UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DeepLearning

NN, CONV, RNN, …

(Versione 10/02/2019)

Stesura a cura di:
Stefano Ivancich

INDICE

Questa dispensa è scritta da studenti senza alcuna intenzione di sostituire i materiali universitari. Essa costituisce uno strumento utile allo studio della materia ma non garantisce una preparazione altrettanto esaustiva e completa quanto il materiale consigliato dall'Università.
Lo scopo di questo documento è quello di riassumere i concetti fondamentali degli appunti presi durante la lezione, riscritti, corretti e completati facendo riferimento alle slide, ai libri di testo e alle seguenti fonti:

- Stanford cs231n
- DeppLearning.ai by Andrew Ng
- Deep Learning Book - Ian Goodfellow

per poter essere utilizzato nella realizzazione di reti neurali come un manuale “pratico e veloce” da consultare. Non sono presenti esempi e spiegazioni dettagliate, per questi si rimanda ai testi citati e alle slide.

Se trovi errori ti preghiamo di segnalarli qui:
www.stefanoivancich.com
ivancich.stefano.1@gmail.com
Il documento verrà aggiornato al più presto.

1. Introduzione

La sfida all'intelligenza artificiale è quella di risolvere compiti facili da eseguire per le persone, ma difficili da descrivere formalmente, problemi che risolviamo in modo intuitivo.
Questo documento parla della soluzione a questi problemi più intuitivi.
Se disegniamo un grafico che mostra come questi concetti sono costruiti uno sopra l'altro, il grafico è profondo, con molti livelli. Per questo motivo, questo approccio all'intelligenza artificiale viene chiamato **Deep Learning**.

Approccio basato sulla conoscenza (knowledge base approach): codificare la conoscenza del mondo tramite linguaggi formali, poi computer usa regole di inferenza logica.
Nessuno dei progetti finora svolti utilizzando questo approccia ha portato risultati di successo.
Le difficoltà incontrate suggeriscono che i sistemi di intelligenza artificiale hanno bisogno della capacità di acquisire le proprie conoscenze, estraendo schemi dai dati grezzi. Questa capacità è nota come Apprendimento Automatico (**Machine Learning**).

Molte attività di intelligenza artificiale possono essere risolte progettando il giusto set di funzionalità (features) da estrarre per quell'attività, quindi fornendo queste funzionalità a un semplice algoritmo di apprendimento automatico.
Tuttavia, per molte attività, è difficile sapere quali funzionalità devono essere estratte.
Representation Learning: soluzione a questo problema, in cui si usa il Machine Learning per scoprire non solo la mappatura dalla rappresentazione all'output, ma anche la rappresentazione stessa.
Questo metodo ha diversi vantaggi:

- ottiene spesso prestazioni migliori rispetto a quelle che si otterrebbero con rappresentazioni progettate a mano.
- permette ai sistemi di AI di adattarsi rapidamente a nuovi compiti, con un intervento umano minimo.
- può scoprire un buon set di funzionalità per un compito semplice in pochi minuti. Mentre questo può richiedere anche decenni per una comunità di ricercatori.

Un esempio di Representation Learning è l'**Autoencoder**. Che è la combinazione di un **encoder**, funzione che converte l'input in una rappresentazione diversa, e il **decoder**, funzione che converte la nuova rappresentazione nel formato originale.

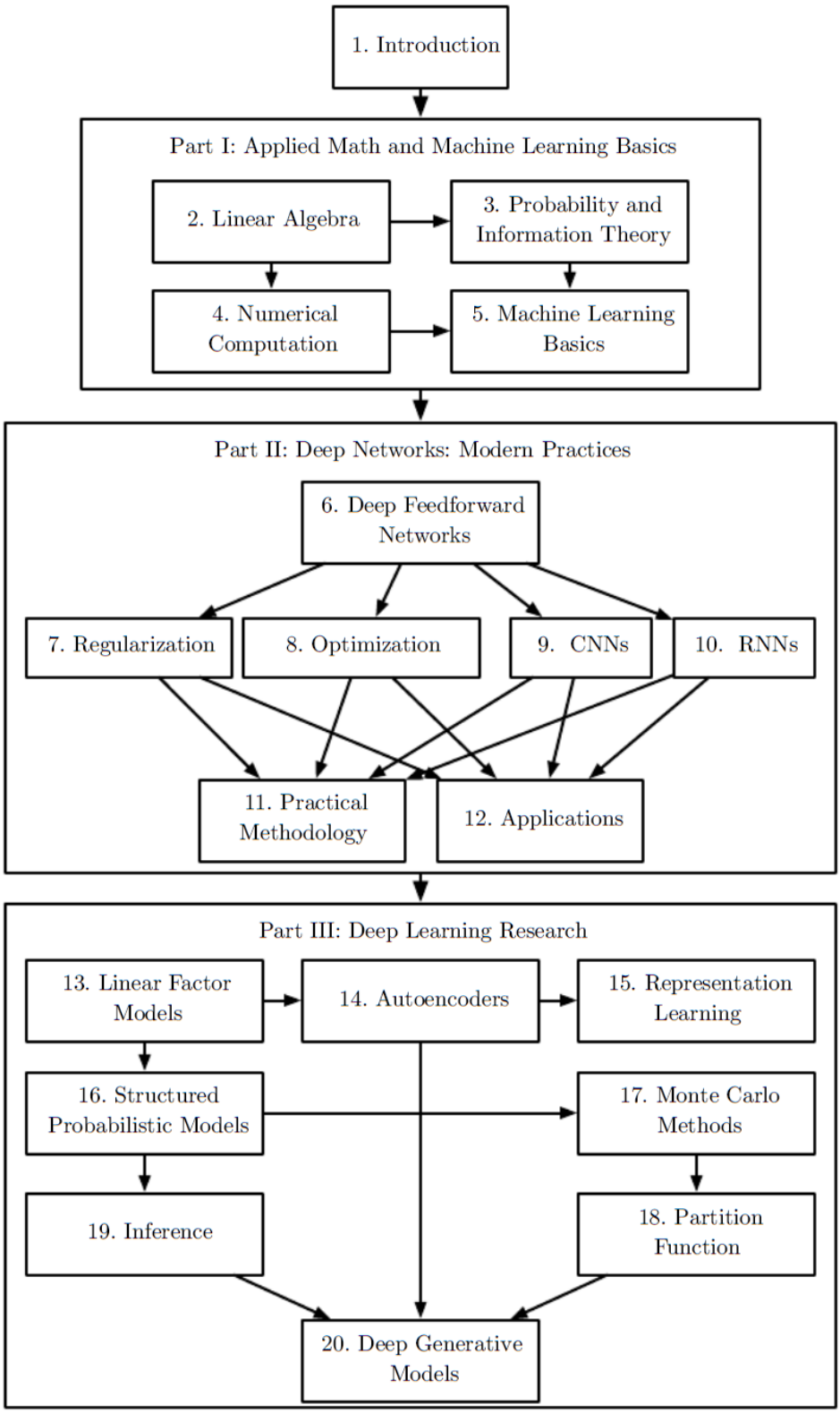
Quando si progettano le features o gli algoritmi, lo scopo è quello di separare i fattori di variazione (**factors of variation**) che spiegano i dati osservati.
Con "fattori" ci si riferisce a fonti di influenza separate, come oggetti non osservabili o forze non osservabili nel mondo fisico che influenzano le quantità osservabili.
La difficoltà in molte applicazioni del mondo reale è che molti dei fattori di variazione influenzano ogni singolo dato che siamo in grado di osservare. Quindi può diventare molto difficile estrarre features dai dati grezzi.
Il **DeepLearning** risolve questo problema cruciale del Representation Learning introducendo rappresentazioni espresse in termini di altre rappresentazioni più semplici. E quindi permette al computer di costruire concetti complessi partendo da concetti semplici.

Un esempio di modello di DeepLearning è il **Multilayer Perceptron** (MLP), che non è altro che una funzione matematica che mappa alcuni set di valori di input in valori di output.

Esistono due modi principali per misurare la profondità di un modello:

- basandosi sul numero di istruzioni sequenziali che devono essere eseguite per valutare l'architettura.
- Tramite modelli probabilistici.

Contenuto del documento:



Parte 1 - Basi matematiche e Machine Learning

2.Algebra lineare

2.1. Oggetti matematici utilizzati

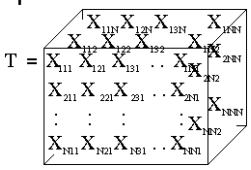
Scalare: numero singolo.
Vettore: sequenza di numeri ordinata.

Ogni numero è identificabile dal suo indice. Es. x_1 è il primo elemento. $\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$

Lo si può pensare come l'identificazione di un punto dello spazio, di cui ogni elemento indica la coordinata lungo un asse.

Matrice: sequenza di numeri ordinata in 2 dimensioni. $\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}$

Tensore: sequenza di numeri ordinata in più di 2 dimensioni.



2.2. Matrici

2.2.1. Operazioni delle matrici

Matrice trasposta: $\boldsymbol{A}^T = scambia\ righe\ con\ colonne$
Traccia: $Tr(\boldsymbol{A}) = \sum_i A_{i,i}$ Somma di tutti gli elementi della diagonale principale.
Somma di matrici: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}\ con\ C_{i,j} = A_{i,j} + B_{i,j}\ solo\ se\ hanno\ la\ stessa\ dimensione$
Somma/moltiplicazione per uno scalare: $\boldsymbol{D} = a * \boldsymbol{B} + c\ con\ D_{i,j} = a * B_{i,j} + c$
Prodotto tra matrici: $\boldsymbol{C} = \boldsymbol{AB}\ dove\ \boldsymbol{C_{i,j}} = \sum_{\boldsymbol{k}} \left(\boldsymbol{A_{i,k}} * \boldsymbol{B_{k,j}}\right)\ con\ A_{m\times n}, B_{n\times p}\ e\ C_{m\times p}$
Prodotto elemento per elemento: $\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$
Prodotto scalare tra vettori: $x^T y = x_1 y_1 + \cdots + x_n y_n = \|x\|\|y\| cos\theta$
Proprietà: distributiva, associativa ma non commutativa. (Il prodotto tra 2 vettori è commutativo)

2.2.2. Tipi speciali di matrici

Matrice Identica: $\boldsymbol{I_n} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$, vettore moltiplicato per questa matrice non cambia.
Matrice Inversa: $\boldsymbol{A^{-1}}\ tale\ che\ \boldsymbol{A^{-1}A} = \boldsymbol{I_n}$
Esiste solo se la matrice è quadrata e le sue colonne sono linearmente indipendenti.
Usata principalmente per questioni teoriche, non viene utilizzata spesso nelle applicazioni software perché può essere rappresentata solo con una precisione limitata dai computer.
Matrice diagonale: $D = diag(\boldsymbol{v}) = \begin{bmatrix} v_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & v_n \end{bmatrix}$
È molto efficiente da usare ed è facile da invertire $diag(v)^{-1} = diag\left(\left[\frac{1}{v_1}, \ldots, \frac{1}{v_n}\right]^T\right)$
Matrice simmetrica: $\boldsymbol{A} = \boldsymbol{A}^T$ matrice uguale alla sua trasposta
Matrice ortogonale: se le colonne sono ortogonali e le righe ortonormali. Quindi $\boldsymbol{A^{-1}} = \boldsymbol{A^T}$

2.3. Spazi Vettoriali e Norme

Spazio vettoriale: insieme infinito di vettori
Combinazione Lineare: $\boldsymbol{a_1 v_1 + \cdots + a_n v_n}$ $con\ a_i\ scalare\ e\ v_i\ vettore$ questa operazione restituisce un nuovo elemento dello spazio.
Sottospazio generato $\langle S \rangle = \{\lambda_1 v_1 + \cdots + \lambda_r v_r | v_i \in S, \lambda_i \in K\}$ insieme di tutte i punti ottenibili dalle combinazioni lineari di un insieme di vettori.
Vettori Linearmente Indipendenti: se l'unica n-pla $(a_1, \dots a_n)$ che annulla la combinazione lineare $a_1 v_1 + \cdots + a_n v_n$ è quella di coefficienti tutti nulli.
Vettore normale(versore): $\|u\| = 1$
Vettori ortogonali: $u \perp v \Leftrightarrow u * v = 0$
Vettori ortonormali: se sono sia ortogonali che normali.

Norma: misura la dimensione di un vettore.
Norma L_p: $\|\boldsymbol{x}\|_{\boldsymbol{p}} = \left(\sum_{\boldsymbol{i}} |\boldsymbol{x_i}|^{\boldsymbol{p}}\right)^{\frac{\boldsymbol{1}}{\boldsymbol{p}}}\ con\ p \geq 1$
Norma euclidea: p=2, distanza dall'origine al punto x. $\|v\| = \sqrt{v * v} = \sqrt{v_1^2 + \cdots + v_n^2}$
Norma L1: cioè con p=1, viene usata quando la differenza tra gli elementi nulli e non nulli è molto importante.
Norma massima: $\boldsymbol{L^{\infty}} = \|\boldsymbol{x}\|_{\infty} = \boldsymbol{\max_i |x_i|}$
Norma di Frobenius: $\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2} = \sqrt{Tr(AA^T)}$ misura la dimensione di una matrice. Simile a L2.

2.4. Decomposizioni

Decomporre le matrici aiuta a ricavare informazioni sulle loro proprietà che altrimenti non si vedrebbero dalla semplice rappresentazione di sequenze di numeri.

2.4.1. Auto-decomposizione

Si decompone una **matrice quadrata** in un insieme di auto-valori e auto-vettori.

Autovettore: vettore non nullo che moltiplicato per la matrice altera solo la propria scala.

$$\boldsymbol{Av} = \lambda \boldsymbol{v}$$

Autovalore: λ corrispondente all'autovettore.

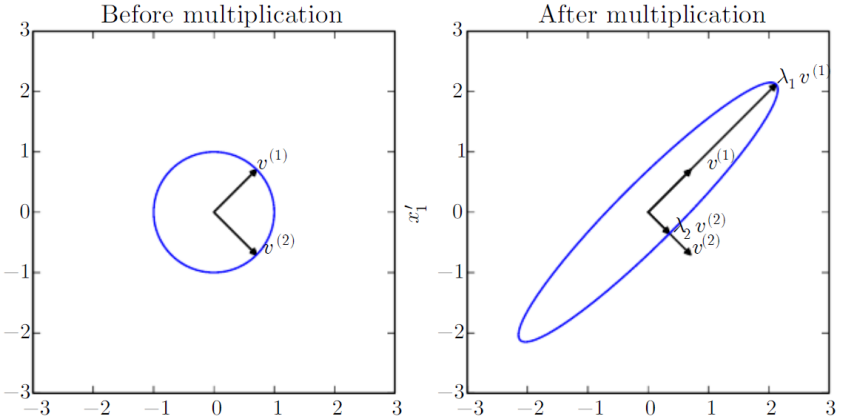
Auto-decomposizione di A: $\boldsymbol{A} = \boldsymbol{V} diag(\boldsymbol{\lambda}) \boldsymbol{V}^{-1}$ $con\ \boldsymbol{\lambda}: autovalori, \boldsymbol{V}: autovettori\ in\ colonne$

Non tutte le matrici possono essere auto-decomposte, in alcuni casi la decomposizione esiste, ma coinvolge numeri complessi.

Ogni matrice simmetrica reale può essere decomposta usando solo reali:

$$\boldsymbol{A} = \boldsymbol{Q\Lambda Q^T}\ dove\ \boldsymbol{Q}\ \text{è}\ ortogonale\ e\ composta\ da\ autovettori, \boldsymbol{\Lambda}\ \text{è}\ diagonale$$

In questo caso si può pensare ad A come un moltiplicatore di scala di v:



L'auto-decomposizione può non essere unica, di solito si ordinano gli autovalori di $\boldsymbol{\Lambda}$ in ordine decrescente. Sotto questa convenzione, l'auto-decomposizione è unica solo se gli autovalori sono tutti diversi.

Se almeno uno degli autovalori è 0 allora la matrice è singolare (le colonne non solo LI).

$f(\boldsymbol{x}) = \boldsymbol{x^T Ax}\ vincolato\ a\ \|\boldsymbol{x}\|_2 = 1$

Il valore massimo di f all'interno di una regione di vincolo è l'autovalore massimo, il minimo è l'autovalore minimo.

Matrice Positiva Definita: se gli autovalori sono tutti positivi non nulli. $\boldsymbol{x^T Ax} = 0 \Rightarrow \boldsymbol{x} = 0$

Matrice Positiva Semidefinita: se gli autovalori sono tutti positivi o nulli. Garantisce $\forall \boldsymbol{x}, \boldsymbol{x^T Ax} \geq 0$

Matrice Negativa Definita: se gli autovalori sono tutti negativi non nulli.

Matrice Negativa Semidefinita: se gli autovalori sono tutti negativi o nulli.

Determinante: $\det(A) = prodotto\ autovalori.$

Funzione che mappa le matrici in numeri scalari reali.

Il valore assoluto del determinante può essere pensato come una misura di quanto la moltiplicazione per la matrice espande o contrae lo spazio

Se il determinante è 0, allora lo spazio viene contratto completamente lungo almeno una dimensione, facendolo perdere tutto il suo volume. Se il determinante è 1, la trasformazione conserva il volume

2.4.2. Decomposizione ai valori singolari (SVD)

Fornisce un altro modo di decomporre la matrice, in vettori singolari e valori singolari.
È applicabile anche alle matrici non quadrate.
Decomposizione: $\boldsymbol{A = UDV^T}$ con $\boldsymbol{A_{mxn}, U_{mxm}, D_{mxn}, V_{nxn}}$
Vettori singolari sinistri: colonne di U. Calcolabili come gli autovettori di $\boldsymbol{AA^T}$.
Vettori singolari destri: colonne di V. Calcolabili come gli autovettori di $\boldsymbol{A^TA}$.
Valori singolari non nulli: radici quadrate degli autovalori non nulli di $\boldsymbol{A^TA}$ e $\boldsymbol{AA^T}$.
Viene utilizzata per invertire matrici non quadrate.

Matrice pseudo-inversa di Moore-Penrose: $\boldsymbol{A^+ = \lim_{\alpha \to 0}(A^TA + \alpha I)^{-1}A^T}$
Algoritmo pratico che viene utilizzato: $\boldsymbol{A^+ = VD^+U^T}$
$dove\ U, D\ e\ V\ sono\ le\ decomposizioni\ singolari\ di\ A$
$e\ D^+\ ricavata\ prendendo\ la\ trasposta\ degli\ elementi\ non\ nulli\ di\ D$

Sistema di equazioni lineari: $\boldsymbol{Ax = b}$ con $\boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m, \boldsymbol{x} \in \mathbb{R}^n$ $vettore\ di\ incognite$
Soluzione: $\boldsymbol{x = A^{-1}b}$ se $\boldsymbol{A}$ è $quadrata$
Quando A ha più colonne che righe: $\boldsymbol{x = A^+b}$ $con\ x\ che\ ha\ norma\ euclidea\ minima$
Quando A ha più righe che colonne: $\boldsymbol{x = A^+b}$ $con\ x\ più\ vicino\ possibile\ a$ $\boldsymbol{\|Ax - b\|_2}$

3. Probabilità e Teoria dell'Informazione

La **teoria della probabilità** è uno strumento matematico per rappresentare affermazioni incerte. Fornisce un mezzo per quantificare l'incertezza e assiomi per derivare nuove affermazioni incerte. Nelle applicazioni di intelligenza artificiale viene utilizzata in due modi principali: dice come i sistemi IA dovrebbero ragionare e per analizzare teoricamente il comportamento dei sistemi di IA.

La **teoria dell'informazione** consente di quantificare la quantità di incertezza in una distribuzione di probabilità.

Si può assumere che la CPU esegua perfettamente ogni istruzione. Errori a livello hardware a volte ci sono, ma sono così rari che la maggior parte delle applicazioni software non devono essere progettate per tenerne conto.
Invece il Machine Learning deve sempre gestire quantità incerte e talvolta può anche essere necessario occuparsi di quantità stocastiche (non deterministiche).

Ci sono 3 fonti principali di incertezza:

- Stocasticità intrinseca nel sistema che si sta modellando.
- Osservabilità incompleta.
- Modellazione incompleta, ovvero quando si usa un modello che deve scartare alcune delle informazioni che si osservano, questo genera delle incertezze nelle previsioni del modello.

In molti casi, è più pratico utilizzare regole semplici ma incerti piuttosto che complesse ma certe.

Probabilità frequentista: correlata direttamente ai tassi in cui si verificano gli eventi.
Probabilità Bayesiana: correlata ai livelli qualitativi di certezza.

La teoria della probabilità fornisce un insieme di regole formali per determinare la probabilità che una proposizione sia vera data la probabilità di altre proposizioni.

3.1. Variabili aleatorie

Variabile aleatoria x: è una variabile che può assumere diversi valori in modo casuale.
A esempio x_1 e x_2 sono possibili valori che la variabile x può assumere.
È solo una descrizione degli stati che sono possibili e deve essere accoppiata con una distribuzione di probabilità che specifica la probabilità di ciascuno di questi stati.
Variabile aleatoria discreta: ha un numero finito o infinito di stati numerabili. Gli stati non sono necessariamente numeri interi, possono non avere alcun valore numerico.
Variabile aleatoria continua: è associata valori reali.

3.2. Distribuzione di probabilità

È una descrizione di quanto sia probabile che una variabile casuale o un insieme di variabili casuali assuma ciascuno dei suoi stati possibili.

3.2.1. Variabili discrete

Funzione Densità Discreta P(x): mappa uno stato di una variabile discreta alla probabilità che la variabile possa assumere quello stato. P(x=x_1) o P(x_1) indica la probabilità che x assuma il valore x_1.
$\mathrm{x} \sim P(\mathrm{x})$ indica che alla variabile x è associata la densità discreta P(x) e verrà specificata più avanti.
Proprietà:

- dominio di P è l'insieme di tutti gli stati possibili di x
- $\forall x \in \mathrm{x}, 0 \leq P(x) \leq 1$
- $\sum_{x \in \mathrm{x}} P(x) = 1$

Distribuzione congiunta (joint): distribuzione su più variabili. P(x=x, y=y) denota simultaneamente la probabilità che x=x e y=y. Viene scritto P(x, y) per brevità.
Distribuzione uniforme: $P(\mathrm{x} = x_i) = \frac{1}{k}$ La variabile aleatoria x ha k stati diversi tutti con la stessa probabilità.

3.2.2. Variabili continue

Densità di probabilità p(x): non dà direttamente la probabilità di uno stato specifico, ma invece la probabilità di cadere all'interno di una regione infinitesimale con volume δx che è data da $p(x)\delta x$.
Proprietà:

- dominio di p è l'insieme di tutti gli stati possibili di x
- $\forall x \in \mathrm{x}, 0 \leq p(x)$
- $\int p(x)dx = 1$

Si può integrare la funzione di densità per trovare la distribuzione di probabilità effettiva di un insieme di punti. Precisamente, la probabilità che x giace in un insieme S è data dall'integrale di p(x) sull'insieme.
Distribuzione uniforme $u(x; a, b)$ dove a e b sono gli estremi dell'intervallo.
Spesso scritto come $\mathrm{x} \sim U(a, b)$

3.3. Distribuzione marginale

A volte si conosce la distribuzione di probabilità su un insieme di variabili e si vuole ricavare la distribuzione di probabilità su un solo sottoinsieme di esse.
Su **variabili discrete:** $\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y)$
Su **variabili continue:** $p(x) = \int p(x, y)dy$

3.4. Probabilità Condizionata

Probabilità di un evento, sapendo che un altro è avvenuto.

$$P(\mathrm{y} = y|\mathrm{x} = x) = \frac{P(\mathrm{x} = x, \mathrm{y} = y)}{P(\mathrm{x} = x)}$$

Regola della catena: $P\left(\mathrm{x}^{(1)}, \dots, \mathrm{x}^{(n)}\right) = P\left(\mathrm{x}^{(1)}\right) \prod_{i=2}^{n} P\left(\mathrm{x}^{(i)} \middle| \mathrm{x}^{(1)}, \dots, \mathrm{x}^{(i-1)}\right)$
Qualsiasi probabilità congiunta su più variabili può essere decomposta in distribuzioni condizionali su una sola variabile.

