_j < 0$
- Se y_k passa da -1 a $+1$, allora $(y_k - y'_k) < 0$ e $\sum_j w_{kj} y_j > 0$

Se si sostituiscono i valori all'interno della relazione, si ottera' sempre che $E - E' > 0$ per cui $E > E'$.

Abbiamo dimostrato che ad ogni iterazione l'energia si abbassa. ■

7.0.4 Pro

1. Sono in grado di completare pattern parziali
2. Imparano a generalizzare: riescono a capire che un input simile ad una memoria fondamentale, appartiene di fatto alla classe di quella memoria fondamentale
3. Fault tolerant: anche se alcune sinapsi vengono rimosse, la rete continua comunque a funzionare (posto che la rete sia sufficientemente grande)
4. Sono in grado di estrarre prototipi: se vengono presentate degli input simili a tante memorie fondamentali, la rete si stabilizzera' in un punto in mezzo che e' prototipo di tutte le memorie fondamentali
5. La regola di apprendimento si basa sul principio di Hebb, che e' un principio *biologicamente plausibile*

7.0.5 Contro/Problematiche

1. Esistono stati stabili imprevisti: come detto, l'opposto di uno stato stabile e' anch'esso uno stato stabile. Inoltre, l'apprendimento di piu' memorie fondamentali insieme introduce altri stati stabili detti *spuri* che non si volevano introdurre. In questo senso, questi stati *interferiscono* con gli stati delle memorie fondamentali.
2. Non tutte le memorie fondamentali possono essere resi stati stabili.
3. La capacita' di storage di memorie fondamentali e' limitata. In generale, una regola ci dice che possiamo memorizzare $0.14 \cdot N$ memorie fondamentali per una rete di N dimensioni.

8 Boltzmann Machines

Problema: *Le reti di Hopfield incorrono in stati* ***spuri***

Come abbiamo visto esiste un limite di memorie calcolabile per evitare di incorrere in questo problema: $0.14 \cdot N$ memorie (dove N e' il numero di neuroni) Se noi non volessimo seguire questa regola, come possiamo eliminare questi stati stabili spuri? Questa e' la domanda che ha dato origine alle Boltzmann Machines. Una

delle idee principali proposte consiste nel lasciare che la rete finisca in uno stato spuro a partire da uno stato iniziale casuale. In caso finisca in questo stato, si applica una procedura di *unlearning* cioe' si va a rimuovere la memoria di quello stato spuro.

> *Il procedimento di* ***unlearning*** *e' stato proposto come modello per spiegare a cosa servano effettivamente i* ***sogni***

Un'altra idea che fu proposta fu anche quella di cambiare l'architettura fondamentale della rete di Hopfield. L'idea fondamentale e' che al posto di utilizzare la rete per salvare *memorie*, la si usa per costruire *interpretazioni* dell'input.

Rispetto alle reti di Hopfield, le RBM vanno a introdurre un layer esterno, rendendo tutti i neuroni connessi (come la rete di Hopfield) un unico layer hidden. Le RBM hanno quindi due layer distinti:

1. **Hidden layer**: servono a trovare correlazioni di attivazioni a livello visible e riforzano questi pattern a livello visible quando ne si presenta una versione corrotta.
2. **Visible layer**: riceve l'input e fa pattern completion.

I neuroni dell'hidden layer sono completamente connessi con tutti i neuroni del visible layer, ma i neuroni di ogni livello non sono connessi tra loro.

La terza idea addizionale e' che le reti di Hopfield vanno solo a decrescere l'energia totale della rete, per cui la probabilita' di finire in un minimo locale e' molto grande in generale. Le reti di Boltzmann vanno a introdurre un elemento di stocasticita' in modo da considerare altri stati dei neuroni e dar modo di cercare altri minimi nella funzione dell'energia.

La regola dell'attivazione dei neuroni nelle Boltzmann Machine ha quindi anche una componente stocastica per tenere in conto questo comportamento. Per i neuroni hidden, la regola e' la seguente:

$$p(s_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}}$$

Dove $\Delta E_i = E(s_i = 0) - E(s_i = 1)$ e' l'*energy gap*, che indica quanto sia propenso al cambiamento il neurone i, e T indica la *temperatura* che quantifica la *stabilita'*.

L'energia $E(s_i = x)$ viene calcolata come l'energia totale della rete supponendo che i abbbia il valore x. In realta', per definizione (e cosi' evitare di calcolare tutte le energie totali a mano ogni volta), l'enegy gap e' definito come:

$$\Delta E_i = \sum_j w_{i,j} \cdot s_i + b_i$$

Ovviamente siamo parlando di una probabilita', per cui l'attivazione sara' un valore in $[0; 1]$, per cui il vero valore sara' ottenuto da una successiva procedura di sampling dalla distribuzione ottenuta.

Le macchine di Boltzmann sono modelli Generativi: Imparano a riprodurre ad un livello visivo il training set. In altri termini, modellano la probabilita' congiunta $P(X, Y)$

L'algoritmo di apprendimento e' chiamato *Contrastive Divergence*. Esso si basa sulla seguente idea: quando si presenta un esempio alla rete, se ne calcola l'attivazione per ogni neurone. Una volta calcolata, si ripete lo stesso passaggio utilizzando pero' la rete il cui stato e' stato calcolato al passo precedente. In altri termini, si fa un "applicazione ripetuta" del passaggio per un numero molto grande di volte. Idealmente, al tendere di questo numero all'infinito, otterremo uno stato verso cui la rete converge.

L'idea di update dell'algoritmo e' quindi quella di assegnare i pesi in modo che la discrepanza tra l'esempio che e' stato presentato alla rete e lo stato finale (quello dopo l'applicazione di infiniti passi di update) e' minima (o nulla).

Sorprendentemente, il numero di passi per ottenere questo stato a cui tende la rete non e' infinito, ma sono solo 2. Con questa conoscenza, si puo' costruire l'algoritmo di apprendimento, che consiste nei seguenti passaggi:

- Si presenta un esempio alla rete forzando l'attivazione dei neuroni visible ad essere pari ai valori delle features dell'esempio (*visible clamping*)
- Si calcolano le attivazioni dei neuroni hidden (facendo samplig dalle probabilita' calcolate)
- Date le attivazioni dei neuroni hidden, si calcolano le attivazioni dei neuroni visible
- Si ri-calcolano nuovamente le attivazioni dei neuroni hidden
- Per ogni coppia di neuroni i, j (i visible, j hidden) si valuta la differenza tra il prodotto dell'attivazione del neurone visible v_i e dell'attivazione del neurone hidden h_j allo stato iniziale, e lo stesso prodotto ma allo stato finale. In altri termini:

$$< v_i h_j >^0 - < v_i h_j >^1$$

dove $< \cdot >^k$ indica lo stato k-esimo. L'update rule e' quindi

$$\Delta w_{i,j} = \eta(< v_i h_j >^0 - < v_i h_j >^1)$$

Le singole unita' hidden vanno a riconoscere determinate porzioni del pattern

Le unita' hidden si fanno carico di attivare determinate porzioni di unita' visible. In questo modo, se un pattern corrotto attiva un determinato neurone hidden, questo andra' ad attivare di conseguenza le unita' visible ad esso collegate per ricostruire il pattern che "riconosce".

9 Deep Restricted Boltzmann Machines

Le RBM possono essre combinate tra di loro per ottenere modelli deep piu' complessi. Il problema delle reti profonde e' che e' difficile trovare dei minimi globali. Al crescere del numero dei layers auenta anche la difficolta' di apprendimento.

Le reti convolutional pongono dei limiti architetturali per sorpassare questo problema: I layer non sono tutti densi, i pesi sono condivisi, ecc.. L'idea delle Deep RBM per risolvere questo problema e' diversa e si basa sul concetto di *pre-training*.

> *Al posto di inizializzare i pesi in modo casuale, si cerca di inizializzarli in modo intelligente*

Nella letteratura, la fase di pretraining e' stata implementata in due modi diversi, a seconda del metodo con cui si inizializzano i pesi in maniera informata:

1. Autoassociators
2. Restricted Boltzmann Machines

Gli autoassociators sono delle reti feedforward con un solo livello hidden, il cui compito e' quello di *impare a riprodurre a livello dell'output l'input stesso*.

> *Associa un determinato input a se stesso passando attraverso un livello hidden*

Il training di un autoassociator viene fatto mediante backpropagation in cui l'input e' anche il target. In questo senso, l'apprendimento e' *non-supervisionato*. Il metodo consiste nel prendere a coppie i livelli della rete profonda in cui fare pretraining, li si considera come un autoassociator e si va a fare l'operazione descritta in precedenza: provando a far matchare input e output. Sperabilmente la rete dovrebbe partire in uno spazio dei pesi vicino ad un buon minimo.

