8. Reti Neurali e Apprendimento Profondo

Dispensa ICon

versione: 30/11/2024, 12:42

Introduzione · Reti Neurali Artificiali Feed Forward · Estensioni (Cenni)

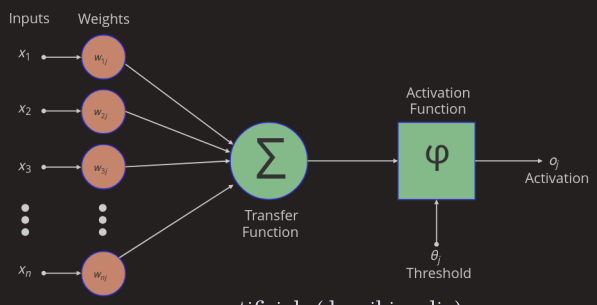
1 Introduzione

Adottare una rappresentazione degli esempi in termini di input grezzo spesso non risulta fruttuoso: servono feature più utili. A tale scopo si può ricorrere a una *progettazione* delle feature utili, generate a partire dall'input, mediante la cosiddetta **feature engineering**.

Attualmente ci si basa sull'apprendimento delle feature, specie allorché si disponga di ingenti quantitativi di dati. I metodi per tali scopi apprendono feature utili per il problema da risolvere anche se spesso queste si rivelano poco interpretabili. Un altro vantaggio dall'apprendimento di feature è che possono anche essere reimpiegate nella soluzione di problemi affini.

Imparare le feature è un obiettivo perseguito attraverso il cosiddetto *feature* **representation learning** ed è in genere svolto attraverso **reti neurali artificiali** (ANN) che sono ispirate alla struttura dei *neuroni* del cervello umano. Le **unità** artificiali risultano diverse dai corrispettivi biologici in quanto molto più semplici (cfr. figura) e meno numerose di quelle nel cervelli umani (numero dell'ordine di 10^{11}).

Tali modelli, attualmente molto popolari, sono impiegati per risolvere problemi che comportano un ragionamento di *basso livello*, con molti dati disponibili, come ad esempio l'*interpretazione delle immagini*, il *riconoscimento del parlato* e la *traduzione automatica*, risultando molto *flessibili* e capaci di *inventare* nuove feature.



neurone artificiale (da wikimedia)

Alcuni motivi d'interesse per le reti neurali artificiali sono i seguenti:

- nelle *neuroscienze*, per comprendere i sistemi naturali, ad esempio per la simulazione di sistemi propri di animali semplici (comportamento);
- per la comprensione dei *meccanismi* funzionali e astratti *dell'intelligenza*: ipotizzando di poter replicare i meccanismi del cervello per realizzarne le funzionalità, testandoli confrontando forme d'intelligenza con e senza tali meccanismi; l'ipotesi alternativa parte da osservazioni come quella degli aerei che, pur sfruttando gli stessi principi del volo degli uccelli, non si basano sullo stesso meccanismo;
- per progettare nuovi *modelli computazionali*: mentre i modelli convenzionali sono costituiti da pochi processori e molta memoria (per lo più inerte), nel cervello si attivano moltissimi processi asincroni distribuiti, senza un master; difatti le ANN vengono spesso implementate su architetture massivamente *parallele*; si consideri anche il nuovo paradigma di programmazione detto *differentiable programming*;
- per puntare a modelli di diversa complessità ovvero *learning bias* rispetto ad altri: le ANN possono rappresentare qualsiasi funzione su feature discrete, come gli alberi di decisione, ma anche molte altre funzioni non sempre realizzabili attraverso alberi.