3.5. Indipendenza e indipendenza condizionale

x e y sono **indipendenti** $\mathrm{x} \perp \mathrm{y}$: se la loro distribuzione di probabilità può essere espressa come un prodotto di due fattori, uno che coinvolge solo x e uno che coinvolge solo y: $\forall x \in \mathrm{x}, y \in \mathrm{y}$, $p(\mathrm{x} = x, \mathrm{y} = y) = p(\mathrm{x} = x)p(\mathrm{y} = y)$

x e y sono **condizionalmente indipendenti** $\mathrm{x} \perp \mathrm{y} \,|z$ data una aleatoria casuale z: se la distribuzione di probabilità condizionale su x e y viene fattorizzata in questo modo per ogni valore di z: $\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in z,\ p(\mathrm{x} = x, \mathrm{y} = y|\mathrm{z} = z) = p(\mathrm{x} = x|\mathrm{z} = z)p(\mathrm{y} = y|\mathrm{z} = z)$

3.6. Valore atteso, Varianza e Covarianza

Valore atteso di una funzione f(x) è **il valore medio**.
Variabili discrete: $\mathbb{E}_{\mathrm{x}\sim P}[f(x)] = \sum_x P(x)f(x)$
Variabili continue: $\mathbb{E}_{\mathrm{x}\sim P}[f(x)] = \int p(x)f(x)dx$
È lineare: $\mathbb{E}_{\mathrm{x}}[\alpha f(x) + \beta g(x)] = \alpha\mathbb{E}_{\mathrm{x}}[f(x)] + \beta\mathbb{E}_{\mathrm{x}}[g(x)]$

Varianza: $\boldsymbol{Var}\big(\boldsymbol{f}(\boldsymbol{x})\big) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]$
Fornisce una misura di quanto variano i valori di una funzione di una variabile aleatoria x quando campioniamo diversi valori di x dalla sua distribuzione di probabilità.

Deviazione standard: $\boldsymbol{\partial}\big(\boldsymbol{f}(\boldsymbol{x})\big) = \sqrt{\boldsymbol{Var}\big(\boldsymbol{f}(\boldsymbol{x})\big)}$

Covarianza: $\boldsymbol{Cov}\big(\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{g}(\boldsymbol{y})\big) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[(g(y)])]$
Dà un senso di quanto due valori siano linearmente correlati tra loro.
Alti valori assoluti di covarianza significa che i valori cambiano molto e sono entrambe lontane dalle loro rispettive medie. Positiva, vuol dire che entrambe le variabili tendono ad avere valori alti simultaneamente. Negativa, vuol dire che mentre una variabile ha valori alti l'altra ha valori bassi e viceversa.
Variabili indipendenti hanno covarianza 0.

Matrice di covarianza di un vettore aleatorio: $Cov(\mathbf{x})_{i,j} = Cov\big(\mathrm{x}_i, \mathrm{x}_j\big)$

3.7. Distribuzioni comuni

Distribuzione di Bernoulli: binaria controllata dal parametro ϕ.

- $P(\mathrm{x} = 1) = \phi$
- $P(\mathrm{x} = 0) = 1 - \phi$
- $P(\mathrm{x} = x) = \phi^x(1 - \phi)^{1-x}$
- $\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$
- $Var_{\mathrm{x}}(\mathrm{x}) = \phi(1 - \phi)$

Distribuzione Multinoulli: su una variabile discreta con k stati diversi. Parametrizzato da un vettore $p \in [0,1]^{k-1}$ dove p_i dà la probabilità dello stato i-esimo
usato per riferirsi a distribuzioni su categorie di oggetti, quindi di solito che lo stato 1 abbia il valore numerico 1, ecc.

Distribuzione normale: $\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$ μ è la media, σ^2 è la varianza

$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x - \mu)^2\right)$

In assenza di una conoscenza preliminare di quale forma dovrebbe assumere una distribuzione sui numeri reali, la distribuzione normale è una buona scelta predefinita perché molte distribuzioni che desideriamo modellare sono vicine alla distribuzione normale e molti sistemi complicati possono essere modellati con successo come rumore normalmente distribuito, anche se il sistema può essere scomposto in parti con un comportamento più strutturato.

Distribuzione normale multivariata: $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sqrt{\frac{1}{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\right)$

Parametrizzata dalla matrice simmetrica positiva $\boldsymbol{\Sigma}$ che è la matrice di covarianza.

Distribuzione esponenziale: $p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x)$

Distribuzione di Laplace: $Laplace(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right)$

Distribuzione di Dirac: $p(x) = \delta(x - \mu)$ Tutta la massa concentrata in un unico punto, dove $x = \mu$

Distribuzione empirica: $\hat{p}(\boldsymbol{x}) = \frac{1}{m}\sum_{i=1}^{m} \delta\left(\boldsymbol{x} - \boldsymbol{x}^{(i)}\right)$ in cui in ogni punto $x^{(i)}$ c'è un picco di massa 1/m

Si può vedere la distribuzione empirica come specificazione della distribuzione che si campiona quando si allena un modello sul dataset. Massimizza la probabilità dei dati di allenamento.

Mistura di distribuzioni: è composta da diverse distribuzioni. È una semplice strategia per combinare più distribuzioni di probabilità per crearne una più ricca.

$P(\mathrm{x}) = \sum_i P(c = i)P(\mathrm{x}|c = i)$ $con\ P(c)\ distribuzione\ multinoulli$

Mistura Gaussiana: il componente $P(\mathrm{x}|c = i)$ è gaussiano. Ogni componente ha una media $\boldsymbol{\mu}^{(i)}$, una covarianza $\boldsymbol{\Sigma}^{(i)}$ e una probabilità a priori $\alpha_i = P(\mathrm{x}|c = i)$ che esprime la credenza su c prima di aver osservato x.

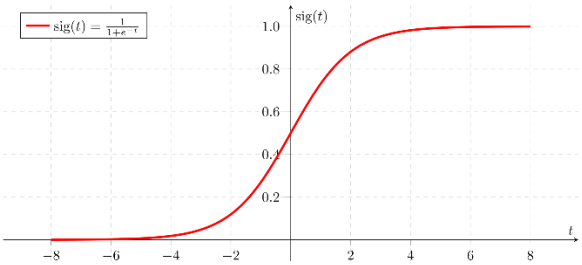
Le misture possono avere vincoli, esempio: $\boldsymbol{\Sigma}^{(i)} = \boldsymbol{\Sigma}, \forall i$

Ogni densità può essere approssimata con una mistura Gaussiana con un numero sufficiente di componenti.

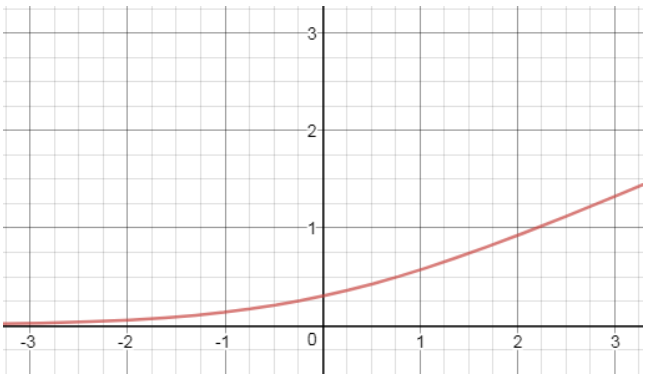
Variabile latente: è una variabile casuale che non possiamo osservare direttamente.

3.8. Funzioni comuni

Sigmoide logistica: $\sigma(x) = \frac{1}{1+e^{-x}}$ usata per produrre il parametro ϕ di una Bernoulli perché il range è (0,1).



Softplus: $\zeta(x) = \log(1 + e^x)$ utile per produrre i parametri β o σ di una distribuzione Normale perché il range è $(0, \infty)$



Proprietà:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$1 - \sigma(x) = \sigma(-x)$$

$$\log \sigma(x) = -\zeta(-x)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x)$$

$$\forall x \in (0,1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right)$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log\left(\exp(x) - 1\right)$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy$$

$$\zeta(x) - \zeta(-x) = x$$

Regola di Bayes: $P(\mathrm{x}|\mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y}|\mathrm{x})}{P(\mathrm{y})}$

Teoria delle misure: fornisce un modo rigoroso di descrivere che un insieme di punti è trascurabilmente piccolo.

3.9. Teoria dell'informazione

In questo contesto, la teoria dell'informazione spiega come progettare codici ottimali e calcolare la lunghezza attesa dei messaggi campionati date specifiche distribuzioni di probabilità ed usando vari schemi di codifica.

- Eventi probabili hanno un basso contenuto informativo, gli eventi sicuri di accadere non hanno contenuto informativo.
- Eventi meno probabili hanno un contenuto informativo più elevato.
- Gli eventi indipendenti dovrebbero avere informazioni aggiuntive.

Autoinformazione di un evento x=x: $I(x) = -\log P(x)$ è la quantità d'incertezza associata all'evento. Misurato in **nat**, quantità di informazione acquisita osservando un evento di probabilità 1/e.

Entropia di Shannon: $H(\mathrm{x}) = \mathbb{E}_{\mathrm{x}\sim P}[I(x)] = -\mathbb{E}_{\mathrm{x}\sim P}[\log P(x)]$

È la quantità di incertezza di una distribuzione è la quantità di informazione attesa in un evento.

Stabilisce il limite inferiore sul numero di nat necessari in media per codificare i simboli dalla distribuzione P.

Distribuzioni in cui il risultato è quasi certo hanno bassa entropia, quelle vicino alla distribuzione uniforme hanno entropia alta.

Entropia differenziale: quando la variabile x è continua.

Divergenza di Kullback-Leiber (KL): $D_{KL}(P||Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}[\log P(x) - \log Q(x)]$

Date due distribuzioni P e Q sulla stessa variabile, misura quanto sono differenti.

Nel caso di variabili discrete, è la quantità extra di informazioni (misurate in bit se usiamo il logaritmo di base 2, ma in machine learning usiamo solitamente nats e il logaritmo naturale) necessarie per inviare un messaggio contenente simboli tratti dalla distribuzione di probabilità P, quando usiamo un codice che è stato progettato per minimizzare la lunghezza dei messaggi estratti dalla distribuzione di probabilità Q.

Proprietà:

- Non negativa.
- È 0 se e solo se P e Q solo la stessa distribuzione.
- Non è simmetrica $D_{KL}(P||Q) \neq D_{KL}(Q||P)$

Entropia incrociata: $H(P, Q) = -\mathbb{E}_{\mathrm{x}\sim P} \log Q(x)$

3.10. Modelli probabilistici strutturati

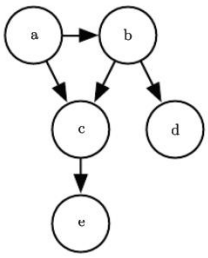
Il Machine Learning spesso coinvolge distribuzioni su un numero molto grande di variabili aleatorie. Usare una singola funzione per descrivere l'intera distribuzione congiunta sarebbe inefficiente. Quindi si divide una distribuzione in molti fattori che vengono moltiplicati insieme.

Vengono usati i grafi diretti o non diretti.
Ogni nodo del grafo corrisponde ad una variabile aleatoria, gli archi stabiliscono che la distribuzione di probabilità è in grado di rappresentare le interazioni dirette tra due variabili.

Modelli diretti: gli archi diretti rappresentano la fattorizzazione in distribuzioni condizionali.
Il modello contiene un fattore per ogni variabile aleatoria x_i, questo fattore consiste in una distribuzione condizionale su x_i dati i parenti (Pa) di x_i:

$$p(\mathbf{x}) = \prod_i p\left(\mathrm{x}_i \middle| Pa_{\mathcal{G}}(\mathrm{x}_i)\right)$$

Esempio:

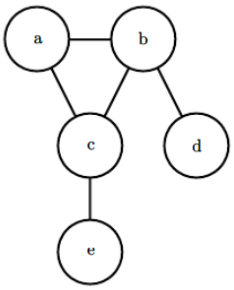


$$p(a,b,c,d,e) = p(a)p(b|a)p(c|a,b)p(d|b)p(e|c)$$

Modelli non diretti: gli archi non diretti rappresentato la fattorizzazione in n insieme di funzioni, queste funzioni di solito non sono distribuzioni di probabilità.
Cricca: insieme di nodi tutti connessi tra di loro.
Ad ogni cricca $\mathcal{C}^{(i)}$ è associato un fattore $\sigma^{(i)}(\mathcal{C}^{(i)})$ che rappresenta una funzione, non una distribuzione. L'output di ogni fattore deve esser non-negativo.
La probabilità di una configurazione di variabili aleatorie è proporzionale al prodotto di questi tutti fattori.
Z: costante che serve per normalizzare il prodotto, calcolata facendo la somma o l'integrale su tutti gli stati del prodotto dei fattori.

$$p(\mathbf{x}) = \frac{1}{Z}\prod_i \sigma^{(i)}(\mathcal{C}^{(i)})$$

Esempio:



$$p(a,b,c,d,e) = \frac{1}{Z} * \sigma^{(1)}(a,b,c) * \sigma^{(2)}(b,d) * \sigma^{(3)}(c,e)$$

4. Analisi numerica

L'analisi numerica viene utilizzata per creare algoritmi che risolvono problemi matematici con metodi che migliorano le stime della soluzione tramite un processo iterativo, piuttosto che derivare analiticamente una formula che fornisce un'espressione simbolica per la soluzione corretta.
Le operazioni includono:

- **Ottimizzazione**: trovare il valore che minimizza o massimizza una funzione. Di solito si minimizza f(x), per massimizzarla basta minimizzare -f(x).
- Risolvere sistemi di equazioni lineari.

Un problema è che si ha un numero limitato di bit per rappresentare infiniti numeri reali. L'approssimazione crea dei problemi, algoritmi che funzionano nella teoria possono fallire nella pratica.
Underflow: numeri vicino allo zero vengono arrotondati a zero.
Overflow: numeri molto grandi vengono approssimati ad +inf o -inf.
Condizionamento: velocità con cui una funzione cambia rispetto ai piccoli cambiamenti nei suoi input. Errori di arrotondamento negli input possono determinare grandi cambiamenti nell'output.

4.1. Funzioni e derivate

Funzione obiettivo (Objective function): funzione che si vuole minimizzare o massimizzare.
Funzione di costo (cost/loss/error function): funzione che si vuole solo minimizzare.
Il valore che minimizza o massimizza la funzione verrà scritto come x^*.

Derivata di f(x): $f'(x) = \frac{d}{dx}$ determina la pendenza della funzione al punto x.
Specifica come ridimensionare un piccolo cambiamento nell'input per ottenere lo stesso cambiamento nell'output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$
Può essere utilizzata per ridurre f(x) muovendo x in piccoli passi di segno opposto alla derivata: $f\left(x - \epsilon \text{sign}\left(f'(x)\right)\right) < f(x)$

Punto stazionario: dove $f'(x) = 0$, quindi non si hanno informazioni in quale direzione muoversi.
Minimo locale: punto dove la funzione è minore di tutti i sui vicini.
Massimo locale: punto dove la funzione è maggiore di tutti i sui vicini.
Punto di sella: punto che non è ne minimo ne massimo.
Minimo globale: punto dove la funzione è minima in assoluto.
Massimo globale: punto dove la funzione è massima in assoluto.

Derivata seconda: $f''(x) = \frac{d^2}{dx^2}$ derivata della derivata. Dice come la derivata cambia quando si cambia l'input. Serve per misurare la curvatura.

4.2. Discesa del gradiente

Gradient descent: tecnica per minimizzare una funzione che ha più input: $f: \mathbb{R}^n \to \mathbb{R}$

Derivata parziale: $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ misura come f cambia quando solo x_i aumenta al punto **x**.

Gradiente di f: vettore contenente tutte le derivate parziali $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. L'i-esimo elemento è la derivata parziale di f rispetto x_i.

Punto critico: ogni elemento del gradiente = 0. È quello che vogliamo trovare.

Derivata direzionale in direzione **u** (vettore unitario): $\boldsymbol{u}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ è la pendenza della funzione in direzione **u**.

Si minimizza f muovendosi nella direzione negativa gradiente $\boldsymbol{u} = -\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$, partendo da un punto **x** causale.

Il nuovo punto $\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$

Learning rate ϵ: scalare positivo che determina la grandezza del passo. Scelto come:

- una semplice costante piccola
- oppure, valutare $f\left(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})\right)$ per diversi valori di ϵ e scegliere quello che la minimizza di più.

Converge quando $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$

Algoritmo:

Si parte da un punto **x** causale.
while($\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \neq 0$){
  Calcolo learning rate ϵ se necessario)
  nuovo punto $\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$
}

4.3. Matrice Jacobiana ed Hessiana

Matrice Jacobiana J(f)(**x**): $J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$

Matrice di derivate parziali di una funzione cha ha più input e più output $f: \mathbb{R}^m \to \mathbb{R}^n$.

In una funzione quadratica (molte funzioni possono essere approssimate a tale):

- Se la derivata 2° = 0: non c'è curvatura, è una linea retta. Il suo valore può essere ricavato usando solo la derivata 1°:
  - Se il gradiente = 1: si può fare un passo ϵ lungo il gradiente negato e la funzione diminuirà di ϵ.
- Se la derivata 2° < 0: la funzione curva verso il basso, quindi diminuirà più di ϵ.
- Se la derivata 2° > 0: la funzione curva verso l'alto, quindi diminuirà meno di ϵ.

Matrice Hessiana H(f)(**x**): $\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x})$

Matrice di derivate seconde parziali di una funzione cha ha più input e più output $f: \mathbb{R}^m \to \mathbb{R}^n$.
È simmetrica, quindi può essere decomposta in autovettori e autovalori.

Derivata seconda direzionale in direzione **d**: $\boldsymbol{d}^T \boldsymbol{H} \boldsymbol{d}$
Se **d** è un autovettore di **H** la derivata seconda in quella direzione corrisponde all'autovalore.
Per le altre direzioni, la derivata seconda è una media pesata degli autovalori.
L'autovalore massimo/minimo determina la derivata seconda massima/minima.

4.4. Capire se il punto critico trovato è un minimo locale

In 1 dimensione:

- Se f'=0 e f''>0: è un minimo locale.
- Se f'=0 e f''<0: è un massimo locale.
- Se f''=0: non si può dire nulla.

In più dimensioni quando $\boldsymbol{\nabla_x} f(\boldsymbol{x}) = 0$ si esaminano gli autovalori e gli autovettori.

- Se tutti gli autovalori sono positivi: è un minimo locale.
- Se tutti gli autovalori sono negativi: è un massimo locale.
- Se almeno uno autovalore è positivo e almeno uno è negativo: è massimo locale su una sezione e minimo locale su un'altra.
- Se tutti gli autovalori non-nulli hanno lo stesso segno, ma almeno uno è nullo: non si può dire nulla.

Metodo di newton: $\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \boldsymbol{H}(f)\left(\boldsymbol{x}^{(0)}\right)^{-1} \boldsymbol{\nabla_x} f\left(\boldsymbol{x}^{(0)}\right) \; con \; \boldsymbol{x}^{(0)} \; punto \; casuale$
Restituisce il valore minimo di una funzione:

- Immediatamente se f è quadratica definita positiva.
- Ripetendolo più volte se f non è quadratica ma può essere approssimata a tale.

Algoritmi del primo ordine: usano solo il gradiente.
Algoritmi del secondo ordine: usano anche la matrice Hessiana.

4.5. Ottimizzazione con vincoli

Quando si vuole trovare il minimo o il massimo in un sottoinsieme $\mathbb{S}$ invece che su tutti i possibili valori di **x**. Si utilizza l'approccio di **Karush–Kuhn–Tucker** (KKT).
$\mathbb{S}$ viene descritto in forma di equazioni $g^{(i)}$ e disequazioni $h^{(j)}$.
$\mathbb{S} = \left\{\boldsymbol{x} \middle| \forall i, g^{(i)}(\boldsymbol{x}) = 0 \; and \; \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\right\}$
Funzione Langragiana: $L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x})$
Valore minimo: $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq \mathbf{0}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$

5. Basi di Machine Learning

Il Machine Learning è una forma di statistica applicata con maggiore enfasi sull'uso dei computer per stimare statisticamente funzioni complicate.
I due approcci alla statistica più importanti sono gli estimatori frequentisti e la inferenza Bayesiana.
La maggior parte degli algoritmi di DeepLearning si basa algoritmi di ottimizzazione chiamati Stochastic gradient descent.

5.1. Algoritmi di apprendimento

Sono algoritmi in grado di apprendere dai dati.
Si dice che un programma apprende dall'esperienza E in relazione ad alcune classi di compiti (task) T e alla misura di prestazione P, se le sue prestazioni dei task in T, misurate da P, migliorano con l'esperienza E.

5.1.1. Task

Task: sono solitamente descritti in termini di come il sistema di machine Learning dovrebbe elaborare un esempio.
Esempio: è una raccolta di caratteristiche che sono state misurate quantitativamente da alcuni oggetti o eventi che vogliamo che il sistema di machine learning elabori.
Tipi di Task più comuni:

- **Classificazione:** al computer è richiesto di specificare il quale di k categoria un input appartiene. Di solito è una funzione $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$ oppure l'output della funzione è una distribuzione di probabilità lungo le classi.
- **Classificazione con input mancanti:** l'algoritmo impara un insieme di funzioni. Ogni funzione corrisponde alla classificazione di x con un sottoinsieme degli input mancanti.
- **Regressione:** predire un valore numerico dato un input. $f: \mathbb{R}^n \rightarrow \mathbb{R}$. Simile alla classificazione tranne che per il formato dell'output.
- **Trascrizione:** osservare una rappresentazione relativamente non strutturata di alcuni tipi di dati e trascriverli in una forma testuale discreta. Esempio: da voce a testo, da immagine a testo…
- **Traduzione:** tradurre da una lingua ad un'altra.
- **Output strutturato:** l'output è un vettore (o una struttura dati) con relazioni importanti tra gli elementi. Esempio: tradurre un testo in un albero in cui i nodi descrivono la struttura grammaticale in verbi, nomi, aggettivi…
- **Rilevamento di anomalie.** Esempio: modellando le abitudini di utilizzo della carta di credito di un utente, la banca può accorgersi se è stato commesso un furto.
- **Campionamento e sintesi:** generare nuovi esempi che sono simili a quelli di allenamento. Esempio: generare degli ambienti in modo automatico nei videogames. Generare della voce da del testo scritto.
- **Assegnare dei valori mancanti**: dato un nuovo esempio $\boldsymbol{x} \in \mathbb{R}^n$ ma con alcun x_i mancanti. L'algoritmo deve fornire una predizione di valori mancanti.
- **Riduzione del rumore:** dato in input un esempio corrotto $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ l'algoritmo deve predire l'esempio pulito $\boldsymbol{x} \in \mathbb{R}^n$, o più in generale predire la distribuzione di probabilità condizionata $p(\boldsymbol{x}|\tilde{\boldsymbol{x}})$
- **Stima della densità di probabilità:** l'algoritmo impara una funzione $p_{\text{model}}: \mathbb{R}^n \rightarrow \mathbb{R}$ dove $p_{\text{model}}(\boldsymbol{x})$ è una densità di probabilità o densità discreta.

La maggior parte delle attività sopra descritte richiede che l'algoritmo di apprendimento catturi almeno implicitamente la struttura della distribuzione di probabilità. La stima della densità ci consente di acquisire esplicitamente tale distribuzione.

5.1.2. Prestazione ed esperienza

La misura di prestazione P di solito è specifica al task T.
Precisione ed **error rate**: misure di prestazione per classificazione, classificazione con input mancanti e trascrizione.
Probabilità logaritmica media: misura di prestazione usata per la stima di densità.

Datapoint: singolo esempio.
Dataset: collezione di molti esempi.
Matrice di Design: descrive un dataset, contiene esempi nelle righe, le colonne descrivono le features.

Gli algoritmi possono essere suddivisi in:

- **Non supervisionati:** Imparano proprietà utili della struttura del dataset come pattern o schemi.
  Coinvolge l'osservazione di diversi esempi di un vettore aleatorio **x**, e il tentativo di apprendere implicitamente o esplicitamente la distribuzione di probabilità p(**x**), o alcune proprietà interessanti di quella distribuzione
- **Supervisionati**: ogni esempio ha associato un'etichetta.
  Coinvolge l'osservazione di diversi esempi di un vettore aleatorio **x** e un vettore o valore associato **y**, e imparare a predire **y** da **x**, solitamente stimando p(**y|x**).
- **Con rinforzo:** l'algoritmo apprende esplorando l'ambiente e ricevendo ricompense nel caso di azioni positive.

5.1.3. Regressione Lineare

Semplice algoritmo di Machine Learning che risolve un problema di regressione.
Sistema che prende in input un vettore $\boldsymbol{x} \in \mathbb{R}^n$ e predice come valore di uscita uno scalare $y \in \mathbb{R}$, ovvero l'output è una funzione lineare dell'input.