Come detto, al posto degli autoassociators possiamo utilizzare anche le Restricted Boltzmann Machines.

> *"To recognize shapes, first learn to generate images"*

Il procedimento e' simile a quello visto in precedenza. Si inizia a considerare l'input layer della rete profonda e il livello immediatamente successivo e li si tratta come una RBM. (Il livello visible sara' l'input layer e quello successivo l'hidden). Successivamente si applica l'algoritmo di contrastive divergence a questi due layer, trattandoli appunto come se fossero una RBM. Cosi' facendo, questi layer saranno in grado di generare al livello visibile il training set con pochi errori. Una volta fatto cio', si passa al prossimo layer, per cui quello che prima era hidden diventa visible, e il layer successivo nella rete profonda diventa il layer hidden della nuova RBM. Essenzialmente si avanza considerando i layers dell'architettura profonda a coppie.

Consideriamo un esempio in cui abbiamo una rete deep a 4 livelli (i, h_1, h_2, o).

- Considera i livelli (i, h_1) come una RBM (i *input*, h_1 *hidden*) e fai il training con l'algoritmo di CD *utilizzando gli esempi del training set*
- Considera i livelli (h_1, h_2) come una RBM (h_1 *input*, h_2 *hidden*) e fai il training con l'algoritmo di CD *utilizzando i segnali di output salvati al livello precedente* h_1
- Considera i livelli (h_2, h_o) come una RBM (h_2 *input*, o *hidden*) e fai il training con l'algoritmo di CD *utilizzando i segnali di output salvati al livello precedente h_2

Dopo aver applicato questa tecnica, si succede una fase cosiddetta di *fine-tuning* in cui viene fatto il training tramite *backpropagation* (Stochastic Gradient Descent

+ Autodiff) Tramite questa tecnica di RBM stacking, e' possibile ottenere diversi modelli. L'applicazione piu' classica di questa tecnica permette di ottenere modelli discriminativi, ma variando il numero di unita' hidden di ogni layer possiamo ottenere anche degli *autoencoders* oppure dei modelli *generativi*.

9.0.1 RBM-Based Deep Autoencoders

L'idea degli autoencoders e' quella di utilizzare queste *pile* di RBM in modo da fare un lavoro di *compressione* dei dati. Per fare cio' si crea un'architettura composta da RBM impilate tra loro con sempre meno *hidden units*, cosi' che si riduca progressivamente la dimensionalita' dei dati. Questa struttura e' detta *encoder* per appunto la sua natura di codifica dei dati in dimensionalita' piu' basse. Il *decoder* e' invece una struttura simile all'*encoder* che viene creata facendo il training dei vari livelli considerandoli in ordine inverso. Il compito del *decoder* e' quello di riportare alla dimensione originale i dati compressi.

9.0.2 Deep Belief Networks

Le DBN sono dei modelli *generativi*, cioe' hanno il compito di approssimare molto bene la distribuzione di probabilita' congiunta di *features* e *target*. Le DBN hanno un'importante differenza rispetto alle altre architetture basate su questo metodo di stacking, ed e' nella fase di *fine tuning*. L'algoritmo per far cio' e' chiamato **Contrastive Wake - Sleep**. Tale algoritmo necessita di aggiungere dei pesi che non sono parte integrante del modello e sono necessari solo all'esecuzione dell'algoritmo detti *pesi discriminativi*, che vanno in direzione dal basso verso l'alto (questo e' ragionevole dal momento che se andiamo dall'alto verso il basso otteniamo un modello discriminativo) L'algoritmo consiste poi nell'alternare due fasi per ogni elemento del training set:

1. Fase di *"wake up"*: L'elemento corrente del training set viene presentato al livello visible propagando l'attivazione dei livelli sovrastanti fino a raggiungere l'ultimo livello. Ogni qual volta che si passa da un livello a quello sovrastante, (si utilizzano dei pesi discriminativi che non sono parte del modello) si vanno ad aggiustare i pesi che vanno invece nella direzione opposta (pesi generativi). In questo modo, dalla configurazione hidden appena ottenuta, aumenteranno le chance in futuro di generare lo stimolo che le ha causate (muovendosi verso il basso).
2. Fase di *"sleep"*: Si parte dal layer piu' esterno (quello di output) che conterra' una configurazione data dalla fase di *"wake up"*, facendo l'inverso: ogni volta che si passa da un livello a quello sottostante, (pesi generativi) allora si vanno ad aggiustare i pesi nella direzione opposta (pesi discriminativi), in modo che il segnale al layer piu' alto possa essere piu' plausibile a partire dal segnale che e' stato ottenuto al livello sottostante.

   I pesi discriminativi fanno parte solo dell'algoritmo e non sono parte del modello generativo in se

L'idea di questo algoritmo e' quella di *"cristallizzare"* in un certo senso i pattern che sono stati appresi nella fase di *pre-training*.

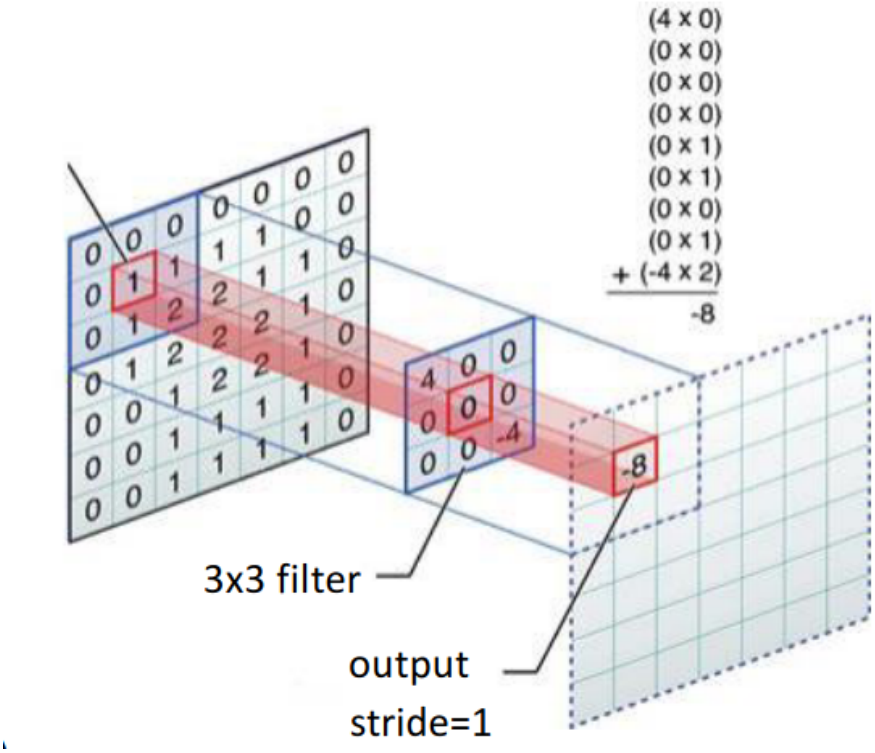
10 Convolutional Neural Networks

Le reti neurali convoluzionali differiscono dal MLP su diversi fronti:

- I neuroni sono connessi ai neuroni del livello precedente **solo localmente**, per cui ogni neurone fa un *processing locale*, diminuendo il numero totale di connessioni
- I pesi sono condivisi in modo che neuroni dello stesso livello processino nello stesso modo diverse porzioni dell'input. Anche in questo caso il numero di connessioni e' drasticamente ridotto

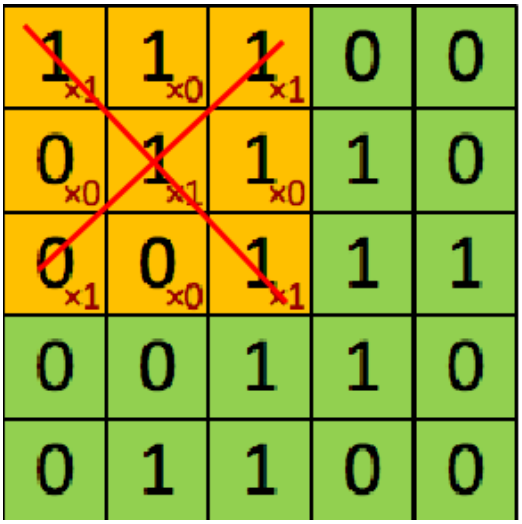
Il processing dei neuroni e' a livello *locale*. Ogni neurone e' connesso solo ad un gruppo di neuroni precedente, ma non tutti, per cui la computazione avviene in un sottospazio di input Questo tipo di processing dell'input sono chiamati *filtri convoluzionali*. L'output e' detto *feature map*. Ogni neurone di una feature map e' il risultato di una computazione di una porzione dello spazio delle features, fatto da neuroni con lo stesso insieme di pesi. Tipicamente, l'architettura di queste reti ha i layer intermedi come quelli descritti, quindi composti da filtri convoluzionali, mentre i layer finali sono tipicamente completamente connessi, in modo da operare da MLP per fare la classificazione finale.