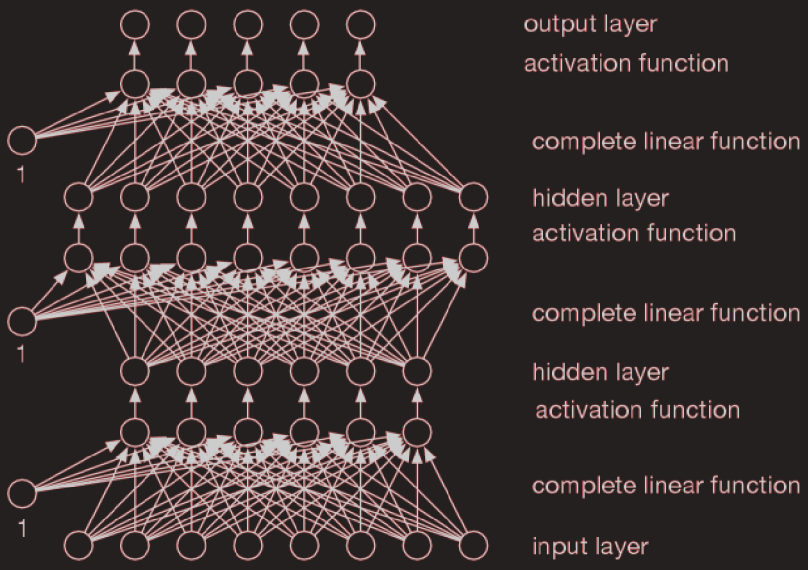
2 Reti Neurali Artificiali Feed Forward

Essenzialmente, una *rete neurale* **feed-forward** implementa una funzione di predizione come segue: dato l'input (vettoriale) x

$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- ogni **layer** / *strato* è costituito da una funzione f_i che mappa un vettore (array o lista) di input su un vettore di output;
- ciascuna componente del vettore di output di un layer costituisce un'*unità*;
- la *profondità* della rete è n, con l'ultimo strato, f_n, detto di *output* della rete mentre gli altri altri layer sono detti *nascosti* / *hidden*, tranne quello di *input*, con x.

In genere le ANN hanno più feature di input ma anche più feature-target, tipicamente a valori *reali*; feature discrete possono essere trasformate in variabili-indicatrici o feature ordinali.



Ogni layer di unità opera in funzione dei precedenti: il layer di *input* ha un'unità per feature di input e viene avvalorato da ciascun esempio; un layer *nascosto* prevede la composizione di una funzione *lineare* f_i con una funzione ϕ *di attivazione* non-lineare, da cui si calcola l'output $out[j]$ in funzione dei valori in input $in[k]$ allo strato

$$out[j] = \phi\left(\sum_k in[k] \cdot w[k,j]\right)$$

si noti l'input aggiuntivo costante $i[0] = 1$ per il *bias* (o *termine noto*); occorrerà apprendere ciascun peso $w[k,j]$, uno per ogni (arco) coppia input-output del layer. Si noti che layer lineari *consecutivi* sarebbero inutili perché trasformabili in un unico layer lineare.

Le funzioni di attivazione ϕ, tipicamente, sono (quasi ovunque) differenziabili.

Ad esempio, si può usare un *rettificatore* o *Rectified Linear Unit* (ReLU) definito

$$\phi(x) = \max(0, x)$$

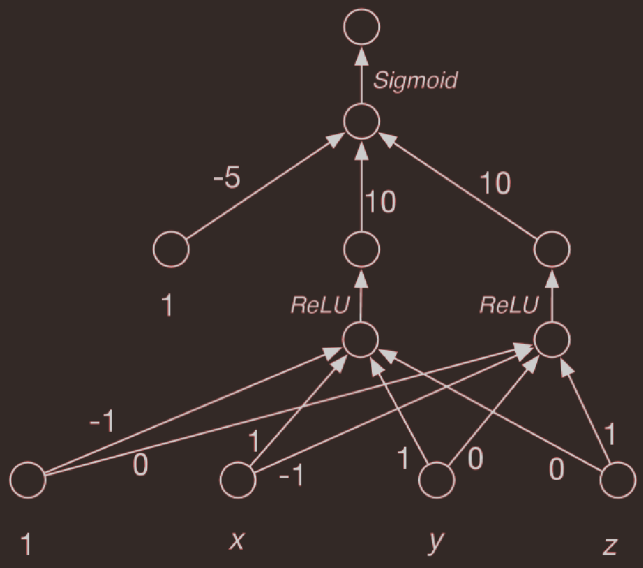
assumendo che la derivata sia nulla in $x = 0$.

La scelta di ϕ per il layer di output può variare in base al tipo di y e alla loss da ottimizzare:

- per y reale e loss quadratica media si può usare $\phi(x) = x$, funzione *identità*;
- per y booleana e log loss binaria, tipicamente si usa la *sigmoide* $\phi(x) = 1/(1+e^{-x})$, giustificabile anche in termini probabilistici;
- per y categorica e log loss categorica si usa invece una *softmax*, considerando un *output layer* con una unità per ciascun valore di y.

Queste scelte comportano derivate semplici dell'errore, proporzionali a $\hat{Y}(e) - Y(e)$.

