

Appunti “Advanced Machine Learning”

Prof. Fabio Galasso. Appunti scritti da [Alessio Lucciola](#) durante l'a.a. 2023/2024.

Sei libero di:

- Condividere: Copiare e distribuire il materiale in qualsiasi mezzo o formato.
- Adattare: Trasformare o modificare il materiale.

Rispettando i seguenti termini:

- Attribuzione: E' necessario fornire i crediti appropriati, un collegamento alla licenza e indicare se sono state apportate modifiche.
- Fine non commerciale: Non è possibile utilizzare il materiale per scopi commerciali.

Le note possono contenere errori di battitura. Se ne vedi uno, puoi contattarmi utilizzando i link nella [pagina Github](#). Se questi appunti ti sono utili, potresti prendere in considerazione l'idea di [offrirmi un caffè](#) 😊.

1. Introduzione

1.1 Cos'è la “Data Science”

La **Data science** è un campo interdisciplinare che utilizza metodi, processi, algoritmi e sistemi scientifici per estrarre conoscenze e approfondimenti da dati rumorosi (noisy), strutturati e non strutturati e applicare conoscenze e approfondimenti utilizzabili dai dati in un'ampia gamma di domini applicativi. La Data Science è correlata al data mining, al Machine Learning e al Big Data. Esistono due fasi per lavorare con i dati:

- **Raccogliere dati:** Tramite computer, sensori, persone, eventi, ecc.;
- **Farsi qualcosa:** Prendere decisioni, confermare ipotesi, acquisire intuizioni, prevedere il futuro, ecc.;

Una volta raccolti i dati è possibile utilizzarli in molti modi, ad esempio:

- Traffico: E' possibile prevedere le congestioni ed i flussi di traffico nel presente e nell'immediato futuro;
- Sistemi di raccomandazione: E' possibile consigliare cose (come film da guardare o prodotti da acquistare) in base agli interessi dell'utente;
- Sport: è possibile fare pronostici su come andrà a finire una partita;

E molti altri ancora, come le previsioni del tempo, la pubblicità online, la diagnosi medica, i mercati finanziari, la gestione delle risorse, le scienze sociali computazionali, gli edifici e le città intelligenti.

1.2 Cos'è il “Machine Learning”

Il **Machine Learning** (ML) è lo studio di algoritmi informatici che possono migliorare automaticamente attraverso l'esperienza e l'uso dei dati. È visto come una parte dell'intelligenza artificiale. Gli algoritmi di machine learning costruiscono un **modello** basato su dati campione, noti come dati di training, per fare previsioni o prendere decisioni senza essere esplicitamente programmati per farlo. Gli algoritmi di ML vengono utilizzati in un'ampia varietà di applicazioni, come in medicina, nel filtraggio della posta elettronica, nel riconoscimento vocale e computer vision, dove è difficile o irrealizzabile sviluppare algoritmi convenzionali per eseguire i compiti necessari.

Alcuni esempi possono essere trovati in:

- Estrazione del database: Grandi set di dati derivanti dalla crescita dell'automazione/web (ad esempio dati sui clic sul web, cartelle cliniche, biologia);
- Applicazioni che non possono essere programmate manualmente: Per esempio. guida autonoma, riconoscimento della scrittura, gran parte del Natural Language Processing (PNL) e Computer Vision;
- Programmi di auto-personalizzazione (che variano o si adattano all'utente in base alla sua esperienza e ai suoi interessi): Per esempio. Amazon, consigli sui prodotti Netflix;
- Comprendere l'apprendimento umano (cervello, intelligenza artificiale reale);

Una **definizione importante di ML** da Tom Mitchell (1998) è la seguente:

“Si dice che un programma apprende dall’esperienza (experience) E rispetto a qualche compito (task) T e a qualche misura di prestazione (performance) P, se la sua prestazione su T, misurata da P, migliora con l’esperienza E.”

EX. Supponiamo che un programma di posta elettronica controlli quali email contrassegnare o non contrassegnare come spam e, in base a ciò, apprende come filtrare meglio lo spam. Ora:

- Compito (T): Classificare le email come spam o non spam;
- Esperienza (E): Osservare come etichettare le email come spam o non spam;
- Performance (P): Il numero (o frazione) di email correttamente classificate come spam/non spam.

C’è stata un’evoluzione nel modo di approcciarsi al ML:

1. Expert Systems (1980+): Oltre 20 anni fa, c’erano sistemi basati su regole, quindi si aveva bisogno di esperti che dovevano ottenere un dottorato di ricerca in linguistica, fare introspezione sulla struttura della loro lingua madre e scrivere le regole. Quindi hanno dovuto studiare tutti i casi possibili riguardanti un problema specifico e poi affrontare il problema con “If-Then-Else”.

EX. Give me direction to Starbucks

If: “give me directions to X”;

Then: directions(here, nearest(X)).

Gli esperti sono **bravi a rispondere a domande su argomenti specifici ma non sono bravi a dire come farlo.**

2. Annotate Data and Learn (1990+): Allora perché non chiedere loro semplicemente di dirti cosa fanno su casi specifici e poi lasciare che ML ti dica come arrivare alle stesse decisioni che hanno preso loro. Quindi ciò che viene fatto (anche nell’odierno ML) è raccogliere frasi grezze $\{x_1, \dots, x_n\}$ e lasciare che gli esperti annotino il loro significato $\{y_1, \dots, y_n\}$.

EX. x_1 : How do I get to Starbucks?

y_1 : directions(here, nearest(Starbucks))

Qual è la differenza tra ML e DS?

Poiché la Data Science è un termine ampio per più discipline, il Machine Learning si adatta alla Data Science. Il ML utilizza varie tecniche, come la regressione e il clustering supervisionato. D’altra parte, i dati nella DS possono o meno evolversi da una macchina o da un processo meccanico. La differenza principale tra i due è che la DS, come termine più ampio, non si concentra solo su algoritmi e statistiche, ma si occupa anche dell’intera metodologia di elaborazione dei dati (ad esempio raccolta e manipolazione degli stessi).

1.3 Cos’è la “Computer Vision”?

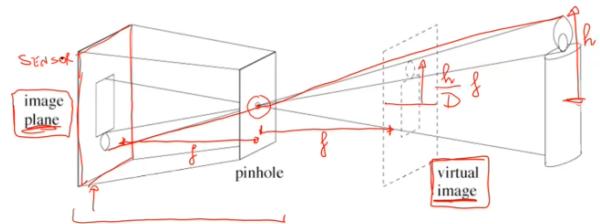
La **Computer Vision** è un campo scientifico interdisciplinare che si occupa di come i computer possono acquisire una comprensione ad alto livello da immagini o video digitali. Dal punto di vista dell’ingegneria, cerca di comprendere e automatizzare le attività che il sistema visivo umano può svolgere. Alcuni esempi possono essere trovati in:

- Riconoscimento targhe (ad esempio tassa sulla congestione di Londra);
- Sicurezza/Sorveglianza: Riconoscimento facciale (Face ID di Apple: possibilità di 1 su 1 milione che una persona a caso possa sbloccare il tuo telefono), il passaporto biometrico (noto anche come passaporto elettronico) ha un chip elettronico incorporato che contiene informazioni biometriche;
- Imaging medico: Segmentazione e misurazioni (semi-)automatiche;
- Guida autonoma;
- Robotica.

1.4 Concetti Base e Terminologia di Image Processing e Computer Vision

Prima di introdurre il concetto di image processing è necessario spiegare cos'è una **macchina fotografica**. Innanzitutto occorre presentare il primo modello di macchina fotografica utilizzato da Leonardo Da Vinci per elaborare la camera oscura (camera obscura) intorno al 1520. La fotocamera stenopeica (**pinhole camera**) è il semplice modello standard e astratto di una macchina fotografica: una fotocamera stenopeica è una semplice macchina fotografica senza lente ma con una piccola apertura (il cosiddetto foro stenopeico o pinhole). Si tratta di una scatola a prova di luce con un piccolo foro su un lato. La luce (ad esempio quella di una candela) passa attraverso l'apertura e proietta un'immagine invertita sul lato opposto della scatola e questo effetto prende il nome di **camera oscura**. È simile a quanto accade con il nostro cervello, infatti il foro stenopeico può essere paragonato alla nostra retina che proietta un'immagine invertita e questa viene elaborata dal cervello.

"Quando immagini di oggetti illuminati... penetrano attraverso un piccolo foro in una stanza molto buia... vedrai [sulla parete opposta] questi oggetti nella loro forma e colore propri, ridotti di dimensioni... in posizione invertita per l'intersezione dei raggi" - Leonardo Da Vinci (1519)



Le moderne fotocamere stenopeiche dispongono di alcuni sensori in grado di trasformare l'immagine acquisita, attraverso l'ottica, in un'immagine digitale (tramite un **digitalizzatore**) dove i colori sono rappresentati con matrici di valori RGB (valori scalari nel caso di un'immagine gray-scale). Esistono sensori che sono in grado di percepire tutti i diversi colori (i tre principali - RGB) e questo accade grazie al fatto che hanno frequenze diverse.

1.4.1 Image Filtering

Data un'immagine, l'obiettivo principale della **computer vision** è capire come sia possibile riconoscere qualcosa da una serie di numeri (in scala di grigi) e anche come percepire la profondità. È esattamente l'opposto della **computer graphics**, il cui obiettivo è trovare un modo per generare una serie di numeri (in scala di grigi) che assomigli a un determinato oggetto.

EX. CV: Data un'immagine (una serie di numeri (in scala di grigi), come possiamo riconoscere i frutti?

CS: Come possiamo generare una serie di numeri (in scala di grigi) che assomiglano a frutti?

Quindi, il caso di studio principale della computer vision è come sia possibile riconoscere gli oggetti e come possiamo far sì che anche i computer vi riescano. Nel cercare di raggiungere questo obiettivo ci sono alcuni problemi che bisogna considerare come dati mancanti, ambiguità e molteplici possibili spiegazioni. Esistono diversi tipi di **problemi di riconoscimento**:

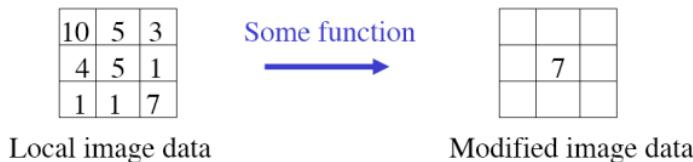
- **Identificazione dell'oggetto (Object Identification)**: Riconoscere un oggetto specifico. Per esempio, riconoscere la tua mela, la tua tazza, il tuo cane;
- **Classificazione di un oggetto (Object Classification)** (detta anche riconoscimento generico degli oggetti o "categoria di livello base"): Riconoscere qualsiasi oggetto di un tipo specifico. Per esempio, riconoscere qualunque mela, qualunque tazza, qualunque cane;

Prima di proseguire ricordiamo che la computer vision è il "rovescio" dell'image processing. Nel momento in cui si va ad effettuare l'elaborazione di un'immagine digitale 2D viene effettuata un'analisi di "**pattern recognition**" che porta alla comprensione dell'immagine. Ricorda che durante il processo di "digitalizzazione" **perdiamo alcune informazioni** come:

- Colori: Poiché la fotocamera è dotata di alcuni sensori che riconoscono solo i 3 colori principali (Rosso, Verde, Blu) e gli altri sono espressi come combinazione dei tre;

- Gamma dinamica: Poiché otteniamo un'immagine 2D, perdiamo l'idea della distanza.

Una delle operazioni principali che si effettuano nell'image processing è il **filtraggio delle immagini (image filtering)**. Il filtraggio delle immagini consiste nell'**applicare alcune funzioni a una patch di immagine locale** (un piccolo frammento nell'immagine) per ottenere un determinato risultato e recuperare, ad esempio, parti di informazioni che sono andate perse durante il processo di digitalizzazione. Quindi abbiamo un certo input che è un pezzo di immagine, effettuiamo una certa operazione su di essa e ricaviamo un output. Ad esempio: Dato un blocco 3x3 (con alcuni valori al suo interno), è possibile estrarre un numero particolare:



Con il filtraggio delle immagini è possibile:

- **Ridurre il rumore:** Partendo dal presupposto che ci sia una certa morbidezza (smoothness), se nel sensore della fotocamera è presente un pixel morto (cioè un pixel nero) è possibile cambiarlo con un pixel che abbia un colore simile a quelli circostanti;
- **Inserire valori/informazioni mancanti:** Come per il rumore ma qui non abbiamo informazioni. Possiamo, ad esempio, riempire un valore mancante decidendo quale usare in base alle informazioni circostanti;
- **Estrarre caratteristiche dell'immagine:** Può essere utilizzato per estrarre bordi/angoli tenendo presente che i colori sono "uniformi" tranne quando è presente qualche bordo (cioè c'è un improvviso cambio di colore);

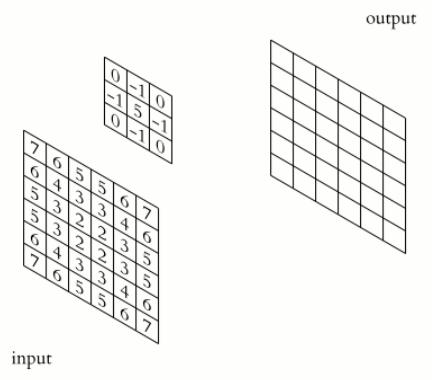
Un caso semplice di filtraggio delle immagini è il **filtraggio lineare (linear filtering)** che consiste nel sostituire ciascun pixel con una **combinazione lineare** dei suoi vicini. Un esempio di combinazione lineare è la **convoluzione 2D**:

Data un'immagine discreta I , i pixel nell'immagine sono rappresentati da due coordinate $[m, n]$ (ad esempio il numero al centro è $I[0, 0]$ mentre quello più a destra in alto è $I[-1, -1]$). Quindi, data una patch locale 3x3, gli indici i e k variano da -1 a 1 (come si può vedere nell'esempio a destra). Possiamo ora introdurre cos'è la convoluzione: Un **filtro digitale** (una piccola **maschera** di peso 2D, chiamata anche **kernel**) scorre sulle diverse posizioni di input. Per ciascuna posizione, viene generato un valore di output sostituendo il pixel sorgente con una somma ponderata di se stesso e dei pixel vicini. La maschera scorre sui nostri dati da sinistra a destra coprendo l'intera matrice di input. Formalmente, la **convoluzione** è il processo di **aggiunta di ciascun elemento dell'immagine ai suoi vicini locali, ponderati dal kernel**.

Nota: Si noti che questo concetto è fortemente correlato alla forma della convoluzione matematica. L'operazione sulla matrice eseguita (convoluzione) non è la tradizionale moltiplicazione di matrici, nonostante sia similmente indicata con $*$.

EX. Quindi nell'esempio sopra, stiamo calcolando la convoluzione per l'elemento centrale (quindi $k, l = 0$), quindi dobbiamo solo moltiplicare ciascun valore del filtro g

$$\begin{aligned}
 & \bullet \text{ 2D convolution (discrete): } f[m, n] = I \otimes g = \sum_{k,l} I[m-k, n-l]g[k, l] \\
 & \quad \begin{array}{l} \text{discrete Image: } I[m, n] \\ \text{filter 'kernel': } g[k, l] \\ \text{'filtered' image: } f[m, n] \end{array} \\
 & = \sum_{\substack{-1 < k < +1 \\ -1 < l < +1}} I[m-k, n-l]g[k, l] \\
 & = I[m+1, n+1]g[-1, -1] + I[m+1, n]g[-1, 0] + I[m+1, n-1]g[-1, +1] + \dots \\
 & \quad \text{SUM} \\
 & \quad \begin{array}{l} (k = -1, l = -1) \\ (k = -1, l = 0) \\ (k = -1, l = +1) \end{array} \\
 & \quad \begin{array}{l} \text{PRODUCT} \\ + \circ \\ + \circ \end{array} \\
 & \quad \begin{array}{l} \approx -1 \\ + 0 \\ + 3 \end{array} \\
 & \quad \begin{array}{l} \dots \\ 1 \end{array}
 \end{aligned}$$



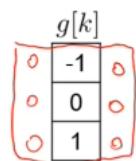
per il rispettivo valore nell'immagine I (ricorda di rispecchiare il filtro) quindi, una volta eseguite tutte le moltiplicazioni, somma semplicemente i risultati parziali:

- $k = -1, l = -1 \rightarrow I[m+1, n+1] g[-1, -1] = 1 * (-1) = -1;$
- $k = -1, l = 0 \rightarrow I[m+1, n] g[-1, 0] = 0 * 4 = 0;$
- $k = -1, l = +1 \rightarrow I[m+1, n-1] g[-1, +1] = 9 * 1 = 9;$
- ...
- $k = +1, l = +1 \rightarrow I[m-1, n-1] g[+1, +1] = 8 * 1 = 8;$

Alla fine si esegue la somma dei risultati parziali e il totale è 18 che viene sostituito all'elemento centrale nell'output.

È anche possibile realizzare una convoluzione (discreta) di un'immagine 2D utilizzando un filtro 1D:

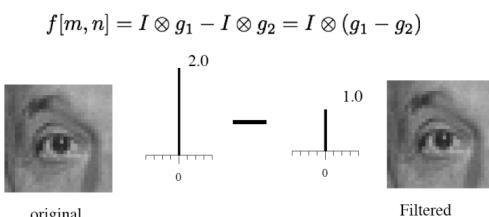
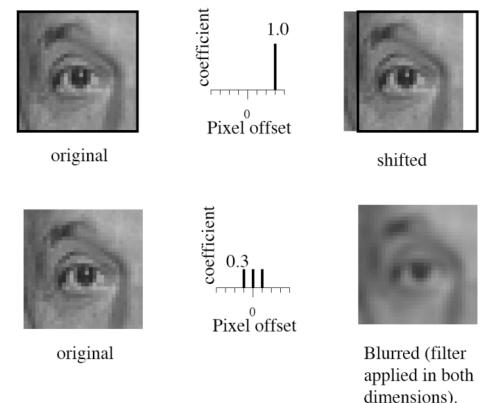
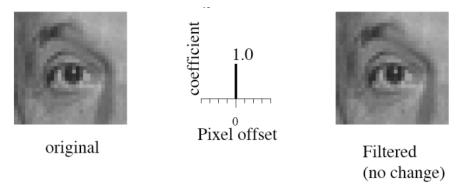
$$f[m, n] = I \otimes g = \sum_k I[m - k, n] g[k]$$



In questo caso scorriamo ancora il kernel ma l'unica direzione che conta è quella rappresentata dall'indice k . Si noti che il risultato è lo stesso dell'utilizzo di una matrice 3x3 con il filtro 1D al centro e tutti zeri su entrambi i lati (possiamo costruire questa matrice utilizzando lo zero padding a partire dal filtro 1D originale). Naturalmente il filtro 1D può essere anche orizzontale, e in questo caso considereremo l'indice l . **Un filtro 2D è spesso separabile in due filtri 1D** (uno orizzontale e uno verticale). L'approccio di separare un filtro 2D in due filtri 1D e applicarli separatamente all'immagine è solitamente preferibile in quanto **computazionalmente più efficiente**.

Alcuni esempi di applicazione della convoluzione 2D sono i seguenti:

- La nostra funzione f è rappresentativa come $f = I \otimes g$ (dove I è l'input originale e g è il kernel). Si noti che g può essere rappresentato come una matrice 1D $[0, 0, 0, 1, 0, 0, 0]$. Tieni presente che utilizzando questo filtro (ricordati di specchiarlo!), l'immagine non cambia affatto. Data una certa posizione nell'immagine, quello che facciamo è prendere il filtro, specchiarlo e scorrere sui k pixel dell'immagine (vicini) creando il prodotto tra il valore dell'immagine e quello del filtro. Infine la somma dei prodotti e, poiché nel filtro avevamo un solo 1 centrale, il risultato finale sarà il valore della posizione iniziale. Applicato all'intera immagine, questo ci dà l'immagine senza alcun cambiamento effettivo.
- In questo caso, poiché $g=[0, 0, 0, 0, 0, 0, 1]$, portiamo il pixel più a destra e lo posizioniamo a sinistra (a causa del mirroring). Il risultato è che spostiamo l'immagine a sinistra di 6 posizioni.
- In questo caso $g=[0, 0, 0.3, 0.3, 0.3, 0, 0]$ e quello che facciamo è ottenere tre pixel adiacenti e mescolarli con lo stesso coefficiente ottenendo lo stesso colore. Si noti che il livello generale di luminosità rimane lo stesso. In generale, dato un **impulso** nell'immagine (ovvero un picco nell'immagine), la sfocatura dell'immagine consente all'impulso di diffondersi sui tre pixel al centro. Dato un **bordo**, sfocare l'immagine renderà il bordo meno nitido (un bordo è un improvviso cambiamento di colore nell'immagine e la transizione tra i colori diventerà più fluida con la sfocatura).