Questa tipologia di reti e' utilizzata sopratutto nell'image processing/recognition. Difatti, il passaggio in un livello convoluzionale equivale ad applicare di fatto dei filtri digitali alle immagini in input. Ogni neurone puo' essere pensato come ad uno *stencil* (maschera) che applica una determinata transformazione in una determinata porzione dell'immagine. Lo spazio "di slittamento" dello stencil e' chiamato *stride*



Si noti dall'immagine come una porzione dell'input (quadrato piu' a sinistra) venga mappata dallo stencil (quadrato nel mezzo) descritto da 9 pesi.

Intuitivamente, se rimaniamo nell'ambito dell'image processing, i pesi che rappresentano uno stencil sono effettivamente una particolare **forma** riconosciuta dal layer.



Nel caso dell'immagine, vediamo come lo stencil (la prozione gialla) vada a riconoscere per mezzo dei pesi assegnati una forma a "X" (delineata in rosso). Cioe' i valori che non sono nulli su una X. In generale, se l'output e' 5, sappiamo che la "X" sara' sicuramente presente (cioe' tutti gli input in quelle posizioni =1)

> *I filtri convoluzionali riescono in qualche modo a ridurre anche la dimensionalta'. Se noi applicassimo un filtro 2x2 in un filtro 3x3, otterremmo un output 2x2.*

Per i valori sui bordi, di solito vengono aggiunti opportunamente dei valori di padding.

> **Tensore**: *Matrice a piu' di due dimensioni.*

10.1 Livelli

Pooling: Il compito del livello di pooling e' quello di prendere un determinato insieme di neuroni del livello precedente e di "aggregarne" successivamente l'informazione. La tipologia di aggregazione determina la tipologia di pooling:

- **Average Pooling**: Aggrega gli input attraverso la media
- **Max Pooling**: Aggreaga gli input facendone il massimo

Lo scopo del pooling e' dare *invarianza* rispetto alle semplice trasformazioni degli input, mantenendo una quantita' di informazione sufficiente per lo scopo *discriminativo* della rete neurale.

> *Il livello di pooling non ha nessun tipo di parametro da imparare.*

Livello ReLU: Livello di neuroni che applica una funzione di attivazione di tipo Rectified Linear Unit. $f(x) = 0$ con $x \leq 0$, mentre $f(x) = x$ con $x > 0$. In modo compatto: $f(x) = max(0, x)$.

Livello di Uscita: Il livello di uscita e' in genere costituito da neuroni che hanno una funzione di attivazione con funzione *softmax*, definita come

$$z_k = f(x_k) = \frac{e^{v_k}}{\sum_{c=1...s} e^{v_c}}$$

dove:

- s e' il numero di unita' in output
- v_k e' il campo in uscita del k-esimo neurone

Cioe' calcola e elevato al campo in uscita del k-esimo neurone, diviso la somma della stessa quantita' per tutti gli altri. Siccome questa e' una normalizzazione, la sommma di tutti i valori z_k e' $= 1$, per cui ogni neurone in output dara' una *probabilita'*.

Funzione di Errore: Al posto di utilizzare l'errore quadratico medio, si utilizza la *Cross-Entropy*. Siano p e q due distribuzioni. La *cross-entropy* misura quanto q differisce da p

$$H(q, p) = -\sum_{v} p(v) \cdot log(q(v))$$

Nel nostro caso, p e' la distribuzione che noi vogliamo ottenere e q quella ottenuta in output.

10.2 Seminario su Convolutionals Neural Networks (Marco Roberti)

Il deep learning ha 4 componenti principali:

- **Dati**: il dataset (*che spesso definisce anche il task da risolvere*)
- **Architettura della rete**: tipologia di rete neurale che stiamo utilizzando
- **Processo di apprendimento**: algoritmo di apprendimento
- **Misura dell'errore**

  *Fissata l'**architettura**, le si danno in pasto i **dati**. L'output della rete viene confrontato con i dati attesi attraverso una **misura d'errore**, che viene utilizzata a sua volta dall'**algoritmo di apprendimento** per correggere i pesi della rete.*

Le reti convoluzionali si prestano particolarmente bene al task di *image processing*. Nonostante siano utilizzate quasi solo unicamente per questo task, non e' l'unico task in cui possono essere impiegate. Un'immagine:

- **Grayscale** puo' essere rappresentata come una matrice $w \times h$, i cui valori tra 0-255 indicano l'intensita' di nero
- **Immagine a colori**: puo' essere rappresentata come un *tensore* $w \times h \times 3$, dove ogni *"matrice"* delle 3 rappresenta un particolare *color channel*

Ipotizziamo di avere il task di riconoscere il numero scritto in delle immagini rappresentanti dei numeri in grayscale. (MNIST) Se volessimo utilizzare un MLP, dovremmo fare il flattening dell'input in primo luogo, per cui ogni pixel e' effettivammente una feature indipendente e distinta dalle altre. L'MLP non e' proprio adatta a risolvere questo tipo di task per diverse ragioni:

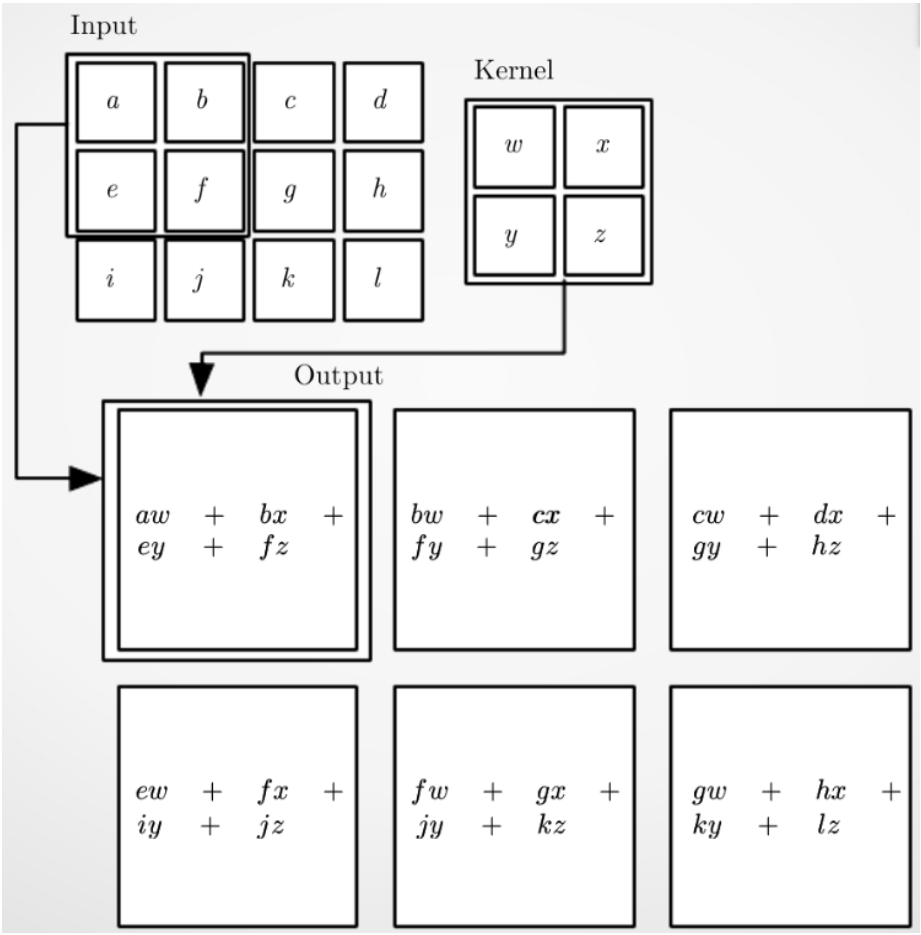
- Il singolo pixel in un'immagine non ha una grande valenza significativa
- Il flattening rimuove le informazioni riguardanti posizione e prossimita' dei pixel
- Troppi parametri -> il numero di features cresce in modo enorme, per cui il collegamento con il livello hidden richiede un numero gigante di pesi. Inoltre, troppi parametri portano spesso a overfitting

Vorremmo quindi avere un "*estrattore di features*" in modo da rendere il percettrone utile. Questo estrattore di features si puo' ottenere combinando due operazioni principali:

- **Convoluzione**
- **Pooling**

10.2.1 Convoluzione

La convoluzione utilizza una matrice in dimensioni piccole (tipicamente $3\times3, 2\times2$) di pesi chiamata *filtro* o *kernel*, che viene fatto scivolare sulla immmagine di input, dando in output una **feature map**. Ogni elemento della feature map e' una somma pesata degli input per i pesi, come nella figura seguente:



La convoluzione ha diversi vantaggi:

- I pesi sono sparsi: invece di connettere ogni neurone in input con ogni neurone in output, i pesi sono molti di meno -> ne connetto solo $3 \times 3 = 9$ ad uno
- I parametri vengono condivisi
- L'analisi e' basata sulla localita': ogni kernel va a definire una determinata caratteristica che si vuole andare a cercare (es. linea, occhio)

  Equivarianza: *la posizione della caratteristica dell'immagine NON influenza la capacita' di essere rilevata dal kernel (questo perche' facciamo sliding window essenzialmente)*

I filtri possono fare diverse cose:

- Identita' (non fare niente)
- Edge Detection
- Sharpening: serve a vedere se ci sono dei motivi molto ripetitivi
- Gaussian Blurring: sfoca l'immmagine

Nell'image processing classico, i pesi dei filtri convoluzionali venivano definiti dagli operatori

Gli iperrparammetri di ogni layer convoluzionale sono:

- Il numero di filtri convoluzionali: questo perche' posso applicare piu' filtri in un passaggio solo (di solito sono potenze di 2) -anche in questo caso abbiamo un tensore che ci rappresenta il filtro
- La grandezza del kernel (tipicamente non piu' grande di 4×4)
- Il padding
- Lo stride

La ***grandezza del kernel*** e' una quantita' che indica quanto locale e' l'analisi che si fa. Se k e' pari alla grandezza dell'immagine otteniamo un **percettrone**, per cui perdiamo i vantaggi discussi.

Il ***padding*** indica il numero di componenti che si aggiungono su tutti i lati (tipicamente con valori pari a 0) in modo da permettere ai kernel di fare l'analisi anche sulle parti piu' vicine ai bordi dell'immagine. Ci sono due valori molto ricorrenti di padding in letteratura:

- *Same* = si aggiunge un padding in modo che la feature map in output abbia le stesse dimensioni di quella in input
- *Full* = fa in modo che ogni pixel dell'immagine venga analizzato lo stesso numero di volte (quindi anche quelli nei corners)

Lo ***stride*** e' la dimensione del *salto* che viene fatto dal kernel ad ogni passo. Se lo stride e' uguale alla dimensione del filtro, allora ogni pixel viene analizzato una sola volta.

> *Attenzione*: *uno stride con valore uguale alla dimensione del kernel, fa perdere la proprieta' di invarianza rispetto a traslazioni (equivarianza)*

10.2.2 Pooling

L'operazione di pooling e' un'operazione molto banale ma di grande importanza in questo contesto. Per chiarire l'utilizzo dell'operazione di pooling, si pensi ad esempio di essere nel caso in cui un layer convoluzionale dia in output una feature map che ha pochi valori presenti in alcuni punti e tutti gli altri nulli.

Questo indica ad esempio che il layer ha trovato delle particolari caratteristiche (*es. una linea*) in diversi punti dell'immagine, mentre in tutti gli altri non l'ha trovata.

Se non si facesse pooling, ci si porterebbe dietro una serie di valori nulli che non danno nessuna informazione ai layer piu' avanti, facendo quindi una serie di calcoli inutili. Il pooling non fa altro che ridurre la dimensionalita' *selezionando* i valori che sono significativi.

> *Gli strati di pooling non necessitano di nessun tipo di peso*

Due tipologie di pooling:

- **Average Pooling**
- **Max Pooling**: si prende il massimo perche' effettivamente i valori piu' alti sono in qualche modo legati ad una *significativita'* nel contesto delle reti convoluzionali

Anche qui bisogna specificare una serie di iperparametri:

- Dimensione del kernel
- Padding
- Stride

In generale, una buona pratica e' evitare che si faccia pooling su zone che si sovrappongono tra di loro, impostando lo stride pari alla dimensione del kernel.

> *Il punto di partenza standard e'* 2×2*, con stride* $= 2$ *(non e' un risultato teorico, semplicemente funziona molto bene la maggior parte delle volte)*

I blocchi convoluzionali della rete sono quindi composti da 3 operazioni principali:

1. Convoluzioni
2. Pooling
3. Attivazione (ReLU)

Essenzialmente, ogni blocco convolutivo va ad estrarre informazioni sempre piu' di alto livello, fino a quando non si arriva verso l'ultimo blocco che dara' il proprio output (che viene essenzialmente *flattened*) in pasto ad un MLP per la classificazione. Ogni passaggio tra i blocchi va a ridurre sempre di piu' le dimensioni dei layers. Questo perche' si passa a dire "*in questa posizione c'e' una riga*" a "*questa immagine contiene un gatto*".

11 Recurrent Neural Networks

Le reti ricorrenti sono nate per modellare task che hanno a che fare con le *sequenze*. Alcuni esempi possono essere:

- Speech Recognition: converti una sequenza di frequenze in una sequenza di parole
- Sentiment Classification: converti una sequenza di parole nel sentimento che viene espresso (rabbia/gioia/ecc..)
- Multi-Omics Sequence Analysis: converti una sequenza di DNA/RNA nella sua corrispondente proteina

Altri task inerenti alla linguistica sono:

- Predirre se un determinato *verbo* e' plurale o singolare
- Predirre la prossima parola
- Determinare la correttezza grammaticale di una frase

Per spiegare meglio questo tipo di reti, considereremo un task specifico: *determinare quali parole di una frase in input sono un nome proprio.*

Esempio:

- Input: $\mathbf{x} =$ "***Harry Potter*** *and* ***Hermione Granger*** *invented a new spell"*
- Output: $\mathbf{y} = (1, 1, 0, 1, 1, 0, 0, 0, 0)$

In cui nel vettore di output ogni valore $\mathbf{y}^{(t)}$ indica che la parola t-esima e' o meno un nome proprio. (*la notazione e' differente rispetto a quella vista fino ad ora per altre reti*) Con $Tx^{(i)}$ e' la lunghezza della sequenza in input i-esima $\mathbf{x}^{(i)}$.

Si noti come in questi modelli la lunghezza $Tx^{(i)}$ puo' sempre variare a seconda dell'esempio i, diversamente da qualsiasi altra rete vista fin'ora dove la lunghezza (numero di features) era fissa

Le frasi sono codificate con una rappresentazione *one-hot*, cioe' vengono rappresentate come dei vettori che hanno tante componenti quante sono le parole del vocabolario, in cui sono le posizioni corrispondenti alle parole che sono presenti sono uguali a 1.

Per motivare le reti ricorrenti, proviamo a porci la domanda se sia possibile in primo luogo risolvere questo problema con una rete Feed-Forward normale. Effettivamente, se abbiamo una frase codificata in one-hot encoding, abbiamo stabilito un numero di features fisso, per cui potremmo utilizzare effettivamente una di queste reti. Questo approccio presenta pero' tre problemi principali:

1. Sia gli input che gli output possono avere lunghezze differenti in diversi esempi
2. Non riesce a tenere conto delle features apprese in diverse posizioni del testo
3. Non puo' tenere conto del *contesto* delle singole parole all'interno delle frasi

Le reti ricorrenti invece riescono a tenere conto degli elementi precedenti della sequenza ad ogni elemento che processano. Una rete ricorrente non e' nient'altro che l'applicazione ricorsiva della stessa rete. Intuitivamente, possiamo vedere la sua versione *unrolled* come l'applicazione di una serie di multi-layer perceptrons, che prendono in input anche lo stato risultante dai MLP precedenti.

Se $g^{(t)}(x^{(t-1)}, x^{(t-1)}, \ldots, x^{(1)})$ e' la funzione che calcola l'intera rete, possiamo rappresentare la stessa funzione con la stessa formula di ricorrenza $f(h^{(t-1)}, x^{(t)}, \theta)$.

Guardare una rete ricorrente in questi termini introduce diversi vantaggi:

1. Il modello appreso avra' sempre la stessa grandezza di input (una sola parola della sequenza nel caso del text processing) indipendentemente dalla lunghezza della sequenza.
2. Dal momento che la funzione di transizione f ha sempre gli stessi parametri per ogni timestep, e' possibile apprendere un singolo modello f per tutti i timesteps t al posto di imparare un modello g per ogni possibile timestep.