Esempio — ANNs per la funzione `if-then-else`



- x, y, z costituiscono l'input;
- la funzione è equivalente a $(x \wedge y) \vee (\neg x \wedge z)$, rappresentabile in termini di ReLU o sigmoidi;
- dato un primo layer con due output: $h_1 = \text{relu}(x + y - 1)$ e $h_2 = \text{relu}(-x + z)$
  si avrà che $h_1 = 1$ se $(x \wedge y)$ vera e $h_2 = 1$ se $(\neg x \wedge z)$ vera,
  altrimenti sono entrambe nulle;
- per l'output, $\text{sigmoid}(-5 + 10 \cdot h_1 + 10 \cdot h_2)$ approssima $h_1 \vee h_2$ con un errore vicino allo 0.7%.

L'*ampiezza* (o *width*) di un layer è pari alle dimensioni del vettore di output. L'ampiezza della rete è pari all'ampiezza massima fra i suoi layer. La sua *profondità* è pari al numero di layer (scelta di progetto). Le dim. dell'input dipendono dal problema.

Si può dimostrare che una rete con un solo layer nascosto, purché sufficientemente ampio, può approssimare qualsiasi funzione continua su un intervallo di n. reali o discreta con input / output discreti. I layer a livelli bassi costruiscono feature utili a quelli dei livelli più alti, cosa che li rende più semplici.

2.1 Apprendimento dei Parametri

L'algoritmo **Back-propagation** (BP) implementa la DG stocastica (SGD) per determinare i pesi del modello: per ogni esempio e del batch, si aggiorna ogni peso w in ragione di $-\frac{\partial}{\partial w}\text{error}(e)$ usando le proprietà[1] delle derivate, ossia le regole di decomposizione lineare e di concatenazione (per le funzioni composte).

La predizione per e, con feature di input x_e, sarà

$$f(e) = f_n(f_{n-1}(\cdots f_2(f_1(x_e))))$$

con le f_i parametrizzate dai pesi. Supponendo che $v_i = f_i(f_{i-1}(\cdots f_2(f_1(x_e))))$, ossia $v_i = f_i(v_{i-1})$, posto $v_0 = x_e$, i valori v_i per $1 \leq i \leq n$ possono essere calcolati in una passata attraverso le funzioni annidate.

Definire le derivate per gli aggiornamenti è più difficile perché è più difficile calcolare la variazione dell'errore determinata da pesi di livelli inferiori a quello di output.

Sia w uno dei pesi nella definizione di una certa f_j (si noti che f_i non dipende da w, per $i \neq j$). La derivata dell'errore rispetto al peso è data da:

$$\begin{aligned}\frac{\partial}{\partial w}\text{error}(f(e)) &= \text{error}'(v_n)\cdot\frac{\partial}{\partial w}f_n(v_{n-1})\\ &= \text{error}'(v_n)\cdot\frac{\partial}{\partial w}f_n(f_{n-1}(v_{n-2}))\\ &= \text{error}'(v_n)\cdot f_n'(v_{n-1})\cdot\frac{\partial}{\partial w}(f_{n-1}(v_{n-2}))\\ &= \text{error}'(v_n)\cdot f_n'(v_{n-1})\cdot f_{n-1}'(v_{n-2})\cdot\frac{\partial}{\partial w}(f_{n-2}(v_{n-3}))\\ &= \text{error}'(v_n)\cdot f_n'(v_{n-1})\cdot f_{n-1}'(v_{n-2})\cdot\ldots\cdot\frac{\partial}{\partial w}(f_j(v_{j-1}))\end{aligned}$$

con f_i' derivata di f_i rispetto ai suoi input. L'espansione prosegue fino a f_j. L'ultima derivata parziale non è istanza della regola di concatenazione non essendo v_{j-1} funzione di w, quindi si usa la regola di decomposizione lineare e w viene moltiplicato per la componente di v_{j-1} appropriata.

BP determina l'aggiornamento per ogni peso in due passate per ogni esempio:

- *Predizione*: per ogni layer, dati i valori degli input, si calcola il valore degli output, v_i (nell'algoritmo che segue ogni v_i nell'array *values* associato al layer);
- *Retro-propagazione*: andando a ritroso attraverso i layer, si determina l'aggiornamento di ciascun peso (dei layer lineari); si calcolano (a partire da 0) $\text{error}'(v_n)\cdot\prod_{i=0}^{k}f_{n-i}'(v_{n-i-1})$ per essere passati ai layer inferiori; tale input costituisce il *termine d'errore* per un layer, combinato con i valori calcolati al passo di predizione per aggiornare tutti i pesi del layer e calcolare il termine d'errore per il layer inferiore.