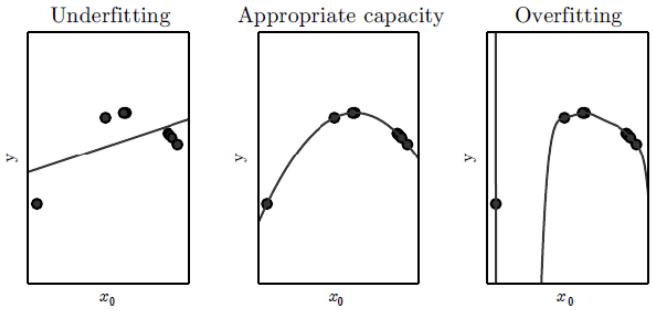
$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} \; con \; \boldsymbol{w} \in \mathbb{R}^n \; vettore \; di \; parametri$$

La soluzione è $\boldsymbol{w} = \left(\boldsymbol{X}^{(train)T}\boldsymbol{X}^{(train)}\right)^{-1}\boldsymbol{X}^{(train)T}\boldsymbol{y}^{(train)}$ con X e y matrici di design del dataset di input.
A volte viene utilizzato $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b \; con \; b \; scalare \; (bias)$

5.2. Capacità, overfitting e DataSet

5.2.1. Underfitting, Overfitting e Capacità

Generalizzazione: capacità di performare bene su input precedentemente non osservati.
Underfitting: quando il modello non è capace di ottenere un valore basso di training error.
Overfitting: quando il modello la differenza tra training erro e test error è troppo grande. Cioè non generalizza ma impara a memoria i dati.
Capacità: abilità di adattarsi a una grande varietà di funzioni. Modelli con bassa capacità possono avere problemi ad adattarsi al training set. Modelli con alta capacità possono avere problemi di overfit.
Spazio delle ipotesi: insieme di funzioni che l'algoritmo può selezionare come possibili soluzioni. Serve per controllare la capacità.



Capacità di rappresentazione: il modello specifica quale famiglia di funzioni può scegliere l'algoritmo di apprendimento quando varia i parametri al fine di ridurre un obiettivo di allenamento.
Capacità effettiva: può essere inferiore alla capacità di rappresentazione perché l'algoritmo di ottimizzazione è imperfetto.
Dimensione di Vapnik-Chervonenkis (VC): misura la capacità di un classificatore binario. Definita come il più grande valore possibile di m per cui esiste un training set di m differenti punti $\boldsymbol{x}$ che il classificatore può etichettare arbitrariamente.
Modelli parametrici: apprendono una funzione descritta da un vettore di parametri le cui dimensioni sono limitate e fissate prima di osservare qualsiasi dato.
Modelli non parametrici: non hanno questo limite. Si possono creare prendendo un algoritmo parametrico e metterlo in un altro che incrementa il numero di parametri quando necessario.

Teorema No Free Lunch: nessun algoritmo di Machine Learning è universalmente migliore degli altri. Vuol dire che l'obiettivo non è cercare un algoritmo universale, ma capire quali tipi di distribuzioni sono rilevanti per il "mondo reale" che un agente AI sperimenta e quali tipi di algoritmi di machine Learning funzionano bene sui dati della distribuzione che ci interessa.

Regolarizzazione: modifica apportata ad un algoritmo di apprendimento in modo da ridurre il suo errore di generalizzazione ma non il suo errore di addestramento.

5.2.2. Iperparametri e DataSet

Iperparametri: impostazioni che si possono usare per controllare il comportamento dell'algoritmo di apprendimento.

Validation set: utilizzato per stimare l'errore di generalizzazione durante o dopo l'allenamento, consentendo di aggiornare gli iperparametri di conseguenza.

Il dataset viene diviso in training set e testing set:

- Random sampling: i due dataset vengono presi a caso
- K-fold cross validation: Suddivido il dataset in K sottoinsiemi, alleno il sistema su K-1 sottoinsiemi e lo testo sul sottoinsieme restante. Itero K volte e prendo la mediana dei risultati.

Il Training set viene ulteriormente diviso in Training set effettivo e validation set.

Procedura di apprendimento:

- **Training**: al sistema viene fornito un insieme di coppie input-output, che adatta il proprio stato interno per classificare correttamente le coppie fornite. (Training set)
  - Durante l'allenamento, testo periodicamente l'accuratezza sul Validation set.
- **Testing**: Al sistema viene fornito un diverso insieme di input (di cui si conosce l'output). Si valuta l'accuratezza del sistema, in termini di percentuale di risposte corrette. (Test Set)

5.3. Stimatori

5.3.1. Proprietà

Stimatore $\widehat{\boldsymbol{\theta}}$ sul parametro $\boldsymbol{\theta}$**:** è il tentativo di fornire la singola previsione "migliore" di una quantità di interesse. La quantità di interesse può essere un singolo parametro, un vettore di parametri o una funzione.

Definito come una variabile aleatoria $\widehat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, .., \boldsymbol{x}^{(m)})$

Proprietà:

- **Bias**: $bias(\widehat{\boldsymbol{\theta}}_m) = \mathbb{E}(\widehat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$ Misura la deviazione attesa dal valore reale della funzione o parametro.
- **Varianza**: $Var(\hat{\theta})$ Misura la deviazione dal valore stimato atteso che un particolare campione di dati è portato a causare.

Se si deve scegliere tra due stimatori, uno con più bias o uno con più varianza si usa il cross-validation oppure **l'errore quadratico medio**: $MSE = Bias(\hat{\theta}_m)^2 + Var(\hat{\theta}_m)$ che misura la deviazione complessiva prevista tra lo stimatore e il valore reale del parametro θ. Bisogna cercare di avere MSE basso.

Consistenza: $plim_{m\to\infty}\hat{\theta}_m = 0$ Assicura che il pregiudizio indotto dallo stimatore diminuisca con il crescere del training set.

5.3.2. Stima della massima verosimiglianza

Principio dal quale possiamo ricavare funzioni specifiche che sono buoni stimatori per diversi modelli.
Cerca di minimizzare la dissomiglianza tra $\hat{p}_{data}$ di distribuzione empirici definiti dal training set e la distribuzione del modello. Cioè cerca di far combaciare la distribuzione del modello alla distribuzione empirica $\hat{p}_{data}$.

Maximum Likelihood: $\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$

Con $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ famiglia parametrica di distribuzioni di probabilità sullo stesso spazio indicizzato da $\boldsymbol{\theta}$**.** Cioè $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ mappa ogni configurazione $\boldsymbol{x}$ in un numero reale stimando la vera probabilità $p_{data}(\boldsymbol{x})$.

Proprietà:

- con l'aumentare degli esempi del training set, la stima del parametro converge con il valore reale del parametro.
- Ha una efficienza statistica, ovvero uno stimatore consistente può ottenere un errore di generalizzazione inferiore per un numero fisso di campioni m da altri stimatori, o ha bisogno di meno esempi per ottenere un livello fisso di errore di generalizzazione.

Conditional Log-Likelihood Estimator: $\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta})$

Stima la probabilità condizionata $P(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta})$ per predire $\boldsymbol{y}$ dato $\boldsymbol{x}$**,** molto utilizzato nel supervised learning.

5.3.3. Statistica Bayesiana

L'approccio è considerare tutti i possibili valori di $\boldsymbol{\theta}$ quando si effettua una previsione.
Il dataset è osservato direttamente, quindi non è casuale.
Il vero parametro $\boldsymbol{\theta}$ è sconosciuto o incerto e quindi rappresentato da una variabile aleatoria, utilizzando la **distribuzione di probabilità a priori** $p(\boldsymbol{\theta})$
Generalmente i metodi bayesiani generalizzano molto meglio quando sono disponibili dati di formazione limitati, ma in genere soffrono di costi computazionali elevati quando il numero di esempi di formazione è ampio

Stima Massimo a Posteriori (MAP): $\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{x}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$

Sceglie il punto di probabilità massima posteriore.

5.4. Apprendimento Supervisionato

algoritmi che imparano ad associare degli input con degli output, dato un training set di esempi di input x e output y.

La maggior parte degli algoritmi supervisionati si basano sulla stima di una distribuzione di probabilità $p(y|\boldsymbol{x})$ tramite una stima Maximum Likelihood per trovare il miglior parametro $\boldsymbol{\theta}$ per una famiglia parametrica $p(y|\boldsymbol{x};\boldsymbol{\theta})$.
Si una la regressione logistica $p(y=1|\boldsymbol{x};\boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$

Support Vector Machines: Forniscono in output solo la classe a cui l'input appartiene.
kernel trick: $\boldsymbol{w^T x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^T \boldsymbol{x}^{(i)}$ consiste nell'osservare che molti algoritmi di apprendimento automatico possono essere scritti esclusivamente in termini di prodotti scalari tra gli esempi. Permette di sostituire $\boldsymbol{x}$ dall'output proveniente da una feature function $\boldsymbol{\phi(x)}$ e dal prodotto scalare con una funzione kernel $k\left(\boldsymbol{x}, \boldsymbol{x}^{(i)}\right) = \boldsymbol{\phi(x)} \cdot \boldsymbol{\phi}\left(\boldsymbol{x}^{(i)}\right)$
Questo ci permette di imparare modelli non lineari e k è più efficiente nella pratica.
Possono essere fatte predizioni usando la funzione $f(\boldsymbol{x}) = b + \sum_i \alpha_i k\left(\boldsymbol{x}, \boldsymbol{x}^{(i)}\right)$
Gaussian kernel: $k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2\boldsymbol{I})$ $dove\ \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ è la densità normale

k -nearest neighbors: è una famiglia di tecniche che possono essere utilizzate per la classificazione o la regressione. Vengono pensati come se non avessero parametri. Possono raggiungere elevate capacità.

Alberi di decisione: ogni nodo è associato con una regione dello spazio di input, i nodi interni suddividono questa regione in sotto-regioni.
Lo spazio è quindi suddiviso in regioni non sovrapposte, con una corrispondenza uno-a-uno tra i nodi foglia e le regioni di input. Ogni nodo foglia di solito mappa ogni punto nella sua regione di input sullo stesso output.

5.5. Apprendimento Non Supervisionato

Il classico compito di apprendimento non supervisionato è trovare la rappresentazione "migliore" dei dati ovvero va alla ricerca di una rappresentazione che conservi quante più informazioni possibili su $\boldsymbol{x}$.
Modi di mantenere la rappresentazione semplice:

- rappresentazioni **dimensionali inferiori**: tentano di comprimere quante più informazioni possibili su x in una rappresentazione più piccola.
- rappresentazioni **sparse**: incorporano il set di dati in una rappresentazione le cui voci sono per lo più zero per la maggior parte degli input. Questo richiede un aumento della dimensionalità.
- rappresentazioni **indipendenti**: tentano di districare le fonti di variazione alla base della distribuzione dei dati in modo tale che le dimensioni della rappresentazione siano statisticamente indipendenti.

Analisi delle componenti principali (PCA): è un algoritmo non supervisionato che impara una rappresentazione dei dati, e quindi fornisce un modo per comprimere i dati.
Basato su due criteri: minore dimensionalità e la indipendenza tra gli elementi.
Impara trasformazione lineare ortogonale dei dati che proietta un input $\boldsymbol{x}$ in una rappresentazione $\boldsymbol{z}$.

k-means Clustering: divide il training set in k diversi gruppi (cluster) di esempi vicini l'uno all'altro.
Se $\boldsymbol{x}$ appartiene al cluster i, allora $h_i = 1$ e tutte gli altri elementi del vettore $\boldsymbol{h}$ sono 0.
Inizializza k centroidi $\{\boldsymbol{\mu}^{(1)}, \dots, \boldsymbol{\mu}^{(k)}\}$ a valori diversi, poi alternandosi dei due step finché converge:

- step 1: ogni esempio del training set è assegnato al cluster i, dove i è l'indice del centroide $\boldsymbol{\mu}^{(i)}$ più vicino.
- Step 2: ogni centroide $\boldsymbol{\mu}^{(i)}$ è aggiornato alla media di tutti gli esempi $\boldsymbol{x}^{(j)}$ assegnati al cluster i.

5.6. Gradient descent stocastico

Quasi tutto il DeepLearning è alimentato da questo algoritmo molto importante.

Un problema ricorrente nell'apprendimento automatico è che sono necessari ampi set di allenamento per una buona generalizzazione, ma i set di allenamento di grandi dimensioni sono anche più dispendiosi dal punto di vista computazionale.

Su ogni passo dell'algoritmo, possiamo campionare un minibatch di esempi $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m')}\}$.
La dimensione de minibatch è relativamente piccola, da 1 a qualche centinaio di esempi.
m' è fisso anche quando m aumenta.

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{\boldsymbol{m'}} \boldsymbol{L}(\boldsymbol{x}^{(i)}, \boldsymbol{j}^{(i)}, \boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g} \; con \; \epsilon \; learning \; rate$$

5.7. Costruzione degli algoritmi

Gli algoritmi di DeepLearning combinano una specifica di un set di dati, una funzione di costo, una procedura di ottimizzazione e un modello.
In alcuni casi, la funzione di costo può essere una funzione che non possiamo effettivamente valutare, per ragioni computazionali. In questi casi, possiamo ancora approssimativamente minimizzarlo usando l'ottimizzazione numerica iterativa.

Il DeepLearning è stato progettato perché generalizzare a nuovi esempi diventa esponenzialmente più difficile quando si lavora con dati ad alta dimensione e i meccanismi utilizzati per ottenere la generalizzazione nell'apprendimento automatico tradizionale non sono sufficienti per imparare funzioni complicate in spazi ad alta dimensione.
Per d dimensioni e v valori da distinguere lungo ciascun asse, occorrono $O(v^d)$ regioni ed esempi.

Se ci sono pochi dati non c'è molta differenza con altri algoritmi di ML

Parte2 - Deep Networks: Pratiche Moderne

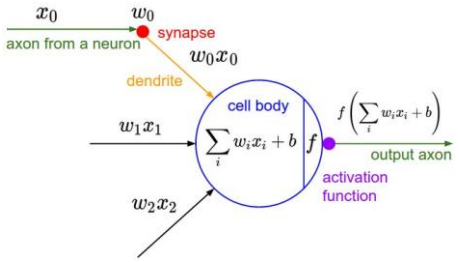
6. Deep Feedforward Networks

6.1. Basics

Un **singolo neurone** non è altro che una regressione Lineare a cui viene applicata una Activation Function, cioè un sistema che prende in input un vettore $\boldsymbol{x} \in \mathbb{R}^n$ e predice come valore di uscita uno scalare $y \in \mathbb{R}$.

$$\hat{y} = g(\boldsymbol{w}^T \boldsymbol{x} + b)$$

con b scalare (bias)
$\boldsymbol{w} \in \mathbb{R}^n$ vettore di parametri
g Activation function



Singolo esempio in input $\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$

m esempi messi in una matrice $\boldsymbol{X} = \begin{bmatrix} \vdots & & \vdots \\ \boldsymbol{x^{(1)}} & \cdots & \boldsymbol{x^{(m)}} \\ \vdots & & \vdots \end{bmatrix}$

Output di un singolo esempio $y \in \{0,1\}$
m esempi in output $\boldsymbol{Y}[y^{(1)} \quad \cdots \quad y^{(m)}] \in \mathbb{R}^{1 \times m}$

Loss Function: applicata ad un solo esempio. $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$
Cost function: è la media delle Loss Function sull'intero dataset. $J(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$

Gradient Descent: algoritmo che serve per imparare i parametri $\boldsymbol{w}$ e b sul dataset, tali che minimizzano $J(\boldsymbol{w}, b)$

repeat{

$\boldsymbol{w} = \boldsymbol{w} - \alpha \frac{\partial J(\boldsymbol{w},b)}{\partial \boldsymbol{w}}$ nel codice scritto come dw

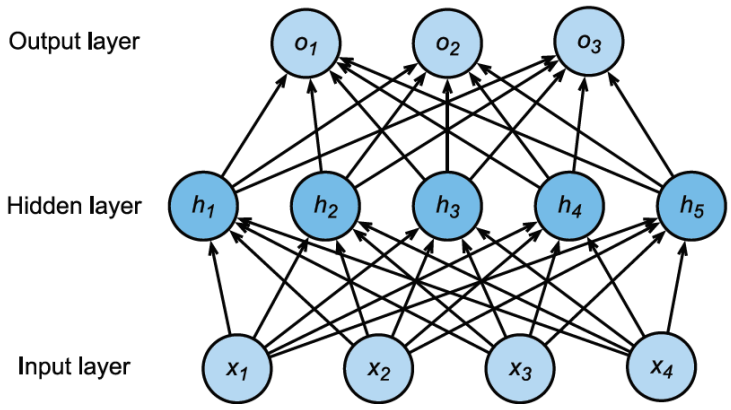
$b = b - \alpha \frac{\partial J(\boldsymbol{w},b)}{\partial b}$ nel codice scritto come db

}

Activation functions: Vengono utilizzate funzioni non lineari perché la composizione di funzioni lineari è sempre una funzione lineare, e quindi anche con 1000 layer non si riuscirebbe mai imparare una funzione non lineare.

	$\boldsymbol{g(z)}$	$\frac{\boldsymbol{d}}{\boldsymbol{dz}}\boldsymbol{g(z)}$	
Sigmoid	$\frac{1}{1+e^{-z}}$	$g(z)\big(1-g(z)\big)$	Da non usare mai, se non nell'output layer della binary classification. È lenta da imparare perché la derivata è quasi 0.
Tanh	$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$1 - \tanh^2 z$	$\in [-1,1]$
ReLu	$\max(0, z)$	$\begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$	Trasforma i negativi in 0.
Leaky ReLu	$\max(\alpha * z, z)$	$\begin{cases} \alpha, & z < 0 \\ 1, & z \geq 0 \end{cases}$	$con\ \alpha \cong 0.001$

6.2. Deep Networks

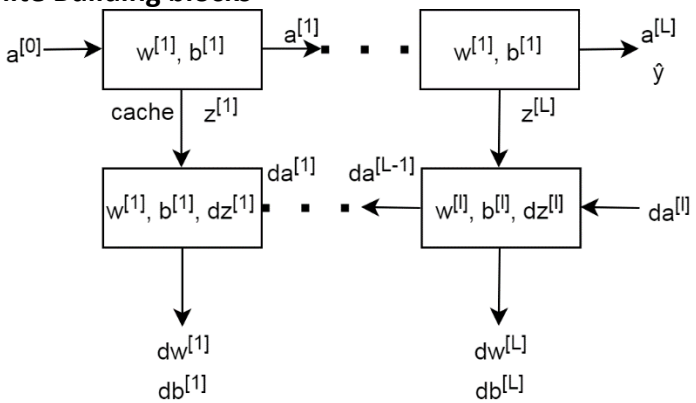


Sia $n^{[l]} = n° \; di \; neuroni \; del \; layer \; l$

con 1 solo esempio	**Dimensioni**
$b^{[l]} = bias \; livello \; l$	$(n^{[l]}, 1)$
$w^{[l]} = pesi \; di \; z^{[l]}$	$(n^{[l]}, n^{[l-1]})$
$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$	$(n^{[l]}, 1)$
$a^{[l]} = g^{[l]}(z^{[l]}) = activation \; layer \; l$	
$a^{[0]} = x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$	$(n^{[l]}, 1)$

Con m esempi	**Dimensioni**	
$b^{[l]} = bias \; livello \; l$	$(n^{[l]}, m)$	$= db^{[l]}$
$W^{[l]} = pesi \; di \; Z^{[l]} = rimane \; uguale$	$(n^{[l]}, n^{[l-1]})$	$= dW^{[l]}$
$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} = \begin{bmatrix} \vdots & & \vdots \\ \boldsymbol{z}^{[l](1)} & \cdots & \boldsymbol{z}^{[l](m)} \\ \vdots & & \vdots \end{bmatrix}$	$(n^{[l]}, m)$	$= dZ^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]}) = activation \; layer \; l = \begin{bmatrix} \vdots & & \vdots \\ \boldsymbol{a}^{[l](1)} & \cdots & \boldsymbol{a}^{[l](m)} \\ \vdots & & \vdots \end{bmatrix}$	$(n^{[l]}, m)$	$= dA^{[l]}$
$A^{[0]} = \boldsymbol{X} = \begin{bmatrix} \vdots & & \vdots \\ \boldsymbol{x}^{(1)} & \cdots & \boldsymbol{x}^{(m)} \\ \vdots & & \vdots \end{bmatrix}$		

Rappresentazione tramite Building blocks



6.2.1. Softmax Regression

Classificazione su più classi, usata sull'ultimo layer, generalizza la Logistic Regression su più di 2 classi, se $c = 2$ equivale ad essa.

- **Activation function:** $a = \frac{e^z}{\sum_{i=1}^{c}(e^z)_i}$
- **Loss function:** $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{c} y_j \log \hat{y}_j$
- **Cost function:** $J(\boldsymbol{w}^{[1]}, b^{[1]}, \dots, \boldsymbol{w}^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
- **Gradient descent:** $dz^{[L]} = \hat{y} - y$

6.2.2. Steps for building a Neural Network

- Define the model structure (such as number of input features)
- Initialize the model's parameters (w,b)
- Loop:
  - forward propagation
  - Compute cost
  - backward propagation (Calculate current gradient)
  - Update parameters (gradient descent)
- Use the learned (w,b) to predict the labels for a given set of examples

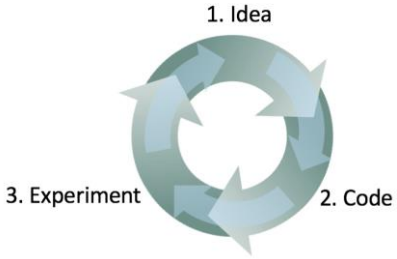
You often build 1-3 separately and integrate them into one function we call model().

6.2.3. Esempio DeepNeuralNetwork con Keras

```
import tensorflow as tf

mnist=tf.keras.datasets.mnist #Dataset of picture's numbers 28x28

(x_train, y_train), (x_test, y_test) = mnist.load_data()

#Normalizzazione, mette i dati tra 0 e 1 per semplificare l'allenamento
x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)

# Modello
model=tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten()) #Trasforma la matrice 28x28 in un
vettore
model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu)) #n° neuroni,
activation
model.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))


model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

#Training
model.fit(x_train, y_train, epochs=3)
```

7.Improving Deep Neural Networks

7.1. Setting up a ML application



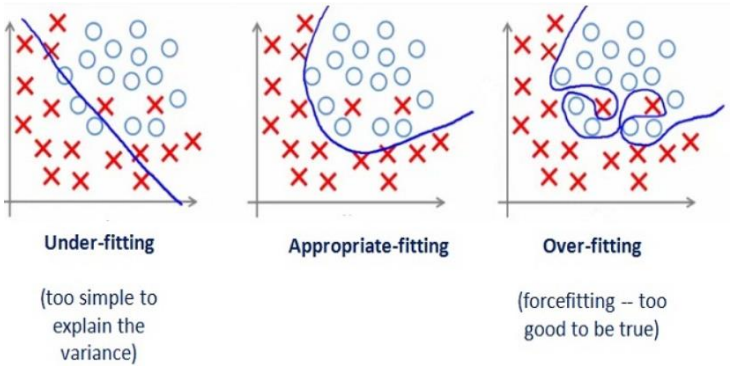
Suddivisone del dataset:

Training	Development	Test

- Dataset piccolo: 70(Training)/30(Test) oppure 60/20/20
- Big data: 98/1/1 oppure 99.5/0.4/0.1

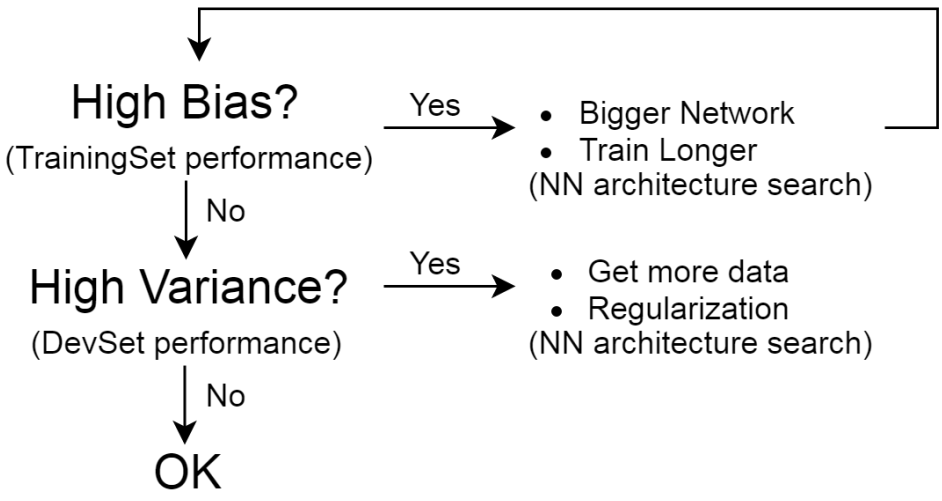
Bias e Varianza:

- **Varianza:** trainingError - DevError
- **Bias:** trainingError - optimalError



Esempio: (con Optimal error = 0%)

Training Error	1%	15%	15%	0.5%
Dev Error	11%	16%	30%	1%
	High Variance	High Bias	High Bias High Variance	Low Bias Low Variance



7.2. Initialization

A well-chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

Zero initialization: $W = 0, b = 0$, The performance is really bad, and the cost does not really decrease, and the algorithm performs no better than random guessing.
In general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and you might as well be training a neural network with $n^{[l]} = 1$ for every layer, and the network is no more powerful than a linear classifier such as logistic regression.
The weights $W^{[l]}$ should be initialized randomly to break symmetry.
It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.