Nota: Si noti che la convoluzione rispetta la proprietà associativa, quindi lo stesso risultato può essere ottenuto in questi modi:

- Applica 2 filtri all'immagine e sottrai i risultati parziali;
- Sottrai i filtri e applica il risultato all'immagine.

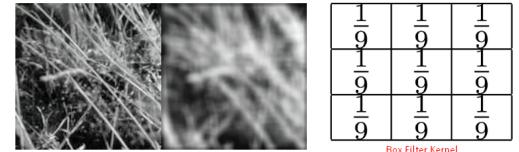
La **convoluzione** è un **filtro lineare** e alcune proprietà di base dei sistemi lineari sono:

- Omogeneità: $T[aX] = aT[X]$ (Applicare un filtro lineare T all'immagine X e poi moltiplicarlo per uno scalare a);

- Additività: $T[X_1 + X_2] = T[X_1] + T[X_2]$;
- Sovrapposizione (Superposition): $T[aX_1 + bX_2] = aT[X_1] + bT[X_2]$ (Combinazione di omogeneità e additività);
- Sistemi lineari \leftrightarrow Sovrapposizione (Un sistema è lineare se e solo se si applica la sovrapposizione).

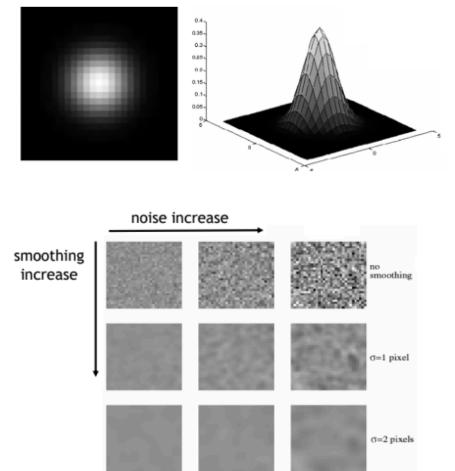
Il nostro obiettivo principale è **ridurre il rumore** che può essere basso (fluttuazioni di luce, rumore del sensore, ecc.) o complesso (ombre, oggetti estranei). Nel caso delle immagini, se assumiamo di avere del rumore, un modo per recuperare il segnale (e ridurre il rumore) è applicare un **average filter**. Ne esistono vari:

- Un filtro è il **BOX filter** che sostituisce ogni pixel con una media del suo quartiere. È fondamentalmente un filtro 2D 3x3 con voci positive che si sommano a 1 (quindi ogni elemento vale 1/9 che sommati fanno 1, si noti quindi che i pesi sono tutti uguali);
- Un altro modo per filtrare è il **Gaussian filter** è rotazionalmente simmetrico e pondera i pixel vicini più di quelli distanti. Quindi, ci sono pixel che contribuiscono in modo diverso: Dato un certo pixel, il suo vicino immediato contribuisce di più e questo contributo diminuisce man mano che ci allontaniamo da quel pixel. Questo dipende dalla σ che è anche chiamata deviazione standard (σ^2 è la varianza): Se riduciamo il sigma (la campana diventa più appuntita) facciamo la media in un'area più piccola (quindi abbiamo uno smoothing meno massiccio, terremo più dettaglio nell'immagine). Al contrario, con un sigma più grande filtriamo di più e avremo uno smoothing più massiccio.



- the pictures show a smoothing kernel proportional to

$$g(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$



Nota: Sia il filtro BOX che il filtro gaussiano sono separabili, il che significa che possiamo applicarli coinvolgendo prima ciascuna riga con un filtro 1D e poi coinvolgendo ciascuna colonna con un filtro 1D. In altre parole, entrambi i filtri possono essere espressi come la convoluzione di un filtro solo orizzontale per righe e di un filtro solo verticale.

$$(f_x \otimes f_y) \otimes I = f_x \otimes (f_y \otimes I)$$

Nuovamente, questa operazione viene effettuata perché computazionalmente più efficiente rispetto ad applicare un singolo filtro 2D.

DIM. Separabilità gaussiana: La separabilità gaussiana è una convoluzione gaussiana n dimensionale equivalente a n convoluzioni gaussiane 1D. $f(i,j)$ è la nostra immagine I e $g(i,j)$ è la funzione gaussiana 2D. L'output $h(i,j)$ è la convoluzione tra i due. Applichiamo quindi la definizione di convoluzione e sostituiamo $g(k,l)$ con la definizione di gaussiana. Ora, il termine k è indipendente da l quindi possiamo portarlo fuori e separarli. Ora abbiamo una gaussiana 1D in orizzontale. Lo calcoliamo e quindi il risultato è un filtro gaussiano 1D verticale. Fondamentalmente è possibile

Example: separable BOX filter

$$\begin{array}{|c|c|c|} \hline f_x \otimes f_y & f_x & f_y \\ \hline \begin{array}{|c|c|c|} \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array} & = & \begin{array}{|c|c|c|} \hline \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline \end{array} \otimes \begin{array}{|c|c|c|} \hline \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline \end{array} \\ \hline \end{array}$$

Example: Separable Gaussian

- Gaussian in x-direction

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

- Gaussian in y-direction

$$g(y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right)$$

- Gaussian in both directions

$$g(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

$$\begin{aligned} h(i, j) &= f(i, j) * g(i, j) = \\ &= \sum_{k=1}^m \sum_{l=1}^n g(k, l) f(i-k, j-l) = \\ &= \sum_{k=1}^m \sum_{l=1}^n e^{-\frac{(k^2+l^2)}{2\sigma^2}} f(i-k, j-l) = \\ &= \sum_{k=1}^m e^{-\frac{k^2}{2\sigma^2}} \underbrace{\sum_{l=1}^n e^{-\frac{l^2}{2\sigma^2}} f(i-k, j-l)}_{h'} = \\ &= \sum_{k=1}^m e^{-\frac{k^2}{2\sigma^2}} h'(i-k, j) \end{aligned}$$

I-D Gaussian horizontally

I-D Gaussian vertically

trasformare la seconda operazione nella convoluzione tra due filtri gaussiani 1D (uno orizzontale e uno verticale) quindi qualcosa del tipo $f(i,j) \otimes g_x(i) \otimes g_y(j)$.

1.4.2 Edge Detection

I **bordi** corrispondono a cambiamenti rapidi in cui l'entità (magnitude) della derivata è grande, quindi **se c'è un bordo e smussiamo l'immagine, il bordo sarà ancora lì**.

Se prendiamo la **derivata prima** dell'immagine smussata, i **punti massimo e minimo** corrispondono ai bordi.
Se prendiamo la **derivata seconda**, gli archi corrispondono agli **"zero crossing"**. Quindi la derivata è la soluzione per trovare i bordi nell'immagine. Facciamo un riepilogo di cosa è una derivata:

$$\frac{d}{dx} f(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \approx f(x+1) - f(x)$$

Questa è un'operazione facile da implementare con un filtro lineare convoluzionale: Per un'immagine avente una funzione f , basta prendere un punto $x+1$ con coefficiente 1 e x con coefficiente -1. Se vogliamo rendere il filtro simmetrico possiamo mettere uno 0 tra -1 e 1. Questo è utile quando abbiamo un numero pari di posizioni nel filtro e non vogliamo disallineare la derivata prima. Quindi aggiungiamo uno 0 in modo da avere un numero dispari e l'allineamento viene preservato (tra l'immagine originale e la differenziazione). Lo zero che aggiungiamo non cambia il calcolo. Per riassumere possiamo semplicemente implementare la derivata prima come un filtro lineare:

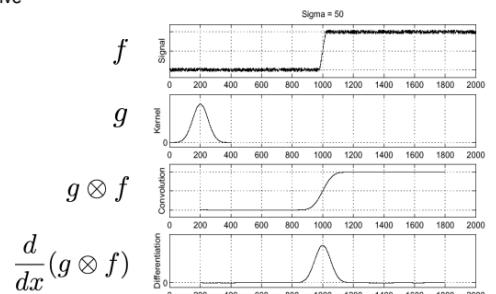
- Diretto: -1 1;
- Simmetrico: -1 0 1;

Quindi, per quanto riguarda la derivata prima: Prendiamo il segnale (che è rappresentato da una funzione f) e vogliamo rimuovere qualsiasi tremore nel segnale (dove è presente un rumore ad alta frequenza) per rilevare eventuali bordi. Quindi si noti che non applichiamo la derivata prima direttamente sull'immagine ma prima la smussiamo con un filtro gaussiano per rimuovere il rumore che altrimenti verrebbe amplificato dalle derivate (la derivata amplifica le alte frequenze). Quindi quello che facciamo è livellare applicando un kernel gaussiano. L'output è il segnale livellato. Successivamente applichiamo la derivata prima da cui possiamo trovare il bordo (saranno il massimo e il minimo della derivata).

Nota: Ricorda che le derivate come la convoluzione sono operazioni lineari, quindi possiamo salvare un'operazione calcolando la derivata del kernel g e quindi applicare la convoluzione tra il risultato e la funzione f . In questo modo, miglioriamo l'efficienza poiché facciamo molte meno moltiplicazioni.

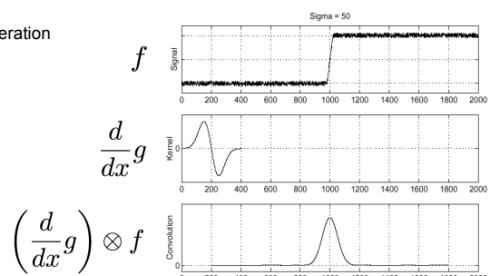
Possiamo trovare bordi anche usando la **derivata seconda**, in particolare quando la si applica si devono trovare gli **"zero crossing"**. Questi sono più efficienti rispetto alla ricerca di massimi e minimi perché nella derivata prima quando dobbiamo prendere il punto sopra un certo threshold, di solito ci ritroviamo con una linea spessa e quindi dobbiamo renderla più piccola attraverso tecniche come la non-maximum suppression. Nella derivata seconda dobbiamo solo cercare gli zero crossing che sono proprio il punto in cui l'immagine passa dal nero al bianco (quindi abbiamo meno calcoli da effettuare). Diamo quindi la definizione di derivata seconda:

- based on 1st derivative:
 - smooth with Gaussian
 - calculate derivative
 - finds its maxima



- Simplification: $\frac{d}{dx}(g \otimes f) = \left(\frac{d}{dx}g \right) \otimes f$
- remember: derivative as well as convolution are linear operations

- saves one operation



$$\begin{aligned} \frac{d^2}{dx^2} f(x) &= \lim_{h \rightarrow 0} \frac{\frac{d}{dx} f(x+h) - \frac{d}{dx} f(x)}{h} \approx \frac{d}{dx} f(x+1) - \frac{d}{dx} f(x) \\ &\approx f(x+2) - 2f(x+1) + f(x) \end{aligned}$$

2nd derivative:

1	-2	1
---	----	---

Ciò significa che possiamo considerare la derivata 2 come un filtro convoluzionale lineare per il quale i valori del kernel sono dati da (1 -2 1). Per individuare gli archi

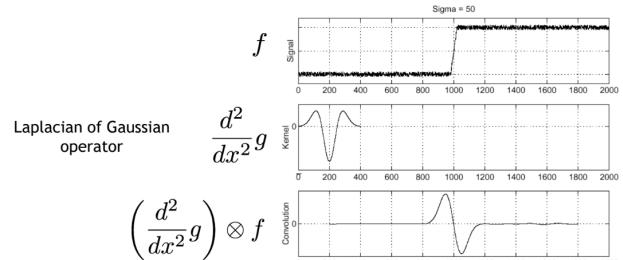
utilizziamo il **Laplaciano** che significa applicare la derivata 2 alla x, alla y e sommare i risultati.

Anche in questo caso, non vogliamo applicare il laplaciano

direttamente sull'immagine, infatti prima smussiamo l'immagine con una gaussiana G ma, per ragioni di efficienza, poiché il laplaciano è un filtro lineare possiamo combinarlo con una gaussiana e poi applicare il risultato all'immagine (anche qui possiamo farlo grazie alla proprietà associativa):

$$\nabla^2(G \otimes f) = \nabla^2G \otimes f$$

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$



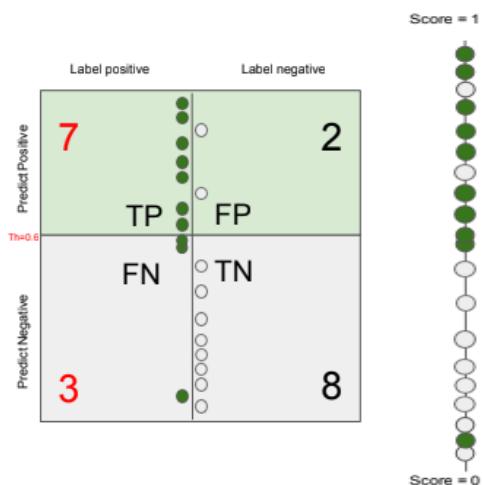
1.5 Performance Evaluation

La **valutazione delle prestazioni (performance evaluation)** ci fornisce gli strumenti per capire se un metodo di confronto è migliore di un altro. Come possiamo dire se il metodo A è migliore del metodo B per lo stesso compito? Potremmo:

- Confrontare un singolo numero: ad es. accuratezza (tasso di riconoscimento - quante volte il metodo riconosce effettivamente un oggetto), accuratezza top-k;
- Confrontare le curve: ad es. precision-recall, ROC curve;

Per quanto riguarda la valutazione basata sul punteggio, iniziamo definendo un po' di terminologia:

DEF. L'algoritmo di riconoscimento identifica (classifica) l'oggetto della query come corrispondente all'immagine di addestramento se la loro somiglianza è superiore a una soglia t.



Immaginiamo di dover classificare un numero di immagini in base alla somiglianza con l'oggetto della query. Abbiamo un punteggio compreso tra 1 (il più alto) e 0 (il più basso) che misura la confidenza del fatto che l'oggetto sia nell'immagine. Il punto pieno è dove l'oggetto si trova effettivamente nell'immagine, mentre il punto vuoto è dove non è nell'immagine. La decisione (se l'oggetto è nell'immagine) dipende da un certo threshold. Vorremmo avere dei posti vuoti in fondo alla partitura e dei posti pieni in alto.

Possiamo quindi suddividere i diversi casi in 4 quadranti: Le immagini possono avere etichetta positiva (se contengono effettivamente l'oggetto) o etichetta negativa (altrimenti). Possiamo prevedere positivo o negativo. I quadranti rappresentano tutti i casi possibili. **La qualità del modello e la soglia determinano il modo in cui le colonne vengono suddivise in righe. Vogliamo che le diagonali siano "pesanti", mentre le diagonali esterne siano "leggere".**

Quindi, tra le immagini abbiamo:

- **TP**: True Positive (c'è l'oggetto, noi prevediamo che ci sia - io prevedo come positivo e l'etichetta è effettivamente positiva);
- **TN**: True Negative (non c'è l'oggetto, prevediamo che non ci sia - prevedo come negativo e l'etichetta è effettivamente negativa);
- **FP** (errore di tipo I): False Positive (l'oggetto non c'è, prevediamo che ci sia - prevedo positivo e l'etichetta in realtà è negativa);
- **FN** (errore di tipo II): False Negative (c'è l'oggetto, prevediamo che non ci sia - io prevedo negativo e l'etichetta in realtà è positiva).

Vorremmo avere un numero più alto di TP e TN mentre dovrebbero esserci FP e FN bassi.

Ci sono alcune misure di prestazione che possiamo usare per dire quanto è buono un algoritmo.

- **Accuracy:** Ci dice quante classificazioni corrette eseguiamo nell'insieme dei test;
- **Precision:** Ci dice quanto è preciso/accurato il modello rispetto ai positivi previsti, quanti di essi sono positivi effettivi (ad esempio tra le immagini che diciamo contengono l'oggetto, quante di queste lo contengono effettivamente e quanto è preciso il modello);
- **Recall** (o Specificity): Calcola quanti dei positivi effettivi vengono catturati dal modello rispetto al numero totale di test etichettati come positivi (ad esempio tra le immagini che contengono un oggetto, quanti di questi abbiamo riconosciuto);
- **False Positive Rate:** Ci dice quanti risultati positivi verranno forniti quando il valore reale è negativo (ad esempio, tra le immagini che non contengono un oggetto, quante di quelle prevediamo che lo contengano);
- **F1-score:** Misura l'accuratezza di un test ed è un equilibrio tra precision e recall. Misura la probabilità di successo, ovvero quante volte il modello ha ragione in media.

$$\text{Overall accuracy} = \frac{(TN + TP)}{N}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \text{True positive rate} = \frac{TP}{TP + FN}$$

$$\text{False positive rate} = \frac{FP}{TN + FP} = 1 - \text{Specificity}$$

$$F_1 = \left(\frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} \right) = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Nota: Dobbiamo scegliere la nostra misura di prestazione in base all'applicazione. Ad esempio, nell'esempio sopra abbiamo una precisione davvero elevata (99%) quindi diremmo che il sistema è quasi perfetto nel fare previsioni. Ma se il modello deve prevedere se qualcuno è portatore di un virus, allora saremmo più interessati all'unico FN (quindi forse dovremmo considerare altre misure come il recall). Inoltre, esiste un compromesso tra precision e recall perché in genere l'uno riduce l'altro e viceversa (sono complementari). Solitamente aumentando il threshold si riducono i falsi positivi e si aumentano i falsi negativi e questa situazione fa aumentare la precisione e diminuire il recall (si verifica una situazione speculare se si diminuisce il threshold). Quindi, ancora una volta, dovremmo scegliere attentamente le nostre misure a seconda dell'applicazione (ad esempio, in uno scenario medico, vorremmo avere un ricordo troppo alto perché significa avere un numero minore di falsi negativi (che è la situazione peggiore che potrebbe verificarsi per un paziente perché significherebbe che alcuni pazienti positivi ad una determinata malattia, vengono definiti falsamente negativi)).

	Predicted/Classified	
	Negative	Positive
Actual		
Negative	998	0
Positive	1	1

Nota: Qualche osservazione:

- Banale recall al 100%: Porta tutto sopra il threshold;
- Banale precision al 100%: Porta tutto sotto il threshold tranne 1 verde in alto (sperando che non ci sia grigio sopra!);
- Cercare una buona precision con un recall del 100%: Portare il verde più basso il più in alto possibile nella classifica;
- Cercare di ottenere un buon recall con una precision del 100%: Spingere il grigio in alto il più in basso possibile nella classifica;

EX. Considerando l'immagine sopra, i risultati per tutte le metriche sono:

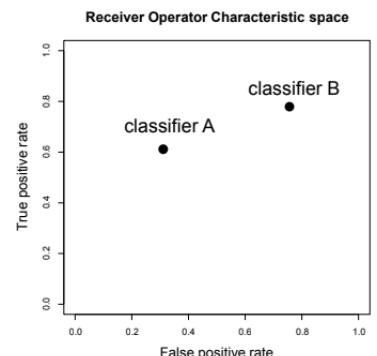
Th	TP	TN	FP	FN	Acc	Pr	Recall	Spec	F1
0.5	9	8	2	1	.85	.81	.9	.8	.857

Nota: Questi valori dipendono fortemente dal threshold. Nonostante il modello, dovremmo scegliere il threshold migliore che ci dà i migliori risultati (ad esempio nell'ultimo esempio dovrebbe essere $t \approx 0,45$). In generale il numero di threshold effettivi è pari al numero di data point + 1.

Quindi abbiamo ormai capito che selezionare il threshold t è un'operazione importante:

- Più basso è t , più immagini di query vengono classificate come corrispondenti (più TP ma anche più FP);
- Più alto è t , meno immagini di query vengono classificate come corrispondenti (più TN ma anche più FN).

Quale valore dovremmo scegliere per t ?



Dipende dal problema. Per avere una rappresentazione di come cambiano le prestazioni, cambiando il threshold, possiamo utilizzare il **Receiver Operator Characteristic (ROC)**, un grafico 2D dove abbiamo l'FPR sull'asse x e il TPR nell'asse y. Ricordati che:

- TPR: Maggiore è il TPR, maggiore è il richiamo delle effettive corrispondenze reali;
- FPR: Maggiore è l'FPR, maggiore è il numero di falsi allarmi;

Quindi, quando abbiamo un threshold basso siamo nella parte in alto a destra del grafico, mentre con una soglia alta siamo nella parte in basso a sinistra del grafico.

EX. L'immagine a destra è un esempio di classificazione di due classificatori A e B con ROC. Si noti che il classificatore A è migliore di B perché anche se ha un TPR leggermente inferiore, ha anche un FPR inferiore, il che è positivo.