BP combina la SGD con la *programmazione dinamica* memorizzando i risultati intermedi.

Modularizzazione

Un *layer* viene costruito come modulo indipendente che implementa le due passate, predizione e retro-propagazione, con l'aggiornamento dei pesi dopo ogni *batch* di esempi.

La *classe* Dense implementa una funzione lineare *densa* in cui ogni unità di output è connessa a tutte quelle di input. Dato l'array w dei pesi, $w[i,j]$ è il peso per la connessione dell'unità di input i all'output j e $w[n_i,j]$ costituisce il bias per l'output j (associato all'input implicito 1). I metodi output e Backprop vengono chiamati su ogni esempio.

- $output(in)$ restituisce i valori di output per i valori di input del vettore in;
- $Backprop(error)$ restituisce il vettore degli errori in input e aggiorna i gradienti
- $update()$ aggiorna i pesi per un batch di esempi

ogni classe memorizza in e out se utili a *Backprop*

```
class Dense(n_i,n_o) // n_i = n. di input, n_o n. di output
    for each 0 ≤ i ≤ n_i e 0 ≤ j < n_o do
        d[i,j] ← 0; w[i,j] ← valore casuale
    method output(in) // in array di lunghezza n_i
        for each j do out[j] ← w[n_i,j] + Σ_i in[i] * w[i,j]
        return out
    method Backprop(error) // error array di lunghezza n_o
        for each i,j do d[i,j] ← d[i,j] + in[i] * error[j]
        for each i do ierror[i] ← Σ_j w[i,j] * error[j]
        return ierror
    method update() // aggiorna tutti i pesi: implementa la SGD
        for each i,j do
            w[i,j] ← w[i,j] − η/batch_size * d[i,j] // η learning rate
            d[i,j] ← 0
```

```
procedure Neural_network_learner(Xs, Ys, Es, functions, η, batch_size)
    Input
        Xs: feature di input, Xs = (X_1, ..., X_n)
        Ys: feature target
        Es: insieme di esempi di training
        functions: sequenza di funzioni che definiscono la rete
        batch_size: n. di esempi in ogni batch
        η: learning rate (passo della GD)
    repeat
        batch ← campione casuale di batch_size esempi
        for each esempio e ∈ batch do
            for each unità di input i do values[i] ← X_i(e)
            for each fun ∈ functions in ordine ascendente do
                values ← fun.output(values)
                for each unità di output j do error[j] := φ_o(values[j]) − Ys[j]
                for each fun ∈ functions in ordine discendente do
                    error := fun.Backprop(error)
        for each fun ∈ functions che contiene pesi do
            fun.update()
    until terminazione
```

- *functions* è una sequenza di funzioni (parametriche) che definisce la rete, tipicamente, per ciascun layer, una lineare e una di attivazione, implementata da una classe che implementi le due passate e memorizzi l'informazione necessaria;
- la prima funzione del layer più basso ha input corrispondenti alle feature di input; per le successive, il numero di input è pari al numero di output della funzione precedente, mentre il numero output della funzione finale è pari al numero di feature target, o al numero di valori per una feature di output categorica (non binaria)
- si assume che il tipo di output della funzione di attivazione sia utile alla funzione di errore (loss), in modo che la derivata sia proporzionale a valore predetto meno il valore effettivo, questo vale per l'identità, la sigmoidale la softmax; ϕ_o è la funzione di attivazione finale, usata in Neural_network_learner, ma non viene implementata come classe; altre combinazioni con funzioni d'errore/loss potrebbero avere forme diverse che vengono automatizzate nelle librerie (*derivazione simbolica*).

```
class ReLU()
    method output(in)
        for each i : 0 ≤ i < n_i do out[i] := max(0, in[i])
        return out
    method Backprop(error)
        for each i : 0 ≤ i < n_i do
            ierror[i] := error[i] if (in[i] > 0) else 0
            return ierror
class sigmoid()
    method output(in)
        for each i : 0 ≤ i < n_i do out[i] := 1/(1 + exp(−in[i]))
        return out
    method Backprop(error)
        for each i : 0 ≤ i < n_i do ierror[i] := out[i] * (1 − out[i]) * error[i]
        return ierror
```

Esempio — Rete vista in precedenza per l'`if-then-else` con i dati visti nel cap.7