Random initialization: each neuron can then proceed to learn a different function of its inputs.
Initializing weights to very large random values does not work well.

He initialization: Works well for networks with ReLU activations.
multiplying np.random.randn(..,..) * np.sqrt(2./layers_dims[l-1])

7.3. Regularization

To prevent overfitting. Introduces the **hyperparameter λ.**
L2 regularization and Dropout are two very effective regularization techniques.

7.3.1. L2 Regularization

L₂ Regularization: $J(\boldsymbol{w}^{[1]}, b^{[1]}, \dots, \boldsymbol{w}^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|\boldsymbol{w}^{[l]}\|_2^2$

$$\text{con } \|\boldsymbol{w}^{[l]}\|_2^2 = \boldsymbol{w}^{[l]T}\boldsymbol{w}^{[l]}$$

In gradient descent:

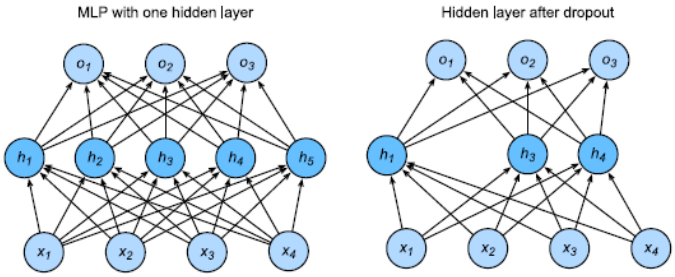
- $dw^{[l]} = (backprop) + \frac{\lambda}{m}w^{[l]}$
- $w^{[l]} = \left(1 - \alpha\frac{\lambda}{m}\right)w^{[l]} - \alpha(backprop), \quad \text{con } \alpha \text{ Learning Rate}$

L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.
L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.
Implications:

- The cost computation: A regularization term is added to the cost
- The backpropagation function: There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"): Weights are pushed to smaller values

7.3.2. Drop-out Regularization

Usato quando si ha un training set piccolo. Aspetto negativo è che la cost function non è più ben definita. Ogni layer ha associata una probabilità di eliminare i suoi nodi, se ne eliminano e si ripete per ogni esempio. Quindi ogni esempio ha associata una rete diversa, che viene allenata.
Non si usa drop out sul test set.
The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.



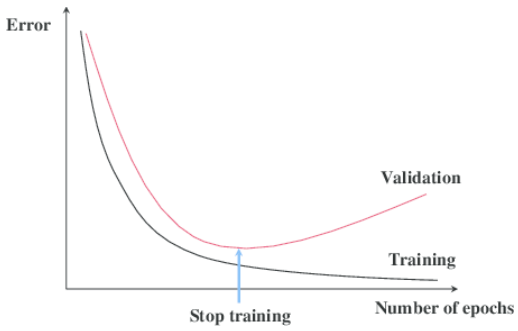
Inverted dropout: (la più usata)

- Forward propagation:
  - $D^{[l]} = np.rnd.rnd(A^{[l]}.shape[0], A^{[l]}.shape[1]) < keepProb$
    Vettore di 0 e 1, è una maschera da applicare ad $A^{[l]}$
  - $A^{[l]} = A^{[l]} * D^{[l]}$ (element-wise product)
  - $A^{[l]} /= keepProb$
- Backward propagation:
  - $dA^{[l]} = dA^{[l]} * D^{[l]}$
  - $dA^{[l]} /= keepProb$

7.3.3. Other regularization methods

Data augmentation: aggiungere dati modificati, ad esempio immagini distorte, mirrorate.

Early Stopping: durante l'allenamento si tiene traccia del training error o della Cost Function J.
In agiunta si tiene traccia anche del DevSet error, che quando sale si ferma l'allenamento.
Aspetto negativo: non si ottimizza la Cost Function J



7.4. Optimizing training

7.4.1. Normalizing input

Utile quando gli input sono su scale diverse. (Es. $x_1 \in [1,1000], x_2 \in [0,1]$)
Inoltre permette di utilizzare un learning rate α più alto.
Media: $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
Varianza: $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ** 2$ (element-wise power)
Inizializzare $x = x - \mu$ e $x = \frac{x}{\sigma^2}$

7.4.2. Vanishing & Exploding gradients

Nelle reti molto profonde le derivate sempre più in profondità possono diventare molto grandi o molto piccole, il che rende il training difficile.
Questo lo si risolve inizializzando i pesi:

- ReLu: $w^{[l]} = random * \sqrt{\frac{2}{n^{[l-1]}}}$
- tanh: $w^{[l]} = random * \sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization)

7.4.3. Gradient Checking

…
Non utilizzare durante il training, solo per il debug
Se fallisce, controllare le componenti db, dw
Se si sta utilizzando la regolarizzazione, ricordarsi del termine $+\frac{\lambda}{2m}\sum_{l=1}^{L}\left\|\boldsymbol{w}^{[l]}\right\|_2^2$
Non funziona col drop-out
Farlo a variabili inizializzate casualmente e anche dopo il training

7.5. Optimization Algorithms

Il DeepLearning funziona molto bene sui Big Data, ma allenare si grandi dataset è molto lento, quindi servono degli algoritmi di ottimizzazione.
Advanced optimization methods that can speed up learning and perhaps even get you to a better final value for the cost function. Having a good optimization algorithm can be the difference between waiting days vs. just a few hours to get a good result.

7.5.1. Gradient Descent

Simple optimization method in machine learning.

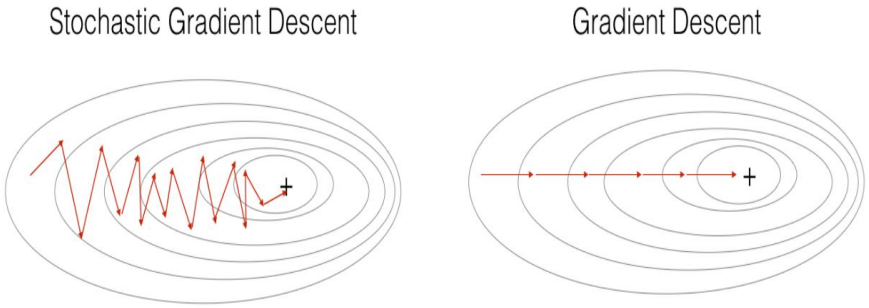
For(l=1 to L){

- $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$
- $b^{[l]} = b^{[l]} - \alpha b^{[l]}$

}

7.5.2. Stochastic Gradient Descent (SGD)

Is equivalent to mini-batch gradient descent where each mini-batch has just 1 example. The update rule that you have just implemented does not change. What changes is that you would be computing gradients on just one training example at a time, rather than on the whole training set.
When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly.



Implementing SGD requires 3 for-loops in total:

- Over the number of iterations
- Over the m training examples
- Over the layers (to update all parameters, from $(W^{[1]}, b^{[1]})$ to $(W^{[L]}, b^{[L]})$)

In practice, you'll often get faster results if you do not use neither the whole training set, nor only one training example, to perform each update.

7.5.3. Minibatch Gradient Descent

$X = \left[x^{(1)} \dots x^{(m)}\right]$ dim: (n_X, m)

$Y = \left[y^{(1)} \dots y^{(m)}\right]$ dim: $(1, m)$

Mini-batch: sottoinsieme del dataset

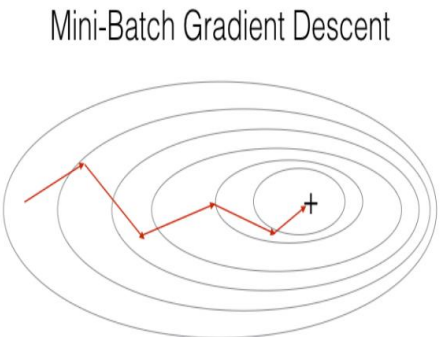
$$X = \underbrace{\left[x^{(1)} \dots x^{(1000)}\right.}_{x^{\{1\}}} \underbrace{\left|x^{(1001)} \dots x^{(2000)}\right|}_{x^{\{2\}}} \dots \underbrace{\left|\dots x^{(m)}\right]}_{x^{\{m/\dots\}}}$$

$(n_x, 1000)$

(i) Esempio

$[l]$ Layer

$\{t\}$ Indice minibatch



Batch Gradient Descent: si esegue sull'intero dataset

Mini-Batch Gradient Descent: si esegue su un singolo sottoinsieme $x^{\{t\}}$ del dataset.

For(t=1 to 5000){

- Forward propagation su $x^{\{t\}}$
- Cost function $J^{\{t\}}$
- Backward propagation

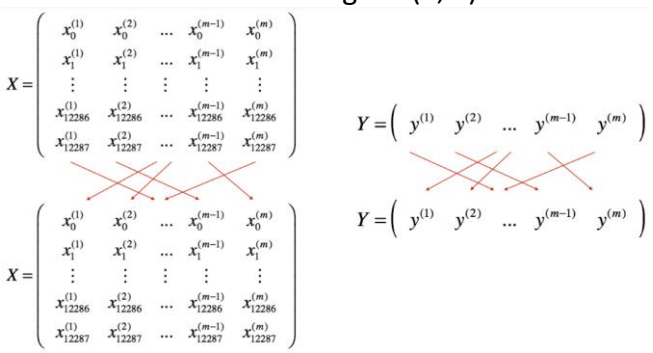
}

Dimensione Mini Batch: è un iperparametro. Assicurarsi che la dimensione mini-batch ci stia nella memoria della CPU/GPU.

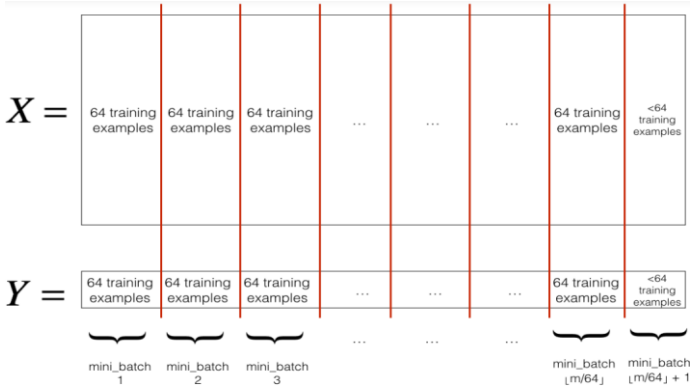
- TrainingSet piccolo: usare Batch, non mini-batch
- Dimensioni tipiche: potenze di 2. Più comuni: $2^6, 2^7, 2^8, 2^9$

Build mini-batches from the training set (X, Y) in two steps:

- **Shuffle**: Create a shuffled version of the training set (X, Y).

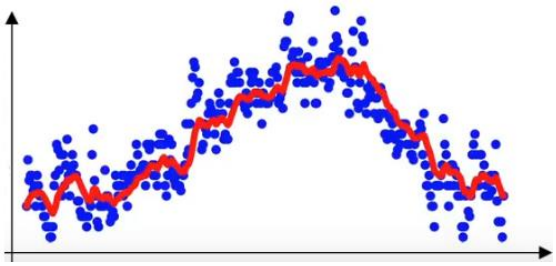


- **Partition**: Partition the shuffled (X, Y) into mini-batches of size mini_batch_size. The last mini batch might be smaller, but you don't need to worry about this.



7.5.4. Exponentially Weighted Averages

Si prende una media dei valori vicini del dataset. β iperparametro.



$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

Implementazione:

$V_\theta = 0$
Repeat{
$V_\theta \leftarrow \beta V_\theta + (1-\beta)\theta_t$
}

Correzione del bias: problema del $V_\theta = 0$. Si prende $\frac{V_t}{1-\beta^t}$

7.5.5. Gradient descent with momentum

Velocizza il gradient descent. Introduce l'iperparametro β, di solito $= 0.9$.

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations.

Takes into account the past gradients to smooth out the update.

$v_{dw} = 0, v_{db} = 0$
Nell'iterazione t:

- Calcola dw, db sul mini-batch corrente
- $v_{dW} = \beta v_{dW} + (1+\beta)dW$
- $v_{db} = \beta v_{db} + (1+\beta)db$
- $W = W - \alpha v_{dW}$
- $b = W - \alpha v_{db}$

If β=0, then this just becomes standard gradient descent without momentum

- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, β=0.9 is often a reasonable default.
- Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

7.5.6. RMSProp (Root Mean Square Propagation)

Velocizza anch'esso il gradient descent.

$S_{dW} = 0, S_{db} = 0$
Nell'iterazione t:

- Calcola dw, db sul mini-batch corrente
- $S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ (element wise power)
- $S_{db} = \beta S_{db} + (1-\beta) db^2$
- $W = W - \alpha \frac{d_W}{\sqrt{S_{dW}} + \varepsilon}$ $con\ \varepsilon = 10^{-8}$
- $b = b - \alpha \frac{d_b}{\sqrt{S_{db}} + \varepsilon}$

7.5.7. Adam

Composizione di Momentum e RMSProp.

$V_{dW} = 0, V_{db} = 0, S_{dW} = 0, S_{db} = 0$
Nell'iterazione t:

- Calcola dw, db sul mini-batch corrente
- $V_{dW} = \beta_1 V_{dW} + (1-\beta_1) dW$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ (Momentum)
- $S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2, S_{db} = \beta S_{db} + (1-\beta) db^2$ (RMSProp)
- $V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t}, V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$
- $S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}, S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$
- $W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon}$
- $b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$

Iperparametri:
α =da testare
$\beta_1 = 0.9$
$\beta_2 = 0.999$
$\varepsilon = 10^{-8}$

advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except α)

7.5.8. Learning rate decay

Diminuisce il Learning Rate α, la parte iniziale la fa velocemente, poi diventa più lento ma più preciso.

- $\alpha = \frac{1}{1 + DecayRate * epochNumeber} \alpha_0$

oppure

- $\alpha = 0.95^{epochNumber} * \alpha_0$

oppure

- $\alpha = \frac{k}{\sqrt{epochNumber}} \alpha_0$

7.5.9. Batch normalization

Velocizza il training.

Dati $z^{[l](1)}, \dots, z^{[l](m)}$ di un layer

$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)}$

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z^{[l](i)} - \mu\right)^2$

$z_{norm}^{[l](i)} = \frac{z^{[l](i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$

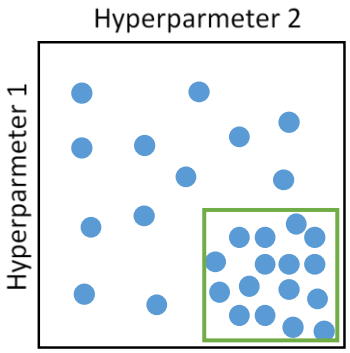
$\tilde{z}^{[l](i)} = \gamma z_{norm}^{[l](i)} + \beta \text{ con } \gamma = \sqrt{\sigma^2 + \varepsilon}, \beta = \mu$

$\beta\ e\ \gamma$ sono parametri da allenare, quindi nella backpropagation si calcolano le loro derivate.
Nel test time si usano $\beta\ e\ \gamma$ allenati.

7.6. Hyperparameter Tuning

Ordine di importanza:

- Learning rate α
- Momentum β
- # hidden units
- MiniBatchSize
- # Layers
- LearningRateDecay
- Adam $\beta_1, \beta_2, \varepsilon$

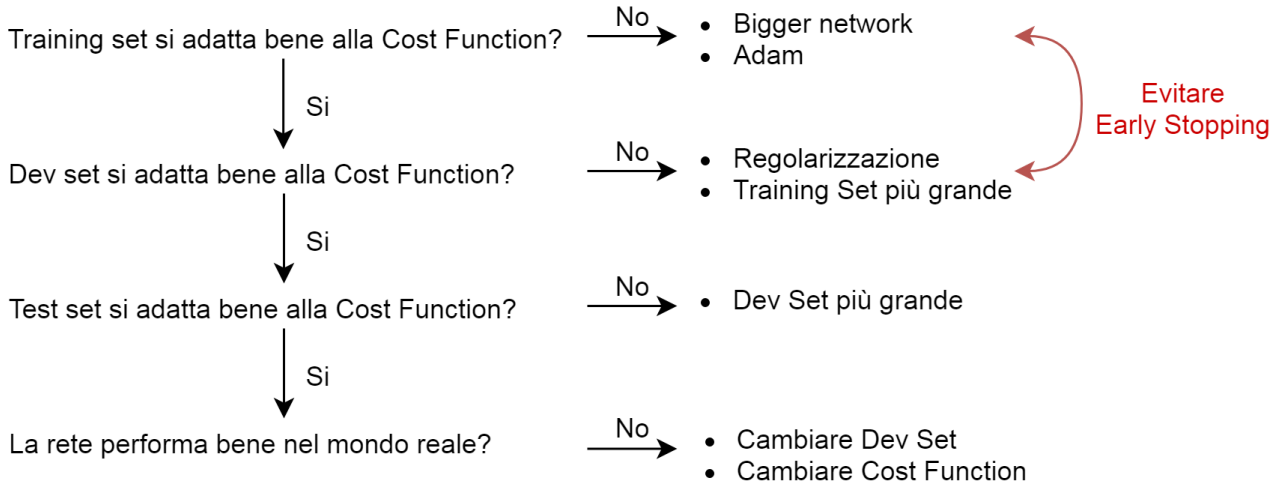


Scelta: Costruire una griglia e scegliere punti a caso.
Poi ci si può accorgere che una certa zona funziona meglio, allora ci si concentra su di essa.
I parametri si possono scegliere su scale diverse:

- Lineare: #Layera, #HiddenUnits
- Logaritmica:
  - Learning rate $\alpha = 0.0001, \dots, 1$
  - $\alpha = 10^r\ con\ r \in random(a, b), a = \log_{10} ValIniz\ , b = \log_{10} ValFin$
- $\beta = 0.9, \dots, 0.999$ si prende la logaritmica di $(1 - \beta)$

7.7. Structuring ML projects

7.7.1. ML Strategy



7.7.2. Setting up the goal

Utilizzare un singolo valore per valutare la performance: precisione o recall, error, …

F_1: media tra precisione e recall $= \frac{2}{\frac{1}{p}+\frac{1}{r}}$

Il **Dev Set e il Test Set** devono provenire dalla **stessa distribuzione** e vanno scelti in modo che riflettano i dati che ci si aspetta di incontrare nel mondo reale.

7.7.3. Comparing to human level performance

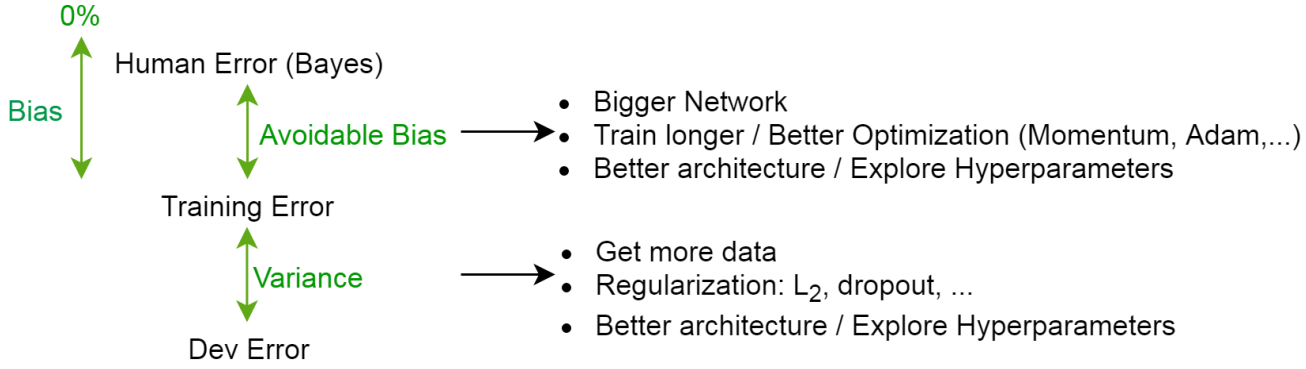
Spesso si comparano le prestazioni delle reti con quelle umane, perché quelle umane sono vicine all'errore ottimo.

Se si superano le performance umane non si riesce a capire se per migliorare bisogna lavorare sul bias o sulla varianza.

$$\boldsymbol{Avoidable\ bias} = Bayes\ Error - Training\ Error \approx \boldsymbol{Human\ Error - Training\ Error}$$

Se è basso non c'è bisogno di fare modifiche.

7.7.4. Improving performances diagram



7.7.5. Error analysis

Andare a vedere manualmente gli esempi classificati male nel DevSet, contare i falsi positivi e falsi negativi, per vedere cosa non va.

Correggere gli esempi etichettati male nel dataset:

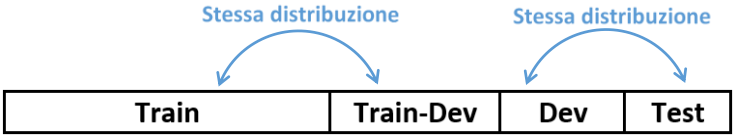
- nel training set: non serve perché gli algoritmi di deeplearning sono robusti agli errori casuali.
- Nel Dev/Test set: correggerli solo se migliorerebbe di molto le performances.

7.7.6. Dealing with Different distributions

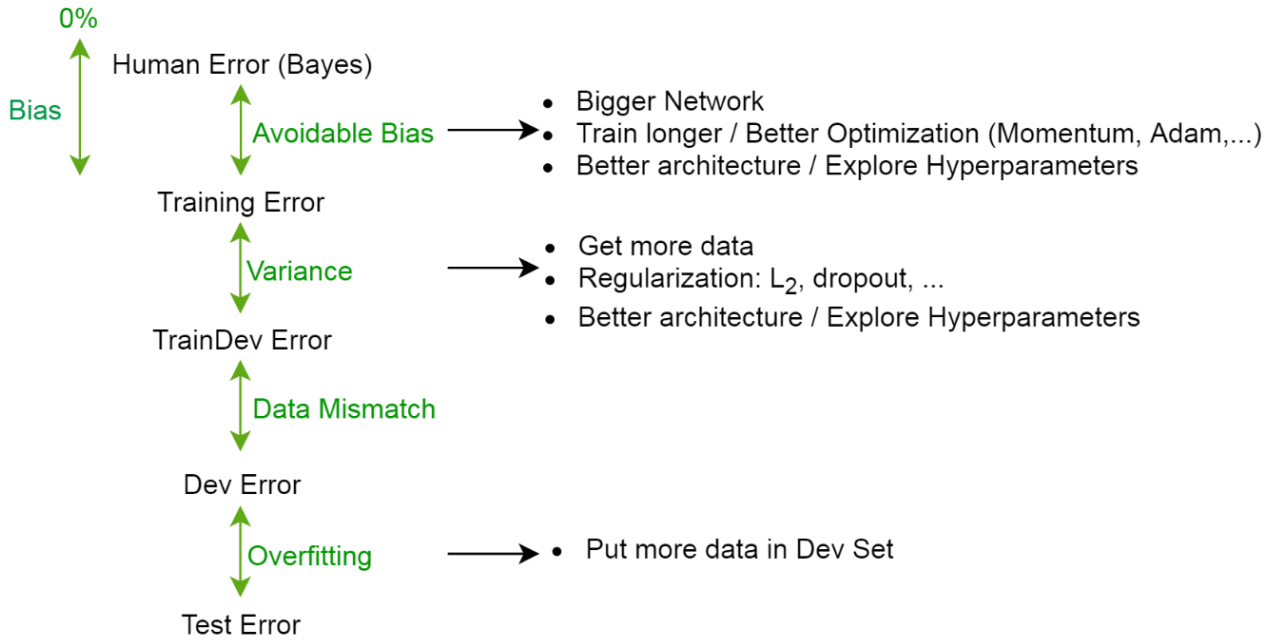
Se il Dev e Test set hanno distribuzione diversa dal training, bias e varianza funzionano in modo diverso.

Se$(DevErr - TrainErr > TrainErr - HumanError)$:

- Si divide il TrainSet in: Train e TrainDev
- Si allena su Train e si testa su TrainDev



- Se$(TrainErr - HumanErr\ is\ \mathrm{HIGH})$: è un problema di Avoidable Bias
- Se$(TrainDevErr - TrainErr\ is\ \mathrm{HIGH})$: è un problema di Varianza
- Se$(DevErr - TrainDevErr\ is\ \mathrm{HIGH})$: è un problema di Data Mismatch
- Se$(TestErr - DevErr\ is\ \mathrm{HIGH})$: è un problema di Overfitting.