Se noi presentiamo un input alla rete, essa funziona nel modo seguente:

- Calcola l'attivazione della rete con $\mathbf{x}^{(1)}$, producendo l'output corrispondente $\hat{\mathbf{y}}^{(1)}$
- Calcola l'attivazione di un MLP con $\mathbf{x}^{(2)}$, ma tenendo conto dell'attivazione del layer *hidden* $h^{(1)}$ del passo precedente, producendo l'output corrispondente $\hat{\mathbf{y}}^{(2)}$

E cosi' via fino alla fine della sequenza.

Piu' formalmente, possiamo calcolare i vari componenti come:

$$h^{(t)} = tanh(\underbrace{W \cdot h^{(t-1)}}_{\text{H.L. precedente}} + \underbrace{U \cdot \mathbf{x}^{(t)} + \mathbf{b}}_{\text{H.L. attuale}})$$
$$\mathbf{o}^{(t)} = softmax(V \cdot h^{(t)} + \mathbf{c})$$

in cui:

- $h^{(0)} = \mathbf{0}$ oppure e' inizializzato con un vettore *casuale*
- W sono i pesi che connettono le unita' hidden con le unita' hidden del passo predente (*hidden-to-hidden*)
- U sono i pesi che connettono le unita' di input input alle unita' hidden (*input-to-hidden*)
- V sono i pesi che connettono le unita' di hidden alle unita' output (*hidden-to-output*)
- b e c sono i bias

I pesi sono sempre gli stessi, condivisi da ogni passo.

L'apprendimento di queste reti e' *supervisionato*, e si allenano mediante *backpropagation*. L'unica differenza e' che l'algoritmo e' detto *Backpropagation Though Time* (BPTT). L'idea e' sempre quella della backpropagation:

- Data una singola frase di input, calcola l'attivazione di tutte la rete (tutti i timestep che consentono di arrivare alla fine della frase)
- Calcola la discrepanza tra l'output restituito ad ogni timestep e l'output desiderato a quel determinato timestep
- Viene calcolato l'errore complessivo della rete come la somma delle discrepanze calcolate ad ogni timestep
- Si propaga il gradiente dell'errore all'indietro

Possiamo definire la funzione di *loss* sul singolo passo nel modo seguente (sia $\mathbf{y}$ il *true target*)

$$\mathscr{L}^{(t)} = y^{(t)} - h^{(t)}$$

possiamo quindi calcolare la loss totale come la somma di tutte le loss ai singoli timestep

$$\mathscr{L}_{tot} = \sum_{t=1}^{Tx} \mathscr{L}^{(t)}$$

quindi l'update dei pesi del gradient descent dovra' tenere conto della loss totale

$$\Delta w_{ij} = \sum_{t=1}^{Tx} \frac{\partial \mathscr{L}^{(t)}}{\partial w_{ij}}$$

Per cui ogni peso terra' conto della loss che e' stata prodotta anche ai passi precedenti (questo e' ragionevole siccome i pesi sono condivisi e vanno a influenzare TUTTI i passaggi).

La backpropagation dovra' essere fatta sia sul flusso computazionale del modello, che su quello del ***tempo****.*

Il problema e' che l'algoritmo di apprendimento BTT non e' un algoritmo stabile e presenta un problema principale: Piu' aumentano i passaggi, piu' c'e' la possibilita' di un *dissoluzione* o *esplosione* numerica del gradiente. Questo perche' piu' sono i passaggi, piu' ci sono moltiplicazioni di gradienti, per cui e' molto probabile che il gradiente svanisca e verso gli step iniziali i cambiamenti sarebbero praticamente nulli.

In questo senso, l'algoritmo cerca di fare degli update considerando l'errore generato da timestep molto indietro nel tempo. E' come se si cercasse di spiegare un comportamento successo attualmente guardando ad anni di distanza. In altri termini, e' difficile che l'errore commesso ai passi precedenti abbia un'influenza significativa sull'errore attuale.

Ci sono diversi modi per risolvere questi problemi:

1. Usare una versione dell'algoritmo di apprendimento chiamato *truncated BPTT*, in cui si vanno a considerare solo delle finestre temporali (in cui possono essere sovrapposte)
2. Fare gradient clipping (nel caso dell'esplosione)

Le LSTM nascono principalmente per risolvere questi problemi tecnici nelle reti ricorrenti

Siccome le reti ricorrenti possono avere diverse dimensioni in input e output, possiamo categorizzarle in base allo schema di mapping:

- **One to Many**: dato un singolo *input* ritorna una sequenza in *output* (es. image captioning)
- **Many to One**: data una sequenza in *input*, ritorna un singolo *output* (es. sentiment analysis)
- **Many to Many**: data una *sequenza* di input, ritorna una sequenza in *output* (es. machine translation)

11.0.1 Generazione di Sequenze

Le reti ricorrenti possono essere utilizzate anche per *generare* le sequenze. Solitamente vengono generate in maniera *autoregressiva* per cui questo tipo di modelli e' detto *autoregressive decoder*. Proprio per la loro natura autoregressiva, questo tipo di architettura deve avere un criterio di STOP, di solito codificato all'interno di un token particolare.

Sempre nell'ambito di generazione di sequenze possiamo parlare di un algoritmo particolare chiamato **Beam Search** che tiene conto solo delle parole scelte allo step precedente.

Diversi score possono essere utilizzati per valutare le sequenze generate:

- Accuracy
- MSE
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation)
- BLEU (Bilingual Evaluation Study)
- Perplexity

- Factual Score (deve essere definito in base al dominio di riferimento)

12 Long Short Term Memory Networks

Come visto precedentemente, le Recurrent Neural Networks soffrono di un problema tecnico: il vanishing/exploding gradient. Le LSTM sono nate proprio per cercare di risolvere questso problema, per cui sono classificate come Reti Ricorrenti.

L'idea che sta alla base di queste reti e' quella di avere una forma di memoria che permetta ad ogni istante di tenere conto della storia passata. L'informazione che viene mantenuta viene pero' regolamentata da dei *gates* che decidono quale informazione tenere e quale buttare via (funzionano come un filtro).

> *Le RNN* ***accumulano*** *tutta l'informazione precedente del layer hidden ad ogni passo, mentre le LSTM la* ***filtrano*** *senza accumularla indiscriminatamente*

Ogni cella LSTM riceve uno *stato* C_{t-1} in input che contiene l'informazione che si vuole propagare dallo stadio precedente. Si puo' pensare a questo stato come ad una flusso di informazioni che arrivano dal livello precedente. Passando per la cella, lo stato viene modificato man mano dai vari gates, che sono:

- Output Gate
- Forget Gate

Una volta combinate tutte le trasformazioni, lo stato risultante C_t viene propagato in output allo stadio successivo.

12.0.0.1 Forget Gate Il compito del forget gate e' quello di decidere quale informazione rimuovere dallo stato della cella. Prende in input:

- L'input x_t corrente (nel caso di NLP, la t-esima parola)
- L'output h_{t-1} della cella al passo precedente

E calcola l'output con la seguente relazione

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

In cui f_t e' un vettore delle stesse dimensioni dello *stato* C_{t-1} le cui componenti sono tra $[0, 1]$. Siccome questo vettore viene poi moltiplicato *element wise* con lo stato C_{t-1}, i valori rappresentano intuitivamente quanta informazione tenere/eliminare per ogni componente di C_{t-1}. Il vettore $[h_{t-1}, x_t]$ e' la *concatenazione* tra h_{t-1} e x_t.

> *Ovviamente, il vettore di pesi* W_f *e il bias* b_f *sono un parametro della rete.*

L'apprendimento di un forget gate fa imparare alla rete quanta informazione filtrare per ogni passo.

12.0.1 Input Gate

L'input gate decide invece quale informazione deve essere aggiunta allo stato della cella. Per fare cio' calcola:

- L'*input gate*: vettore che decide quali elementi o meno devono essere aggiunti allo stato (e in che misura)

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- Lo *stato candidato*: vettore che rappresenta il nuovo stato calcolato sulla base di quello precedente e l'input attuale

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

  Anche in questo caso, questi due vettori vengono combinati con un prodotto *element-wise* (*Prodotto Hadamard*)

$$R = i_t \odot \tilde{C}_t$$

  In questo senso, i_t dice quale informazione aggiungere allo stato e in che misura. Il risultato R viene poi aggiunto con una *somma* allo stato della cella.