- si invoca `Neural_network_learner` assegnando a *functions* la lista `[Dense(3,2),ReLU(),Dense(2,1)]`
- un run con 10000 epoche, learning rate = 0.01 e batch di 8 esempi produce i pesi seguenti (3 cifre significative):
  - $h_1 = \text{relu}(2.47 * x + 0.0000236 * y - 2.74 * z + 2.74)$;
  - $h_2 = \text{relu}(3.62 * x + 4.01 * y + 0.228 * z - 3.84)$;
  - $output = \text{sigmoid}(-4.42 * h_1 + 6.64 * h_2 + 4.95)$;
- la funzione risultante approssima con un errore nel caso pessimo dell'1%;
  - predizione con la massima log loss 0.99 per l'esempio $\neg x \wedge \neg y \wedge z$, per il quale doveva essere 1
- al variare dei run si ottengono diversi pesi
- in tale caso semplice si ottiene un target interpretabile (approssimativamente):
  - h_1 positiva a meno che $x = 0$, $z = 1$
  - h_2 grande solo quando $x = 1$, $y = 1$

Nell'apprendimento *sub-simbolico* si ha un *contrasto apparente* con l'ipotesi di sistema di simboli fisico intrinseco: la rete codifica un significato che si può solo intravedere attraverso i valori delle unità nascoste ma a tali unità non è associato un *significato preciso*. Solo in alcuni casi si può interpretare cosa rappresentino ed esprimerlo in forma concisa.

cfr. XAI: eXplainable AI

3 Estensioni (Cenni)

3.1 Migliorare l'Ottimizzazione

Le seguenti tecniche sono state studiate per rendere più efficace l'ottimizzazione dei parametri, specie nei casi in cui siano molti (anche milioni). In particolare va settato opportunamente il *passo* di apprendimento per permettere alla SGD di risolvere problemi di minimizzazione in cui lo spazio si configura con cosiddetti **canyon** con una funzione d'errore a U in una data dimensione mentre nelle altre presenta una pendenza, oppure con **punti di sella** dove un minimo in una dimensione si incontra con un massimo in un'altra.

Per ovviare a tali problemi si deve adattare il passo nel corso del processo di ottimizzazione. I modi principali sono due e prevedono passi diversi per diverse dimensioni:

- se il segno del gradiente non cambia, si può estendere il passo, se, invece, continua a cambiare allora lo si deve ridurre;
- ogni aggiornamento dovrebbe seguire la direzione del gradiente e la grandezza dovrebbe dipende dal fatto che il gradiente sia maggiore o minore del suo valore storico.

I due modi possono essere combinati.

Il **momentum** per ciascun parametro costituisce la *velocità* dell'aggiornamento del parametro, mentre l'aggiornamento della SGD si comporta come l'accelerazione. Il momentum determina la *dimensione* del passo: va incrementato se l'accelerazione ha lo stesso segno, e decrementato altrimenti. Può servire a risolvere problemi come quelli dei canyon o del rumore comportato dal fatto che un batch non contenga molti esempi.

Il metodo di ottimizzazione **RMS-Prop** (*root mean squared propagation*) si basa sull'idea che la grandezza del cambiamento per ogni peso dipende dal confronto fra il (quadrato del) gradiente e il suo valore storico (anziché il suo valore assoluto). Per ogni peso si aggiorna una media mobile del quadrato del gradiente. Ciò serve a determinare di quanto il peso debba cambiare. Serve anche una correzione utile a non incappare nelle problematiche relative alla divisione per numeri molto piccoli.

Adam (*adaptive moments*) è un ottimizzatore che usa sia il momentum sia il gradiente quadratico e correzioni necessarie per l'inizializzazione comune a 0 dei parametri, stima non buona per calcolare medie.

Inizializzazione: ai fini della convergenza conviene normalizzare i parametri in input e usare circospezione nell'inizializzare i pesi. Le feature (continue) di input potrebbero avere diverse unità di misura e utilità per cui andrebbero standardizzate (e rese indipendenti). Per feature di input categoriche tipicamente rappresentate attraverso *variabili indicatrici* binarie per i diversi valori del dominio (*one-hot encoding*). I pesi delle unità latenti non possono essere inizializzate tutti con lo stesso valore per evitare che tutte le funzioni dello stesso layer convergano verso una stessa funzione (*lockstep*). Pertanto sarà meglio partire da valori casuali. Per le unità di output, il bias andrebbe inizializzato con il miglior valore da predire quando gli input sono tutti a zero. Ad esempio la media sul dataset per problemi di regressione o la sigmoidale inversa della probabilità empirica nei problemi categorici binari. Gli altri pesi possono partire dal valore nullo. Ciò consente di apprendere dagli input e non da medie pregresse.