Dobbiamo quindi scegliere il **miglior threshold t** per il **miglior compromesso**:

- Costo della mancata identificazione di un oggetto;
- Costo per la generazione dei falsi allarmi;

Come possiamo utilizzare la curva ROC per misurare le prestazioni di un metodo?

Dato un metodo, dobbiamo calcolare la quantità di TPR e FPR variando la soglia da 0 a 1.

Quello che otteniamo è una curva (**ROC curve**) che ci dice come operare. Data la curva, possiamo calcolare la **prestazione media di un metodo** prendendo l'area sotto la curva.

L'**area sotto la curva ROC (AUROC)** dà un'idea approssimativa di come il metodo si comporta indipendentemente dal threshold.

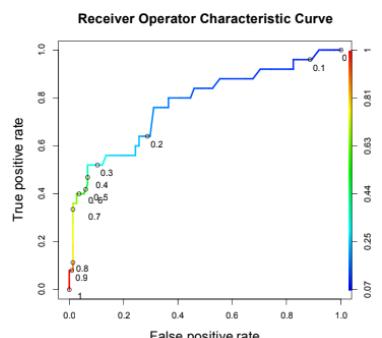
Come possiamo confrontare metodi diversi?

Una volta che abbiamo le curve ROC per, diciamo, 2 metodi, dobbiamo integrare l'area sotto la curva per entrambi i metodi e confrontarli. Quest'area sarà proporzionale al numero di volte in cui prenderemo la decisione giusta.

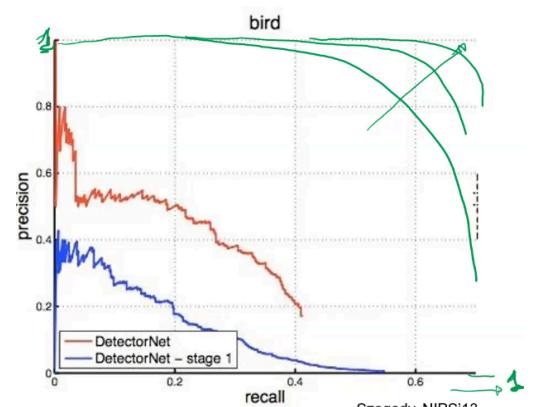
Qual è un buon AUROC?

- Massimo 1 (previsione perfetta);
- Minimo 0,5 (perché la somma di TPR e FPR deve essere 1, se è inferiore allora stiamo prendendo più decisioni sbagliate che giuste);

Nota: Un AUROC di 0,5 corrisponde a una curva diagonale (quindi una funzione lineare) e corrisponde più o meno a un'ipotesi casuale.



Un'altra curva è la **precision-recall curve** che è preferita per il **rilevamento**, dove i TN sono altrimenti indefiniti (questo significa che se dobbiamo rilevare un oggetto in un'immagine probabilmente avremo uno o pochi TP in cui troviamo quegli oggetti e molti TN in tutti gli altri punti dove l'oggetto non è presente). Quindi, siamo interessati al TPR (recall) ma non siamo interessati al FPR poiché sarebbe sempre vicino a 0 (ci sono molti TN quindi il denominatore è grande). In generale vorremmo avere una curva che si comporti come quella verde (quindi una curva che abbia sempre un'elevata precision e recall).



1.6 Image Classification, Linear Classifiers and Losses

La **classificazione delle immagini** è un compito fondamentale nella computer vision. Supponiamo di avere un dato insieme di etichette discrete (ad esempio cane, gatto, camion, ecc.), la classificazione delle immagini consiste nell'assegnare un'etichetta o una classe a un'intera immagine assumendo di avere una sola classe per ciascuna immagine (quindi dobbiamo dire, ad esempio, se l'elemento nella foto è un cane). Un'immagine è rappresentata da una griglia di numeri RGB, quindi assumendo di avere un'immagine $n \times m$ la dimensione della griglia è $n \times m \times 3$ (dove 3 sono i canali RGB). La classificazione delle immagini non è un compito facile, infatti dobbiamo affrontare alcune sfide:

- **Variazione del punto di vista:** Tutti i pixel dell'immagine cambiano quando la telecamera si muove.

- **Disordine dello sfondo:** Significa che l'immagine contiene un gran numero di cose, rendendo difficile per l'osservatore individuare l'oggetto desiderato (le immagini hanno molto rumore).
- **Illuminazione:** Il sistema dovrebbe essere in grado di far fronte anche alle variazioni di illuminazione. Pertanto, se offriamo al nostro sistema di classificazione delle immagini un'immagine dello stesso articolo con diversi livelli di luminosità (Illuminazione), il sistema dovrebbe essere in grado di assegnare loro la stessa etichetta.
- **Deformazione:** L'oggetto potrebbe apparire in una forma o posizione diversa rispetto a quelle abituali.
- **Occlusione:** L'oggetto potrebbe essere parzialmente occluso da un altro oggetto rendendone difficile l'identificazione.
- **Variazione intraclasse:** Variazioni all'interno della stessa classe (es. parenti, razze, ecc.).

Quindi dobbiamo inventare un classificatore di immagini che ci permetta di identificare un particolare oggetto nell'immagine. Esistono diversi approcci.

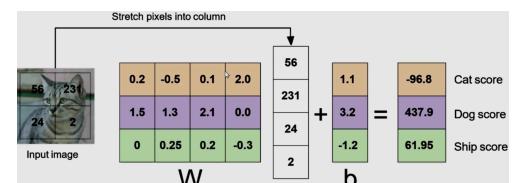
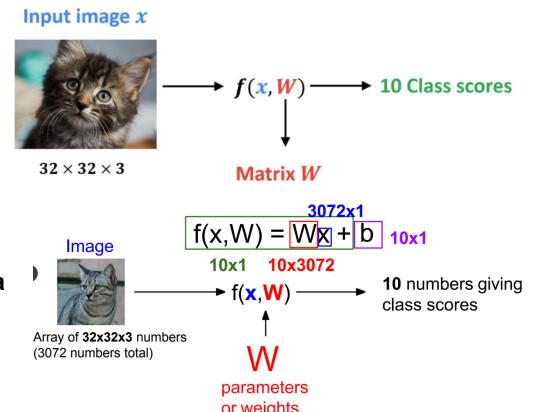
Nel tempo sono stati sperimentati vari algoritmi per effettuare la classificazione delle immagini ma non c'è stato modo di creare uno efficiente in maniera hard-coded. Un metodo efficiente che possiamo utilizzare prevede l'utilizzo del Machine Learning. Si tratta di un approccio "**Data-Driven**" in quanto:

- **Raccogliamo un set di dati di immagini ed etichette** (abbiamo etichette quindi sappiamo già qual è l'oggetto nell'immagine)
- Utilizziamo il ML per addestrare un classificatore
- Possiamo **valutare il classificatore su nuove immagini**.

Come possiamo utilizzare un approccio parametrico per effettuare una classificazione lineare?

Supponiamo di prendere una determinata immagine x con dimensione $32 \times 32 \times 3$ dal set di dati CIFAR-10. Abbiamo una funzione f che prende un'immagine x e un insieme di pesi W come input. L'immagine x è un'**immagine vettorizzata**, quindi prendiamo semplicemente ogni pixel dell'immagine e lo inseriamo in un vettore ad 1-dimensione (quindi la dimensione del vettore sarà 3072×1). W è un insieme di **pesi** c dove c è il numero di classi possibili (quindi supponendo di avere 10 classi la dimensione del vettore sarà 10×3072). La relazione tra il classificatore e il punteggio di output è lineare, quindi otteniamo Wx (nel nostro caso il vettore sarà 10×1) in cui otteniamo i **punteggi delle classi**. La **classe con il punteggio più alto è quella a cui appartiene l'immagine**.

Si noti che abbiamo anche un **vettore di polarizzazione b** (bias vector) che verrà utilizzato per **regolare gli iperpiani** prodotti da questa moltiplicazione. Si può pensare a b come a un termine di intercetta. Ad esempio, se x ha dimensione 2, avremo un piano nello spazio. Possiamo quindi spostare questo piano, su e giù per regolarlo e avere una classificazione ottimale. Tuttavia, in questo caso, b fungerà solo da parametro che aiuterà questi iperpiani. Il vettore b ha dimensione $1 \times c$ dove c è ancora il numero di classi (nel nostro caso $c=10$). Si noti nell'esempio a sinistra che la classificazione finale è errata in quanto ciò è dovuto al dato insieme di pesi e al vettore di distorsione. Ciò è dovuto al fatto che all'inizio partiamo con un'**inizializzazione casuale** (quindi impostiamo valori casuali per W e b) che ci dà più o meno gli stessi punteggi per tutte le classi. Quindi dobbiamo aggiustare W e b in modo da poter fare una classificazione corretta.



$$f(x, W) = x \cdot W + b$$

I classificatori lineari sono gli elementi costitutivi delle Reti Neurali. Vediamo adesso come possiamo allenare una Rete Neurale utilizzando tali classificatori. Finora abbiamo definito una funzione di punteggio (lineare) $f(x, W) = Wx + b$ ma dobbiamo ancora trovare un modo per quantificare quanto è buono W . Dobbiamo quindi definire una **funzione di perdita (loss function)** che quantifichi la nostra insoddisfazione rispetto ai punteggi ottenuti nei training data. Dato un set di dati di n esempi (x_i, y_i) , la loss rispetto al dataset è una media della loss ottenuta sui singoli esempi:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Quello che andiamo a fare è confrontare il ground-truth y_i con la predizione data dal classificatore lineare. Andiamo a calcolare la loss per l'esempio corrente (tra poco verrà spiegato come) ed infine facciamo la media di tutte le loss di tutti gli esempi nel dataset.

Vogliamo eseguire **MLE (Maximum Likelihood Estimation)**, il che significa che vogliamo massimizzare la probabilità dei nostri dati, quindi è conveniente interpretare i punteggi del classificatore come **probabilità**. Quindi, dato $s=f(x_i; W)$ il punteggio calcolato dell'i-esimo esempio, possiamo calcolare la probabilità di ciascuna classe utilizzando la **funzione Softmax**:

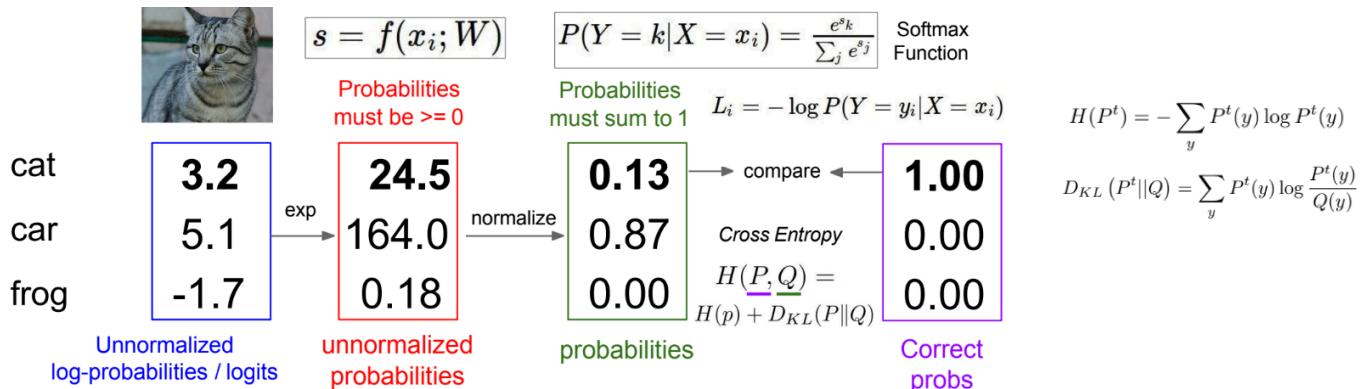
$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Dato un certo punteggio, prendiamo l'esponenziale di tale punteggio e lo andiamo a dividere per la somma di tutti i punteggi (effettuiamo una normalizzazione). Questo fa sì che i punteggi, espressi come probabilità, siano tutti positivi e si sommino ad 1. La **loss per l'i-esimo training example** è calcolata come:

$$L_i = -\log P(Y = y_i|X = x_i)$$

Massimizzare la probabilità dei training data equivale a confrontare con la **Cross Entropy** le probabilità per ciascuna classe che otteniamo con il one-hot vector che rappresenta il ground truth (il ground-truth viene rappresentato tipicamente con un vettore one-hot).

EX.



La Cross Entropy ci dice quanto vicini sono i due vettori di probabilità (ground truth e predizione). Questa corrisponde alla somma dell'entropy di P (vettore di probabilità che rappresenta il ground-truth) e la distanza tra i vettori P e Q (dove Q è la nostra predizione).

Quindi, nel nostro caso l'output target e l'output della rete neurale sono:

$$P^t(y) = \begin{cases} 1 & y = y_i \\ 0 & y \neq y_i \end{cases} \quad Q(y|x_i) = P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Quindi nel vettore P dei ground-truth abbiamo 0 ovunque tranne che nella classe corretta. Nel vettore Q della predizione Q invece abbiamo le probabilità calcolate grazie alla funzione di Softmax.

La **Cross-Entropy Loss** viene calcolata come:

$$L_i = L(x_i) = -\sum_y P^t(y) \log Q(y|x_i) = -1 \cdot \log Q(y_i|x_i) = -\log P(Y = y_i|X = x_i) = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Nota: La loss minima che otteniamo è 0 mentre questa non ha limiti

superiori (può eventualmente raggiungere infinito).

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \lambda R(W)$$

Nota: E' possibile applicare una **regolarizzazione** per evitare che il modello funzioni troppo bene con i dati di training. Questo viene fatto fondamentalmente per ridurre la possibilità di overfitting e:

- Eprimere preferenze sui pesi;
- Rendere semplice il modello in modo che funzioni sui dati di test (i.e. permette di evitare l'overfitting);
- Migliorare l'ottimizzazione aggiungendo curvatura;

Si noti che λ è un iper parametro. Alcuni esempi di funzioni di regolarizzazione sono:

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W)$$

Esistono anche formule più complesse (es. Dropout, Batch normalization, Stochastic depth, ecc.).

Come troviamo il miglior set di pesi W ?

Proprio come negli altri algoritmi, dobbiamo apportare alcune ottimizzazioni per ridurre la loss. In più dimensioni, il **gradiente** è il vettore di (derivate parziali) lungo ciascuna dimensione. La **pendenza** (slope) in qualsiasi direzione è il **prodotto scalare della direzione con la pendenza**. La direzione della discesa più ripida è il **gradiente negativo**. Per calcolare il gradiente e aggiornare i pesi, possiamo utilizzare un **approccio analitico** che prende il nome di **Gradient Descent**:

$$W_{t+1} = W_t + \alpha_t d_t \quad d_t = -\nabla f(W_t)$$

Fondamentalmente, quello che facciamo è muoverci nella direzione opposta al gradiente. Nel caso in cui abbiamo più dimensioni, ricordiamo che il gradiente punta sempre nella direzione ortogonale alla linea di contorno prima di finire al minimo.

Possiamo **ottimizzare il GD**, utilizzando GD stocastico o GD mini batch. Supponiamo che la loss sia:

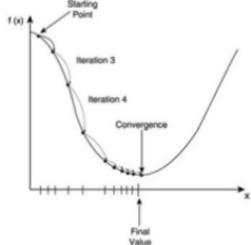
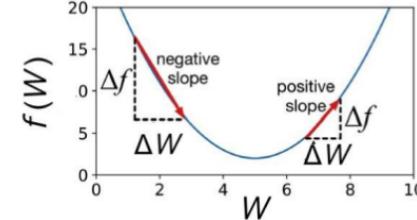
$$L(W) = \frac{1}{n} \sum_i^n L_i(W)$$

Con n il numero di campioni di addestramento e L_i la loss per il campione di addestramento x_i . Possiamo utilizzare diversi approcci:

- **Stochastic Gradient Descent:** Scegli casualmente un campione di addestramento x_i e aggiorna i pesi in base alla loss $L_i(W)$. Ricorda che in questo caso non abbiamo alcuna garanzia che la linea sarà ortogonale alla direzione del tracciato del contorno, ma alla fine otterremo lo stesso risultato del batch GD.

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$



- **Mini-batch training:** Elabora un sottoinsieme di campioni di addestramento $M \subset \{1, \dots, n\}$ e aggiorna i pesi in base a:

$$L_M(W) = \frac{1}{|M|} \sum_{i \in M} L_i(W)$$

Questo approccio è più veloce della GD batch e la convergenza è più fluida rispetto al metodo stocastico.

- **Batch training:** Elabora tutti i campioni di addestramento e aggiorna i pesi in base a:

$$L(W) = \frac{1}{n} \sum_i^n L_i(W)$$

Si tratta dell'approccio tradizionale che abbiamo visto all'inizio.

$$\frac{f(W)}{dW} = \lim_{h \rightarrow 0} \frac{f(W + h) - f(W)}{h}$$

Esiste anche un **approccio numerico** per il calcolo del gradiente che si **avvale semplicemente della definizione di derivata prima**. Quindi, dato un incremento h calcoliamo il valore della funzione in $f(W+h)$ e lo sottraiamo per $f(W)$, quindi infine lo dividiamo per h . Ora, notiamo che W è un vettore n -dimensionale così come h quindi, ad esempio, se

current W :	$W + h$ (second dim):	gradient dW :
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, loss 1.25347	[0.34, -1.11 + 0.0001 , 0.78, 0.12, 0.55, 2.81, loss 1.25353	[-2.5, 0.6 , ?, (1.25353 - 1.25347) / 0.0001 = 0.6

vogliamo incrementare la seconda dimensione, dobbiamo avere qualcosa come $H = [0, h, 0, 0, ..]$.

Questo metodo è lento perché è necessario ripeterlo per tutte le dimensioni (immagine ad una rete neurale da 1k parametri). Inoltre, **è un'approssimazione** perché stiamo calcolando il lim per $h \rightarrow 0$ ma siamo limitati dell'approssimazione numerica che possiamo avere. E' però facile da scrivere ed è utile per verificare l'implementazione quando si utilizza il metodo analitico (questo è chiamato **controllo del gradiente**).

Al contrario, un problema con il metodo analitico è che è **soggetto a errori**, nel senso che se ci troviamo in una grande NN con molti livelli, se cambiamo alcuni parametri allora tutto quello che abbiamo fatto deve essere buttato via e riscrivere di nuovo il gradiente: ecco perché abbiamo bisogno di un approccio migliore per calcolare il gradiente e questo è la **backpropagation**.

1.7 Neural Network e Backpropagation

Tipicamente, quando alleniamo un modello con le reti neurali, non abbiamo un singolo layer ma più layer. Esistono delle formule che ci permettono di calcolare gli score e di aggiornare i pesi di conseguenza. Una volta che abbiamo calcolato la loss L , possiamo calcolare il gradiente di L rispetto ai pesi W , e utilizzare poi i gradienti per aggiornarli (usando ad esempio la gradient descent come visto prima). Le reti neurali possono essere molto grandi e avere molti parametri, per questo motivo viene utilizzata la backpropagation. Vediamo adesso un caso semplice per introdurre le reti neurali e l'operazione di **backpropagation**.

Cominciamo guardando un esempio. Supponiamo di avere una funzione $f(x, y, z) = (x + y)z$. Abbiamo che $x=-2$, $y=5$, $z=-4$, quindi se calcoliamo la funzione con questi valori, il risultato è la loss. Dobbiamo essere in grado di calcolare il gradiente di f rispetto a tutti i parametri per cambiare i loro valori in modo che il valore di f cambi. Quindi, vogliamo:

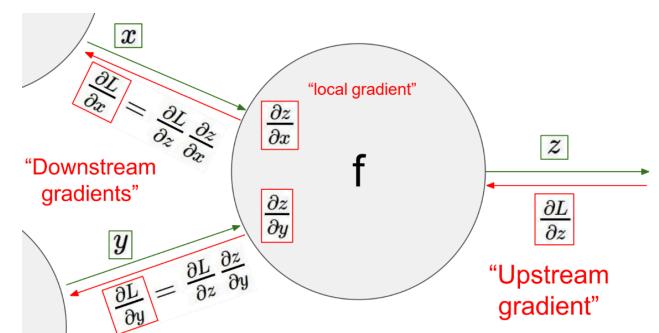
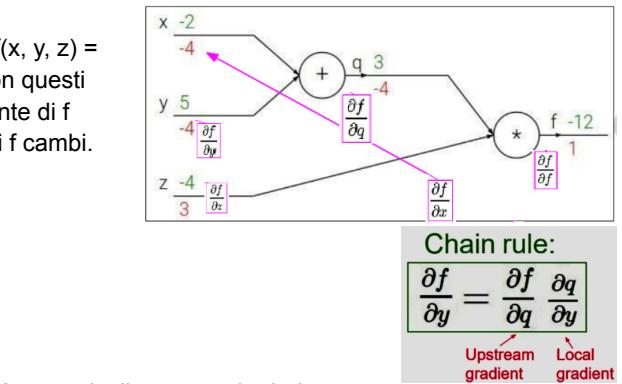
$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

Possiamo notare che:

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1 \quad f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Quindi, iniziamo dalla derivata di f e applichiamo la backpropagation finché non calcoliamo tutte le derivate e, infine, calcoliamo la nuova loss. Nel calcolare la derivata di f rispetto a y possiamo utilizzare il caso generale che consiste nell'utilizzare la **chain rule** effettuando la moltiplicazione tra l'**upstream gradient** e il **local gradient**. Quindi in questo caso avremo il prodotto tra quello che arriva da monte (-4) e il gradiente locale (1), il risultato infatti è -4. Lo stesso vale per la derivata di f rispetto a x .