12.0.2 Output Gate

L'output gate definisce l'output dell'intera cella LSTM h_t, ottenuto combinando lo stato finale C_t. Funziona in maniera molto simile all'input gate, per cui calcola:

- L'*output gate*: vettore che decide quali elementi o meno devono essere mantenuti nell'output

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- L'*output* della cella

$$h_t = o_t \odot tanh(C_t)$$

12.1 Architettura Encoder-Decoder

Le singole celle LSTM se prese da sole sono semplicemente un componente di base per delle architetture piu' complesse. In questa architettura, l'encoder si occupa di processare l'input e codificarlo in una rappresentazione piu' succinta detta *summary vector*. Il decoder fa l'inverso, cioe' prende in input il summary vector e da in output l'input originale (in questo caso una sequenza). Ad esempio, nel task della machine translation, il decoder si occupa di generare la traduzione a partire dal *summary vector.*

Nell'encoder, tutti gli output delle singole celle (considerando la versione unwrapped) sono scartati, per cui si costruisce il summary vector utilizzando lo stato C_t dell'ultimo passo.

Come detto, il decoder non fa altro che fare l'inverso, cioe' ottenere parole dalla rappresentazione sommaria. L'output pero' sono dei vettori, per cui per ottenere la parola a partire dal vettore di output si applicano le seguenti trasformazioni:

- Applica una trasformazione lineare in modo da espandere le dimensioni del vettore a quelle del vocabolario che si vuole utilizzare in output. Questo vettore in output e' chiamato logits
- Applica una *softmax* ad ogni elemento di logits , ottenendo un vettore di probabilita' (ogni parola ha associata una probabilita').

  Il realta' si utilizza una log softmax per rendere il calcolo numericamente piu' stabile

Infine, per ottenere la parola basta fare $argmax\ log\_probs$.

12.1.1 Training (Teacher Forcing)

Per fare il training di questo tipo di architettura, si utilizza per prima cosa una loss *Cross-Entropy*

$$L = -\sum_{i=0}^{n} y_i \cdot log(\hat{y}_i)$$

Dove $\hat{y}_i$ e' il risultato della rete al passo i

Durante la fase di training, al decoder viene data la sequenza di target y, ma *shiftata* di una parola siccome la prima parola e' un token costante <START>. In questo modo si puo' calcolare la loss per ogni passo. In altri termini, si da ad ogni timestep in input al decoder la parola target al passo precedente.

12.1.2 Inferenza (Autoregressive Generation)

Nell'inferenza si fa una cosa simile al training del punto di vista del decoder. Essenzialmente ogni output diventa l'input del prossimo timestep. Per cui si lascia che il decoder generi l'intera frase da solo. Questo evidentemente puo' provocare problemi a cascata, per cui anche solo un errore potrebbe invalidare l'intera sequenza risultato.

Il problema di questa architettura sta nell'encoder: Se la frase e' molto grande, il summary vector (che ha grandezza finita) deve codificare semanticamente molta informazione. Per questo motivo, la rete tende a perdere informazioni sulle parti iniziali delle frasi al crescere della lunghezza della frase.

Idealmente, si vorrebbe modificare l'architettura in modo da tenere conto di **tutti** gli hidden state una volta arrivati alla fine dello stage di encoding, e che ci sia un meccanismo nella fase di decoding che sfrutti queste informazioni in modo intelligente. Questo meccanismo e' l'*attention*.

L'idea dell'attenzione puo' essere spiegata tramite un esempio. Ipotizziamo di voler tradurre la seguente frase

"*Are you free tomorrow?*" -> "*Sei libero domani?*"

Il modello avrebbe molta piu' facilita' se esistesse un meccanismo per concentrarsi sugli input in modo da generare le traduzioni corrispondenti. Esempio: sto considerando *"libero"*, allora l'attenzione deve essere posta su *"free"*.

12.1.3 Attention

L'attenzione viene implementata solo sul *decoder* perche' e' il *decoder* che riceve dall'*encoder* tutti gli hidden state delle parole in input su cui applicare il meccanismo dell'attenzione. Come primo step dell'attenzione, il decoder confronta l'hidden state al timestep attuale con tutti gli hidden state risultanti dall'encoder, generando un vettore di *score*s. Come secondo passo, applica una softmax a questo vettore e moltiplicalo per ogni vettore corrispondente agli hidden state dell'encoder. Infine, somma tutti i vettori risultanti, ottenendo un singolo vettore detto *context vector* (che e' l'output dell'attenzione).

> La softmax (detta anche softargmax) e' una funzione che normalizza un vettore di valori ad un vettore di probabilita' es. $A = (13, 9, 9)$
>
> $$argmax(A) = (1, 0, 0)$$
> $$softmax(A) = (0.96, 0.02, 0.02)$$

Riassumendo:

1. Accumula tutti gli hidden states dell'encoder $h = (h_1, \dots, h_{Tx})$
2. La generica componente j del vettore di score al generico passo i del decoder e' calcolabile come (d_i e' l'hidden state del decoder al passo i. U_a, W_a sono dei parametri apprendibili dalla rete)

$$e_{ij} = (W_a \cdot d_i) \cdot (U_a \cdot h_j)$$

3. Applica la softmax al vettore e. Siccome il risultato e' una distribuzione di probabilita', ogni componente ci dira' in quale hidden state si deve porre di piu' l'attenzione

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{Tx} exp(e_{ik})}$$

4. Moltiplica ogni hidden state dell'encoder per ogni elemento dello score corrispondente e somma i risultati per ottenere un unico vettore

$$C_i = \sum_{j=1}^{Tx} a_{ij} h_j$$

C_i e' il generico elemento i del vettore contesto C.

In questo modo prendiamo solo la informazione che ci interessa sfruttando il contesto (che e' solo un vettore).

L'output dell'attenzione viene calcolato ad ogni step e viene concatenato con l'hidden state (risultato) della rete. Quello che si fa e' poi utilizzare il meccanismo descritto in precedenza (word embedding) per trovare la parola piu' probabile.

12.1.4 Bidirectional Encoder-Decoder

Un'architettura tipica di questo tipo e' detta bidirectional ed e' costituita nel modo seguente:

- Il primo layer e' un layer di *word embedding*, in cui vengono trasformate le parole in vettori numerici
- Il secondo layer e' un *encoder* composto da reti neurali ricorrenti, in cui le sequenze possono essere codificate non solo da sinistra a destra ma anche nella direzione inversa. Questo permette di ottenere rappresentazioni aggiuntive che potrebbero essere utili alla rete.
- Il terzo layer tra encoder e decoder e' il layer dell'attenzione in cui confluiscono i vettori contesto
- Il quarto layer e' il decoder che sfrutta l'attenzione per dare in output i vettori
- L'ultimo layer e' un layer di embedding che serve a tradurre i vettori numerici in parole

12.1.5 Copy Mechanism

Cosa deve succedere se vogliamo una parola che non e' nell'output vocabulary ma che compare nella input sentence? Come soluzione, si vorrebbe che questa parola di input venga "copiata".

L'idea e' quella di prendere la distribuzione dell'attenzione (che per definizione e' costruita a partire dall'input) e ogni qual volta che si vuole copiare una parola non presente nel vocabolario, si fa semplicemente sampling sulla distribuzione dell'attenzione (al posto di fare il sampling dalla distribuzione data in output).

Per scegliere quando fare sampling tra una o l'altra delle due distribuzioni, si introduce un *soft switch* p_{gen}. p_{gen} e' una parte della rete che viene anch'essa appresa e va in qualche modo a decidere se generare la prossima parola dalla distribuzione dell'attenzione oppure dalla distribuzione in output.

$$p_{gen} = \sigma(w_p \, p_{gen,t-1} + W_s S_{t-1} + W_c C_t + W_o O_t)$$

L'idea e' quella di generare una distribuzione finale che tenga conto di entrambe le due distribuzioni: sia quella dell'*attenzione* che quella di output. Per ottenerla, si fa la somma delle due distribuzioni, ognuna scalata per i valori di p_{gen}. Intuiviamente, se p_{gen} tende a 1, allora sara' piu' predominante la distribuzione dell'*output*, mentre se tende a 0 sara' piu' predominante quella dell'*attenzione*. In altri termini p_{gen} regola quali tra le due devono favorite nella scelta.

$$p_{final}(c) = p_{out} \, p_{gen} + (1 - p_{gen}) \sum_{i:c_i=c} a_i$$

Ovviamente i parametri del soft switch sono apprendibili. Una volta appresa la rete, si andra' a fare sampling direttamente da p_{final} per ottenere le parole in output.

13 Transformers

L'idea principale dell'architettura transformer e' quella di generare un'intero modello sul meccanismo dell'*attenzione*. Nasce quindi come un'evoluzione dei modelli Encoder-Decoder basati su modelli ricorrenti, solamente che rimuove la ricorrenza a favore di moltiplicazioni di matrici per calcolare l'attenzione e ottenere cosi' ottimi risultati. Per come e' stato concepito, il transformer e' in parte parallelizzabile rispetto alle RNN che necessitano di processare a timestep.