7.7.7. Data Augmentation

Quando il dataset ha pochi dati, si può utilizzare qualche trucco per aumentarli.
Esempi:

- Mirroring
- Random cropping
- Less common (perhaps due to their complexity):
    - Rotation
    - Shearing (3D rotation)
    - Local warping
- Color shifting (R +20, G -20, B +20)

Per risparmiare spazio nella memoria, le distorsioni possono essere applicate durante il training prima che il dato venga inserito nella rete neurale.

7.7.8. Tips on Doing Well on Benchmarks / winning competitions

Ensemble: train several networks independently (3-15) and average their outputs.

- Can increase performance by 0.5 – 2%.
- Too little and too expensive.
- Used in competitions but not in production.

Multi-crop at test time: run classifier on multiple versions of the dataset (e.g. applying distortions to the dataset) and average the results.

7.8. Extended Learning

7.8.1. Transfer learning

Utilizzare una rete allenata per un task su un altro task.
Si rimpiazza l'ultimo layer con una nuova rete.
La rete vecchia viene utilizzata come una funzione e quindi non si allenano i suoi pesi.
Questo metodo si utilizza quando:

- i due task hanno lo stesso input (es immagini)
- si hanno molti più dati sul primo task che sul secondo.
- Features di basso livello possono essere utili per imparare il secondo task.

7.8.2. Multi-task learning

Si vuole che una rete faccia più cose. (Es. riconosca le automobili e anche i segnali stradali)
Questo metodo si utilizza quando:

- La quantità di dati a disposizione per ogni task è simile.
- Si può allenare una rete abbastanza grande da essere performante su tutti i task.

7.8.3. End-to-End Deep Learning

Avvolte un sistema che richiede una pipeline di processi, questi possono essere sostituiti da una rete neurale.
Es:

Audio ---> Features ---> Fonomi ---> Parole ---> Trascrizione
Sostituito da: Audio ---> Trascrizione

Questo metodo si utilizza quando:

- Pro:
  - Utilizza bene i dati
  - Minore bisogno di progettare componenti a mano
- Contro:
  - Servono molti dati
  - Esclude potenziali componenti progettate a mano

8. Convolutional Networks CNN

Molto simili alle normali reti neurali:

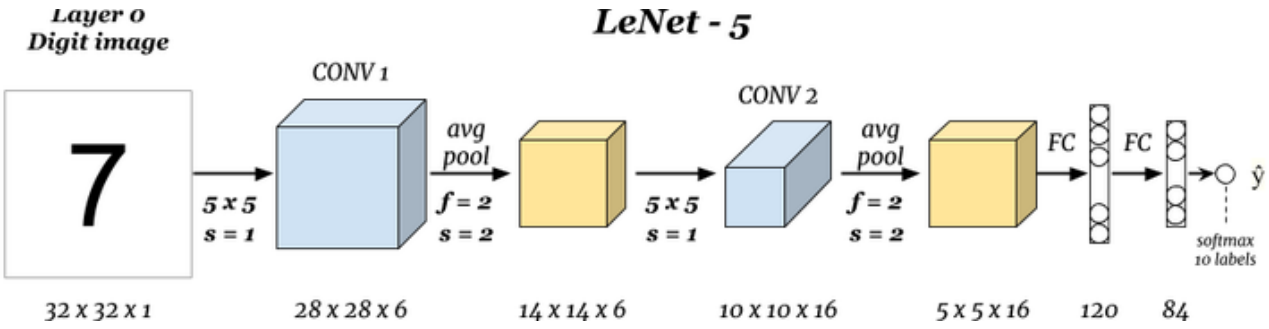
- Composte da neuroni che hanno pesi e bias addestrabili.
- Ogni neurone riceve alcuni input, esegue un prodotto a cui segue opzionalmente una non linearità.
- L'intera rete esprime una singola funzione di punteggio differenziabile: dai pixel dell'immagine grezza da un lato ai punteggi di classe dall'altro.
- Hanno una loss function (ad esempio SVM / Softmax) sull'ultimo livello (completamente connesso).

In aggiunta:

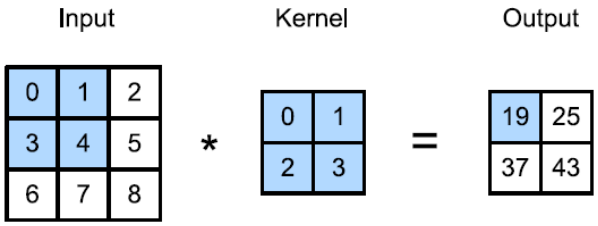
- Rendono la forward function più efficiente da implementare.
- Riducono notevolmente la quantità di parametri nella rete.
- Sparsity of connection: In ogni layer l'output dipende solo da pochi parametri
- Transaction invariance: immagine shiftata, risultato finale non cambia

Le reti neurali normali non scalano bene, ecco un esempio:

Si vuole creare un classificatore di immagini di 1000 classi.
Quindi in ingresso si ha un'immagine 1000x1000x3 (HxWxRGB) = 3milioni di neuroni e in uscita si hanno 1000 neuroni.
La matrice W dei pesi tra i soli due layer avrebbe 3 miliardi di parametri. Il costo computazionale non è accettabile.



Prodotto di Convoluzione (semplificato) 2D: ogni cella della matrice risultante è data dalla somma dei prodotti elemento per elemento sovrapponendo la matrice filtro alla matrice di ingresso + bias (1 solo bias per filtro).



$$out_{1,1} = \big((0 * 0) + (1 * 1) + (3 * 2) + (4 * 3)\big) + b = 19 + b$$

Data una matrice NxN e una più piccola FxF (filtro) la matrice risultante sarà N-F+1
I filtri e i bias sono valori che vengono allenati.

8.1. Fundamentals

8.1.1. Edge detection

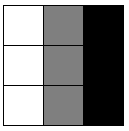
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

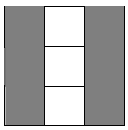
$*$

1	0	-1
1	0	-1
1	0	-1

$=$

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0





Filtri comuni: di solito 3x3, 5x5 o 7x7

1	0	-1
1	0	-1
1	0	-1

Verticale

1	1	1
0	0	0
-1	-1	-1

Orizzontale

1	0	-1
2	0	-2
1	0	-1

Sobel

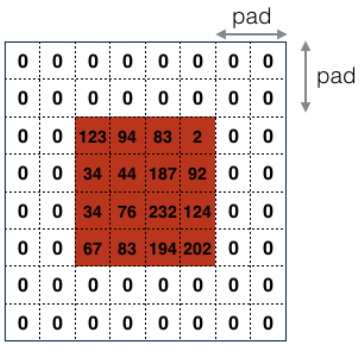
3	0	-3
10	0	-10
3	0	-3

Scharr

8.1.2. Padding

Con l'utilizzo di filtri l'immagine risultante diventa più piccola, se ci sono tanti step l'immagine può diventare piccolissima fino a perdere di significato.
Inoltre, nel prodotto di convoluzione, i pixel nei bordi vengono utilizzati di meno rispetto a quelli centrali.
Quindi **si aggiungono pixel ai bordi**.

- **Zero padding:** si aggiungono zeri
- **Valid padding:** non si aggiunge nulla, $p = 0$
- **Same padding**: dimensione input = dimensione output, $p = \frac{f-1}{2}$



8.1.3. Stride

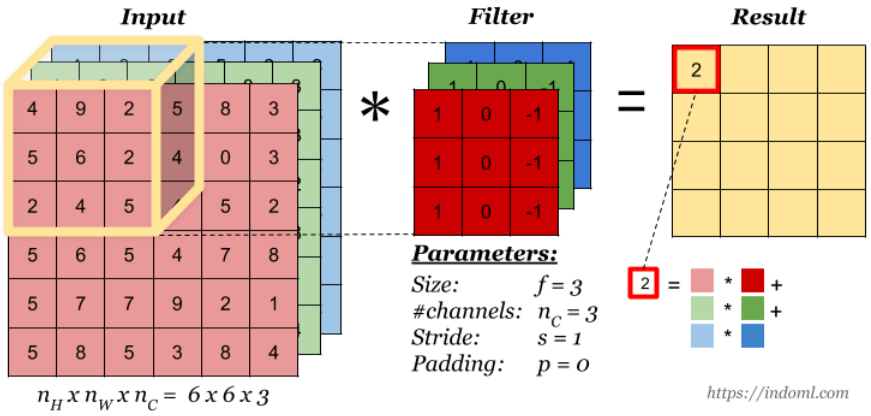
Invece di fare 1 passo se ne fanno di più. Non viene fatto il prodotto se il filtro va fuori dal bordo.

$$Dim\ Output = \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

8.1.4. Convolution over volume

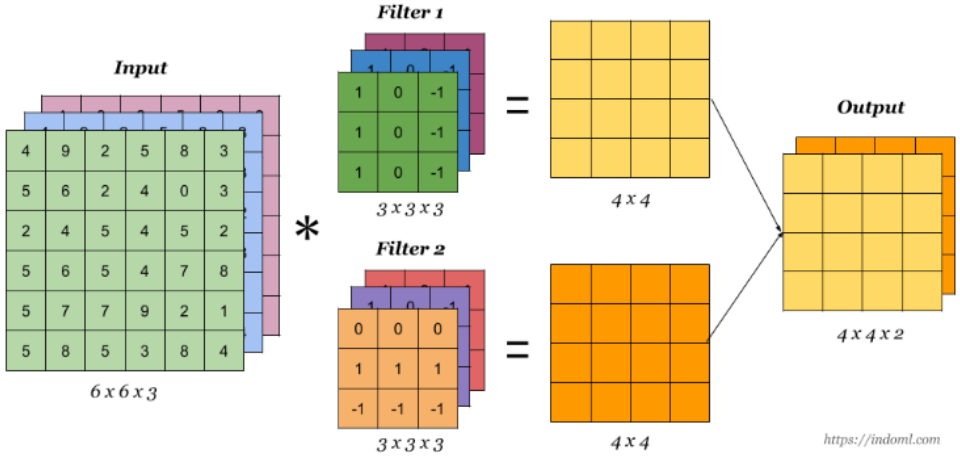
When the input has **more than one channels** (e.g. an RGB image), the filter should have matching number of channels. To calculate one output cell, perform convolution on each matching channel, then add the result together.

Multiple channel:

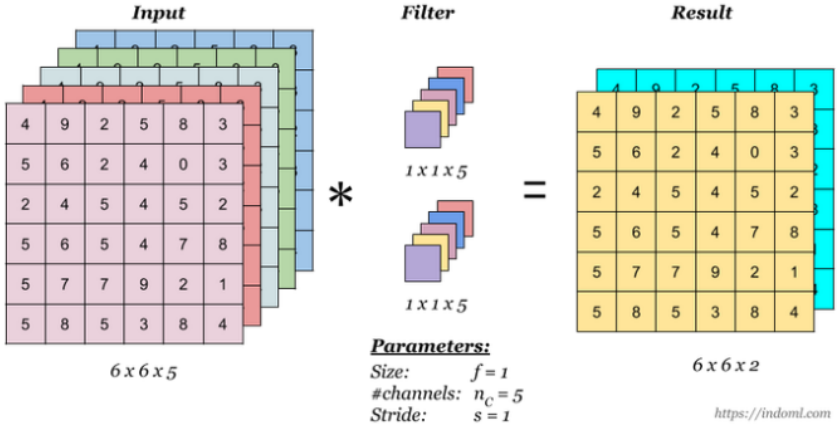


$$(n \times n \times n_c) * (f \times f \times n_c) = (n - f + 1) \times (n - f + 1) \times 1$$

Multiple filters: can be used in a convolution layer to detect multiple features. The output of the layer then will have the same number of channels as the number of filters in the layer.

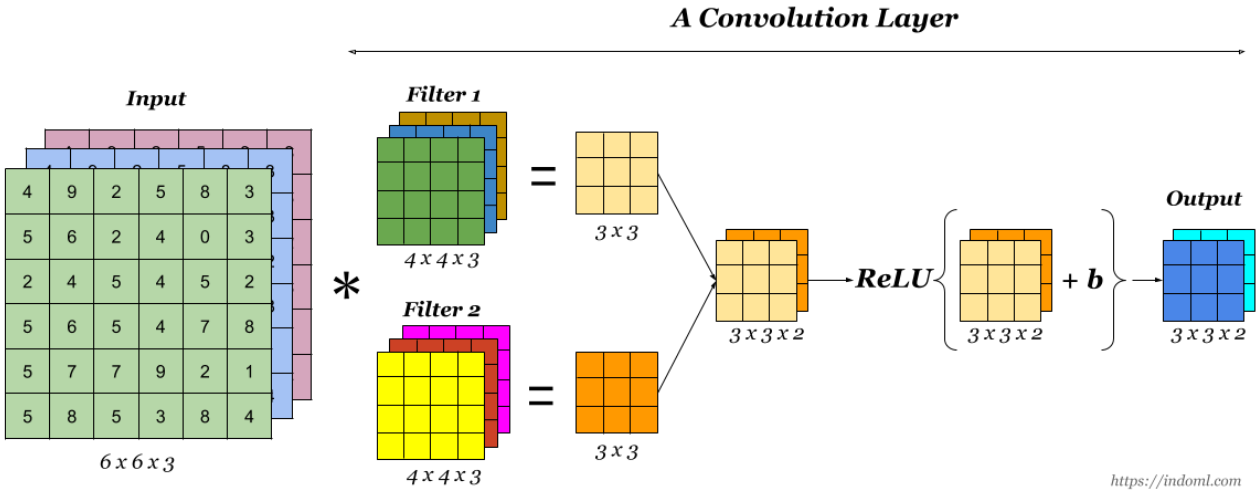


1 x 1 Convolution: The effect is to flatten or **"merge" channels together**, which can save computations later in the network.



8.1.5. One Convolution Layer

Finally to make up a convolution layer, a bias (ϵ R) is added and an activation function such as ReLU or tanh is applied. Bias is a vector with one parameter for each channel.
It doesn't matter how big the input is, the learnable parameters w & b depends only on the number of filters and their sizes.



$l = layer\ attuale$
$f^{[l]} = dimensioen\ filtro$
$p^{[l]} = padding$
$s^{[l]} = stride\ (passo)$
$n_c^{[l]} = n°\ filtri\ (n°\ canali)$
$Dim\ Input = n_h^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$
$Dim\ Output = n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

$$n_{h/w}^{[l]} = \left\lfloor \frac{n_{h/w}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

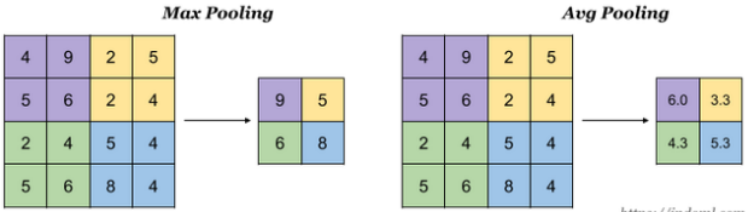
Attivazione $a^{[l]} \rightarrow n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$
$A^{[l]} \rightarrow m \times n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$
Pesi $W^{[l]} \rightarrow f^{[l]} \times f^{[l]} \times n_c^{[l]}$
Bias $b^{[l]} \rightarrow \left(1,1,1,n_c^{[l]}\right)$

8.1.6. Pooling Layer

Pooling layer is used to reduce the size of the representations and to speed up calculations, as well as to make some of the features it detects a bit more robust.
Non ha parametri da imparare quindi spesso non viene considerato come un layer.
Utilizza gli Iperparametri:

- $f: dimensione$
- $s: stride$
- Type: max o average
- $p: padding$ (poco utilizzato)

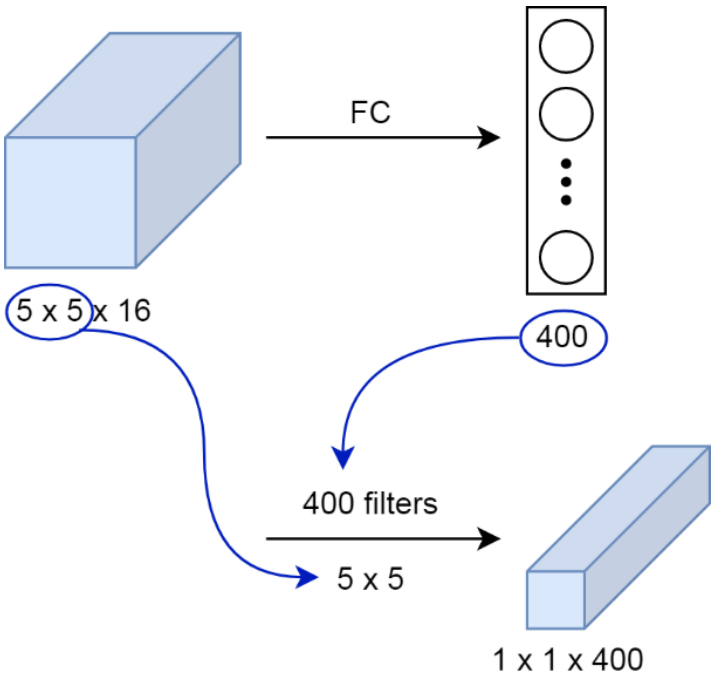
Max pooling: come la convoluzione ma prende il massimo della regione anzi che sommare.
Average pooling: prende la media.



Applicato su più canali: riduce altezza e larghezza ma mantiene invariato il numero di canali n_c.

$$Dim\ output = \left\lfloor \frac{n_h - f}{s} + 1 \right\rfloor \times \ldots \times n_c$$

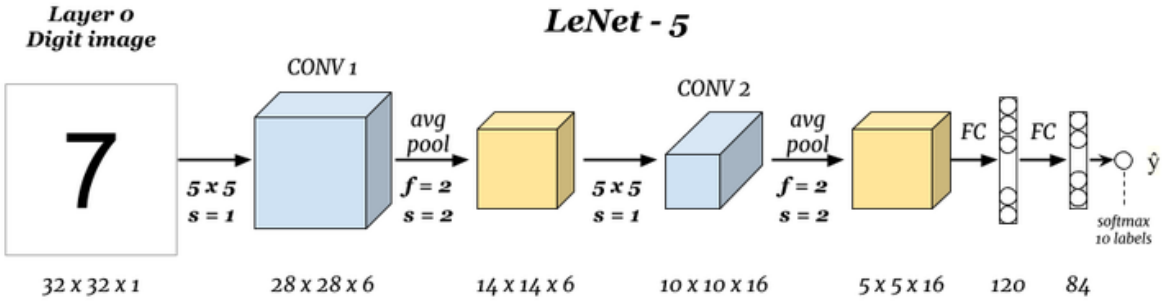
8.1.7. Transform Fully Connected layers in Convolutional layers



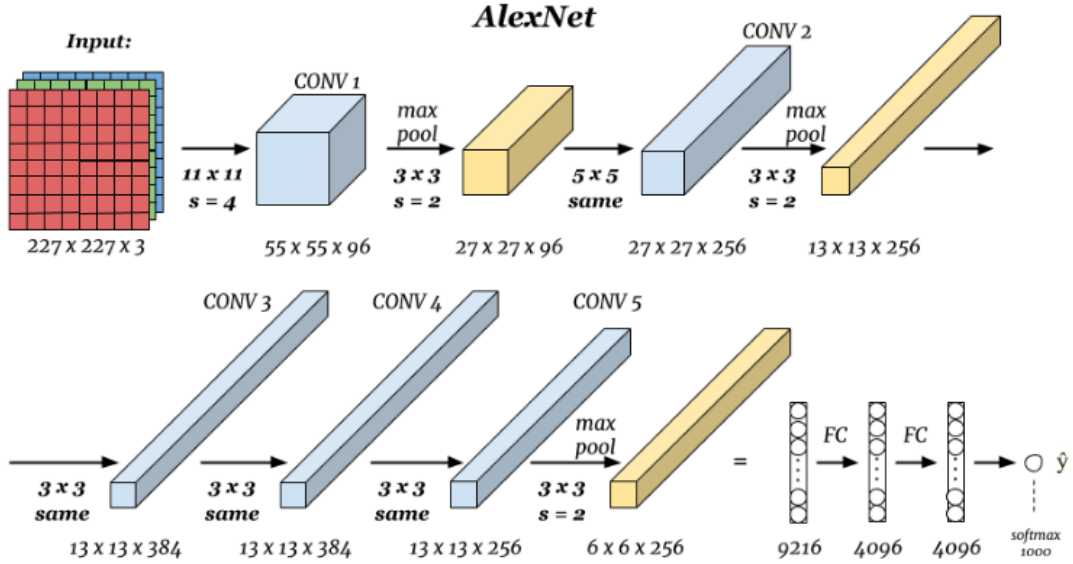
8.2. Common Architectures

8.2.1. Classic Networks

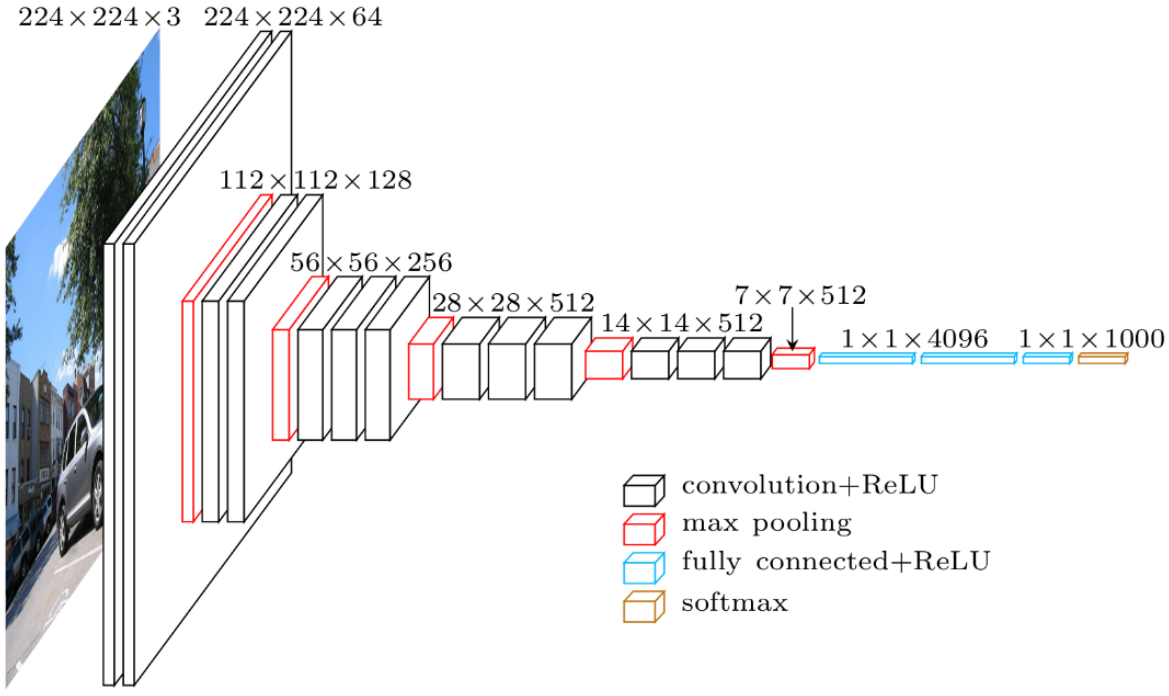
LeNet-5: riconosce numeri scritti a mano in scala di grigio. Utilizza 60k parametri.



AlexNet: riconosce immagini in RGB. 60 milioni di parametri, utilizza ReLu.



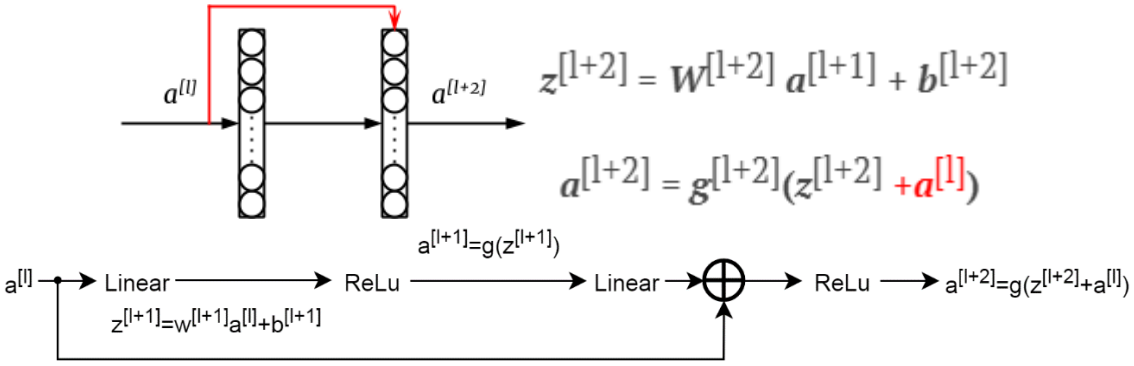
VGG-16: 138 milioni di parametri. CONV: 3x3, s=1, same. MaxPool:2x2, s=2.