Proviamo a dare una formulazione generale: Abbiamo un certo nodo e in quel nodo i **local gradients** sono le derivate di z rispetto a x e rispetto a y . Poi abbiamo un **upstream gradient** che risale dalle deep neural network e questo è la derivata della loss L rispetto a z (questo è tutto ciò che dobbiamo sapere per la backpropagation). Per quanto riguarda la derivata di L rispetto a x , essa verrà semplicemente calcolata come il prodotto del local gradient per l'upstream gradient (lo stesso per y). Quindi, la derivata di L rispetto a x , diventerà l'upstream gradient per lo layer successivo e così via. In generale, in ogni nodo in cui ci troviamo, partendo dalla loss e tornando indietro avremo sempre un upstream gradient e il local gradient che possiamo utilizzare per effettuare la backpropagation.



Nota: Si noti che per eseguire la backpropagation è necessario calcolare solo l'ultimo upstream gradient calcolato, ma sono necessari tutti i local gradient. Avendo questi valori e conoscendo le istruzioni per fare tutti i calcoli, abbiamo tutto per eseguire la backpropagation durante il training.

Si noti che abbiamo alcune **porte** (gates) e il calcolo eseguito in queste porte è praticamente sempre lo stesso:

- Add gate (+): È un distributore di gradiente, prendiamo l'upstream gradient e lo copiamo su tutti gli input;
- Mul gate (*): È un moltiplicatore di scambio, prendiamo l'upstream gradient e lo moltiplichiamo per l'altro dei due ingressi;
- Copy gate (): È un gradiente sommatore, copiamo le informazioni sommandole;
- Max gate (max): è un router a gradiente, retro propaghiamo il valore da cui proviene il massimo (nota che non possiamo dire nulla sul minimo, quindi è impostato su 0 durante la backpropagation).

EX. Supponiamo di avere alcuni valori per le x e le w . Vogliamo cambiare i pesi per diminuire la loss, quindi ci preoccupiamo solo di calcolare le derivate della loss rispetto ai pesi w . Per fare questo calcolo, dobbiamo conoscere la derivata di ciascuno degli elementi. Quindi nel primo nodo abbiamo che l'upstream gradient è 1 e il local gradient è calcolato come la derivata di f rispetto a x ($f(x)$ us 1/ x) cioè $-1/x^2$, quindi poiché $x=1.37$ allora il local gradient è $-1/1.37^2=-0.53$.

Realizziamo il prodotto tra il local gradient e l'upstream gradient e il risultato è sempre -0,53 (questo diventa l'upstream gradient per il layer successivo). Quindi, ancora una volta, il gradiente di $f(x)=x+c$ è solo 1, quindi il risultato è -0,53 (upstream gradient)*1=-0,53 e così via. Notiamo che ad un certo punto abbiamo un blocco (il fork): in quel caso calcoliamo il risultato semplicemente utilizzando lo stesso upstream gradient. Quindi, nel caso di w_2 il local gradient è 1, l'upstream gradient è 0,20 e il prodotto è ancora 0,20. Nell'ultimo quadrato dobbiamo moltiplicare l'upstream gradient 0,20 per il local gradient -1 (cioè la derivata di rispetto a $f(x)=ax$ cioè a) dell'altro nodo ottenendo $0.20*(-1)=-0.2$. Questi due ultimi esempi sono i cosiddetti gates, ce ne sono diversi e ognuno di essi ha un determinato comportamento (come abbiamo visto sopra).

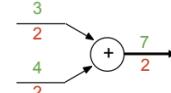
Quindi algoritmamente eseguiamo tutti i nodi in ordine topologico e una volta ottenuta la loss, li prendiamo in ordine inverso, prendiamo il gradiente a monte, lo moltiplichiamo per il local gradient e torniamo indietro. Quindi ripropaghamo ricorsivamente il gradiente su tutti i nodi seguendo i local gradient. Ancora una volta dobbiamo salvare alcuni valori da utilizzare all'indietro. Nell'esempio a destra dobbiamo salvare x e y che verranno utilizzati in fase di backpropagation per calcolare il loro gradiente (in tal caso utilizzeremo il gradiente di z calcolato in fase di esecuzione).

Finora abbiamo visto che possiamo avere derivate per scalari, quindi dati due scalari $x, y \in \mathbb{R}$, la derivata di y rispetto a x ci dice quanto cambia y se x viene incrementato un po'.

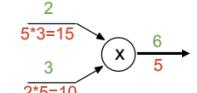
Cosa succede se introduciamo i vettori? Abbiamo due casi:

- Se $x \in \mathbb{R}^N, y \in \mathbb{R}$ (x è un vettore n dimensionale) allora la derivata di y rispetto a tutti i diversi x è il **gradiente** (quindi ci dice quanto cambia y se tutti gli x vengono incrementati un po').

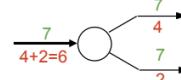
add gate: gradient distributor



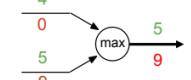
mul gate: "swap multiplier"



copy gate: gradient adder

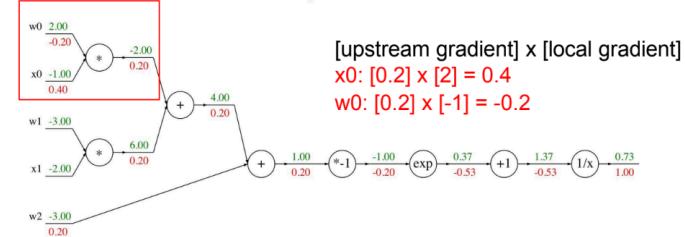


max gate: gradient router



Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2 x_2)}}$$



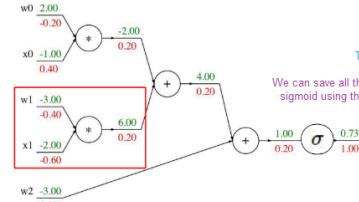
[upstream gradient] x [local gradient]
 $x_0: [0.2] \times [2] = 0.4$
 $w_0: [0.2] \times [-1] = -0.2$

$$\begin{array}{lll} f(x) = e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) = ax & \rightarrow & \frac{df}{dx} = a \end{array} \quad \left| \begin{array}{lll} f(x) = \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) = c + x & \rightarrow & \frac{df}{dx} = 1 \end{array} \right.$$

Backprop Implementation:
 "Flat" code

Forward pass:
 Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

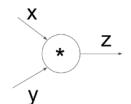


The gradient of the loss with respect to the loss is always 1.

We can save all the other operations by computing the sigmoid using this formula (so we use the loss)

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Multiply gate



$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

- Se $x \in \mathbb{R}^N$, $y \in \mathbb{R}^M$ (sono entrambi vettori) allora la derivata di ciascuna delle y rispetto a ciascuna delle x è la **jacobiana** cioè una matrice con tutte le possibili derivate (quindi ci dice quanto ciascuna delle y cambia se ogni x viene incrementato leggermente).

Come funziona la backpropagation con i vettori?

Supponiamo di avere un vettore in x (D_x dimensionale), un vettore in y (D_y dimensionale) e quindi esiste una funzione che ci restituisce z (D_z dimensionale). La derivata di L rispetto a z è D_z dimensionale poiché L è uno scalare e abbiamo che la derivata ci dice come cambia L quando tutti gli z cambiano leggermente. Per quanto riguarda i **local gradient** (ad esempio la derivata di z rispetto a x), sono **matrici jacobiane** poiché entrambi gli elementi sono vettori e possiamo avere più derivate nella matrice. La chain rule funziona ancora perché il local gradient ha dimensionalità $D_x * D_z$ e quando lo moltiplichiamo per D_z il risultato è D_x (ricorda le proprietà delle matrici).

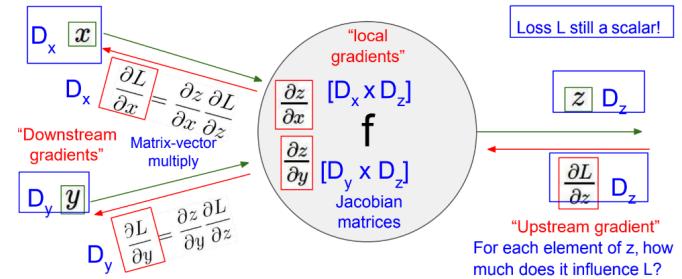
EX. Vediamo un esempio di come funziona: Abbiamo un input 4D x , una funzione $f(x) = \max(0, x)$ e l'output è un output 4D y . Eseguiamo la backpropagation quindi calcoliamo il gradiente di L rispetto a y utilizzando l'upstream gradient (quindi supponiamo di averne uno) e l'output è quello che vediamo. Ora, per la chain rule abbiamo bisogno della derivata di y rispetto a x , che è una matrice in cui tutti gli elementi sono sulla diagonale principale e abbiamo l'elemento i -esimo che è 1 o 0 a seconda se l'elemento fosse positivo o negativo. Lo moltiplichiamo per la derivata di L rispetto a y (che abbiamo) e otteniamo il risultato.

Il problema quando si lavora con le matrici è la memoria. Salvare le matrici Jacobiane può essere costoso quindi dobbiamo pensare a qualcosa di più intelligente. Nota nell'esempio precedente che lo **Jacobiano è sparso**, quindi le **voci fuori diagonale sono sempre zero e non serve salvarla tutta!** Per questo motivo possiamo semplicemente utilizzare la **moltiplicazione implicita**. Quindi possiamo semplicemente prendere l'upstream gradient così com'è e, anziché moltiplicarlo per la matrice jacobiana, possiamo semplicemente applicare una formula che rende 0 i valori negativi ottenendo esattamente lo stesso risultato di prima.

Come funziona la backpropagation con le matrici?

Supponiamo di avere una matrice x ($D_x * M_x$ dimensionale), un vettore in y ($D_y * M_y$ dimensionale) e quindi esiste una funzione che restituisce z ($D_z * M_z$ dimensionale). Per lo stesso motivo di prima, il gradiente della loss rispetto a z è $D_z * M_z$ dimensionale. Anche in questo caso, la chain rule funziona sempre: Supponendo di calcolare il gradiente di L rispetto a x , calcoliamo il local gradient che ha dimensionalità $(D_x * M_x) * (D_z * M_z)$ e poi moltiplichiamolo per l'upstream gradient ottenendo una matrice $D_x * M_x$ dimensionale. Il **problema è la dimensionalità!**

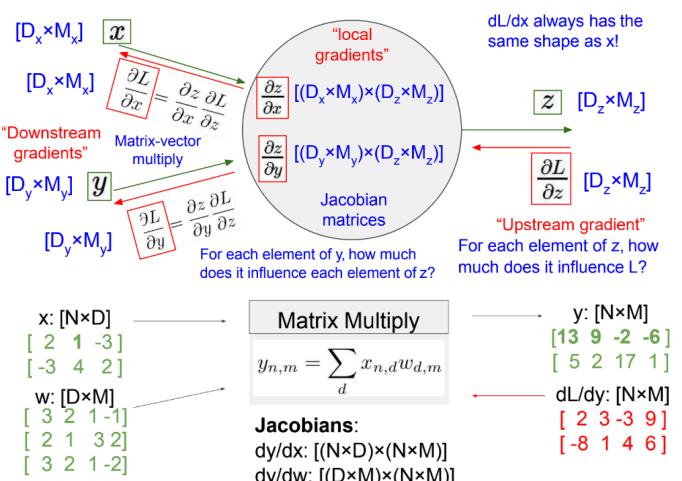
EX. Vediamo un esempio di come funziona: Abbiamo un input x $N*D$ e un input y $D*M$, una funzione che esegue la moltiplicazione di matrici e l'output y è $N*M$ dimensionale. Eseguiamo la backpropagation quindi calcoliamo il gradiente di L rispetto a y utilizzando l'upstream gradient (quindi assumiamo di nuovo di averne uno) e l'output è di nuovo $N*M$ dimensionale. Ora, per la chain rule abbiamo bisogno della derivata di y rispetto a x e della derivata di y rispetto a w che sono due Jacobiani e **occupano troppa memoria**.



$$\begin{array}{l} \text{4D input } x: \\ \begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow \\ \text{4D output } y: \\ \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix} \end{array} \quad f(x) = \max(0, x) \quad (\text{elementwise})$$

$$\begin{array}{l} \text{4D } dL/dx: \\ \begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} dy/dx \\ dL/dy \end{bmatrix} \\ \text{4D } dL/dy: \\ \begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} dy/dx \\ dL/dy \end{bmatrix} \end{array}$$

$$\begin{array}{l} \text{4D } dL/dx: \\ \begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} dy/dx \\ dL/dy \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \leftarrow \begin{pmatrix} \frac{\partial L}{\partial x} \end{pmatrix}_i = \begin{cases} \left(\frac{\partial L}{\partial y} \right)_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases} \\ \text{4D } dL/dy: \\ \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow \begin{bmatrix} dy/dx \\ dL/dy \end{bmatrix} \end{array}$$



Possiamo fare meglio?

Le matrici Jacobiane non servono. Possiamo utilizzare queste formule per effettuare la backpropagation:

[N×D] [N×M] [M×D]

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y} \right) w^T$$

[D×M] [D×N] [N×M]

$$\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y} \right)$$

Finora stiamo considerando una x che è un'immagine e un insieme di parametri W . Abbiamo una funzione di punteggio $f = Wx$ che calcola un punteggio per ogni classe e infine classifichiamo l'immagine con una classe. Questo può essere considerato come una **rete neurale 1-layer** più grande.

Ora, potremmo prendere l'output del primo prodotto W_1x , applicare una certa **non linearità** che è semplicemente il massimo tra 0 e l'output W_1x e moltiplicare il risultato per W_2 : questa è una **rete neurale a 2 layer**. Questa è chiamata anche "fully connected layer" infatti nel nostro caso ogni pixel corrisponde all'output.

Quindi, dato D il numero di pixel nell'immagine, il numero di classi qui è dato da $H*C$. W_1 è l'insieme di parametri che mappano da D a H . Potremmo avere anche altri strati, nel caso di 3 strati abbiamo:

Quindi mappiamo la dimensione H_1 con D , la dimensione H_2 con H_1 e C con H_2 .

EX. Riprendendo i primi esempi che abbiamo fatto introducendo l'argomento, partiamo da 3072 che sono tutti colori per tutti i pixel di un'immagine CIFAR-10, lo mappiamo con W_1 e otteniamo 100 e infine, visto che abbiamo 10 classi, mappiamo con C e otteniamo 10.

La funzione **max(0, z)** è detta **funzione di attivazione (activation function)**. Ciò è necessario perché dobbiamo ridurre la dimensionalità ad ogni passaggio. Se costruiamo una rete neurale senza, finiamo per avere di nuovo un classificatore lineare:

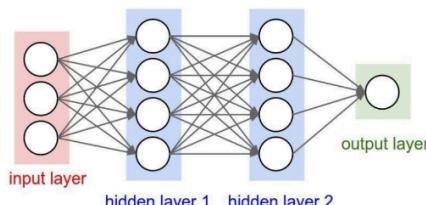
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

Ci sono poche funzioni di attivazione (non linearità) che possiamo considerare in una rete neurale:

Di solito **ReLU** è la scelta ideale per la maggior parte dei casi.

Nota: Se vogliamo codificare come funziona una rete neurale, assumendo che il sigmoid abbia la funzione di attivazione, dobbiamo semplicemente moltiplicare l'input del passaggio precedente con l'insieme di parametri fornito e aggiungere il bias. Quindi applichiamo la funzione sigmoidea. Infine, quando calcoliamo gli hidden layers e calcoliamo l'output neuron. Questo è come funziona il **forward pass**. Il **backward pass** è un po' più complicato.

Partiamo dalla loss che è il quadrato dei valori previsti e della ground truth. Calcoliamo i gradienti rispetto a w_1 e w_2 in modo da poterli modificare e arrivare, si spera, a diminuire la loss. Alla fine cambiamo w_1 e w_2 a seconda dei gradienti e del learning rate.



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

$$f = Wx$$

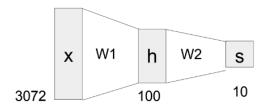
$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

$$f = W_2 \max(0, W_1 x)$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

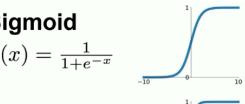
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$



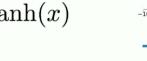
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