3.2 Migliorare la Generalizzazione

Occorre anche assicurare in partenza che il modello possa adattarsi al training set, evitando il sovradattamento.

Per assicurarsi che si stia apprendendo qualcosa l'errore sul training set dovrebbe almeno essere inferiore a quello dei modelli-base più semplici (stime fatte ignorando le feature di input). Altrimenti bisogna cambiare la configurazione del modello.

Se risulta migliore dei modelli base ma non abbastanza occorre cambiare modello. Se già non va benissimo sul training set non potrà migliorare sul test set o su nuovi esempi. Una performance non ottimale è un indice di sotto-adattamento: il modello scelto è troppo semplice (ad esempio la regressione logistica su un problema non linearmente separabile). Il problema, in ultima analisi, potrebbe dipende da feature non abbastanza informative per cui le stime di base potrebbero rappresentare il meglio di quello che si può fare.

Se l'algoritmo funziona sul training set ma non su un validation set allora non riesce a generalizzare e va in sovradattamento, pertanto il modello andrebbe semplificato. Se i dati sono pochi i modelli da usare dovrebbero essere semplici.

Sarebbe utile operare il tuning degli iperparametri via *cross validation*. Tale processo può essere automatizzato prendendo il nome di **autoML**. Gli iperparametri da considerare comprendono:

- l'algoritmo,
- il numero di layer, l'ampiezza di ciascun layer,
- il numero di epoche (per consentire l'*early stopping*),
- il learning rate,
- le dimensioni del batch di esempi,
- i parametri di regolarizzazione (L1 o L2 in caso di carenza di dati).

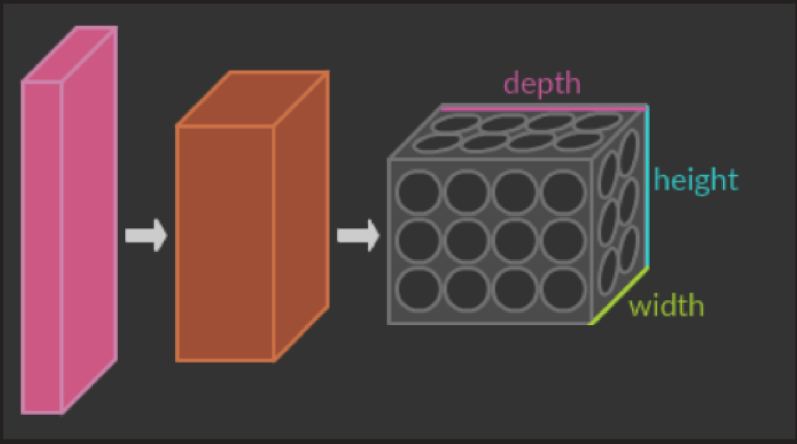
Nel caso delle reti profonde si può prendere in considerazione il meccanismo di **dropout** che porta a ignorare casualmente alcune unità in fase di training, ossia settare temporaneamente l'output a 0. tale meccanismo viene controllato dal parametro `rate`, ossia la proporzione di valori annullati (tipicamente pari a 0.5 per le unità latenti e 0.2 per quelle di input).

Altri metodi:

- raccogliere più dati può essere il modo più economico;
- *data augmentation*: nuovi esempi, trasformando quelli disponibili (ad esempio, nel caso di immagini, mediante traslazione, rotazione, riscalatura) anche con l'aggiunta di rumore o cambiando i contesti;
- usare la *feature engineering*;
- riusare esempi disponibili per problemi correlati, condividendo rappresentazioni delle feature a basso livello. Ad esempio nel *multi-task learning* si condividono i layer inferiori appresi per un problema nella soluzione di altri.

3.3 Reti Convolutive

Una **rete convolutiva** (*convolutional neural network*, CNN) è specializzata nel riconoscere immagini o nella visione sfruttando il principio di *località* e la *condivisione* dei parametri.