8.2.2. Residual Networks (ResNet)

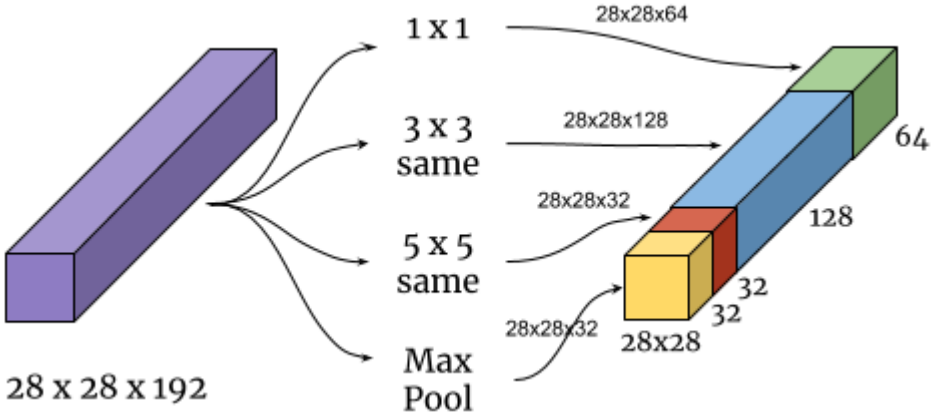
The problem with deeper neural networks is that they are harder to train and once the number of layers reach certain number, the training error starts to raise again. Deep networks are also harder to train due to exploding and vanishing gradients problem.

ResNet (Residual Network), solves these problems by implementing skip connection where output from one layer is fed to layer deeper in the network.



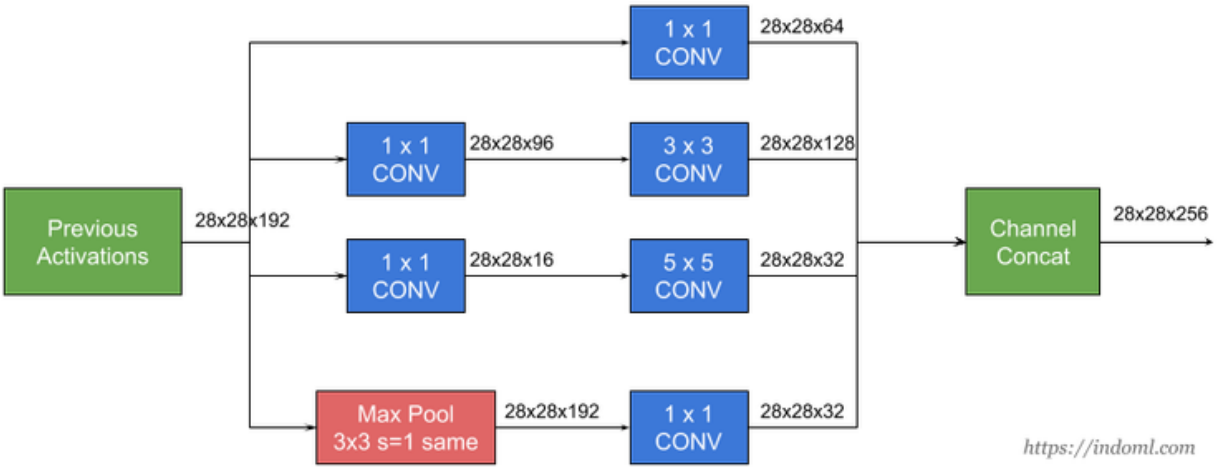
8.2.3. Inception network

The motivation of the inception network is, rather than requiring us to pick the filter size manually, let the network decide what is best to put in a layer. We give it choices and hopefully it will pick up what is best to use in that layer.



Problema: costo computazionale alto.

Soluzione: inserire convoluzione 1x1 prima delle convoluzioni e dopo i pool. Riduce del 90% il costo.



8.3. Detection algorithms

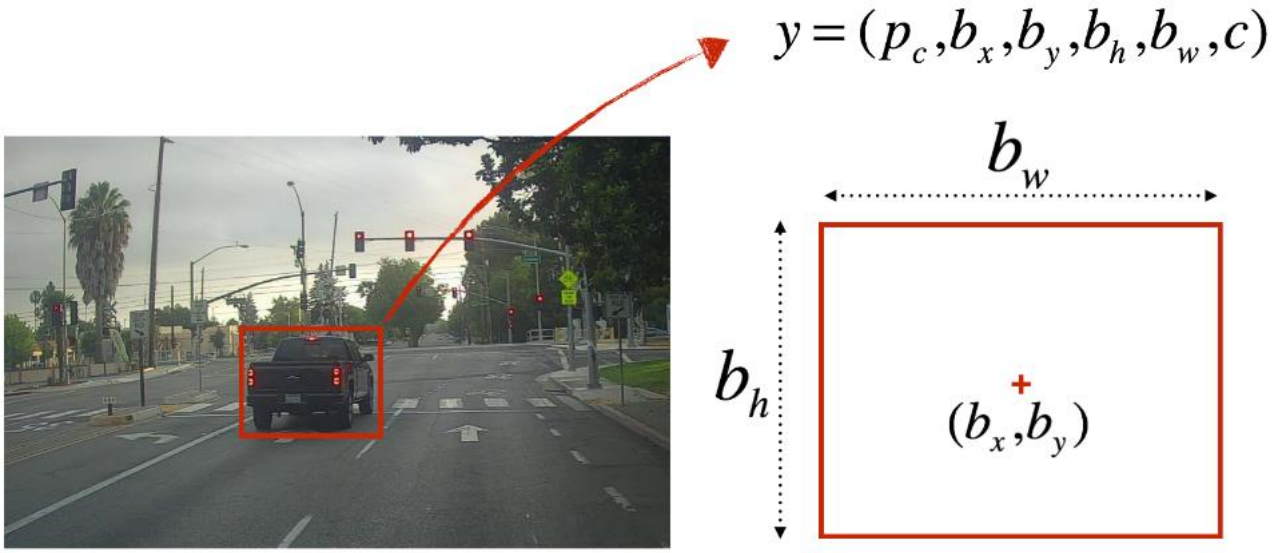
Classification with localization: 1 single object in the picture.
Detection: more objects.
Landmark detection: e.g. parts of the body or face

8.3.1. Object localization

The output is composed by:

- probability that one of the objects recognizable is in the picture.
- the coordinates of the object
- the classes of the object (e.g. car, motorbike, pedestrian, …)

If you have 80 classes that you want YOLO to recognize, you can represent the class label c either as an integer from 1 to 80, or as an 80-dimensional vector (with 80 numbers) one component of which is 1 and the rest of which are 0.



$p_c = 1$: confidence of an object being present in the bounding box

$c = 3$: class of the object being detected (here 3 for “car”)

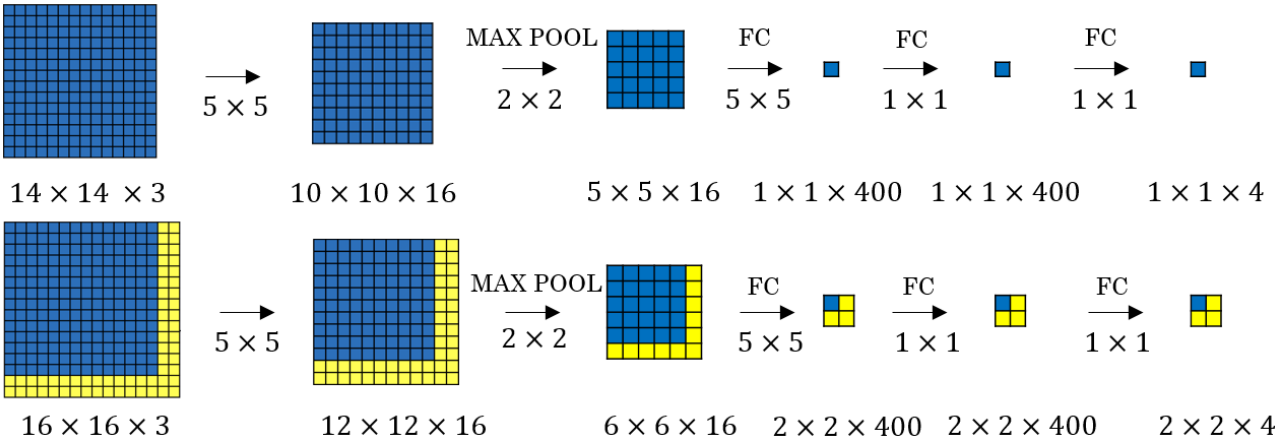
8.3.2. Sliding window detection

Algoritmo di base: (inefficiente)

- prendere un piccolo rettangolo (o quadrato) dell'immagine
- si controlla con una CNN se contiene l'oggetto cercato
- lo si muove per tutta l'immagine e si controlla ancora …
- poi si ricomincia da capo con un rettangolo più grande.

Problema: costo computazionale alto. Molti dei calcoli fatti sono condivisi tra i rettangoli (due rettangoli che hanno un solo pixel di distanza sono praticamente uguali).

Soluzione:
Invece di fare una convoluzione per ogni rettangolo dell'immagine, si fa una singola **convoluzione sull'intera immagine** applicando i filtri come se fosse un'immagine più piccola.



La prima è una normale CNN che restituisce un output del tipo $y = (P_c, c_1, c_2, c_3)$
La seconda, ovvero quella che si utilizza, nell'output si nota che il segmento in alto a sinistra rappresenta il risultato della CNN applicata in alto a sinistra, il segmento in alto a destra rappresenta il risultato della CNN applicata in alto a destra, e così via …
In questo modo si ottiene lo stesso risultato di una sliding window.

Altri problemi:
L'oggetto può non venir catturato da nessuno dei quadrati e quindi non rilevato.
Perché i quadrati sono troppo piccoli, troppo grandi, l'oggetto non ha una forma quadrata ma più rettangolare, lo step tra un quadrato ed un altro è troppo lungo e l'oggetto si trova in mezzo, …
Inoltre non restituisce in output le coordinate dell'oggetto ma vanno ricavate successivamente.
Meglio usare l'algoritmo YOLO.

8.3.3. YOLO (You Only Look Once)

Si divide l'immagine in celle (e.g. 19x19) e ad ogni cella si applica l'algoritmo di rilevamento.
Ogni **bounding box** (cella) avrà il label:

$$y = (P_c, b_x, b_y, b_h, b_w, c_1, \dots, c_n)$$

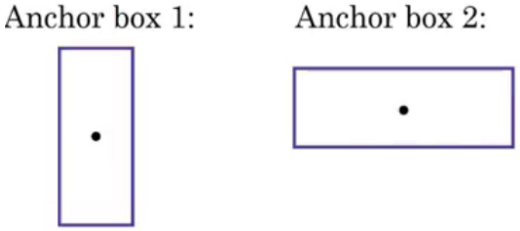
$$con\ b_x, b_y \in [0,1], \qquad b_h, b_w > 0\ (> 1\ se\ l'oggetto\ occupa\ più\ celle)$$

Se il centro/il punto medio di un oggetto cade in una cella della griglia, quella cella della griglia è responsabile della rilevazione di quell'oggetto.

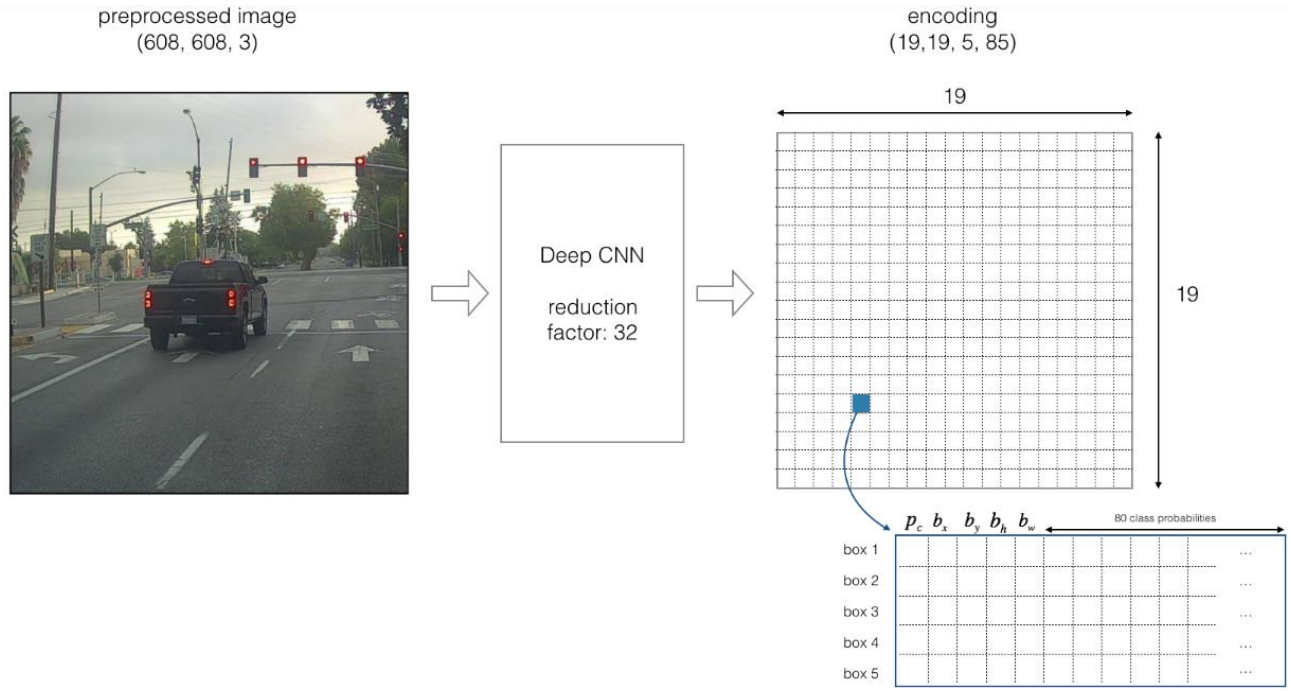
Anchor boxes: usate quando si hanno oggetti sovrapposti. Si usano assegnano più box per ogni cella (di solito 5-10).
Il label di ogni cella quindi sarà:

$$y = (P_c, b_x, b_y, b_h, b_w, c_1, \dots, c_n, P_c, b_x, b_y, b_h, b_w, c_1, \dots, c_n, \dots.)$$

Anchor Box 1 Anchor Box 2

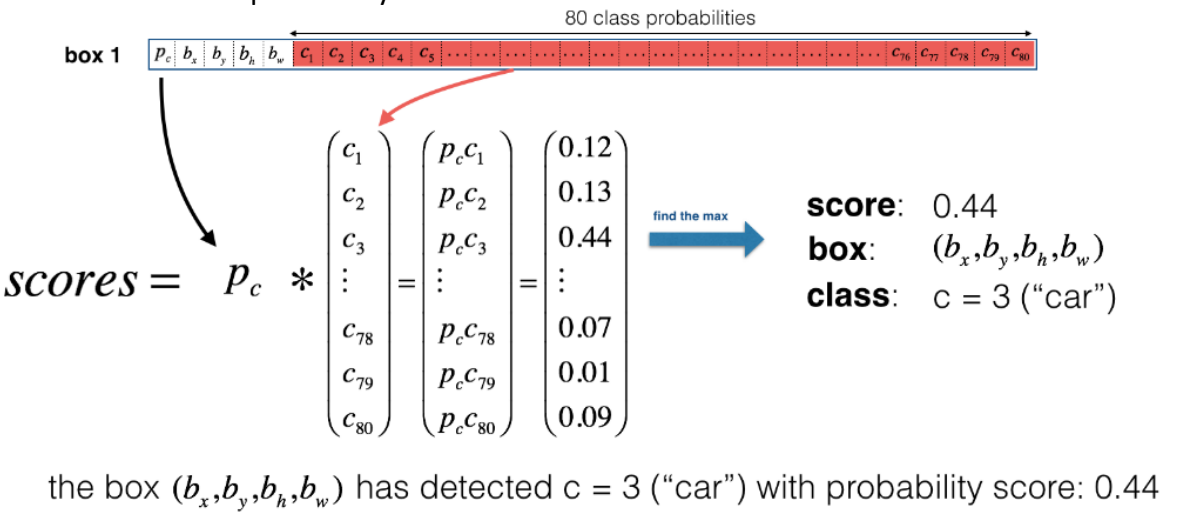


Dimensione output: $n \times n \times (n_{anchor} \times (5 + n_{classes}))$

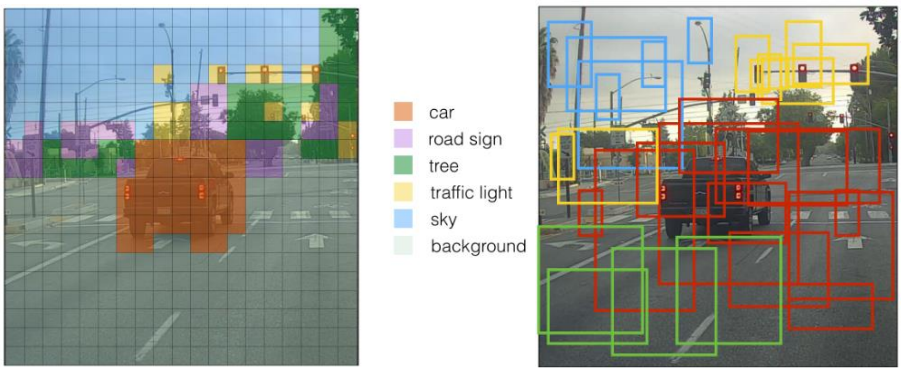


Yolo Algorithm:

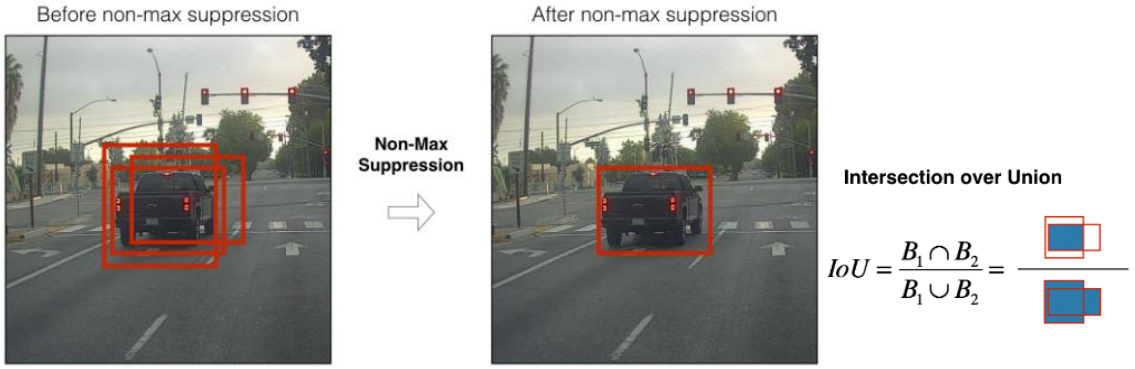
- For each cell:
  - For each anchor box (in the cell): compute the following elementwise product and extract a probability that the box contains a certain class.



- **Score-thresholding:** throw away boxes that have detected a class with a score less than the threshold (e.g $score \leq 0.6$).
  For each of the 19x19 grid cells, find the maximum of the probability scores (taking a max across both the 5 anchor boxes and across different classes).



- **Non-Max suppression:**
  Finché ci sono bounding box:
  - Prendere il box B con la più grande p_c e usarlo come output per quell'oggetto
  - Scartare tutti i box rimanenti che hanno $IoU \geq 0.5$ con il box B.



Example of YOLO

- Input image (608, 608, 3)
- The input image goes through a CNN, resulting in a (19,19,5,85) dimensional output.
- After flattening the last two dimensions, the output is a volume of shape (19, 19, 425):
    - Each cell in a 19x19 grid over the input image gives 425 numbers.
    - 425 = 5 x 85 because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes.
    - 85 = 5 + 80 where 5 is because (pc,bx,by,bh,bw) has 5 numbers, and and 80 is the number of classes we'd like to detect
- You then select only few boxes based on:
    - Score-thresholding: throw away boxes that have detected a class with a score less than the threshold
    - Non-max suppression: Compute the Intersection over Union and avoid selecting overlapping boxes
- This gives you YOLO's final output.

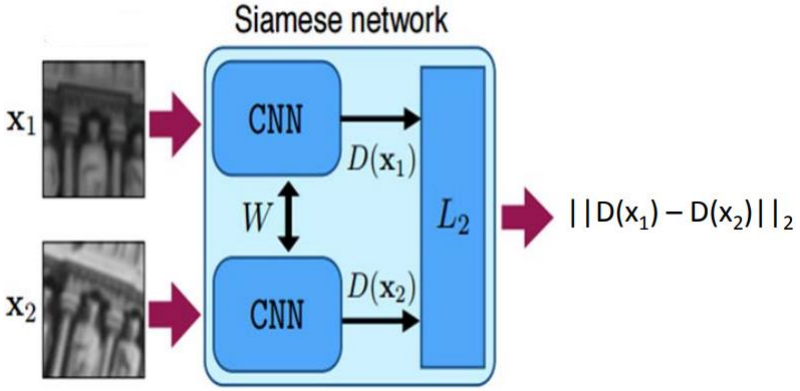
8.4. Face recognition

Face Verification: "is this the claimed person?". Input: picture and id. Output: yes/no
Face Recognition: "who is this person?". Database of k people. Input: picture, output: person id or "not recognized".

One shot learning: learn from just one example. It's not possible to use a softmax because there ar not enough examples and if we want to insert a new person in the database we need to retrain the network. So we will learn a similarity function.

Similarity function: $d(img_1, img_2) = \text{degree of difference}$

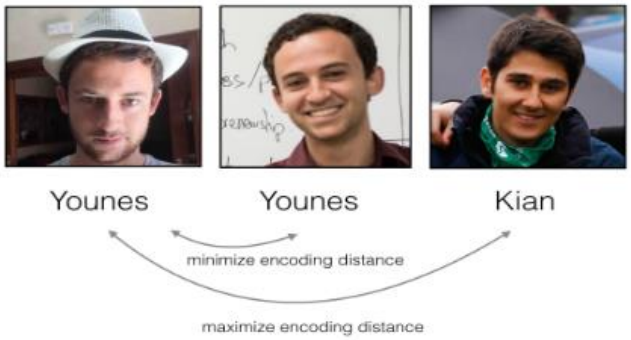
Siamese network: learn parameters in in order to minimize $\|D(x_1) - D(x_2)\|_2^2$



8.4.1. The Triplet Loss

For an image x, we denote its encoding $f(x)$, where f is the function computed by the neural network.
Training will use triplets of images (A, P, N):

- A is an "Anchor" image--a picture of a person.
- P is a "Positive" image--a picture of the same person as the Anchor image.
- N is a "Negative" image--a picture of a different person than the Anchor image.



You'd like to make sure that an image $A^{(i)}$ of an individual is closer to the Positive $P^{(i)}$ than to the Negative image $N^{(i)}$ by at least a margin α:

$$\left\|f\left(A^{(i)}\right) - f\left(P^{(i)}\right)\right\|_2^2 - \left\|f\left(A^{(i)}\right) - f\left(N^{(i)}\right)\right\|_2^2 + \alpha \leq 0$$

Loss function: $\mathcal{L}\left(A^{(i)}, P^{(i)}, N^{(i)}\right) = \max\left(\left\|f\left(A^{(i)}\right) - f\left(P^{(i)}\right)\right\|_2^2 - \left\|f\left(A^{(i)}\right) - f\left(N^{(i)}\right)\right\|_2^2 + \alpha, 0\right)$
Triplet Cost function: $J = \sum_{i=1}^{m} \mathcal{L}\left(A^{(i)}, P^{(i)}, N^{(i)}\right)$

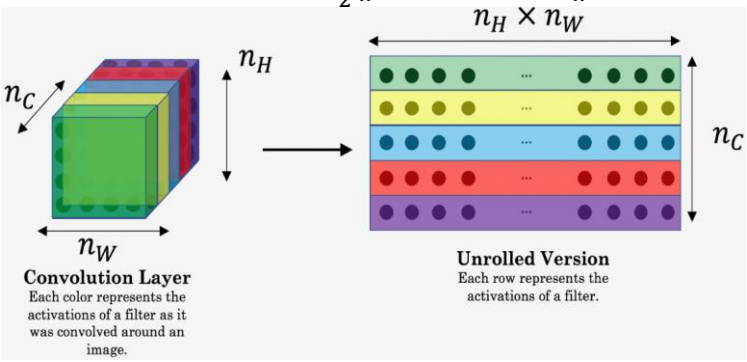
Choose A, P and N in such way that $d(A, P) \cong d(A, N)$ so the network will be more efficient.