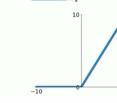
tanh

$$\tanh(x)$$



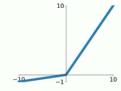
ReLU

$$\max(0, x)$$



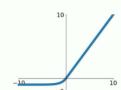
Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14 grad_y_pred = 2.0 * (y_pred - y)
15 grad_w2 = h.T.dot(grad_y_pred)
16 grad_h = grad_y_pred.dot(w2.T)
17 grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19 w1 -= 1e-4 * grad_w1
20 w2 -= 1e-4 * grad_w2
```

Cosa succede se impostiamo più capacità?

Se aggiungiamo più neuroni (hidden layers) alla Rete Neurale, stiamo sostanzialmente aggiungendo più parametri, quindi i confini che dividono i diversi campioni in classi diventano meno regolari e più complessi in modo che possano adattarsi perfettamente ai dati di addestramento (notare che tutti i dati di addestramento sono classificati correttamente e l'errore è 0). Come già sappiamo, questa non è necessariamente una buona cosa perché non abbiamo abbastanza generalizzazione, quindi ci stiamo adattando eccessivamente (**overfitting**). Vorremmo avere un gran numero di neuroni quindi un numero elevato di parametri evitando i confini complessi e possiamo ottenerlo applicando la **regolarizzazione** (in generale, maggiore è la regolarizzazione scegliendo una λ maggiore, più "regolari" saranno i confini).

Il **deep learning** si basa sulla disponibilità di set di dati di grandi dimensioni, sull'enorme potenza di calcolo parallelo e sui progressi nell'apprendimento automatico nel corso degli anni. L'approccio tradizionale consisteva nel fare l'estrazione delle caratteristiche spesso realizzata e fissata manualmente, per poi applicarle su un algoritmo ML per la classificazione. Nelle Reti Neurali invece si parla spesso di **sistema end-to-end** nel senso che la rete apprende le caratteristiche ed il classificatore tutti insieme come visto prima. Quindi, ancora una volta abbiamo un numero di strati completamente connessi con una certa parametrizzazione, e dopo ogni strato abbiamo alcuni risultati W più una certa non linearità (la funzione di attivazione), questo è il forward pass. Alla fine calcoliamo l'errore e partendo da questo propagheremo l'errore all'indietro per aggiornare i parametri.

Quindi le idee principali del Deep Learning sono:

- Eseguiamo l'estrazione delle funzionalità apprendendole (su molti livelli);
- Sistemi efficienti e addestrabili mediante elementi costitutivi differenziabili;
- Composizione di architetture profonde tramite moduli non lineari (utilizzando la funzione di attivazione);
- Formazione "end-to-end": nessuna differenziazione tra estrazione delle feature e classificazione. I parametri del modello di classificazione vengono appresi nello stesso modo in cui vengono appoggiate le caratteristiche;

2. Embodied AI

In un paper risalente al 1950, Alan Turing esprimeva il desiderio di poter riuscire a creare delle macchine che funzionavano come il cervello di un bambino quindi che potevano apprendere grazie all'educazione e via via crescere come e diventare un adulto. Seppur questa sia una vecchia affermazione, ricalca grossolanamente quelli che sono gli obiettivi dell'AI moderna. Ci sono alcuni step che hanno portato via via alla definizione di cos'è l'**Embodied Cognition** (Six Lessons from Babies - Linda Smith & Michael Gasser)

- Essere multimodale: Usare input sensoriali multipli (vista, suono, odore, ..);
- Esplorare: Muoversi in un certo ambiente (pensa ad un bambino che inizia a camminare e inizia a muoversi nell'ambiente e a conoscere quello che lo circonda);
- Essere incrementale: Apprendere con il tempo grazie ad approccio trial and error;
- Utilizzare un approccio fisico: Interagire con il mondo;
- Essere sociale: Interagire con altri agenti;
- Utilizzare il linguaggio: Per comunicare.

Quindi, l'**Embodied AI** si può definire come lo studio di **agenti intelligenti** che possono:

- Vedere: Oppure, in senso più ampio, percepire l'ambiente circostante attraverso la visione, l'udito o altro modality sensoriali;
- Parlare: Cioè impegnarsi in conversazioni in linguaggio naturale radicate nel loro ambiente;
- Ascoltare: Ovvero comprendere e rispondere all'input audio proveniente da varie parti;
- Agire: Ovvero navigare nel loro ambiente e interagire con esso;
- Ragionare: Cioè considerare le conseguenze a lungo termine delle loro azioni.

E' interessante introdurre il **paradosso di Moravec**. Per un macchina è molto più difficile effettuare task che per noi umani appaiono semplici (es. percepire e interagire con il mondo fisico come afferrare oggetti). Al contrario, la macchina riesce ad apprendere con maggiore facilità ragionamenti di alto livello e risoluzione di problemi astratti (es. si pensi agli scacchi). Una spiegazione a questo paradosso potrebbe essere il fatto che abbiamo bisogno di molti più dati per apprendere task statici piuttosto che quelli dinamici (J.Malik).

Alcuni malintesi dell'Embodied AI sono:

1. L'Embodied AI è solo robotica: Questa include anche agenti virtuali, come i chatbot, che interagiscono con il mondo fisico attraverso sensori e altri dispositivi;
2. L'Embodied AI si occupa solo del movimento: Questa implica anche percezione, cognizione e processo decisionale.
3. L'Embodied AI è solo per compiti fisici: ad esempio, l'intelligenza artificiale incorporata può essere utilizzata per creare assistenti virtuali che aiutano le persone con compiti come la pianificazione e gestione della posta elettronica.
4. Tutti i sistemi di intelligenza artificiale che interagiscono con il mondo fisico vengono considerati Embodied AI: Ad esempio, un'auto a guida autonoma può interagire con il mondo fisico, ma non ha necessariamente un corpo fisico che utilizza per interagire con l'ambiente.

Analizziamo ora qual è l'**impatto dell'Embodied AI**. Il campo dell'Embodied AI si concentra su come l'intelligenza emerge dalle **interazioni di un agente con il suo ambiente**. Si pensi all'azione di afferrare oggetti, la navigazione o la risposta a domande visuali (es. Sono finiti i cereali? L'Embodied IA può rispondere grazie alla vista). Tutti questi esempi si possono estendere a sistemi sanitari, fabbriche intelligenti e altro ancora (in generale tutte le situazioni in cui un agente esterno potrebbe interagire ed esplorare un ambiente).

Per quanto riguarda gli elementi costruttivi dell'Embodied IA, è bene fare una differenza con l'IA tradizionale. Nell'IA tradizionale gli algoritmi vengono utilizzati per analizzare i dati statici, è difficile controllare movimenti complessi e nella maggior parte dei casi non utilizza input sensoriali. L'embodied AI è progettata per interagire con l'ambiente, utilizza i dati sensoriali per adattarsi al suo ambiente e può essere utilizzata in un'ampia gamma di applicazioni, dai robot mobili agli agenti conversazionali.

Un caso studio ad alto livello che è bene introdurre è la **Point Goal Navigation**.

Nella Point Navigation un agente viene generato in una posizione iniziale e con un orientamento casuale in un ambiente mai visto prima. Un possibile **obiettivo** potrebbe essere chiedere all'agente di navigare verso le coordinate di destinazione specificate relative alla posizione di partenza dell'agente (ad es. "Vai 5 m a nord, 3 m a ovest rispetto all'inizio"). Per quanto riguarda l'input, non è disponibile alcuna mappa di verità e l'agente deve utilizzare solo il suo input sensoriale (una telecamera in prima persona RGB-D) per navigare.

Per sviluppare un sistema del genere occorre introdurre il **Reinforcement Learning**. I modelli di Reinforcement Learning apprendono da un **ambiente**.

L'ambiente ha una serie di regole e solitamente si presume che sia deterministico. L'interazione con l'ambiente avviene attraverso un **agente**. L'agente ha uno stato nell'ambiente e questo può eseguire **azioni** che modificano lo stato dell'agente nell'ambiente. Quando l'agente intraprende un'azione, l'ambiente la riceverà come input e restituirà lo **stato** risultante e una **ricompensa** (l'obiettivo dell'agente è in genere quello di massimizzare la somma cumulativa delle ricompense che riceve nel tempo, a cui spesso ci si riferisce come "ritorno" (return)).

Cosa ottimizziamo?

Una **politica (policy)** che ci dice quali azioni intraprendere in base allo stato attuale.

Per riassumere:

1. Un **agente** interagisce con l'**ambiente** attraverso azioni (determinate da una **policy**);
2. Queste **azioni** modificano lo **stato** dell'ambiente;
3. L'obiettivo del modello è determinare quali **azioni** porteranno alla massima **ricompensa**.

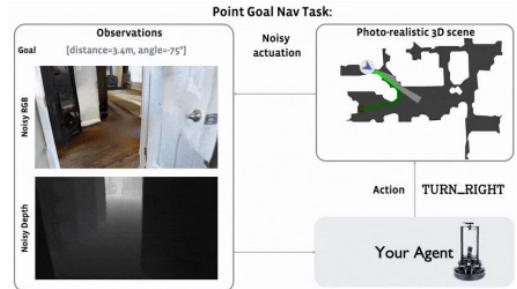
Per determinare l'azione migliore, il Reinforcement Learning stima il valore delle azioni. Il valore di un'azione indica quanto è buona un'azione, ad es. quanto è buono un movimento in un ambiente. Quindi il **Reinforcement Learning ruota attorno a questa idea di stima della funzione del valore ottimale**.

EX. Nell'esempio precedente, si pensi all'agente come un macchina in grado di muoversi nell'ambiente (es. Un appartamento). Le possibili azioni sono muoversi in avanti o ruotare e le possibili ricompense vengono assegnate quando ci si muove verso l'obiettivo e si evitano collisioni.

3. Convolutional Neural Networks

Supponiamo di dover rilevare un bordo in un'immagine 1000x1000. Se vogliamo verificare se esiste un bordo in ciascuna posizione, avremo bisogno di 1 milione di hidden units (tanti quanti sono i pixel nell'immagine). Questi ci descrivono un neurone che può essere in grado di descrivere una singola feature nell'immagine. Noi siamo più interessati a conoscere un milione di feature nell'immagine e questo richiede circa 10^{12} parametri (cioè un milione per un milione che è un numero enorme). Ora, il bordo è un'informazione locale (per trovare un bordo possiamo usare un filtro, non c'è bisogno di considerare l'intera immagine), quindi potremmo considerare filtri più piccoli (ad esempio 10x10), quindi in tal caso avremo bisogno solo di 100 milioni di parametri (perché rimuoviamo le cose che non utilizzeremo, utilizziamo solo patch locali dell'intera immagine). Quindi, per comprendere un'immagine possiamo semplicemente **guardare le informazioni locali (locality)**. Ora, si può fare un altro passo notando che le statistiche sono simili in diverse posizioni dell'immagine (ad esempio, se guardiamo un bordo verticale, ha gli stessi parametri ovunque lo troviamo, quindi il bordo verticale è lo stesso ovunque). Quindi possiamo **condividere gli stessi parametri in luoghi diversi** dell'immagine **utilizzando più filtri (stationarity)**. In questo caso, utilizzando 100 filtri 10x10, avremo bisogno solo di 10k parametri.

In sostanza, prendiamo dei filtri (ad esempio quelli che ci permettono di rilevare un bordo nell'immagine) di piccole dimensioni e li mettiamo in convoluzione con l'immagine.



Pertanto, la rete neurale standard applicata alle immagini si ridimensiona quadraticamente con la dimensione dell'input e non sfrutta la stazionarietà. La soluzione è connettere ciascuna unità nascosta a una piccola porzione dell'input e condividere i pesi attraverso l'unità nascosta: Questa è chiamata **Convolutional Neural Network**.

Per riassumere:

- Località (Locality): La "località" in una CNN si riferisce al principio che i neuroni in uno specifico layer della rete dovrebbero rispondere principalmente a caratteristiche localizzate in un'area ristretta dell'input. Questo significa che i neuroni sono specializzati nell'individuare pattern o feature all'interno di una regione specifica dell'immagine di input, come bordi, linee, texture o forme. Questo concetto di località è fondamentale perché consente alla CNN di catturare dettagli spaziali locali, che sono le componenti di base per la comprensione delle immagini. Nei layer iniziali di una CNN, i neuroni hanno receptive field (campo visivo) piccoli e rispondono a dettagli molto locali. Man mano che ci si sposta verso i layer più profondi, il campo visivo dei neuroni si allarga per catturare relazioni spaziali più complesse.
- Stazionarietà (Stationarity): La "stazionarietà" in una CNN si riferisce al principio secondo cui le stesse operazioni di convoluzione e pooling vengono applicate in modo identico in tutto l'input o in diverse parti dell'input. In altre parole, le operazioni di convoluzione e pooling sono invarianti rispetto alla posizione. Questo principio è importante perché permette alla CNN di rilevare le stesse caratteristiche o pattern indipendentemente dalla loro posizione nell'immagine. Ad esempio, se una CNN è addestrata per riconoscere gatti in foto, la stazionarietà garantirà che il modello possa riconoscere un gatto ovunque si trovi nell'immagine, anziché solo in una posizione specifica.

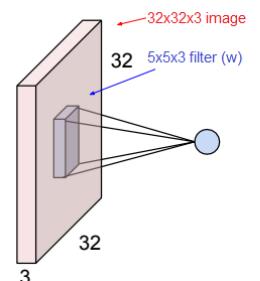
Oltre a questo, si può decidere di raggruppare le risposte dei filtri (ad esempio prendendo il massimo o la media) in posizioni diverse ottenendo robustezza rispetto all'esatta posizione spaziale delle feature. Questa operazione viene chiamata **pooling**. Più in dettaglio, il pooling è un'operazione utilizzata per ridurre la dimensione spaziale dei dati di input, riducendo il numero di parametri e calcoli nella rete. Il pooling è comunemente utilizzato dopo le operazioni di convoluzione per estrarre le caratteristiche salienti dell'immagine di input.

L'operazione di pooling coinvolge una piccola finestra o un kernel, di solito 2x2 o 3x3, che scorre sull'immagine di input. Durante questo processo, il pooling estrae una rappresentazione semplificata dell'area coperta dalla finestra. Le operazioni di pooling più comuni sono il max pooling e il mean pooling (o average pooling). Il pooling è spesso utilizzato per ridurre la dimensione spaziale dei dati e rendere il modello meno sensibile alle piccole variazioni locali nell'immagine. Inoltre, aiuta a ridurre il numero di parametri nella rete, il che può portare a una maggiore generalizzazione e a una minore possibilità di overfitting.

Un parametro importante da specificare quando si applica il pooling è la dimensione della finestra di pooling e il passo (**stride**) con cui viene spostata lungo l'immagine. Questi parametri determinano quanto l'immagine verrà ridotta in dimensione. Ad esempio, un pooling 2x2 con un passo di 2 ridurrà l'immagine in input alla metà delle dimensioni originali.

Prima di spiegare più in dettaglio come funziona una CNN, è necessario introdurre la struttura tipica. Data un'immagine in input, effettuiamo l'operazione di convoluzione sull'input, applichiamo l'activation function (**non-linearity**) come ad esempio la ReLU e infine l'operazione di pooling per aggregare informazioni e ridurre la risoluzione. Quella appena descritta è l'**operazione di feed forward**. Una volta ottenuto l'output possiamo calcolare l'errore (si noti che siamo nel dominio della supervised learning) e, una volta calcolato, l'obiettivo è quello di ridurlo il più possibile utilizzando l'**operazione di backpropagation** con il gradient descent.

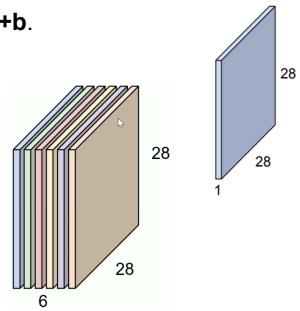
Riassumiamo in breve come funzionano le fully connected layer: Supponiamo di avere un'immagine 32x32x3. L'input sarà l'immagine appiattita 3072x1 (si deve applicare l'operazione di flattening) per calcolare il prodotto con il vettore W dei pesi e si ottiene un vettore 10x3072. Infine calcoliamo l'attivazione (un neurone che è il risultato della moltiplicazione tra x e W) che restituisce il punteggio per le 10 classi. Ancora una volta, questo è un **fully connected layer** poiché ogni numero calcolato dall'attivazione è collegato a tutti i valori R, G, B nei pixel dell'immagine (quindi ogni elemento nell'attivazione è il risultato del prodotto scalare tra una riga di W e l'input, un prodotto scalare di 3072 dimensioni).



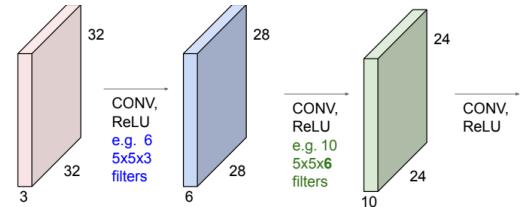
Nella CNN non appiattiamo l'immagine in un vettore poiché si vuole **mantenere la struttura spaziale dell'immagine**. Quindi supponiamo di avere un'immagine $32 \times 32 \times 3$. Abbiamo un filtro $5 \times 5 \times 3$ e lo convolviamo con l'immagine facendolo scorrere sull'immagine calcolando "spazialmente" i prodotti scalari. Si noti che **la profondità tra l'immagine e il filtro corrisponde sempre**, quindi nel kernel abbiamo anche un valore per tutti gli R, G, B. Per ogni posizione dell'immagine in cui effettuiamo l'operazione di convoluzione con il filtro (ricorda che facciamo scorrere il filtro sull'immagine), otteniamo un **valore** che è il risultato del prodotto scalare tra il filtro e una piccola porzione $5 \times 5 \times 3$ dell'immagine (ovvero $5^2 \times 3 =$ prodotto scalare a 75 dimensioni + bias): $\mathbf{w}^T \cdot \mathbf{b}$. Eseguendo la convoluzione su tutte le posizioni spaziali tra l'immagine in input e il filtro, arriviamo a una nuova immagine $28 \times 28 \times 1$ chiamata **activation map** (non sono altro che le feature, un vettore che ci permette di capire come identificare una certa caratteristica nell'immagine).

Si noti che perdiamo una posizione in tutti gli angoli a causa dell'applicazione del filtro: Se vogliamo mantenere la stessa dimensione dell'immagine iniziale possiamo applicare l'operazione di **padding** aggiungendo un angolo composto da tutti 0 attorno all'immagine. Chiaramente l'operazione di padding non è totalmente affidabile in quanto si sta aggiungendo informazione parziale. Ci sono vari motivi per cui si effettua l'operazione di padding:

- **Mantenimento delle dimensioni:** L'operazione di convoluzione senza padding ridurrebbe gradualmente le dimensioni dell'immagine mano a mano che la convoluzione scorre attraverso di essa. Questo potrebbe portare a una riduzione eccessiva delle dimensioni, specialmente nelle deep neural networks. Applicando il padding, è possibile preservare le dimensioni dell'immagine in output dopo la convoluzione, il che può essere importante per mantenere informazioni spaziali cruciali.
- **Gestione dei bordi:** Senza padding, i pixel ai bordi dell'immagine avrebbero meno opportunità di influenzare l'output della convoluzione rispetto ai pixel al centro. Applicando il padding, si assicura che i pixel dei bordi siano trattati in modo simile a quelli nel centro, migliorando la rappresentazione delle caratteristiche lungo tutto l'input.
- **Controllo della dimensione dell'output:** L'uso del padding consente di controllare la dimensione dell'output della convoluzione. Questo può essere importante per garantire che l'output abbia le dimensioni desiderate e sia compatibile con le successive operazioni nella rete.



Ora, la funzione di attivazione che abbiamo calcolato può essere utilizzata per rilevare qualcosa nell'immagine (ad esempio i bordi verticali). Se vogliamo **rilevare più cose contemporaneamente** (ad esempio vogliamo rilevare anche i bordi orizzontali) possiamo calcolare altre mappe di attivazione. Quindi **aggiungiamo altri canali quanti sono i filtri che consideriamo** ciascuno di dimensione $28 \times 28 \times 1$ (es. se consideriamo 6 mappe di attivazione arriveremo ad un'immagine $28 \times 28 \times 6$).



Dal primo livello di convoluzione possiamo ottenere una mappa di attivazione che assomiglia a un'immagine con l'unica differenza che perdiamo i canali RGB per k canali dove k è il numero di kernel convoluzionali (ad esempio 6 nell'ultimo esempio). Perché i 3 canali dell'immagine originale diventano un solo canale nella mappa di attivazione? Il motivo è da ricercarsi nel fatto che quando si effettua l'operazione di convoluzione effettuiamo la somma di R, G, B.

Possiamo quindi definire un nuovo layer convoluzionale e così via (si noti che dobbiamo usare filtri con profondità pari a quella della mappa di attivazione come già definito poco fa, es. 6 nell'esempio precedente). Si ricorda inoltre che in una fully connected layer dobbiamo applicare la **non linearità** nel mezzo altrimenti ci ritroveremo con un solo strato. Lo stesso problema si verifica nella CNN a causa dell'**associatività delle convoluzioni**: Dati x un'immagine e w_1, w_2 due kernel, possiamo applicare $[x(w_1)]w_2$ (quindi potremmo applicare il primo kernel all'immagine e poi il risultato alla seconda immagine) ma anche $(w_1w_2)x$ (quindi moltiplichiamo i kernel ottenendo un nuovo kernel w_3 e poi applichiamo all'immagine ottenendo un singolo layer). Quindi applichiamo ReLU che restituisce uno 0 se abbiamo un valore negativo, altrimenti il valore effettivo. In generale, questa operazione è necessaria perché molte relazioni tra le caratteristiche dei dati non possono essere modellate efficacemente da una semplice combinazione lineare di pesi. Ecco alcune ragioni chiave per cui si utilizzano le funzioni di attivazione tra i layer in una CNN:

- **Introduzione della non linearità:** Le funzioni di attivazione introducono non linearità nei neuroni della rete. Questo significa che la rete può apprendere a rappresentare relazioni complesse tra le caratteristiche dei dati. Senza non linearità, una rete neurale sarebbe essenzialmente una combinazione lineare di pesi e bias, il che limiterebbe notevolmente la sua capacità di apprendimento.
- **Mappatura delle attivazioni a valori reali:** Le funzioni di attivazione trasformano l'output dei neuroni in valori compresi in un determinato intervallo o in una distribuzione specifica. Ad esempio, la funzione ReLU (Rectified Linear Activation) mappa gli input negativi a zero e gli input positivi rimangono invariati. Questa mappatura è utile per diverse ragioni, tra cui la stabilità numerica e la facilità di ottimizzazione durante l'addestramento.
- **Apprendimento di rappresentazioni più complesse:** Le reti neurali profonde, come le CNN, sono in grado di apprendere rappresentazioni gerarchiche e complesse dei dati. Le funzioni di attivazione permettono a ciascun neurone di apprendere diversi tipi di pattern o feature dai dati di input, contribuendo così alla creazione di rappresentazioni sempre più sofisticate.

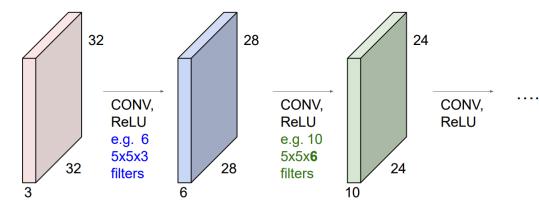
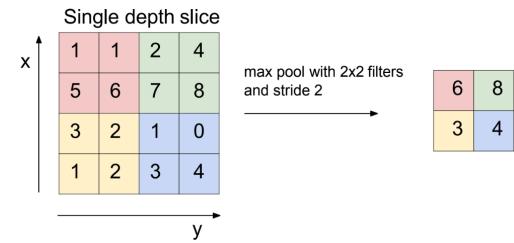
Nota: Se scegliamo un filtro più grande (diciamo un filtro grande quanto l'immagine in input) avremo un'operazione equivalente ad una completamente connessa. In tal caso otterremo una mappa di attivazione 1×1 e non è altro che come avere un layer 1D completamente connesso. L'output è una funzione di tutti i pixel nell'immagine. Se consideriamo patch più piccole, l'output sarà funzione di un numero minore di pixel (quelli corrispondenti alla patch) ma se consideriamo più layer, l'attivazione successiva nella rete sarà funzione di un percorso più ampio nella rete. immagine in ingresso.

Per rendere le rappresentazioni più piccole e più gestibili possiamo applicare un **livello di pooling** che opera su ciascuna mappa di attivazione in modo indipendente (quindi va applicato in ciascun canale in modo indipendente, ottenendo lo stesso numero di canali nella nuova immagine sottocampionata). Un esempio è il max pooling: supponiamo di avere una determinata mappa di attivazione, se consideriamo i filtri 2×2 e lo stride 2 durante il pooling, per ciascuno dei blocchi consideriamo il massimo. L'effetto è che otteniamo un'altra mappa di attivazione grande la metà. Il pooling è un'alternativa alla convoluzione con stride. Ultimamente si sta sempre di più abbandonando l'operazione di pooling a favore della convoluzione con stride. Ci sono alcune differenze:

- **Convoluzione con Stride:** La convoluzione con stride è un'operazione utilizzata per estrarre le caratteristiche dell'immagine di input. Durante questa operazione, una piccola finestra (kernel o filtro) scorre sull'immagine di input in modo da moltiplicare i valori dei pixel nell'area di copertura della finestra con i pesi del kernel e sommare il risultato. La convoluzione con stride consente di creare una mappa delle caratteristiche (feature map) che evidenzia le caratteristiche rilevanti dell'immagine, come bordi, texture e forme. La convoluzione con stride può ridurre le dimensioni dell'immagine in uscita rispetto all'input, a meno che non venga utilizzato il padding. Il passo (stride) determina di quanti pixel la finestra di convoluzione si sposta durante ciascuna iterazione.
- **Pooling:** Il pooling è un'operazione utilizzata per ridurre la dimensione delle feature map generate dalla convoluzione. Il suo scopo principale è quello di semplificare la rappresentazione dei dati, riducendo la dimensione spaziale e il numero di parametri da elaborare nei layer successivi. Il pooling seleziona un valore (ad esempio, il massimo o la media) da un'area specifica dell'immagine di input coperta da una finestra di pooling. Questo valore rappresenta una versione semplificata delle informazioni in quell'area. Il pooling riduce le dimensioni delle feature map riducendo il numero di pixel considerati in ciascuna finestra di pooling. Tuttavia, il pooling di solito non modifica il numero di canali nelle feature map.

Nota: Se si prende un'immagine 32×32 e la si mette in convoluzione con 10 filtri 5×5 , questa operazione riduce spazialmente i volumi ($32 \rightarrow 28 \rightarrow 24 \rightarrow \dots$). In generale, restringere l'immagine troppo velocemente non va bene. Quello che è buona norma fare è applicare uno (zero) padding uguale a 2 e applicare la convoluzione con uno stride pari ad 1 se vogliamo mantenere la dimensione spaziale. Possiamo calcolare la grandezza dell'output come:

(N - F) / stride + 1