Come detto, il transformer e' sempre un'architettura Encoder-Decoder, per cui l'input e l'output sono sequenze di token di parole. Sia gli encoder che i decoder sono pensati come delle pile di 6 sotto-encoder/sotto-decoders. Questi sottopassaggi possono essere visti come diversi layer, in cui il singolo layer prende in input l'ouput di quello precedente e cosi' via. All'interno di ognuno di questi layer degli encoder, sono presenti due layer:

1. Self Attention
2. Feed Forward

Mentre all'interno di quelli del decoder:

1. Self-Attention
2. Encode-Decoder Attention
3. Feed-Forward

13.1 Encoder Layer

Vediamo di preciso cosa succede all'interno del layer dell'encoder. Prima di tutto, in input si ricevono le parole per cui si fa un'operazione di word embedding, ottenendo cosi' i vettori che rappresentano le parole.

> *A differenza delle reti ricorrenti, nei Transformers esiste un limite al numero di parole possibili*

Ognuno di questi vettori passa all'interno del layer di self attention, per cui il risultato di questa operazione saranno delle *rappresentazioni alternative* di ogni parola. Infine, queste rappresentazioni alternative vengono passate ad un layer feedforward per ottenere l'output dell'encoder layer.

Si noti come il layer feedforward prenda in input tutte le singole rappresentazioni alternativa, ma i parametri del layer sono sempre gli stessi, per cui e' possibile parallelizzare questa operazione su tutte le rappresentazioni alternative.

In realta', tra l'output della *self attention* e l'*input* ci sono delle **connessioni residuali**, cioe' delle connessioni che portano parte dell'input non processato all'output della self attention. Queste connessioni vanno a finire come inpout di un *layer normalization*, che applica una normalizzazione alla somma tra la matrice che contiene gli embeddings delle parole (ottenuta mediante le connessioni residuali) e la matrice ottenuta dagli outputs della self attention.

$$LayerNorm(X + Z)$$

Dove X e' la matrice degli embedding e Z e' la matrice di output della self attention.

La layer normalization non fa altro che *rimuovere* la media da ogni elemento e a normalizzare rispetto alla *varianza*. In altri termini:

$$y = \frac{x - \mathbb{E}(x)}{\sqrt{Var(x)}}$$

Lo scopo di questo layer e' quello di evitare che i gradienti esplodano o svaniscano durante la *backpropagation*.

13.1.1 Self Attention

Il layer di self attention e' il piu' importante di questa architettura. L'idea e' la seguente: dato un *input* (una frase) si vorrebbe che il modello ponga l'attenzione a specifiche *parti* (parole) dell'input. Per ogni word embedding in input, il self attention va a creare 3 vettori corrispondenti:

- Query
- Key
- Value

Questi vettori vengono creati semplicemente moltiplicando il vettore dell'embedding della parola per delle matrici W^Q, W^K, W^V (Q per ottenere la *query*, K per ottenere la *key*, V per ottenere il *value*). Tipicamente, questi vettori risultanti sono piu' piccoli delle word embedding in input.

$$Q = W^Q \cdot x \tag{7}$$
$$K = W^K \cdot x \tag{8}$$
$$V = W^V \cdot x \tag{9}$$
$$\tag{10}$$

Questi vettori sono utilizzati per capire dove porre l'attenzione. Una volta calcolati Q, K, V per ogni word embedding in input:

- Prendi la Q e moltiplicala (*DotProduct*) per tutte le K calcolate (sia la propria che quelle delle altre parole) ottenendo un vettore di score
- Dividi ogni elemento dello *score* per la radice della dimensione del vettore chiave $\sqrt{d_k}$ (e' semplicemente una normalizzazione)
- Applica una softmax al vettore *score* normalizzato, in modo da darci una distribuzione di probabilita' sull'input. Il valore massimo ci dira' su quale parola porre di piu' l'attenzione
- A questo punto (come per le reti ricorrenti), calcoliamo il prodotto element-wise tra i valori della softmax dello score per i vettori value di ogni parola corrispondente ($score_i \cdot V_i$)
- Infine, questi valori vengono sommati tra di loro in modo da ottenere un singolo vettore in output per ogni parola.

  Si chiama self-attention perche' l'output pone l'attenzione sull'input stesso

13.1.1.1 Versione Matriciale

Siccome gli acceleratori grafici sono molto veloci a fare moltiplicazioni di matrici, la self attention sarebbe migliorabile se si riformulasse in termini di operazioni di matrici. Per primo lugoo, le word embeddings possono essere messe tutte in una stessa matrice $X(w \times v)$ in cui w e' il numero di parole della frase e v e' il numero di parole del vocabolario. In questo modo, se moltiplichiamo

- $X \cdot W^Q = Q$
- $X \cdot W^K = K$
- $X \cdot W^V = V$

Otteniamo le matrici Q, K, V le cui singole righe sono i corrispondenti vettori Query, Key, Values per ogni word embedding in input. Una volta che abbiamo ottenuto queste matrici e' possibile calcolare l'output della self-attention Z in formato matriciale

$$Z = softmax(\frac{Q \cdot K^T}{\sqrt{d_k}}) \cdot V$$

in cui:

- $Q \cdot K^T$ mette in relazione tutte le parole con tutte le chiavi (e' la matrice di *score*, in cui la singola entrata i, j indica l'attenzione che da' la parola i alla parola j)

13.1.2 Multi-Head Attention

Una delle altre cose introdotte dal modello transformer e' la *multi head attention*, che non e' altro che applicare il meccanismo di self-attention piu' volte. L'idea e' che applicare questo meccanismo piu' volte renda potenzialmente il modello piu' potente, cosi' da far focalizzare ogni attention-head su particolari domini. Ad esempio, possiamo pensare che un determinato attention-head si focalizzi particolarmente sui nomi, uno a cosa si riferiscono gli articoli, uno sul sesso, ecc.. In questo senso espande l'abilita' del modello di concentrarsi su posizioni differenti.

Siccome ogni attention head produce un output della self attention, per ogni parola ci saranno tanti ouput quanti gli attention head. Il problema e' che il layer dopo la self attention (feed-forward) si aspetta in input un input di dimensione fissa. Per risolvere questo problema, si concatenano tra loro tutti i vettori risultato da ogni attention-head e si moltiplicano per una matrice W^O.

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h) \cdot W^O$$

dove:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

e h e' il numero di *attention-head*.

> *La trasformazionee lineare W^O riporta la concatenazione di questi vettori alla dimensione adatta in input del layer feed-forward.*

13.1.3 Positional Encoding

Da come vengono processati gli input (ogni embedding e' una riga di X), e' evidente come l'ordine di questi embedding (delle righe) non cambi il risultato della self-attention. L'ordine nel testo e' pero' importante, per cui e' necessario trovare un modo per tenere conto anche delle posizioni in cui occorrono gli embedding (tokens). Il *positional embedding* e' un modo per codificare le parole aggiungendo l'informazione della posizione all'interno della frase. Per ottenere questi vettori, si prendono gli embeddings delle parole x_i e si sommano *element-wise* per dei vettori t_i detti *positional encodings*.

> *Ogni t_i e' creato esclusivamente per tenere conto dell'informazione posizionale dell'embedding*

I vettori di positional encoding vengono calcolati per posizioni pari e dispari mediante due funzioni periodiche

$$t_{(pos,2i)} = sin(pos/1000^{2i/d_{model}}) \tag{11}$$

$$t_{(pos,2i+1)} = cos(pos/1000^{2i/d_{model}}) \tag{12}$$

$$\tag{13}$$

dove *pos* e' la posizione e i e' la dimensione.

> *In realta', la matrice di tutti i positional encodings viene appresa direttamente come parametro della rete.*

13.1.4 Encoder-Decoder Layer

Il layer Encoder-Decoder attention funziona proprio come la mulit-headed attention con l'eccezione che crea le proprie matrici Q_i dal layer del decoder precendete e prende le matrici K_i e V_i dall'output dell'ultimo encoder. In un certo modo, simula cio'e che succede nei modelli ricorrenti basati sull'attenzione.

(Per questo motivo *visualmente* l'ultimo encoder ha connessioni con tutti i decoders)

Nel layer di self attention nel decoder, inoltre, e' permesso porre l' attenzione solo su posizioni precedenti nella sequenza di output. Per cui si pongono le posizioni future ad un valore pari a $-\infty$. Questa operazione e' detta *masking*.