8.5. Neural Style Transfer

The earlier (shallower) layers of a ConvNet tend to detect lower-level features such as edges and simple textures, and the later (deeper) layers tend to detect higher-level features such as more complex textures as well as object classes.

We would like the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image. In practice, you'll get the most visually pleasing results if you choose a layer in the middle of the network--neither too shallow nor too deep.

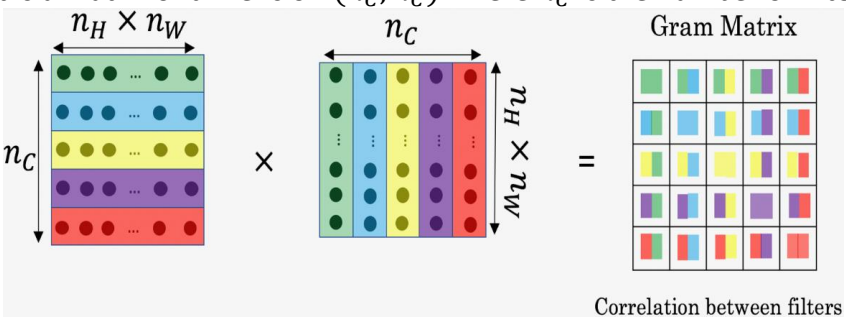
- l is one particular hidden layer to use that you picked.
- set the image C as the input to the pretrained VGG network and run forward propagation.
- $a^{[l](C)}$ is the hidden layer activations in the layer you had chosen. This is a $n_H \times n_W \times n_C$ tensor. That are the height, width and number of channels of the hidden layer.
- Repeat this process with the image G: Set G as the input, and run forward propagation.
- Let $a^{[l](G)}$ be the corresponding hidden layer activation.
- Content cost function $\boldsymbol{J_{content}(C,G)} = \frac{1}{2}\left\|\boldsymbol{a}^{[l](C)} - \boldsymbol{a}^{[l](G)}\right\|^2$ (element-wise subtraction)



The style matrix is also called a "Gram matrix."

In linear algebra, the Gram matrix G of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$. In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.

- Compute the Style matrix by multiplying the "unrolled" filter matrix with their transpose. The result is a matrix of dimension (n_C, n_C) where n_C is the number of filters.



- Style cost function:

$$J_{style}^{[l]}(S,G) = \frac{1}{\left(2 \times n_C^{[l]} \times n_H^{[l]} \times n_W^{[l]}\right)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} \left(G_{ij}^{[l](S)} - G_{ij}^{[l](G)}\right)^2$$

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S,G)$$

- Put it together to get $J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$

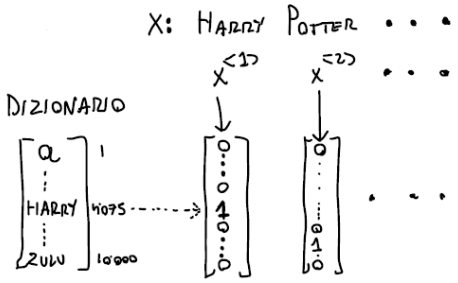
9.Sequence Models

Useful for speech recognition, music generation, sentiment classification, DNA sequences analysis, machine translation, video activity recognition, name entity generation.
Recurrent Neural Networks (RNN) are very effective for Natural Language Processing and other sequence tasks because they have "memory". They can read inputs $x^{\langle t \rangle}$ (such as words) one at a time, and remember some information/context through the hidden layer activations that get passed from one time-step to the next. This allows a uni-directional RNN to take information from the past to process later inputs. A bidirectional RNN can take context from both the past and the future.

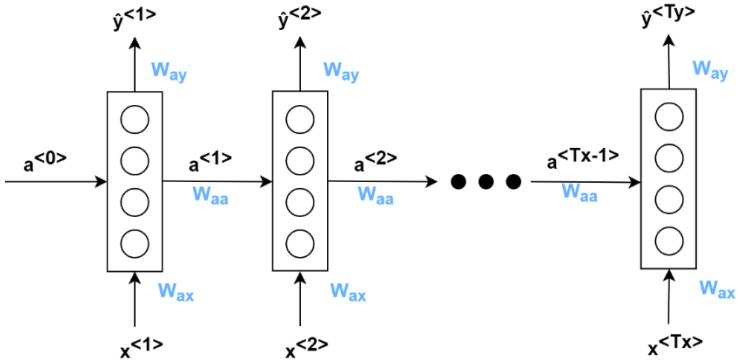
9.1. Recurrent Neural Networks RNN

9.1.1. Notation

$x^{\langle t \rangle}$: input in un certo istante t
$y^{\langle t \rangle}$: output in un certo istante t
T_x: lunghezza input
T_y: lunghezza output
$x^{(i)\langle t \rangle}$: esempio i



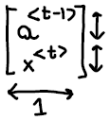
9.1.2. Forward propagation



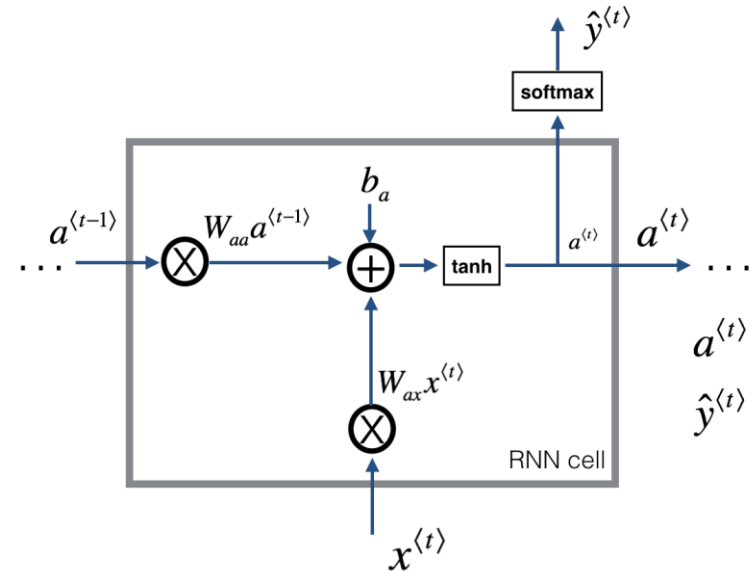
$a^{\langle 0 \rangle} = \vec{0}$
$a^{\langle t \rangle} = g\left(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a\right)$ ReLu/tanh
$\hat{y}^{\langle t \rangle} = g\left(W_{ya}a^{\langle t \rangle} + b_y\right)$ Sigmoid

Questa notazione viene semplificata unendo:

- W_{aa} *e* W_{ax}: $\boldsymbol{W_a} = [\boldsymbol{W_{aa}}|\boldsymbol{W_{ax}}]$
- $a^{\langle t-1 \rangle}$*e* $x^{\langle t \rangle}$: $\left[\boldsymbol{a}^{\langle \boldsymbol{t-1} \rangle}, \boldsymbol{x}^{\langle \boldsymbol{t} \rangle}\right] = \begin{bmatrix} \boldsymbol{a}^{\langle \boldsymbol{t-1} \rangle} \\ \boldsymbol{x}^{\langle \boldsymbol{t} \rangle} \end{bmatrix}$ vettori in verticale

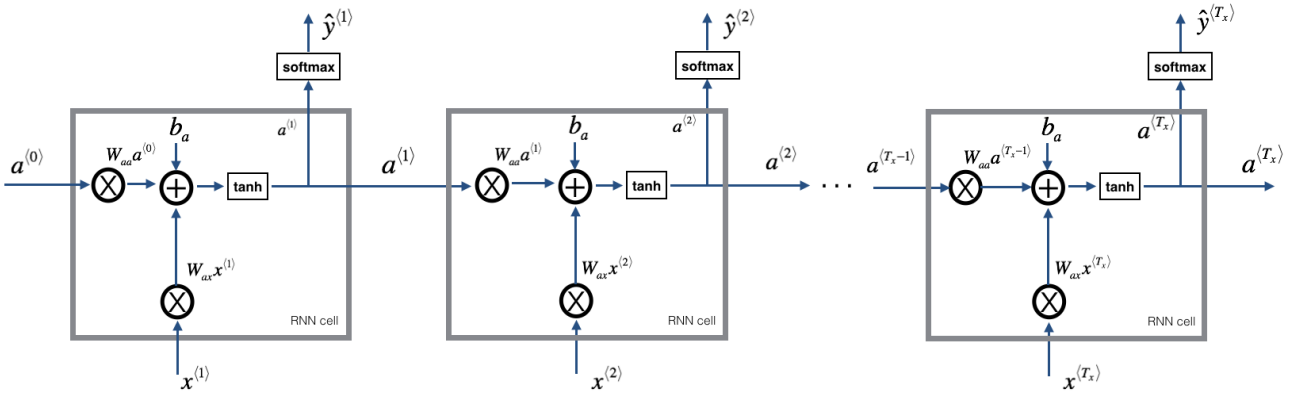


$$\begin{cases} a^{\langle t \rangle} = g\left(W_a\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_a\right) \\ \hat{y}^{\langle t \rangle} = g\left(W_y a^{\langle t \rangle} + b_y\right) \end{cases}$$



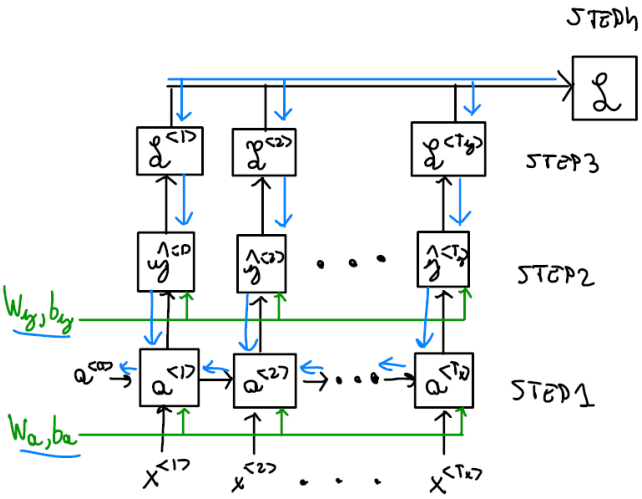
$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$



This will work well enough for some applications, but it suffers from vanishing gradient problems. So it works best when each output $y^{\langle t \rangle}$ can be estimated using mainly "local" context (meaning information from inputs $x^{\langle t' \rangle}$ where t' is not too far from t).

9.1.3. Back propagation through time



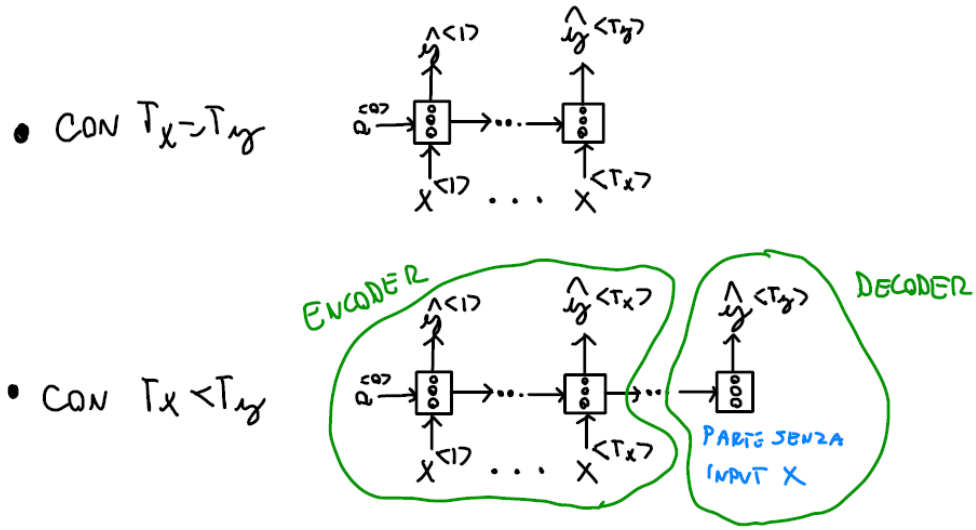
- Forward Prop : Step 1→4
- Backward Prop : Step 4→1

- $\mathcal{L}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -y^{\langle t \rangle}\log\hat{y}^{\langle t \rangle} - (1-y^{\langle t \rangle})\log(1-\hat{y}^{\langle t \rangle})$
- $\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y}\mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$

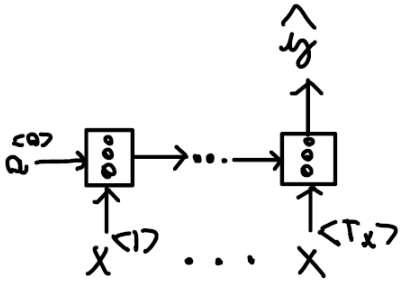
9.1.4. Types of RNN

Not always $T_x = T_y$

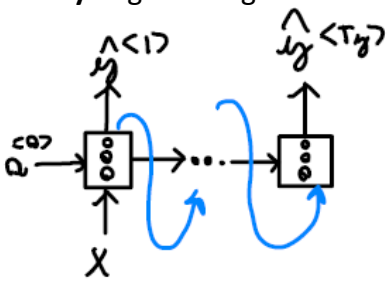
Many to many:



Many to one: e.g. sentiment analysis (x: text, y: 0/1)



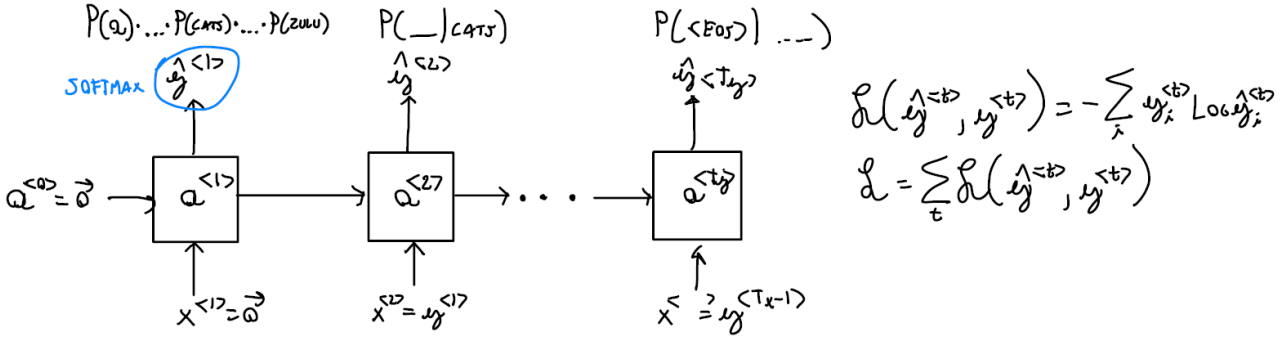
One to many: e.g. music generation



9.1.5. Language modelling

Data una frase restituisce la probabilità che la prossima frase. ??????????????

Training:



Ad ogni step la parola futura $P(y^{\langle 1 \rangle}, \dots, y^{\langle T_y \rangle}) = P(y^{\langle 1 \rangle})P(y^{\langle 2 \rangle}|y^{\langle 1 \rangle}) * \dots * P(y^{\langle T_y \rangle}|y^{\langle 1 \rangle}, \dots, y^{\langle T_{y-1} \rangle})$

Sampling ?????????????

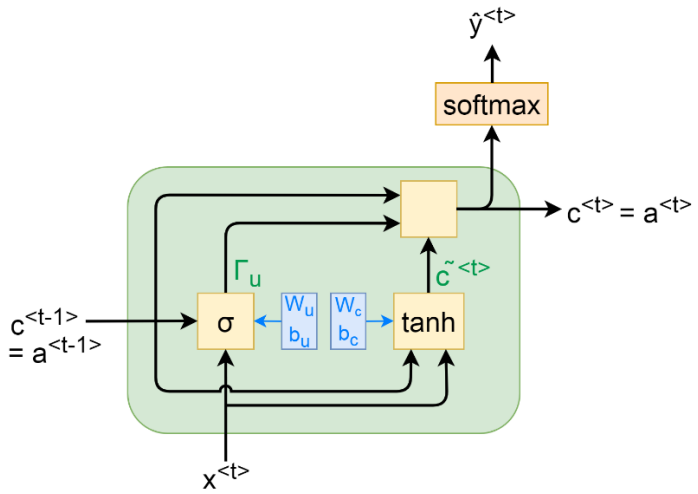
Vanishing gradients: problema per le reti profonde, le derivate diventano piccole/grandi rendendo il training difficile.

Soluzione all'exploding grandients: Gradient Clipping: se gradiente > soglia, lo si ri-scala.

Recall that your overall loop structure usually consists of a forward pass, a cost computation, a backward pass, and a parameter update. Before updating the parameters, you will perform gradient clipping when needed to make sure that your gradients are not "exploding," meaning taking on overly large values.

9.1.6. Gated Recurrent Unit (GRU)

Modification to the RNN hidden layer that makes it much better at capturing long range connections and helps a with the vanishing gradient problem.



When $c^{\langle t \rangle} = a^{\langle t \rangle}$

Candidate $\tilde{c}^{\langle t \rangle} = \tanh\left(W_c\left[c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right)$

Update Gate $\Gamma_u = \sigma\left(W_u\left[c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_u\right)$

$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}$

(element-wise product)

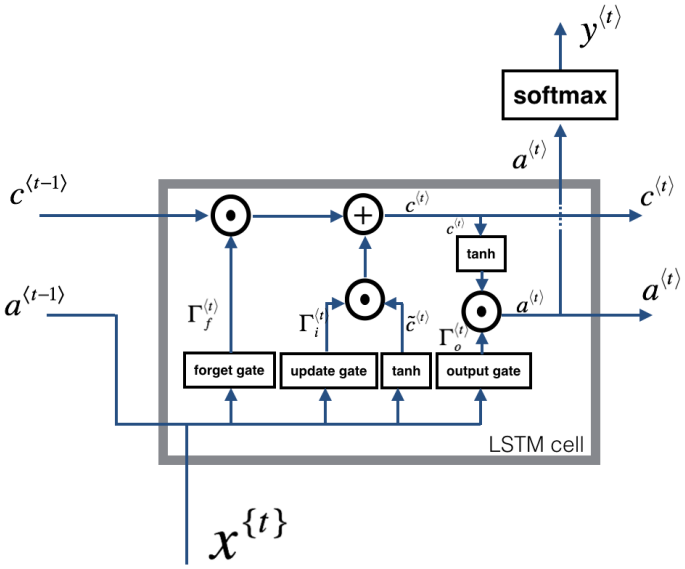
If $\Gamma_u = 1$ will be chosen $\tilde{c}^{\langle t \rangle}$ else $c^{\langle t-1 \rangle}$

Extended GRU:

$$\tilde{c}^{\langle t \rangle} = \tanh\left(W_c\left[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right)$$
$$\Gamma_r = \sigma\left(W_r\left[c^{\langle t \rangle}, x^{\langle t \rangle}\right] + b_r\right)$$

9.1.7. Long Short-Term Memory (LSTM)

Is better at addressing vanishing gradients. Will be better able to remember a piece of information and keep it saved for many timesteps. But requires more computation. When $c^{\langle t \rangle} \neq a^{\langle t \rangle}$



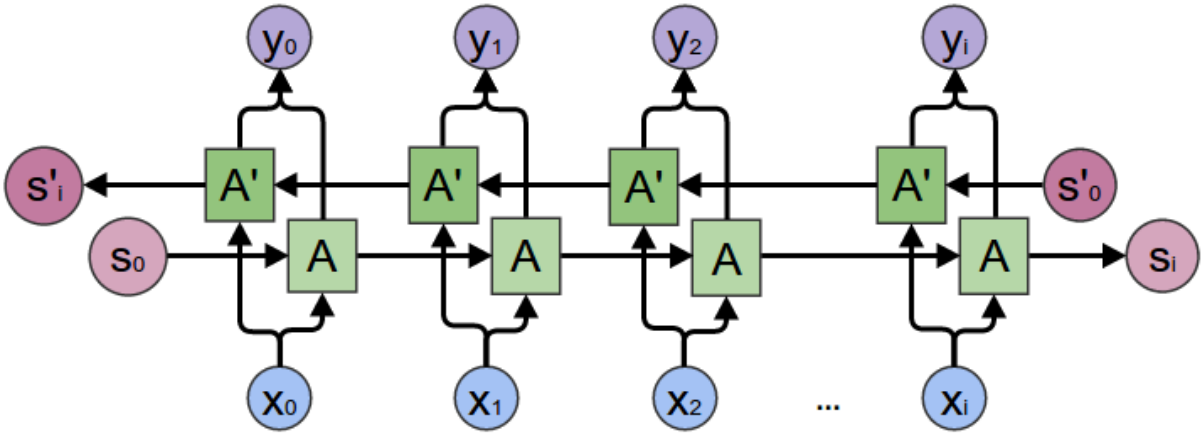
$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\langle t \rangle} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Gates:

- **Forget gate:** vector with values between 0 and 1. This forget gate vector will be multiplied element-wise by the previous cell state c⟨t−1⟩. So if one of the values of $\Gamma^{\langle t \rangle}_f$ is 0 (or close to 0) then it means that the LSTM should remove that piece of information in the corresponding component of c⟨t−1⟩. If one of the values is 1, then it will keep the information.
- **Update gate:** vector of values between 0 and 1. This will be multiplied element-wise with c̃⟨t⟩, in order to compute c⟨t⟩.
- **Output gate:** decide what to output using a sigmoid function

9.1.8. Bidirectional RNN (BRNN)

Take as input also the future, so requires the entire input. (It's not usable in real time)

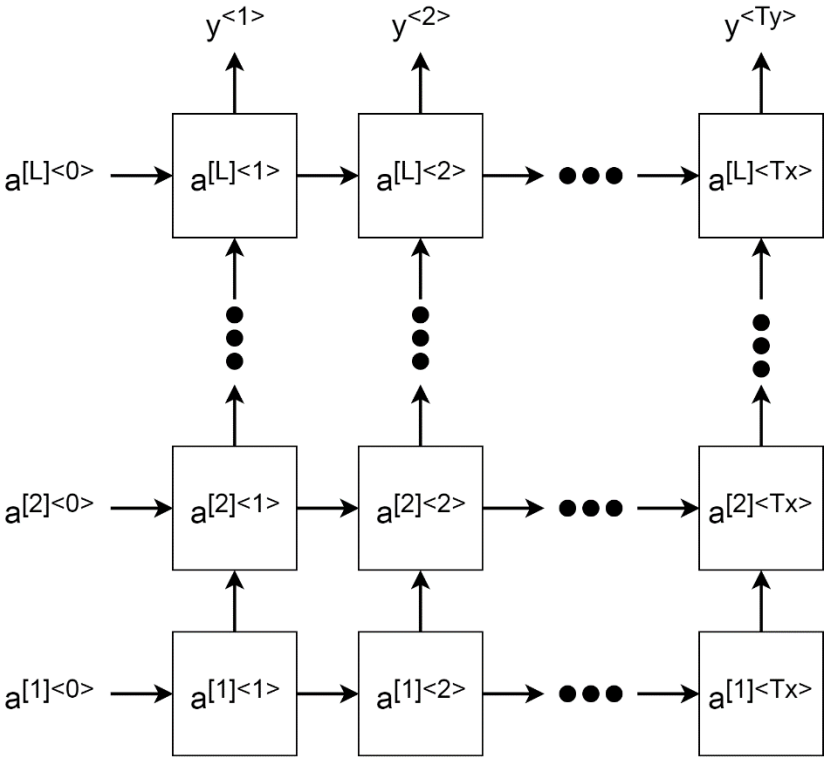


Forward propagation is done in two steps:

- We move from left to right, starting with the initial time step we compute the values until we reach the final time step
- We move from right to left, starting with the final time step we compute the values until we reach the initial time step

9.1.9. Deep RNN

Usually 3 layers are enough.



$$a^{[l]\langle t\rangle} = g\left(W_a^{[l]}\left[a^{[l]\langle t-1\rangle}, a^{[l-1]\langle t\rangle}\right] + b_a^{[l]}\right)$$

9.2. Word Embeddings

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.