Con N il numero di pixel in una dimensione dell'immagine, F la grandezza del kernel e $stride$ la quantità di stride da applicare all'immagine durante la convoluzione.

EX. Nell'esempio precedente:

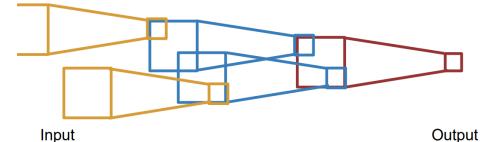
Output volume size: $(32+2*2-5)/1+1 = 32$ spatially, so $32x32x10$

Come facciamo a calcolare il numero di parametri?

Ogni filtro ha $5*5$ (dim. filtri) * 3 (canali) + 1 (bias) = 76 parametri * 10 (filtri) = 760

Nota: In una CNN, il termine "receptive field" o "receptive filter" si riferisce all'area dell'input che influenza la risposta di un neurone in uno specifico layer della CNN. Ogni neurone in uno specifico layer di una CNN è associato a un receptive field, che è una regione dell'input che contribuisce alla sua attivazione. Il receptive field è definito dalla dimensione del kernel utilizzato nelle operazioni di convoluzione e dal passo (stride) con cui il kernel scorre sull'input durante la convoluzione. Funziona in questo modo:

- Layer iniziali: Nei layer iniziali di una CNN, i neuroni hanno un piccolo receptive field. Ad esempio, se si utilizza un kernel $3x3$ nelle operazioni di convoluzione, ciascun neurone risponderà solo a una piccola regione $3x3$ dell'input. Inizialmente, questi neuroni catturano dettagli locali come bordi, linee e piccole texture.
- Layer successivi: Man mano che ci si sposta verso i layer più profondi della CNN, i neuroni hanno receptive field sempre più ampi. Questo perché i layer successivi combinano le informazioni estratte dai layer precedenti per catturare relazioni spaziali più complesse nelle immagini. I neuroni in layer profondi hanno un campo visivo più ampio che copre un'area più estesa dell'input. Ogni successiva convoluzione aggiunge $K - 1$ alla dimensione del receptive field. Con L layer la dimensione del receptive field è $1 + L * (K - 1)$.



Il receptive field è importante perché determina quali caratteristiche dell'input sono prese in considerazione da ciascun neurone e come vengono elaborate. Aumentando il receptive field nei layer più profondi, la CNN è in grado di catturare caratteristiche gerarchiche sempre più complesse e di apprendere rappresentazioni più astratte dell'input. Chiaramente c'è un problema perché per **immagini di grandi dimensioni abbiamo bisogno di molti layer** per ciascun output per "vedere" l'intera immagine. Per risolvere questo problema **possiamo applicare un downsampling all'immagine** con, ad esempio, operazione di **stride**. Ogni volta che si effettua downsampling dell'immagine, si duplica la finestra del receptive field e dopo si procede linearmente nei filtri che seguono.

SUM. Assumiamo che abbiamo un input $W_1 \times H_1 \times C$. Il layer di una CNN richiede: K filtri di grandezza F . Inoltre si deve specificare lo stride S e lo zero padding P . L'operazione di convoluzione produrrà un output $W_2 \times H_2 \times K$ dove:

- $W_2 = (W_1 - F + 2P)/S + 1$;
- $H_2 = (H_1 - F + 2P)/S + 1$.

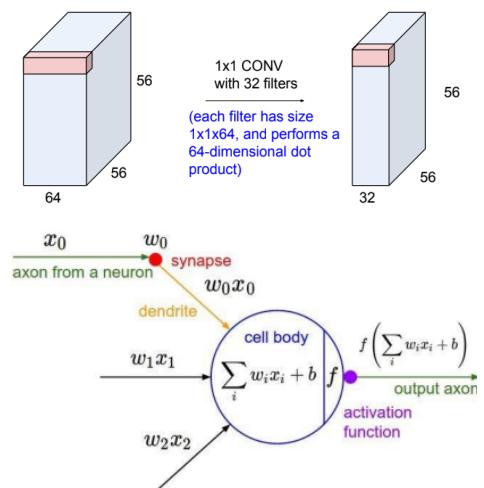
$K = (\text{powers of 2, e.g. } 32, 64, 128, 512)$

- $F = 3, S = 1, P = 1$
- $F = 5, S = 1, P = 2$
- $F = 5, S = 2, P = ?$ (whatever fits)
- $F = 1, S = 1, P = 0$

Il numero di parametri è dato da: F^2CK and K biases.

A destra un esempio di una possibile configurazione. Si noti che lo stride non viene applicato a tutti i layer (quando abbiamo uno stride pari ad 1 lasciamo la dimensione spaziale dell'output inalterata).

Come già accennato in precedenza, tipicamente si **preferisce utilizzare filtri piccoli in quanto molto più efficienti** (e.g. $3x3, 5x5$). Talvolta possiamo **utilizzare anche filtri 1x1**. In questo caso non si va ad incrementare il reception field ma, come si può notare, si agisce solamente sui canali difatti durante la convoluzione si coinvolgono tutti i canali ma solo nella specifica posizione del pixel in cui stiamo applicando la convoluzione. Un motivo per cui scegliere un filtro di tali dimensioni è quando vogliamo cambiare il numero di canali (e.g. abbiamo due branch da combinare, quindi andiamo da 64 canali a 2 da 32 e dopo concateniamo i risultati).



Nota: Nello studiare le reti neurali è bene chiedersi perché si chiamano in questo modo ed il motivo è da ricercarsi nel fatto che prendono ispirazione dai neuroni nel

nostro cervello. Possiamo equiparare il risultato in un convolutional layer con quello di un singolo neurone nel cervello. Un neurone nel cervello riceve un input x_0 proveniente da un assone (axon) che è il terminale di un altro neurone. Questo input viene pesato dalla sinapsi (synapse) e poi viene effettuata la somma nel corpo cellulare (cell body) del neurone. Infine viene applicata una activation function il cui output porta all'attivazione di un altro neurone. Quindi, nel caso di una activation map 28x28 che è l'output di un neurone:

1. Ciascuno è collegato a una piccola regione nell'input;
2. Tutti condividono parametri.

EX. Con un "filtro 5x5" abbiamo un "receptive field 5x5 per ciascun neurone". Invece, con 5 filtri, lo strato CONV è costituito da neuroni disposti in una griglia 3D (28x28x5). Ci saranno 5 neuroni diversi che guarderanno tutta la stessa regione nel volume di input.

Nota: Questo comportamento è differente da quello che succede in una fully connected layer in cui ogni neurone guarda tutte le posizioni dell'input.

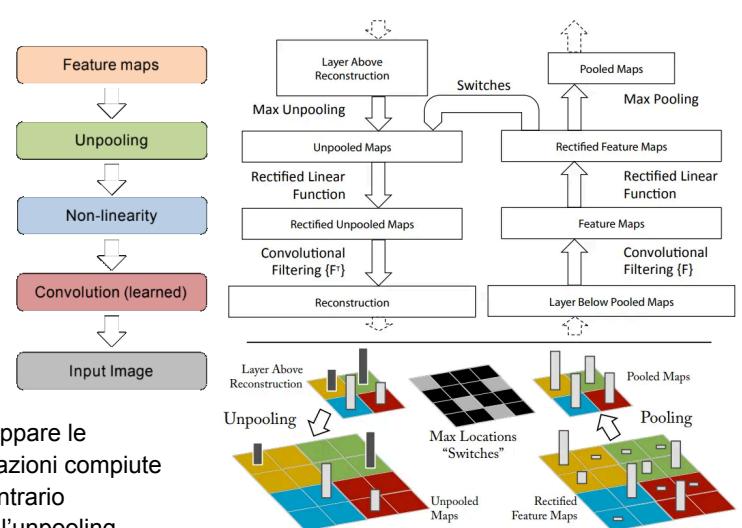
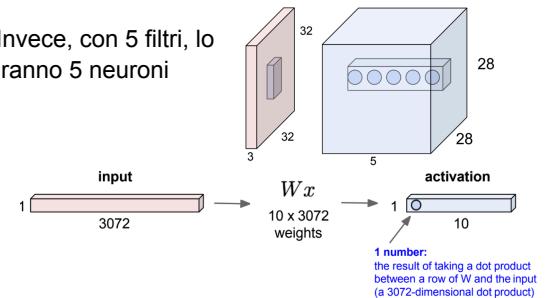
Ci sono delle proprietà relative all'**invarianza spaziale** che vorremmo ottenere da un'architettura basata su Convolutional Neural Network:

- **Invarianza alla traslazione:** Una CNN dovrebbe essere invariante rispetto alle traslazioni dell'oggetto nell'immagine. In altre parole, il riconoscimento di una caratteristica o di un pattern dovrebbe avvenire indipendentemente dalla posizione dell'oggetto nell'immagine. Questa invarianza spaziale consente al modello di generalizzare correttamente anche quando l'oggetto si trova in posizioni diverse;
- **Invarianza alla scala:** L'invarianza alla scala significa che una CNN dovrebbe essere in grado di riconoscere lo stesso oggetto o la stessa caratteristica a diverse scale, ad esempio, sia quando l'oggetto è vicino alla telecamera che quando è lontano. Questo è particolarmente importante in applicazioni in cui gli oggetti possono apparire a diverse distanze;
- **Invarianza alla rotazione:** In alcune applicazioni, come il riconoscimento di caratteri o il riconoscimento facciale, l'invarianza alla rotazione è essenziale. Una CNN dovrebbe essere in grado di riconoscere oggetti o pattern anche se sono ruotati di diversi gradi nell'immagine.

Per ottenere queste invarianze, **dobbiamo utilizzare CNN più profonde**. Nei layer più profondi della CNN, è possibile utilizzare feature map con receptive field più ampi. Questo consente ai neuroni di rilevare caratteristiche in aree più grandi dell'immagine, contribuendo all'invarianza alla scala e alla località (quindi l'invarianza è una proprietà che emerge in layer più profondi e non in quelli iniziali). Talvolta il **pooling** contribuisce all'invarianza spaziale, poiché una piccola variazione nella posizione di una caratteristica non influirà in modo significativo sull'output oppure la **regolarizzazione** (e.g. Dropout) può aiutare a ridurre l'overfitting e a promuovere l'invarianza, permettendo al modello di apprendere rappresentazioni più robuste.

Per quanto riguarda la visualizzazione delle CNN, dobbiamo tenere conto del fatto che i **layer iniziali di una CNN sono generalmente più facili da visualizzare** e interpretare rispetto ai layer più profondi. Ciò è dovuto al fatto che i layer iniziali catturano caratteristiche locali e dettagli specifici dell'input (es. bordi), rendendo le loro attivazioni più interpretabili. Al contrario, i **layer più profondi estraggono caratteristiche sempre più astratte e complesse**, che possono essere più difficili da interpretare direttamente. Sono stati sviluppati vari approcci per effettuare una migliore visualizzazione delle feature ad alto livello e uno di questi è chiamato

"**Deconvolutional Networks**" che fornisce un modo per mappare le attivazioni ai livelli più alti fino all'immagine in input. Le operazioni compiute sono le stesse che vengono effettuate in una CNN ma al contrario (unpooling, convoluzione di mappe a cui abbiamo applicato l'unpooling, ecc..) quindi invece di mappare i pixel sulle features, deconvnets proietta le activation maps (output di convoluzione) nei pixel dell'immagine originale.



Per visualizzare una Convnet, una DeConvnet è **collegata a ciascuno dei suoi livelli**, fornendo un percorso continuo di ritorno ai pixel dell'immagine. Quindi, un'immagine in input viene inviata alla Convnet che calcola le features nei vari layer. Per esaminare una certa activation, impostiamo tutte le altre activation nel layer su zero e passiamo le feature maps come input al layer DeConvnet collegato. Quindi successivamente effettuiamo (i) unpool, (ii) rectify (cioè applichiamo la funzione di attivazione sull'activation map a cui abbiamo applicato l'unpooling) e (iii) filtri (applichiamo la deconvoluzione) per ricostruire l'attività nel layer sottostante che ha dato origine all'activation scelta. Questo viene quindi ripetuto finché non viene raggiunto lo spazio dei pixel in input. Per quanto riguarda l'**unpooling**, abbiamo un meccanismo di routing in cui prendiamo le feature più grandi (quelle che abbiamo selezionato con il max pooling nella ConvNet) e le mettiamo dove erano inizialmente prima della sua applicazione mentre il resto delle posizioni rimarranno semplicemente zero.

4. Training Neural Networks

Finora abbiamo visto come funziona una CNN e in generale, una fully connected layer. Si ricorda inoltre che per effettuare il training di una rete neurale possiamo utilizzare il **Mini-Batch Stochastic Gradient Descent** che funziona in questo modo:

1. Ottieni un batch di dati (una porzione dell'intero training set);
2. Effettua il Forward Pass nella rete per ottenere la loss;
3. Effettua Backpropagation per calcolare i gradienti;
4. Aggiorna i parametri utilizzando il gradiente.

4.1 Activation Functions

Ora, ci sono alcuni aspetti da analizzare meglio e uno di questi è il funzionamento della **funzione di attivazione** (activation functions). Ci sono svariate funzioni di attivazione, ognuna delle quali è preferibile in determinati scenari.

Partiamo dalla funzione **sigmoid**. Questa funzione schiaccia i numeri nell'intervallo [0,1]. Questa funzione di attivazione è storicamente popolare poiché interpreta bene il funzionamento di un neurone ha però alcuni problemi:

- **Problema di svanimento del gradiente:** Durante l'addestramento di deep neural networks, la funzione sigmoide può soffrire di un problema noto come "**vanishing gradient**." Questo significa che la derivata della funzione sigmoide è massima quando x è vicino a 0, ma diminuisce rapidamente quando x si sposta lontano da zero. Questo può causare una lenta convergenza dell'addestramento o addirittura la scomparsa dei gradienti nei layer più profondi della rete, rendendo difficile l'addestramento di reti neurali molto profonde.
EX. Immagina che abbiamo la funzione al lato con valori che vanno da -10 a 10. Se calcoliamo $\sigma(-10) = \sim 0$ e andando a calcolare il suo gradiente avremo:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) = 0(1 - 0) = 0$$

Quindi nell'applicare la backpropagation, propagheremo uno 0 che equivale ad un "non apprendimento". La stessa cosa avviene quando calcoliamo $\sigma(10) = \sim 1$ infatti:

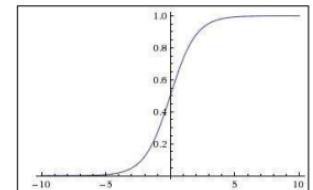
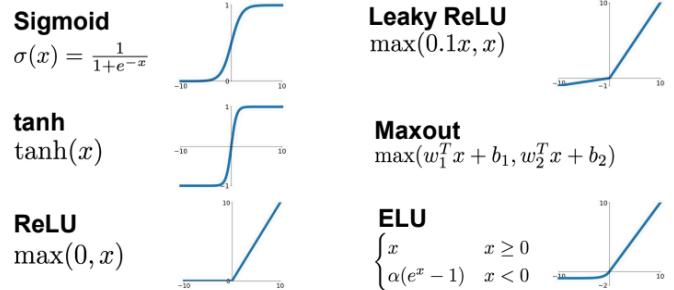
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) = 1(1 - 1) = 0$$

Quindi possiamo vedere che il gradiente è massimo quando x è vicino allo 0 e svanisce a mano a mano che ci allontaniamo da questo numero.

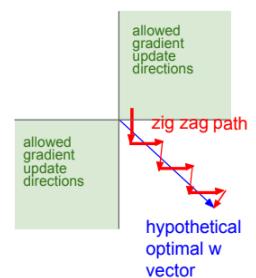
- **Output non zero centrico:** La funzione sigmoide non è centrata su zero, il che può causare problemi in alcune situazioni. Le reti neurali che utilizzano il sigmoide come funzione di attivazione possono richiedere più tempo per convergere perché i gradienti possono essere spostati lontano da zero.

EX. Si consideri il caso in cui l'input di un neurone è sempre positivo. Cosa possiamo dire dei gradienti di w (dove w sono i pesi)?

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}$$



$$f\left(\sum_i w_i x_i + b\right)$$



Sappiamo che il gradiente locale del sigmoide è sempre positivo e supponiamo che x sia sempre positivo ma questo significa che il segno del gradiente per tutti i w_i è lo stesso del segno del gradiente scalare **a monte** (upstream scalar gradient). Questo significa che i gradienti di w sono sempre o tutti positivi o tutti negativi (sempre in base al segno dell'upstream scalar gradient) e questa è una cosa negativa perché la soluzione ottimale potrebbe richiedere una situazione mista (es. aumentare w_1 , diminuire w_2 , lasciare w_3 invariato, ecc..). La situazione che si può verificare è quindi uno **zig-zag nella diminuzione delle loss che può rallentare la convergenza**. Nota che i pesi vengono calcolati per ogni elemento (immagine) quindi per mitigare questo problema possiamo usare mini batches.

- Il calcolo dell'esponente $\exp()$ è computazionalmente pesante.

Possiamo quindi decidere di utilizzare il **tanh** che è un'altra funzione di attivazione che **risolve alcuni problemi del sigmoid**. Questa funzione di attivazione schiaccia gli elementi nell'intervallo $[-1, 1]$ ed inoltre è centrata sullo zero (risolvendo il secondo problema). Il problema che permane è quello dello svanimento del gradiente.

Un'altra funzione di attivazione è la **ReLU** che calcola semplicemente il massimo tra 0 e x (quindi se x è negativo diventa 0, x altrimenti). ReLU **non ha il problema dello svanimento del gradiente (nella regione positiva)**, è **computazionalmente molto efficiente** (calcola semplicemente il massimo tra due elementi) e **converge più velocemente** del sigmoid e tanh in pratica. Un problema è il fatto che l'output **non è centrato sullo zero**.

Inoltre, [cosa succede al gradiente quando \$x < 0\$?](#)

Il comportamento di ReLU può portare a una situazione nota come il "**dying ReLU problem**" (problema del ReLU morto). Se un neurone con funzione di attivazione ReLU riceve input negativi costanti, il suo gradiente rimarrà zero e il neurone smetterà di apprendere. In altre parole, il neurone diventerà "morts" perché non contribuirà all'addestramento della rete. Questo è uno dei problemi principali associati all'uso di ReLU nelle reti neurali.

[Come possiamo risolvere il problema?](#)

Per affrontare il problema del ReLU morto, sono state introdotte varianti della funzione ReLU, come la **Leaky ReLU** oppure la **Parametric Rectifier (PReLU)**, che aggiungono un piccolo gradiente negativo quando $x < 0$. Questo consente al neurone di continuare ad apprendere anche quando l'input è negativo. Ad esempio, una funzione Leaky ReLU è definita come segue:

$$f(x) = \max(\alpha x, x)$$

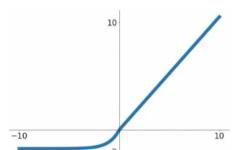
Dove α è un valore positivo piccolo (spesso nell'intervallo 0 a 1) che determina la pendenza della parte negativa della funzione. Questa pendenza leggermente negativa permette al neurone di continuare ad apprendere anche quando l'input è negativo. La principale differenza tra la Leaky ReLU e la PReLU è che nella prima α è un valore fisso mentre nella seconda α diventa un parametro appreso dalla rete (quindi appreso durante il training e aggiornato durante la backpropagation).

Abbiamo anche la **Exponential Linear Units (ELU)** che ha tutti i benefici della ReLU, **non soffre del problema della saturazione** (in ELU, la derivata è diversa da zero sia per i valori positivi che per quelli negativi, consentendo una maggiore velocità di apprendimento) ed ha un **output medio vicino allo zero**. Di contro, potrebbe non essere la migliore soluzione in quanto bisogna calcolare l'esponenziale $\exp()$ e questo **richiede una certa potenza computazionale**.

Un'altra funzione di attivazione è la Scaled Exponential Linear Units (SELU) che è una versione scalata di ELU che lavora meglio con le reti profonde. Garantisce l'**auto-normalizzazione** che in una rete di neuroni profonda con attivazioni SELU e pesi inizializzati correttamente, la distribuzione delle attivazioni rimane centrata su zero e con una varianza costante durante il passaggio

attraverso la rete. Questo può contribuire a mitigare il problema del "vanishing gradient" in reti neurali profonde e favorire una convergenza più rapida.

La funzione di attivazione **Maxout** è un'altra alternativa alla funzione di attivazione ReLU utilizzata in reti neurali. La caratteristica distintiva della funzione Maxout è la sua semplicità e capacità di apprendere una varietà di funzioni non lineari. In una rete che utilizza Maxout, ogni neurone non ha una funzione di attivazione specifica, ma è piuttosto un "blocco" costituito da più "pezzi" o "unità" chiamate "neuroni Maxout". Ogni neurone Maxout prende in input una combinazione lineare delle feature di input,



$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

simile a come avviene in altri neuroni, ma la differenza risiede nella funzione di attivazione. Ogni neurone Maxout calcola il massimo tra gli input lineari per determinare l'output. In altre parole, se un neurone Maxout ha più input, calcolerà il massimo tra questi input.

Maxout generalizza ReLU e LeakyReLU, è lineare, non soffre del problema della saturazione e non "muore". Di contro, **richiede il doppio dei parametri/neuroni.**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Cosa scegliere all'atto pratico?

Conviene partire da ReLU che è la funzione di attivazione più semplice ed eventualmente esplorare LeakyReLU, Maxout e SELU in maniera da provare a ricavare un guadagno marginale. Conviene non usare sigmoid e tanh per il problema del vanishing gradient.

4.2 Data Preprocessing

Il Data Preprocessing è un passaggio fondamentale per garantire il corretto funzionamento delle CNN e il loro addestramento efficace. La corretta preparazione dei dati può aiutare a **migliorare la stabilità** (si diminuisce la sensibilità a piccoli cambiamenti ai pesi) e le prestazioni della rete, nonché a ridurre i problemi come l'overfitting. Si assume di avere dei dati in input, solitamente quello che si può fare è **normalizzare** i dati: si sottrae la media dei dati e si divide per la deviazione standard (z-score normalization) per standardizzare i dati in modo che abbiano una media di zero e una deviazione standard di uno (dati compresi tra 1 e -1). Un aspetto positivo della normalizzazione è che può evitare situazioni in cui il gradiente sia sempre positivo o negativo in quanto questa ci garantisce che i dati abbiano una media di zero.

Si noti che c'è una correlazione tra la prima e la seconda dimensione dei dati. **PCA** è una tecnica di riduzione della dimensionalità utilizzata per trasformare i dati in un nuovo sistema di coordinate in cui le features sono linearmente scorrelate. Ciò viene ottenuto individuando le componenti principali dei dati, che sono combinazioni lineari delle features originali. La prima componente principale cattura la massima varianza nei dati, la seconda componente principale cattura la seconda massima, e così via. Lo **Whitening** è una tecnica utilizzata dopo la PCA o da sola per ulteriori elaborazioni dei dati. Lo scopo del Whitening è rendere i dati trasformati con media zero e varianza unitaria e garantire che le componenti principali siano scorrelate. In altre parole, **rimuove eventuali correlazioni lineari tra le componenti principali**. Nell'applicare lo Whitening, si ridimensionano le componenti principali per la radice quadrata dei loro autovalori, il che garantisce che i dati trasformati abbiano varianza unitaria lungo ciascuna dimensione.

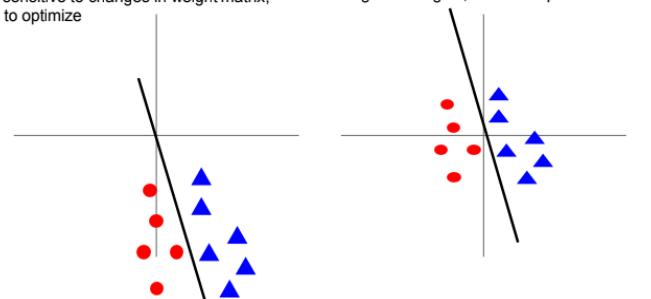
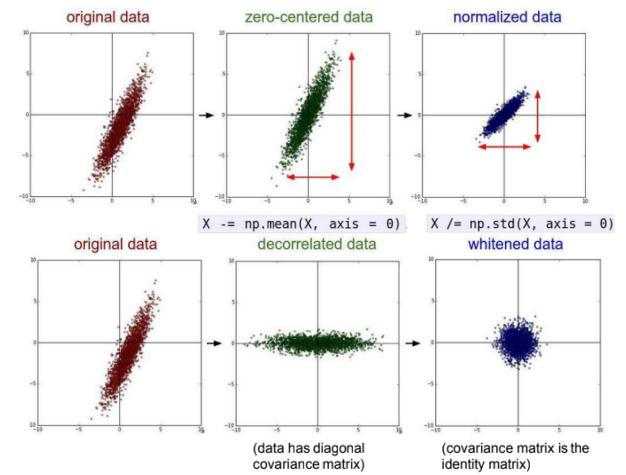
Cosa fare all'atto pratico?

Considera l'esempio CIFAR-10 con immagini [32,32,3]:

- Sottrai l'immagine media (ad esempio AlexNet) (immagine media = [32,32,3] array);
- Sottrai la media per canale (ad esempio VGGNet) (media lungo ciascun canale = 3 numeri);
- Sottrai la media per canale e dividi per std per canale (ad esempio ResNet) (media lungo ciascun canale = 3 numeri).

Nota: PCA e Whitening non sono molto comuni.

Nota: Questi metodi vanno applicati al training set. Non al testing set che non dovremmo conoscere!



4.3 Weight Initialization

Per quanto riguarda l'inizializzazione dei pesi (**weight initialization**) ci sono varie idee che si possono utilizzare:

- **Inizializzazione con zero:** Prevede l'inizializzazione di tutti i pesi a zero. E' molto semplice da implementare ma porta a problemi di simmetria in quanto tutti i neuroni in uno stesso layer avranno lo stesso output, quindi non è raccomandato, poiché le unità apprenderanno gli stessi pesi durante l'addestramento.
- **Inizializzazione casuale:** Prevede l'inizializzazione dei pesi con valori casuali prelevati da una distribuzione uniforme o gaussiana. E' semplice da implementare ma può richiedere una ricerca più lunga dei minimi locali durante l'addestramento, specialmente in reti profonde. Il problema che si potrebbe verificare è che **la standard deviation tende a ridursi ad ogni attivazione tendendo a zero** (mentre la media rimane sempre a zero). Questo comportamento è negativo poiché nel momento in cui si va a calcolare il gradiente dL/dW questo sarà pari a zero (o un numero molto vicino) quindi la rete neurale smetterà di apprendere. Se si decide di incrementare la standard deviation da 0,01 a 0,05 (vedi il valore che moltiplica i valori random nella definizione di W), si verifica il problema opposto. In generale, se la deviazione standard è troppo elevata, i pesi vengono inizializzati con valori molto lontani da zero. Questo può causare problemi di **saturazione delle funzioni di attivazione**, specialmente con funzioni di attivazione come la sigmoid o la tanh. In questo caso, i neuroni possono saturare in modo permanente a valori molto alti o bassi, rallentando la convergenza dell'addestramento. In questo caso i local gradients varranno tutti zero e questo significa che anche in questo caso la rete neurale smetterà di apprendere.
- **Inizializzazione Xavier/Glorot:** Prevede l'inizializzazione dei pesi con valori casuali estratti da una distribuzione uniforme o gaussiana con una varianza specifica in base al numero di input e output del layer. Questo metodo è particolarmente efficace per funzioni di attivazione come la sigmoid e la tanh. Questa inizializzazione prevede di dividere i valori randomici per la radice della dimensione dell'input.

Nota: Nel caso di una ConvNet la formula è la stessa.

Din si calcolerà come $\text{kernel_size}^2 * \text{input_channels}$.

Il principio chiave di questa inizializzazione è quello di mantenere la varianza dell'output di ciascun neurone (nella fase forward) approssimativamente uguale alla varianza dei suoi input. Ciò favorisce una propagazione del segnale stabile attraverso la rete infatti **previene problemi di saturazione** (quando l'output di un neurone è prossimo a 0 o 1) e l'**esplosione del gradiente** (quando i gradienti crescono in modo incontrollato) durante l'addestramento.

Nota: Andiamo di più nel dettaglio: sia y l'output delle rete neurale dove $y = Wx = x_1w_1 + x_2w_2 + \dots + x_D w_D$.

Vogliamo che la varianza dell'input e l'output siano

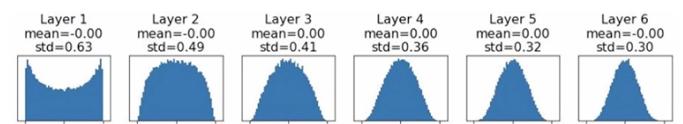
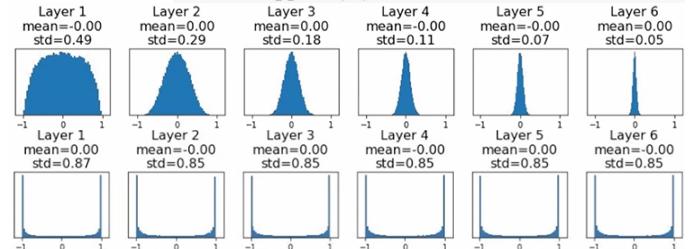
uguali quindi $\text{Var}(y) = \text{Var}(x)$. Assumiamo che x e w sono indipendenti e identicamente distribuiti (iid) allora la varianza della somma di variabili indipendenti è la somma delle varianze ma quindi possiamo

estrarre Din e esprimere $\text{Var}(y)$ come il prodotto tra

Din e $\text{Var}(x_i w_i)$. Sappiamo che se x e w sono indipendenti, la varianza è uguale al valore atteso delle variabili stocastiche alla seconda meno il valore atteso alla seconda delle variabili stocastiche. Assumendo che x e w sono centrate sullo zero allora i due valori sono esattamente la varianza di x_i e w_i . Ma quindi se la varianza di w è uguale per tutti i pesi w_i e pari a $1/Din$ allora abbiamo dimostrato l'uguaglianza.

- Continua a 28:30

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```



$$\begin{aligned} \text{Var}(y) &= \text{Din} * \text{Var}(x_i w_i) && [\text{Assume } x, w \text{ are iid}] \\ &= \text{Din} * (\text{E}[x_i^2]\text{E}[w_i^2] - \text{E}[x_i]^2 \text{E}[w_i]^2) && [\text{Assume } x, w \text{ independent}] \\ &= \text{Din} * \text{Var}(x_i) * \text{Var}(w_i) && [\text{Assume } x, w \text{ are zero-mean}] \end{aligned}$$

$$\text{If } \text{Var}(w_i) = 1/\text{Din} \text{ then } \text{Var}(y) = \text{Var}(x)$$

- Q la varianza dell'input è uguale alla varianza dell'output.

Questo metodo purtroppo **non funziona bene con funzioni di attivazione come la ReLU** poiché è asimmetrica e non ha una variazione simmetrica intorno a zero. Difatti, avremo alcuni valori di x nei layer iniziali che valgono comunque zero ed il rischio è che le **attivazioni collassino a zero** portando ad un non apprendimento della rete neurale. Per evitare questo problema si può calcolare la standard deviation come $\text{std} = \sqrt{2/\text{Din}}$ e questo aiuta a fare in modo che la varianza dell'input venga preservata nell'output. Questa inizializzazione prende il nome di **inizializzazione Kaiming / MSRA**.