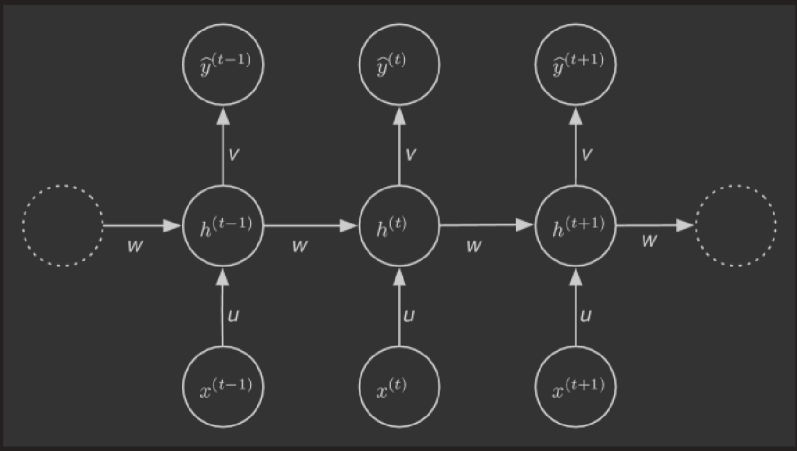
Una CNN utilizza un **kernel** (o *convolution mask*, o *filtro*) come operatore lineare parametrico, da applicarsi a regioni locali (*patch*). Tipicamente si usano kernel monodimensionali (ad es. per sequenze di parole) o bidimensionali (per immagini e video). I parametri del kernel vanno appresi dai dati. Feature locali determinano i loro valori in funzione di punti vicini. Lo stesso kernel (ossia una configurazione dei suoi parametri) viene applicato in parti diverse dell'istanza in input (ad es. un'immagine). Si possono applicare più kernel in maniera concorrente. In tal caso di avranno più output, ad esempio più array, detti *canali*. Questa suddivisione può essere prevista già in input.

In un **pooling layer**, si applica una funzione prefissata degli output delle unità in un kernel, anziché un kernel lineare. Un tipico pooling layer è il *max-pooling*, che è come la circonvoluzione, ma applica il massimo invece di una funzione lineare. Una tipica architettura di CNN usa un layer convolutivo, seguito da una funzione non lineare, come la ReLU, seguita da un pooling layer, il cui output va in input a layer convolutivo e così via.

3.4 Modelli per Sequenze

A volte i dati si presentano come sequenze di lunghezza diversa, non prefissata (come per le immagini). Si pensi alle frasi nel linguaggio naturale.

Una **rete ricorrente** (*recurrent neural network*, RNN) vanno oltre il meccanismo di *feed-forward*, ammettendo connessioni con unità in layer non superiori per formare cicli.



Ciò permette l'utilizzo di uno dei layer come *memoria di stato*, e consente, fornendo in input una *sequenza temporale* di valori, di modellarne un comportamento dinamico dipendente dalle informazioni ricevute in istanti di tempo precedenti. Tale tipo di reti è molto utilizzato per predizioni su serie temporali in cui vada previsto il prossimo output.

Le reti **Long Short-Term Memory** (LSTM) costituiscono un tipo speciale di RNN progettate in modo che sia mantenuta la memoria a meno di sostituzioni con nuova informazione. Ciò consente una miglior cattura delle dipendenze di lungo termine rispetto alle RNN tradizionali. Intuitivamente, invece di apprendere la funzione da $h^{(t-1)}$ a $h^{(t)}$, si apprende il cambiamento $\Delta h^{(t)}$.

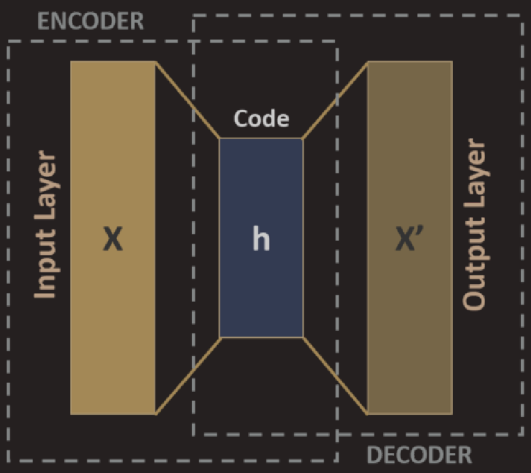
Un modo per aumentare l'efficacia dei modelli sequenziali è quello di mettere in grado il modello di soffermarsi su particolari parti delle sequenze. Il meccanismo di **attention** prevede l'uso di una distribuzione di probabilità sulle diverse parti dell'input (parole, immagini) per assegnare loro un diverso peso all'interno di una stessa *finestra*, consentendo così di sfruttare informazioni di contesto.