9.2.1. Representation

We use a dictionary $V = [a, aaron, \dots, zulu, < unk >]$ $|V| = 10'000$ 1-hot representation

Featured representation: each word is associated with features

	Man	Woman	King	Queen
Gender	-1	1	-0.95	0.97
Royal	0.01	0.02	0.93	0.95

$$e_{man} = column\ of\ "man"$$

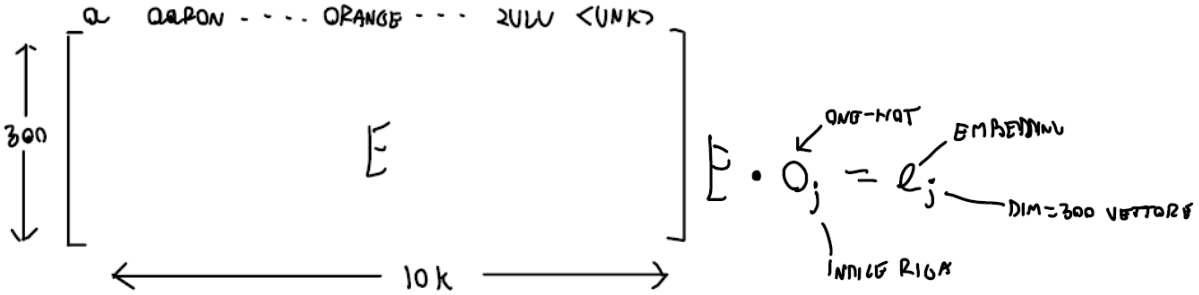
Visualizing word embeddings: 300 dim --> 2 dim using T-SNE algorithm

Find analogies: $e_{man} - e_{woman} \approx e_{king} - e_w$

Find word w that has $\underset{w}{\text{argmax}}\ similarity(e_w, e_{king} - e_{woman} + e_{man})$

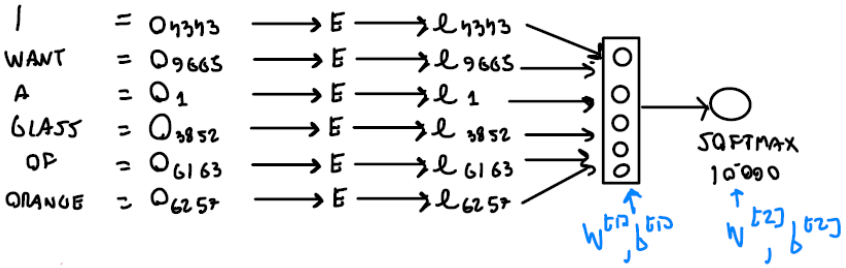
Cosine similarity: $sim(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$

Embedding matrix E:



9.2.2. Learning Word Embeddings

Method 1: Language model



Method 2: Word2Vec: is a method to construct an embedding.

Skip-Gram:

- Context: random word of the sentence
- Target: another random word of the sentence

$$o_c \to E \to e_c = E * o_c \to softmax \to \hat{y}$$

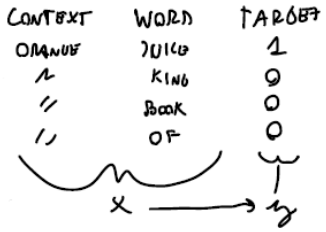
Softmax: $P(t|c) = \frac{e^{\theta_j^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$ $con\ \theta_t$: parameter associated with output t

$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i \log \hat{y}_i$
In practice is used Hierarchical Softmax to save computation.

Negative sampling:

- Pick a context/target pair as positive example
- Pick a few context+random as negative examples

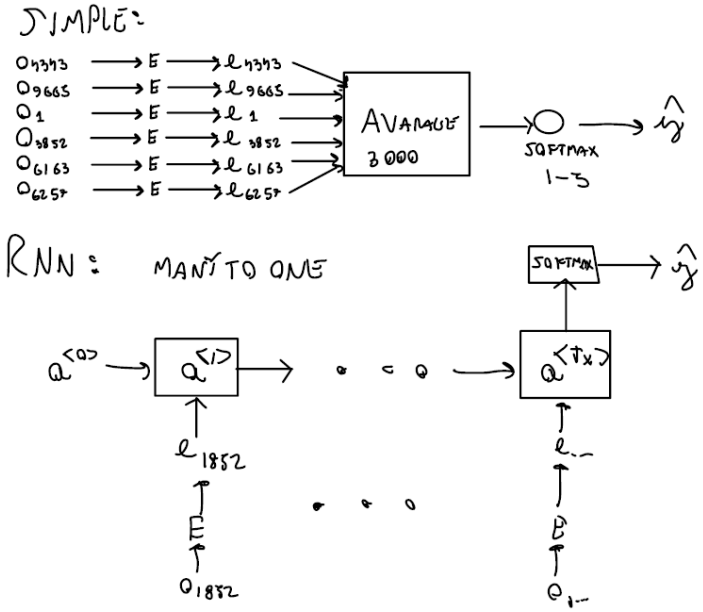
Create a supervised learning algorithm that has to learn the output y.



$P(y = 1|c, t) = \sigma(\theta_t^T e_c)$ Binary classifier

9.2.3. Applications

Sentiment classification:



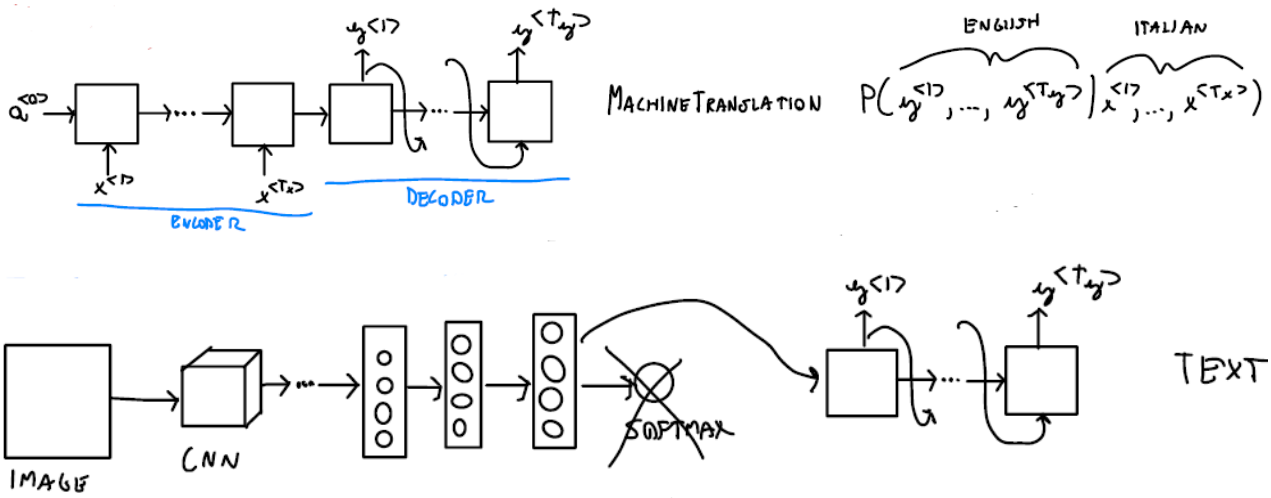
If you have an NLP task where the training set is small, using word embeddings can help your algorithm significantly. Word embeddings allow your model to work on words in the test set that may not even have appeared in your training set.

Training sequence models in Keras (and in most other deep learning frameworks) requires a few important details:

- To use mini-batches, the sequences need to be padded so that all the examples in a mini-batch have the same length.
- An Embedding() layer can be initialized with pretrained values. These values can be either fixed or trained further on your dataset. If however your labeled dataset is small, it's usually not worth trying to train a large pre-trained set of embeddings.
- LSTM() has a flag called return_sequences to decide if you would like to return every hidden states or only the last one.
- You can use Dropout() right after LSTM() to regularize your network

9.3. Architectures

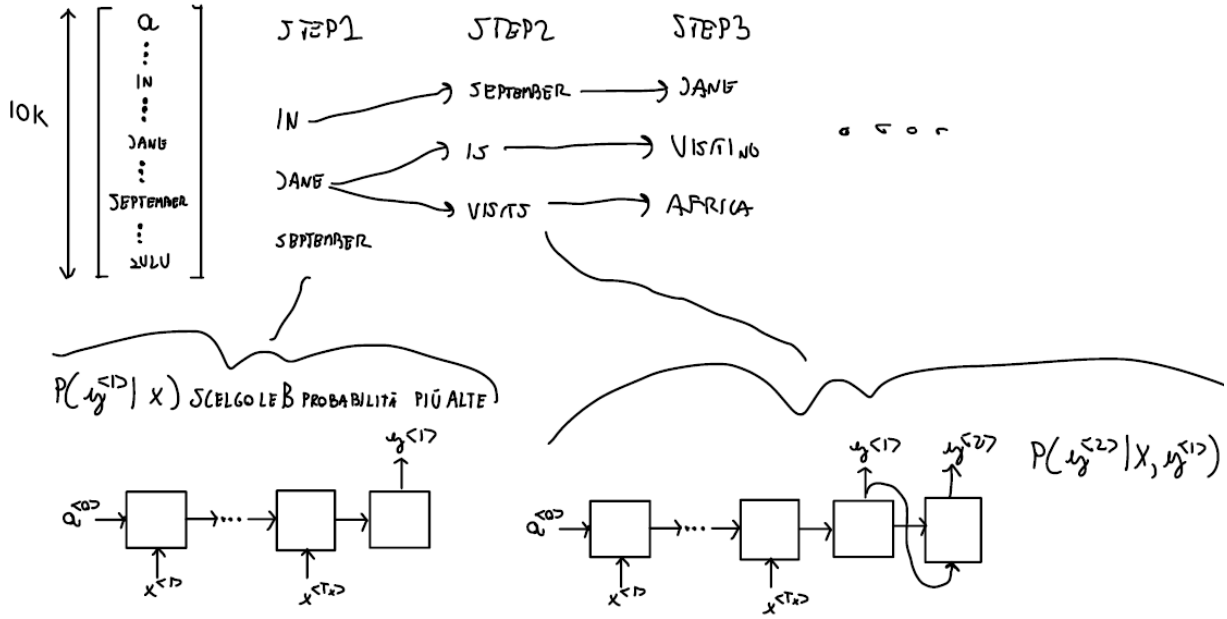
9.3.1. Basic models



Pick the most likely sentence: $\underset{y^{\langle 1\rangle},\ldots,y^{\langle T_y\rangle}}{\operatorname{argmax}} P\left(y^{\langle 1\rangle},\ldots,y^{\langle T_y\rangle}\middle|x^{\langle 1\rangle},\ldots,x^{\langle T_x\rangle}\right)$ (ENGLISH, ITALIAN)

9.3.2. Beam search

The parameter B considers more possibilities



Refinements to beam:

- $\operatorname{argmax}\prod_{t=1}^{T_y} P\left(y^{\langle t\rangle}\middle|x,y^{\langle 1\rangle},\ldots,y^{\langle t-1\rangle}\right)$
  Problem: multiplying probabilities result in a small number (overflow), so use this just with short sentences
- $\frac{1}{T_y^{\alpha}}\sum_{t=1}^{T_y}\log P\left(y^{\langle t\rangle}\middle|x,y^{\langle 1\rangle},\ldots,y^{\langle t-1\rangle}\right)\ con\ \alpha = 0.7$

Error Analysis Beam:

UMANO y^* RNN $\hat{y}$

SE $P(y^*|x) > P(\hat{y}|x)$: BEAM HA FALLITO, PROVA UN B PIÙ GRANDE

ALTRIMENTI

RNN HA SBAGLIATO, QUINDI MIGLIORA LA RNN

9.3.3. Bleu Score

SE 2 UMANI DICONO COSE DIVERSE :

$\forall$ PAROLA $\in$ FRASE VIENE ASSEGNATO UN VALORE C_i = # VOLTE CHE APPARE NELLA FRASE , $\text{COUNTCLIP} = \sum_i C_i$, $\text{PRECISIONE} = \frac{\text{COUNTCLIP}}{\#\text{ PAROLE NELLA FRASE}}$

$\forall$ BIGRAMMA (COPPIA PAROLE) VIENE ASSEGNATO UN VALORE C_i = # VOLTE CHE APPARE NELLA FRASE , $\text{COUNTCLIP} = \sum_i C_i$, $\text{PRECISIONE} = \frac{\text{COUNTCLIP}}{\#\text{ BIGRAMMI POSSIBILI}}$

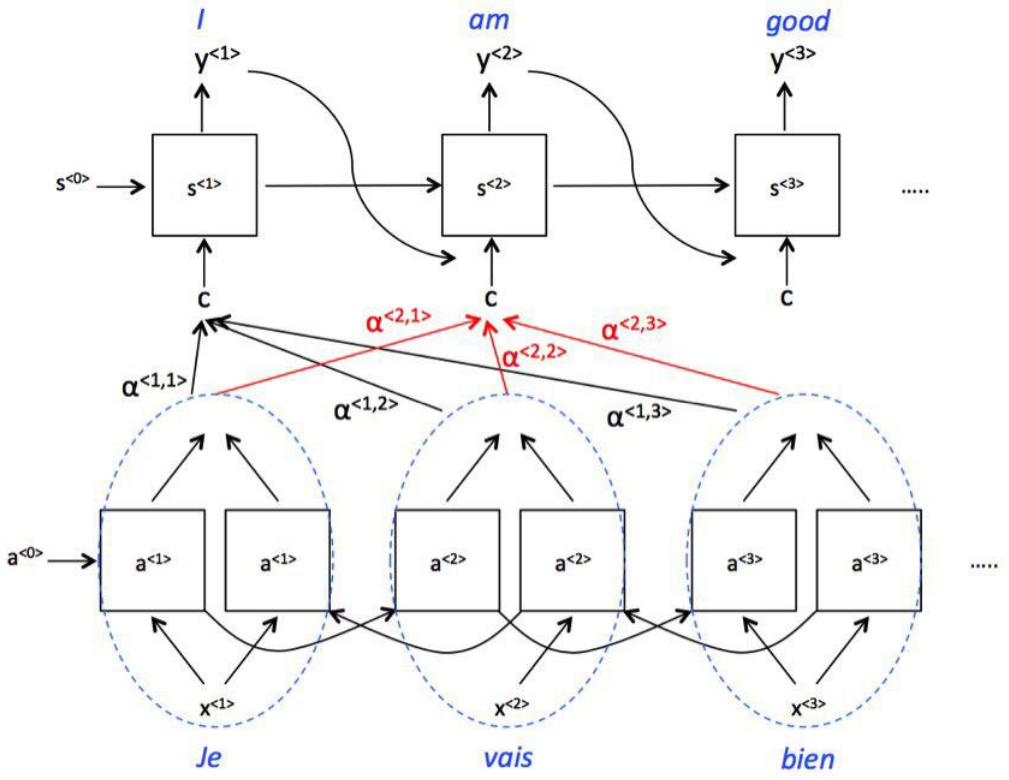
$$\text{PRECISIONE } P_n = \frac{\sum_{n\text{-gram} \in \hat{y}} \text{COUNTCLIP}(n\text{-gram})}{\sum_{n\text{-gram} \in \hat{y}} \text{COUNT}(n\text{-gram})} \quad / \; n\text{-gram}$$

METRICA DA UTILIZZARE:

$$BP \cdot \exp\left(\frac{1}{m}\sum_{n=1}^{m} P_n\right) \quad \text{CON} \quad BP = \begin{cases} 1 & \text{SE LUNGHEZZA OUTPUT MT} > \text{LUNGHEZZA OUTPUT UMANO} \\ \exp(1 - \text{LUNGHEZZA OUTPUT MT} / \text{LUNGHEZZA OUTPUT UMANO}) & \text{ALTRIMENTI} \end{cases}$$

9.3.4. Attention Model

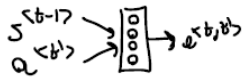
Solution for long sequences: translate a little a time using only parts of the sentence as context.



$a^{\langle t,t' \rangle}$ = quantity of attention that $y^{\langle t \rangle}$ has to pay to $a^{\langle t' \rangle} = \exp\left(e^{\langle t,t' \rangle}\right) / \sum_{t'=1}^{T_x} \exp\left(e^{\langle t,t' \rangle}\right)$

$a^{\langle t' \rangle} = \left(\vec{a}^{\langle t' \rangle}, \overleftarrow{a}^{\langle t' \rangle}\right)$

$c^{\langle 1 \rangle} = \sum_{t'} \alpha^{\langle 1,t' \rangle} a^{\langle t \rangle}$

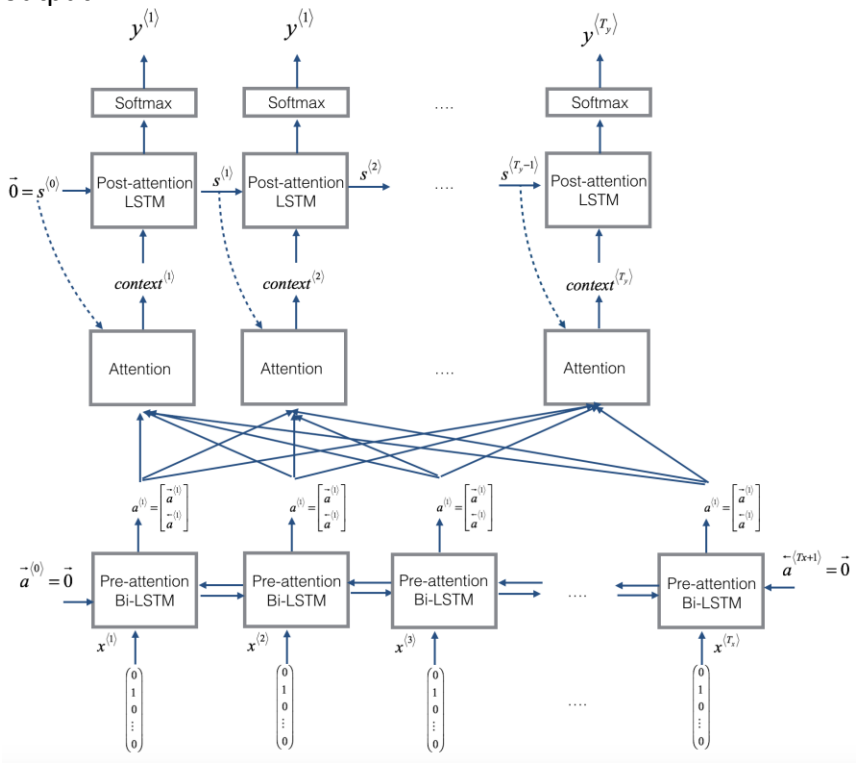
CON 

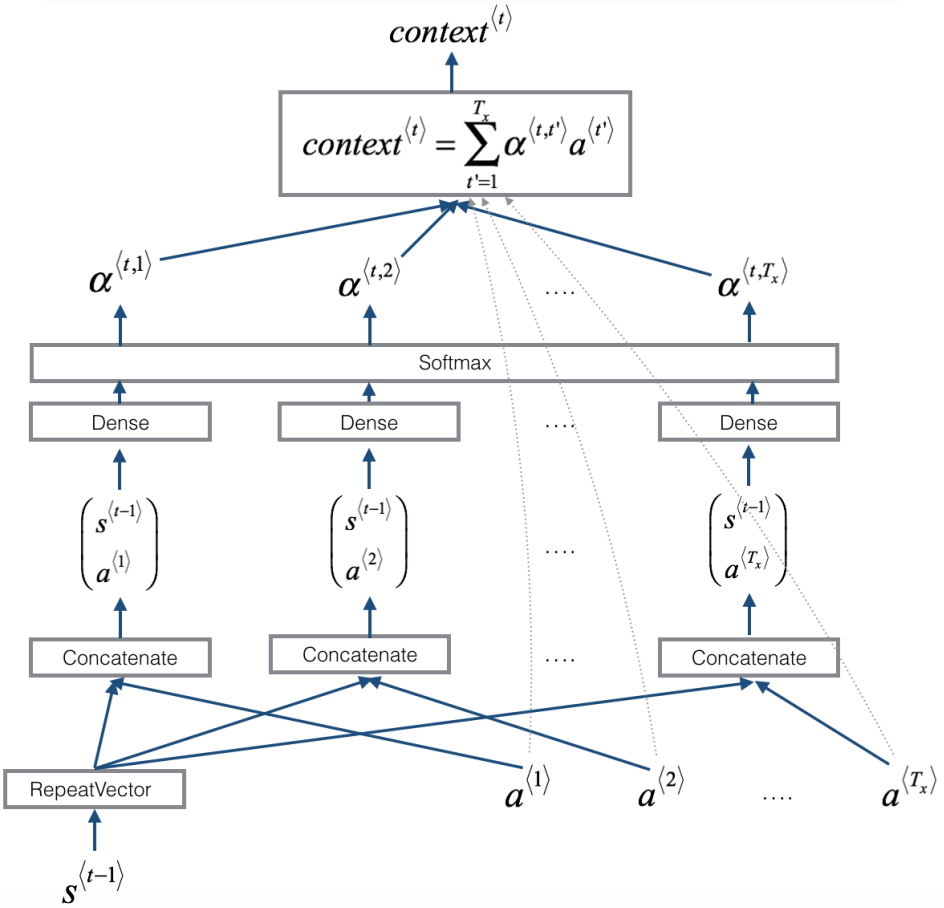
Machine translation models can be used to map from one sequence to another. They are useful not just for translating human languages (like French->English) but also for tasks like date format translation.

An attention mechanism allows a network to focus on the most relevant parts of the input when producing a specific part of the output.

A network using an attention mechanism can translate from inputs of length Tx to outputs of length Ty, where Tx and Ty can be different.

You can visualize attention weights α⟨t,t′⟩ to see what the network is paying attention to while generating each output.





9.3.5. Trigger Word Detection

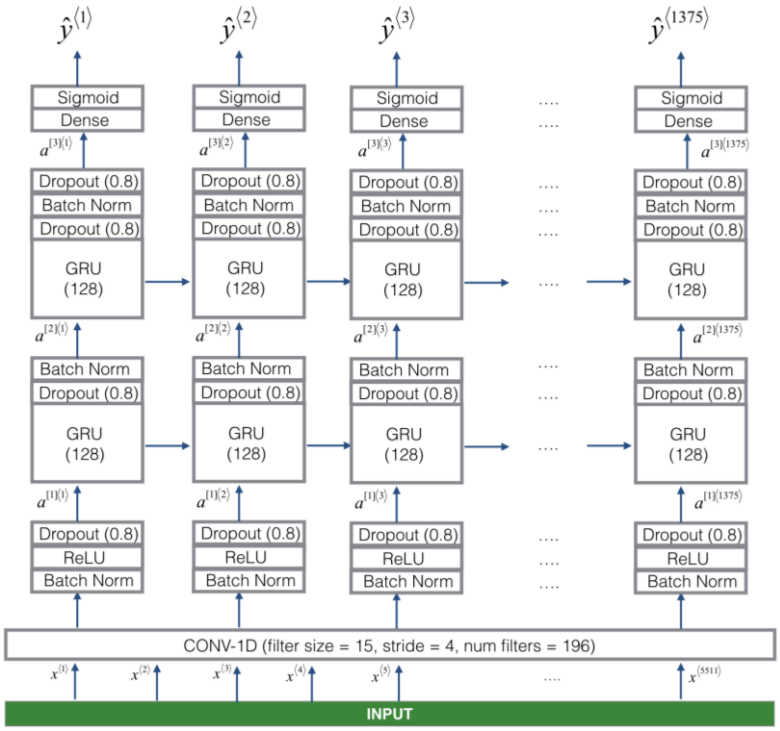
Trigger word detection is the technology that allows devices like Amazon Alexa, Google Home, Apple Siri, and Baidu DuerOS to wake up upon hearing a certain word.

A speech dataset should ideally be as close as possible to the application you will want to run it on. You thus need to create recordings with a mix of positive words ("activate") and negative words (random words other than activate) on different background sounds.
A microphone records little variations in air pressure over time, and it is these little variations in air pressure that your ear also perceives as sound. You can think of an audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone.

It is quite difficult to figure out from this "raw" representation of audio whether the word "activate" was said. In order to help your sequence model more easily learn to detect triggerwords, we will compute a spectrogram of the audio. The spectrogram tells us how much different frequencies are present in an audio clip at a moment in time.
The dimension of the output spectrogram depends upon the hyperparameters of the spectrogram software and the length of the input.

Because speech data is hard to acquire and label, you will synthesize your training data using the audio clips of activates, negatives, and backgrounds. It is quite slow to record lots of 10 second audio clips with random "activates" in it. Instead, it is easier to record lots of positives and negative words, and record background noise separately (or download background noise from free online sources).
To synthesize a single training example, you will:

- Pick a random 10 second background audio clip
- Randomly insert 0-4 audio clips of "activate" into this 10sec clip
- Randomly insert 0-2 audio clips of negative words into this 10sec clip



Using a spectrogram and optionally a 1D conv layer is a common pre-processing step prior to passing audio data to an RNN, GRU or LSTM.