13.2 GPT (Generative Pre-Training)

GPT e' un modello di OpenAI basato su Transformer. GPT si pone di risolvere al problema che la quantita' di dati etichettati per un task specifico (Machine Translation, Summarization, Q&A) siano di fatto molto scarse. La soluzione proposta con GPT si basa sull'utilizzo di testo non strutturato per far apprendere alla rete una rappresentazione universale del testo per poi (transfer learning) riadattare il modello a risolvere determinati task, utilizzando la minore quantita' di testo etichettato.

La prima idea e' chiamata *unsupervised pre-training*: Dato del testo non etichettato, predici la prossima parola.

Sia $U = \{u_1, \ldots, u_n\}$ il testo non etichettato, l'idea e' apprendere un modello che massimizzi la seguente likelihood:

$$L_1(U) = \sum_i log\ P(u_i \mid u_{i-k}, \ldots, u_{i-1}; \Theta)$$

Cioe' si vuole massimizzare la probabilita' di ottenere u_i dato il contesto (tutte le parole che c'erano prima in una finestra di lunghezza k). P e' modellata da una rete neurale con parametri Θ.

La seconda idea e' invece chiamata *supervised-finetuning*: Dato il modello pretrainato in modo non supervisionato, fanne il training con dati etichettati in modo supervisionato per adattarlo al task specifico. Si vuole essenzialmente massimizzare la funzione:

$$L_2(C) = \sum_{(x,y)} log\ P(y \mid x^1, \ldots, x^m)$$

dove $P(y \mid x^1, \ldots, x^m)$ e' approssimato con $softmax(h_l^m W_y)$, in cui h_l^m e' l'output dell'ultimo layer del transformer appreso in precedenza e W_y e' una matrice di pesi appresa per predirre y.

14 Appendice: Reverse Mode Automatic Differentiation

Nel corso abbiamo visto come differenziare l'output di una rete neurale rispetto al suo errore in funzione dei suoi pesi. Ovviamente, fare questa operazione a mano (derivare la regola di update dei pesi) e' infattibile a livello pratico per architetture piu' complesse. I moderni framework di deep learning sono in grado di calcolare in modo autonomo la derivata senza alcun ausilio esterno. Per poter fare cio', si appoggiano ad un metodo numerico/matematico chiamato *differenziazione automatica*. In realta', la backpropagation non e' nient'altro che un caso specifico di questo metodo matematico: la differenziazione automatica con accumulo dei gradienti all'inverso. Consideriamo il grafo computazionale della equazione $f = (a + b) \cdot b$.

$$\begin{aligned} y &= v_2 \\ v_2 &= v_1 \cdot v_0 \\ v_1 &= v_{-1} + v_0 \\ v_0 &= b \\ v_{-1} &= a \end{aligned}$$

Vogliamo calcolare la derivata di ogni nodo intermedio rispetto al valore della funzione f, chiamato *aggiunto*. Utilizzando semplicemente la regola di derivazione a catena

$$\frac{\partial f}{\partial v_i} = \sum_j \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial v_i}$$

per tutti i nodi intermedi i. A questo punto uno potrebbe chiedersi il perche' di una sommatoria. Di fatto, quando applichiamo la regola di derivazione a catena, dobbiamo scegliere un *"percorso"* all'interno del grafo, scegliendo tutte le quantita' intermedie rispetto a cui derivare per arrivare fino in fondo. Facciamo un esempio e ipotizziamo di voler calcolare $\bar{b} = \bar{v_0}$. Notiamo subito che abbiamo due modi per arrivare fino in fondo al grafo:

$$\frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial v_0} \text{ oppure } \frac{\partial f}{\partial v_2}\frac{\partial v_2}{v_0}$$

Scegliendo solo un percorso, non teniamo in conto di fatto il contributo che puo' dare al risultato della funzione *"passando"* per l'altro percorso, per cui risolviamo il problema sommandoli, ottenendo

$$\begin{aligned}
\frac{\partial f}{\partial v_0} &= \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial v_0} + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{v_0} \\
&= \bar{v}_2\frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial v_0} + \bar{v}_2\frac{\partial v_2}{v_0} \\
&= 1 \cdot v_0 \cdot 1 + 1 \cdot v_1 \\
&= v_0 + v_1
\end{aligned}$$

Ora che abbiamo giustificato intuitivamente la formula, andiamo avanti e utilizziamola a questo punto per calcolare tutti gli *aggiunti* (*adojoints*). Sta volta pero' partiamo dall'alto verso il basso, in modo che in caso comparissero delle definizioni di aggiunti calcolati precedentemente possiamo eventualmente sostituirli.

$$\begin{aligned}
\bar{f} &= 1 \\
\bar{v}_2 &= \bar{f} \cdot \frac{\partial f}{\partial v_2} = 1 \\
\bar{v}_1 &= \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = v_0 \\
\bar{v}_0 &= \bar{v}_1 \cdot \frac{\partial v_1}{v_0} + \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_0} = v_0 + v_1 \\
\bar{v}_{-1} &= \bar{v}_1 \cdot \frac{\partial v_1}{\partial v_{-1}} = v_0
\end{aligned}$$

Osserviamo che per calcolare un singolo $\bar{v}_i$ gli ingredienti principali sono:

- Gli aggiunti dei valori v_j intermedi in cui compare almeno una volta v_i
- I parziali rispetto agli input del passo successivo (essenzialmente, quanto cambia la prossima funzione in cui compare v_i rispetto a v_i)

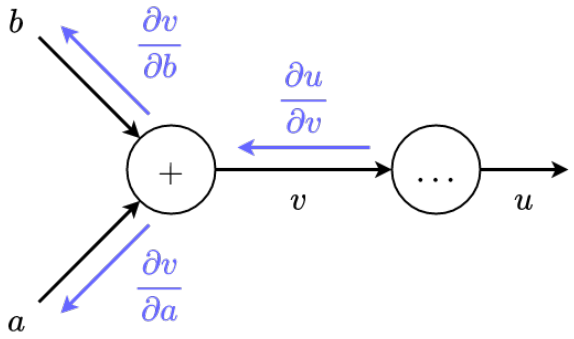
E' evidente che ad ogni passo, queste quantita' le posso calcolare solamente al passo successivo (per questo sostituiamo le definizioni). Per questa ragione partiamo dall'output e scendiamo verso gli input, propagando ai propri i vari parziali/aggiunti e moltiplicandoli/aggiungendoli man mano fino ad arrivare ai nodi di input. Il singolo nodo quindi ricevera' gli ingredienti necessari per calcolare il proprio aggiunto, lo calcola e si salva il valore, per poi propagare all'indietro ad ogni input il suo parziale corrispondente.

Ricapitolando, se io ho un nodo, ad esempio $v = a + b$, che a sua volta sara' l'input di un nodo intermedio quasiasi $(\dots)$ (con i puntini intendo che puo'

esserci qualsiasi operazione tra $+, -, /, \cdot$ ecc) che produce il risultato u, per calcolare il suo aggiunto $\bar{v}$ necessito:

- L'aggiunto $\bar{u}$
- La derivata $\frac{\partial u}{\partial v}$

Notiamo come l'unico a poter sapere la derivata parziale $\frac{\partial u}{\partial v}$ e' u, per cui essenzialmente lo propaga all'indietro (della serie: *il mio input e' il tuo output*). La situazione e' rappresentata graficamente nella figura seguente.



Si noti inoltre che le derivate rispetto ai propri input sono delle espressioni che sono definite in termini degli input, per cui ogni nodo dovra' salvarsi o i valori di input che riceve, oppure direttamente le derivate rispetto agli input. Questi sono dettagli dal punto di vista implementativo ma vale la pena menzionarli. Una prima opzione e' appunto salvare gli input nei singoli nodi, per poi utilizzare questi valori per calcolare i parziali da propagare indietro, mentre un'alternativa e' appunto quella di calcolare direttamente i parziali rispetto agli input nel passo di forward in avanti (senza propagarle in avanti) e salvandole opportunamente per poterle propagare all'indietro nel passo di backward.

Questo per il parziale, ma invece per gli aggiunti? Anche qui possiamo fare due cose. La prima consiste nel calcolare l'aggiunto ogni volta che siamo in un nodo per poi propagarlo all'indietro cosi' come facciamo per il parziale (anche moltiplicandolo con esso volendo), oppure creiamo un *tape*, cioe' proprio una lista condivisa che mantiene i valori degli aggiunti mano a mano che scendiamo verso i nodi (variabili) di inputs.

Di seguito sono riportate alcune signatures possibili corrispondenti ai metodi descritti:

- v.backward(adjoint, dudv) – Soluzione con accumulo degli aggiunti in ogni nodo
- v.backward(dudv) – Soluzione con accumulo in un *tape*