Questo calcolo di pesi può avvenire in maniera simultanea attraverso modelli **transformer**, basati su più meccanismi di attenzione (*multi-heads*), vanno a inframmezzarsi con layer densi e le funzioni di attivazione. Essi trasformano sequenze in sequenze operando una **codifica posizionale**, risultando più facili da apprendere rispetto a RNN e LSTM.

Per tutti questi modelli, algoritmi ed esempi di utilizzo si trovano nel cap. 8 di [PM23].

3.5 Altri Modelli Neurali Generativi

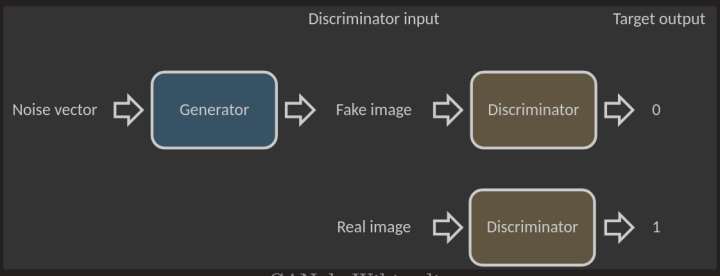
Un **autoencoder** è una rete con uno strato/modello *encoder* con input e output uguali ricostruiti da uno strato/modello **decoder** ed è atta a costruire una *rappresentazione* interna vettoriale (a dimensioni ridotte) che può essere usata come modello generativo (ad esempio di immagini o testo casuale).



Autoencoder da wikipedia

Con tali modelli si mira a trasformare i dati in ingresso (ad esempio le parole nelle frasi) in forma vettoriale con un meccanismo di codifica dell'input in una rappresentazione (low-rank) intermedia in uno strato latente che garantisca la loro ricostruzione nello strato di output. Un tipico utilizzo può essere quello di generare immagini artificiali a partire da testo. In generale, tale meccanismo è molto utilizzato in NLP, ad esempio nei modelli **Word2vec** e derivati, come quelli relativi al *Knowledge Graph embedding* rappresentati da triple.

Una *adversarial network* come la **generative adversarial network** (GAN) impara a generare esempi con una distribuzione simile a quella degli esempi di training. Essa viene allenata sia a predire parte dell'output, sia a non essere in grado di predire altro output. È formata da un *generatore* di esempi con l'obiettivo di ingannare il *discriminatore* che invece deve determinare se siano veri o *fake*.



GAN da Wikipedia

Un possibile utilizzo delle GAN può essere quello di escludere dal modello alcune feature sensibili (*debiasing GAN*), ad esempio per evitare che predizioni che riguardino persone, ad esempio per selezioni di lavoro, vengano influenzate dal genere o dal colore della carnagione.

Un **modello di diffusione** rappresenta un metodo efficace di IA generativa (*image generation*): esso costruisce una sequenza di esempi (ad esempio immagini) più rumorose, a partire dai dati disponibili, aggiungendo ripetutamente rumore fino ad avere solo rumore, imparando poi a invertire questo processo. DALL-E è una tipica applicazione di questo modello combinato con alcuni dei precedenti.

Riferimenti Bibliografici

[**PM23**] D. Poole, A. Mackworth: *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press. 3a ed. 2023 (ch.7)
[**Fla12**] P. Flach: *Machine Learning*. Cambridge University Press. 2012
[**HTF09**] T. Hastie, R. Tibshirani, J. Friedman: *The Elements of Statistical Learning* Springer. 2nd ed [ESL] 2009
[**McK05**] D.J.C. MacKay: *Information Theory, Inference, and Learning Algorithms* Cambridge University Press. 2005. disponibile su [ITPRN]
[**Mit97**] T. Mitchell: *Machine Learning*. McGraw Hill. 1997
[**Ros58**] F. Rosenblatt: *The perceptron: a probabilistic model for information storage and organization in the brain* Psychological Review 65 (6), pp. 386-408. 1958

Link per Approfondimenti

[**DLBook**] http://www.deeplearningbook.org
[**ITPRN**] http://www.inference.org.uk/itprnn/book.html
[**HPNN**] https://www.researchgate.net/publication/271841595_HISTORY_AND_PHILOSOPHY_OF_NEURAL_NETWORK
[**Alice**] https://arxiv.org/abs/2404.17625#

Note

Note su PyTorch e Keras in Appendice B.2 di [PM23]