```

dims = [4096] * 7           "Xavier" initialization:
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)

```

4.4 Normalization (Batch)

Un altro aspetto importante è la **Batch Normalization** che può aiutare ad avere attivazioni centrate sullo zero (zero-mean) e con varianza unitaria (unit variance) ad ogni layer. La Batch Normalization (BN) è una tecnica utilizzata nelle reti neurali profonde per normalizzare i dati in uscita da ciascun layer all'interno di un batch durante l'addestramento. Consideriamo un batch di attivazioni in un certo layer. Per fare in modo che ogni dimensione sia centrata sullo zero (zero-mean) e con varianza unitaria possiamo calcolare:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Una volta calcolata la x , la si può sottrarre al valore atteso di x e dividere il risultato per la varianza di x .

Più nel dettaglio, immaginiamo di avere un certo input x che ha dimensione $N \times D$ (dove N è il numero di elementi, es. immagine, e D la la loro dimensione, es. $32 \times 32 \times 3$ nel caso di CIFAR-10). Possiamo calcolare **la media μ per ogni canale j** facendo scorrere i su ogni pixel delle immagini nel batch. Si effettua la somma del pixel e si divide per il numero di immagini ottenendo la media per ogni canale:

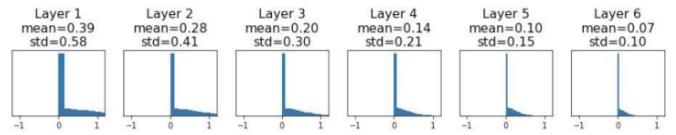
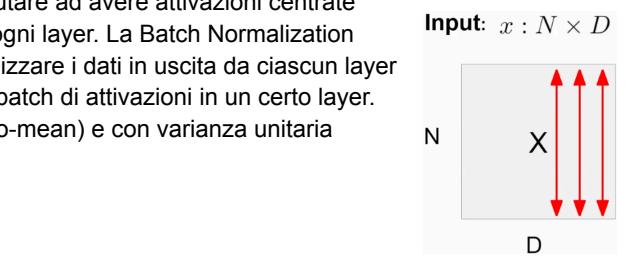
$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is } D$$

Allo stesso modo di può calcolare la **varianza σ dei valori nel batch per ogni canale j** :

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is } D$$

Infine si va a normalizzare utilizzando la z-score cioè sottraendo l'attivazione originale alla media e dividendo tutto per la varianza. Si noti che viene incluso un valore ϵ per stabilità numerica in quanto potremmo avere una varianza molto vicina allo zero (in generale, vogliamo evitare divisioni per zero):

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \text{Normalized } x, \text{ Shape is } N \times D$$



```

dims = [4096] * 7           "ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)

```

Nota: Imporre il vincolo di avere attivazioni centrate sullo zero e con varianza unitaria ad ogni layer potrebbe essere una considerazione troppo forte ed in generale è stato dimostrato che non apporta molti vantaggi. Per questo motivo si può dare la possibilità alla rete di stabilire un valore per la media e la varianza che non siano necessariamente zero ed uno. Si possono introdurre quindi due parametri $\gamma, \beta \in D$ apprendibili (imparati dalla rete durante il training) per preservare quindi la capacità espressiva del layer. Dopo la normalizzazione, l'output normalizzato viene riscalato con il parametro γ (gamma) e spostato con il parametro β (beta) per ottenere l'output finale del layer. In generale, l'introduzione di questi parametri porta ad una convergenza più veloce, migliore robustezza e accuratezza finale.

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is } N \times D$$

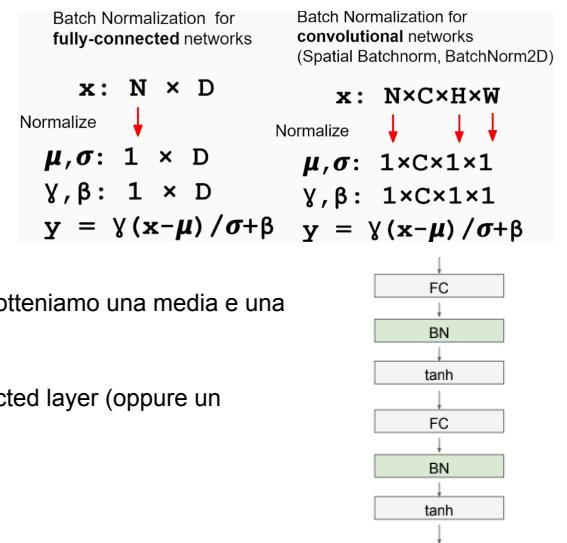
$$\mu_j = \frac{\text{(Running) average of values seen during training}}{\cdot} \quad \text{Per-channel mean, shape is } D$$

$$\sigma_j^2 = \frac{\text{(Running) average of values seen during training}}{\cdot} \quad \text{Per-channel var, shape is } D$$

Nota: Nelle fasi di test non abbiamo un batch (ricorda che la media e la varianza le calcoliamo su batch dell'input) ma una singola immagine quindi non possiamo applicare i metodi visti finora durante la fase di test. Per ovviare a questo problema possiamo prendere la media μ e la varianza σ che avevamo calcolato durante la fase di training.

Nota: Perché non calcolare la media e la varianza prendendo l'intero input invece che dei batch? Questo dipende dal fatto che consideriamo l'input di un generico layer dove tale input è l'output del layer precedente cioè Wx dove W viene appreso durante il training. Non riusciamo mai a calcolare la media e la varianza poiché W cambia. L'unico caso in cui riusciremmo a calcolarle è alla fine del training quando W è stato appreso e non cambia.

Ora, vediamo come possiamo applicare la **Batch Normalization alle ConvNet**. Nelle fully connected layer la media e la deviazione standard vengono calcolate su tutte le attivazioni in un fully connected layer quindi otteniamo una media e una varianza di un batch per ogni canale. Nelle ConvNet, la batch normalization viene di solito applicata in modo indipendente a ciascun canale all'interno di un layer. Ogni canale in un convolutional layer ha la sua **media e varianza calcolate lungo le dimensioni spaziali (altezza e larghezza) nello stesso batch**. Quindi, la differenza chiave sta nel modo in cui la batch normalization calcola le statistiche. Una rete fully connected, calcola le statistiche su tutti i neuroni nello stesso layer, mentre la ConvNet calcola le statistiche separatamente per ciascuna feature map e lungo le dimensioni spaziali. Questo perché le ConvNets applicano un filtro indipendentemente dalla posizione nell'immagine (e.g. i bordi verticali sono gli stessi in ogni posizione dell'immagine). Si noti che anche nelle ConvNet otteniamo una media e una varianza per canale.



Nota: Solitamente la Batch Normalization viene inserita dopo un fully connected layer (oppure un convolutional layer) e prima dell'applicazione della non linearità.

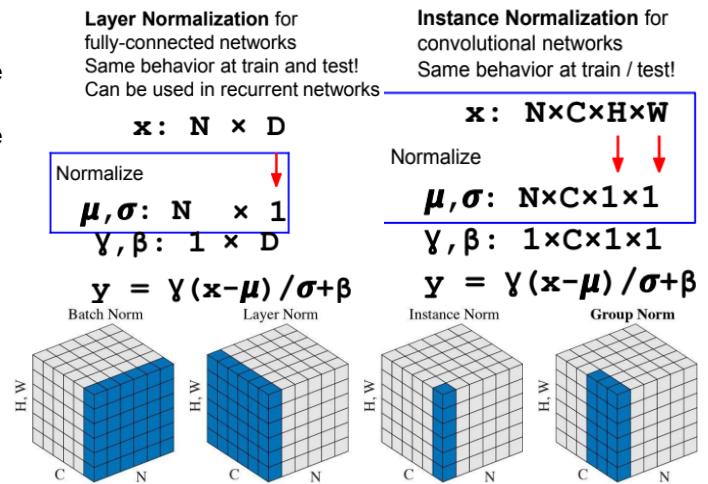
Quindi, la Batch Normalization:

- Rende le reti profonde molto più facili da addestrare;
- Migliora il flusso del gradiente;
- Consente learning rates più elevati e quindi una convergenza più rapida;
- Le reti diventano più robuste all'inizializzazione;
- Agisce come regolarizzazione durante il training;
- Zero overhead durante il test: Può essere fuso con il layer convoluzionale;

Ricorda che BatchNorm si comporta diversamente durante il training e i test (come abbiamo visto prima) e questa è una fonte di bug molto comune. Inoltreabbiamo necessariamente bisogno di un batch: Se il batch diventa piccolo, la stima della media e della varianza saranno poco accurate.

Esistono altri tipi di normalizzazioni:

- **Layer Normalization:** Le statistiche vengono calcolate lungo la dimensione delle feature, ossia per ciascun neurone (unità) nello stesso layer. Questo significa che le medie e le deviazioni standard sono calcolate per ciascun neurone in uno stesso layer. Quindi, contrariamente alla BN dove la normalizzazione viene effettuata sulla base delle immagini nel batch, nella LN questa viene effettuata sui canali (sulle dimensioni) e questo è conveniente in quanto non viene richiesta una dimensione minima per i batch. Questa normalizzazione è spesso utilizzata per normalizzare le attivazioni in sequenza, il che è utile nelle Recurrent Neural Network (RNN).
- **Instance Normalization:** Le statistiche vengono calcolate indipendentemente per ciascuna feature



map (canale) all'interno di uno stesso layer. Questo significa che le medie e le deviazioni standard sono calcolate per ciascuna feature map all'interno di uno stesso esempio. Quindi anziché calcolare la media e la varianza sulla base degli esempi nel batch (come avviene nel BN), si calcolano sulla base nell'altezza H e la lunghezza W dell'activation map. Questa normalizzazione è spesso utilizzata per normalizzare le feature map all'interno di un'immagine o uno spazio, ad esempio, per l'estrazione di stili in applicazioni di stile di immagine.

- **Group Normalization:** Si raggruppa qualche canale e si effettua la normalizzazione partendo da quel raggruppamento.

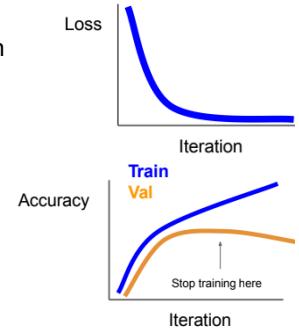
Nota: Nella LN e IN si applica lo stesso comportamento durante il training e il testing.

4.5 Training vs Testing Error

Nella maggior parte dei casi, il training loss verrà decrementato ad ogni iterazione grazie all'applicazione dello stochastic gradient descent (e altre possibili ottimizzazioni) e questo è un bene in quanto è il nostro obiettivo ma non è l'unica cosa di cui ci si deve preoccupare. Difatti vogliamo che lo stesso avvenga anche per il validation/testing set.

In generale, quello che potrebbe accadere è che il training loss diminuisce ad ogni iterazione portando l'accuracy del training set ad aumentare ma quella del validation set potrebbe rimanere invariata o addirittura diminuire dopo una certa iterazione portando ad un sperimentare l'**overfitting**. A noi ci interessa l'errore sui nuovi dati quindi non vogliamo che ci sia una grande differenza tra l'accuracy del training set e quella del validation set perché questo significherebbe che il modello non è in grado di generalizzare. Ci sono vari approcci che si possono usare:

- **Early Stopping:** Interrompi il training del modello quando la precisione sul validation set diminuisce o esegui il training per un lungo periodo, ma tieni sempre traccia dello snapshot del modello che ha funzionato meglio su validation set (e alla fine scegli quello);
- **Model Ensembles:** Addestra più modelli indipendenti e al momento dei test, effettua la media dei risultati (prendi la media delle distribuzioni di probabilità previste, quindi scegli argmax). Altrimenti, anziché allenare modelli indipendenti, si possono utilizzare degli snapshots di un singolo modello ottenuti durante il training.



4.6 Regularization (Dropout)

La **regolarizzazione** è una tecnica utilizzata **prevenire l'overfitting e migliorare la capacità di generalizzazione di un modello**. L'overfitting si verifica quando un modello si adatta troppo ai dati di addestramento, acquisendo dettagli e rumore anziché catturare i pattern generali nei dati. Possiamo notare questo comportamento quanto l'accuracy del training set si discosta (è più alta) di quella calcolata nel validation/testing set (come visto prima). Per regolarizzare, si può aggiungere un termine di regolarizzazione durante il training:

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

Dove:

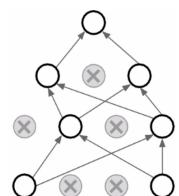
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

La **regolarizzazione L1** (nota anche come regolarizzazione Lasso) aggiunge il valore assoluto dei pesi alla funzione di costo, incoraggiando la sparsità nei pesi, cioè alcuni pesi diventano esattamente zero, riducendo così il numero di feature rilevanti. La **regolarizzazione L2** (nota anche come regolarizzazione Ridge) aggiunge il quadrato dei pesi alla funzione di costo, impedendo ai pesi di diventare troppo grandi, il che limita la complessità del modello e favorisce una migliore generalizzazione. L'**elastic net regularization** combina le due e regola il loro contributo per mezzo di un parametro β .

Il **dropout** è una tecnica di regolarizzazione specificamente utilizzata nelle reti neurali. Durante l'addestramento, il dropout elimina casualmente alcuni neuroni (o unità) in modo temporaneo. Quindi durante il forward pass, alcuni neuroni vengono impostati a zero in maniera randomica. La probabilità di disattivare un neurone è un hyperparameter e viene tipicamente inizializzato a 0.5. Questo previene che



la rete si affidi eccessivamente a determinati neuroni, migliorando la robustezza del modello. Forza la rete ad avere una rappresentazione ridondante impedendo il **coadattamento** delle features (quindi evita che i neuroni si concentrino sulle stesse features). Detto in un altro modo, quello che vogliamo fare è disattivare dei neuroni per vedere se la rete neurale è ancora in grado di arrivare ad un certo output. Il dropout introduce quindi **variabilità** nel modello durante l'addestramento, poiché a ogni passaggio il set di neuroni attivi cambia. Questo rende il modello più robusto e meno incline all'overfitting, poiché non può fare affidamento su nessun neurone specifico. Quindi fa sì che la rete debba imparare a essere robusta indipendentemente dalla presenza di neuroni specifici.

Siccome ad ogni iterazione disattiviamo dei neuroni in maniera casuale, questo significa che otteniamo sempre delle reti diverse. Quindi un'altra interpretazione di Dropout può essere che Dropout sta addestrando un **grande insieme** (ensembles) **di modelli** (che condividono parametri) dove ogni maschera binaria è un modello.

Nota: Un fully connected layer con 4096 unità ha $2^{4096} \sim 10^{1233}$ maschere possibili!

Poiché l'output è randomico, **non si può utilizzare Dropout durante la fase di test** (in cui vogliamo che la randomicità sia fissata). Potremmo fare la media della randomicità ma questa prevede il calcolo di un integrale che non è facile da calcolare:

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Si può quindi cercare di approssimare l'integrale. Si consideri un singolo neurone. Durante la fase di test abbiamo:

$$E[a] = w_1x + w_2y$$

Durante la fase di training invece dobbiamo considerare tutte le possibili combinazioni di Dropout con la stessa probabilità (es. lascia entrambi i neuroni, disattivane uno, disattiva l'altro, ecc..):

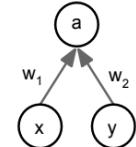
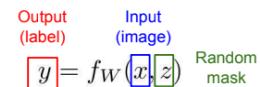
$$E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) = \frac{1}{2}(w_1x + w_2y)$$

Si noti come questo sia equivalente a prendere l'intera rete e dividerla per due che è esattamente la probabilità di disattivare un singolo neurone. Al momento del test tutti i neuroni sono sempre attivi quindi dobbiamo **scalare le attivazioni (moltiplicando per la probabilità di Dropout)** in modo che per ciascun neurone **l'output al momento del test sia l'output atteso al momento del training**. Quindi, durante il training lasciamo i parametri come sono ma ne disattiviamo alcuni con una probabilità p . In fase di test lasciamo tutti i neuroni attivi ma cerchiamo di simulare la disattivazione effettuata in fase di training prendendo solo una loro porzione e questo di ottene moltiplicando per la probabilità di Dropout p .

Nota: Si può anche applicare il cosiddetto "**inverted dropout**" per **evitare di toccare le attivazioni in fase di test**. Quello che si fa è dividere le maschere (U_1, U_2) per p direttamente durante la fase di training (senza moltiplicare per p nella fase di test) e queste due operazioni sono equivalenti.

Un altro modo per regolarizzare, è l'applicazione della **Data Augmentation**. La data augmentation è una tecnica utilizzata per aumentare la quantità e la diversità dei dati disponibili. L'obiettivo della data augmentation è migliorare le prestazioni del modello, rendendolo più robusto e in grado di generalizzare meglio ai dati non visti. Questa tecnica è particolarmente utile quando il dataset di addestramento è limitato o **non rappresentativo di tutte le variazioni possibili nel testing set**. La data augmentation coinvolge la creazione di nuovi esempi di addestramento attraverso la manipolazione dei dati esistenti. Le trasformazioni vengono applicate in modo casuale e controllato ai dati di addestramento, producendo nuovi campioni che sono simili, ma leggermente diversi da quelli originali. Alcune trasformazioni comuni utilizzate nella data augmentation:

- **Rotazione:** Ruotare l'immagine. **Non abbiamo tipicamente bisogno della rotazione** perché nel momento in cui ruotiamo l'immagine



```

p = 0.5 # probability of keeping a unit active. higher = less dropout

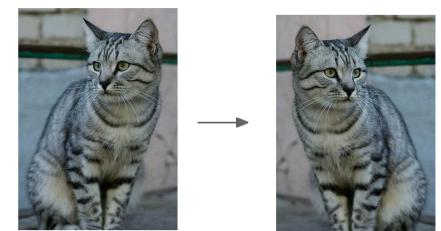
def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1) Drop in forward pass
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

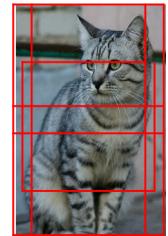
def predict(X):
    """ ensembled forward pass """
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3 Scale at test time

```



potremmo perdere il senso dell'oggetto nel mondo (e.g. immagina la foto di un gatto, se lo riflettiamo orizzontalmente è ancora un gatto e ci può essere utile ma se lo ruotiamo di 90° non ci aiuta nel mondo reale perché non è solito avere foto di gatti con quell'angolatura).

- **Riflessione orizzontale/verticale:** Riflettere l'immagine orizzontalmente o verticalmente per ottenere una versione speculare.
- **Traslazione:** Spostare l'immagine o i dati orizzontalmente e verticalmente in modo casuale.
- **Cropping/Zooming:** Ritagliare (o zoomare) parti casuali dell'immagine. La procedura durante il training prevede di scegliere uno scalare L nel range [256, 480], si ritaglia l'immagine in maniera che il lato corto abbia una lunghezza pari ad L e dopodiché si ottengono delle patch randomiche 224x224. Questa operazione è utile per insegnare al modello a riconoscere oggetti in diverse posizioni all'interno dell'immagine. Durante il testing, per fare in modo di fare della media della casualità introdotta durante la fase di training e aumentare la possibilità di riconoscere l'oggetto, si ridimensiona l'immagine su 5 scale (224, 256, 384, 480, 640) e per ogni grandezza si usano 10 porzioni 224x224 dell'immagine: 4 angoli + centro, + ribaltamenti. Infine si ottengono i risultati e si fa la media (quindi anziché usare un'immagine, usiamo 5*10 porzioni e facciamo la media).
- **Regolazione del contrasto e della luminosità:** Modificare il contrasto e la luminosità dell'immagine. Alcuni metodi più complessi prevedono di applicare PCA a tutti i pixel [R, G, B] nel training set in maniera tale da ottenere la massima variazione, si ottiene un "color offset" lungo le direzioni dei componenti principali e si aggiunge l'offset a tutti i pixel dell'immagine.



Un pattern generale che tipicamente si usa quando si applica la regolarizzazione, prevedere **l'aggiunta di una certa casualità** (randomness) durante il training che poi viene **rimossa o marginalizzata durante la fase di test** facendo la **media della casualità** (a volte approssimandola).

EX: Si pensi al BatchNorm. Durante il training, la casualità viene data dalla media e dalla standard deviation quindi si effettua la normalizzazione utilizzando statistiche di mini batch casuali. Durante la fase di test si usano statistiche fissate per normalizzare.

Training: Add some kind of randomness

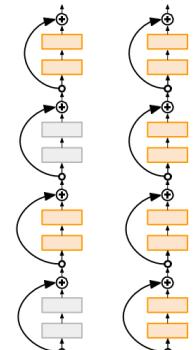
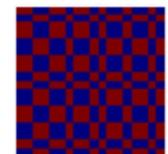
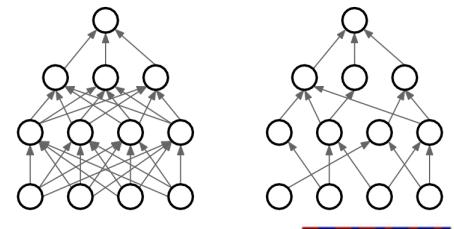
$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Esistono anche altre tecniche di regolarizzazione:

- **Drop Connect:** Questa tecnica è simile al Dropout, ma invece di disabilitare casualmente le unità (nodi) in un layer, Drop Connect disabilita casualmente (con una certa probabilità) le connessioni tra le unità di un layer e le unità di un layer successivo impostando i pesi a 0. Durante la fase di test o inferenza, Drop Connect non è attivo. Tutte le connessioni tra le unità sono attive, il che significa che il modello utilizza tutte le informazioni apprese durante l'addestramento.
- **Fractional Pooling:** È una tecnica di pooling utilizzata per eseguire l'operazione di pooling su una frazione specifica di una regione rettangolare randomica all'interno di un'immagine o di una feature map. Durante la fase di test si fa la media della predizione di varie regioni. L'obiettivo principale di Fractional Pooling è aumentare la flessibilità nell'aggregazione delle informazioni spaziali e migliorare la robustezza del modello, in particolare quando si tratta di dati in cui le dimensioni dell'input sono varie.
- **Stochastic Depth:** In Stochastic Depth, durante il training, vengono selezionati casualmente (con una certa probabilità) alcuni layer della rete e vengono temporaneamente "disabilitati" o "saltati" durante l'elaborazione dei dati. In altre parole, i dati passano attraverso solo una frazione dei layer durante un'epoca di addestramento specifica. Questa disabilitazione casuale di layer "profondi" riduce effettivamente la profondità della rete in ogni iterazione. Di conseguenza, il modello addestrato su un sottoinsieme dei layer può apprendere rappresentazioni più semplici o "superficiali" dei dati. Durante la fase di test o inferenza, tutti i layer della rete sono attivi, e il modello utilizza l'intera profondità della rete per effettuare previsioni.
- **Cutout:** Seleziona una regione casuale all'interno dell'immagine e la "taglia" o "maschera" impostando tutti i pixel all'interno di quella regione su un valore predefinito (solitamente zero o il valore medio dei pixel nell'immagine). L'immagine con la regione di taglio viene utilizzata per addestrare il



modello. Questo significa che il modello deve imparare a fare previsioni anche quando alcune parti dell'immagine sono oscurate o mancanti. Nella fase di test si usa semplicemente l'intera immagine.

- **Mixup:** Combinare due o più campioni di addestramento (di solito coppie di input e label) per creare un nuovo campione di addestramento che è una combinazione lineare dei campioni originali (es. si possono miscelare foto di diversi oggetti, 40% foto di un gatto e 60% foto di un cane). Nella fase di test si usano le immagini originali.



In generale:

- Considera l'eliminazione per i layer di grandi dimensioni completamente connessi;
- La Batch Normalization e la Data Augmentation sono quasi sempre una buona idea;
- Prova il Cutout e il Mixup soprattutto per i dataset di classificazione di piccole dimensioni.



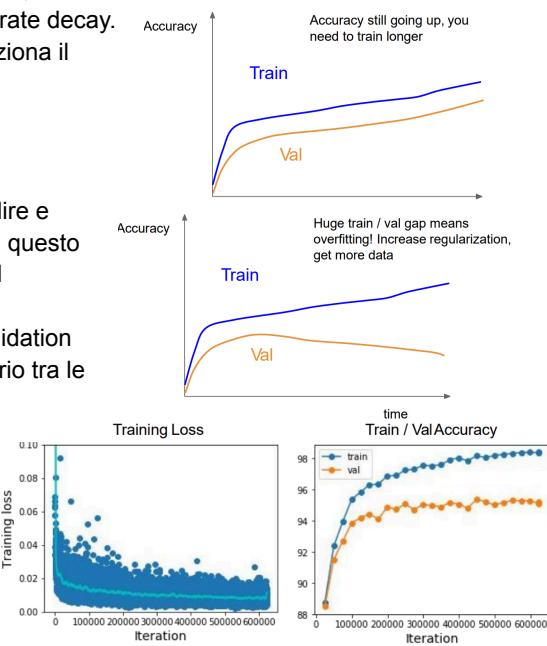
4.7 Hyperparameters

Nel momento in cui creiamo una nuova architettura, alcuni possibili step da compiere possono essere i seguenti:

1. **Controlla la loss iniziale:** All'inizio sappiamo quanto dovrebbe valere la loss se effettuiamo una inizializzazione randomica (e disattivando la weight decay). Ad esempio, assumendo di avere C classi allora dovremmo avere $\log(C)$ per la softmax per ogni classe.
2. **Adattare eccessivamente un piccolo campione:** Prova ad effettuare il training il modo da avere una training accuracy del 100% su un piccolo campione di dati di addestramento (~5-10 minibatch). Si può modificare l'architettura, il learning rate e la weight initialization per provare ad ottenere questo risultato. In generale se la loss non scende potrebbe essere per via di un learning rate troppo basso o una cattiva inizializzazione. Se il loss esplode a infinito (o NaN) allora il learning rate potrebbe essere troppo alto oppure, di nuovo, potremmo avere una cattiva inizializzazione.
3. **Trova un learning rate che faccia abbassare la loss:** Utilizzare l'architettura del passaggio precedente, questa volta usando tutto il training data, attivare un piccolo weight decay e trovare un learning rate che faccia diminuire significativamente la loss entro circa 100 iterazioni. Buoni learning rate sono 1e-1, 1e-2, 1e-3, 1e-4.
4. **Effettua il train per 1-5 epochs:** Scegli alcuni learning rate e weight decay che hanno funzionato dal passaggio 3, addestra alcuni modelli per circa 1-5 epochs (quindi più tempo rispetto a prima). Alcuni weight decay da provare potrebbero essere 1e-4, 1e-5, 0.
5. **Effettua il training più a lungo:** Scegli i modelli migliori dal passaggio 4, addestrali per un periodo più lungo (~ 10-20 epochs) senza il learning rate decay.
6. Guarda alla **curve della loss e dell'accuratezza** per capire come funziona il modello.
7. Torna indietro al passo 5.

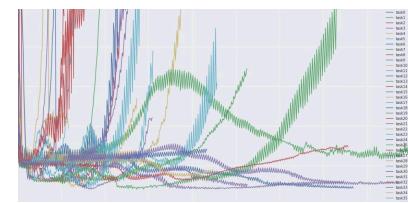
Nel punto 6, potremmo avere una di queste situazioni:

- L'accuratezza (sia del training set che del validation set) continua a salire e quella della validation è ad una certa distanza da quella del training. In questo caso, visto che l'accuratezza continua a salire, conviene continuare ad allenare per un tempo maggiore.
- Se l'accuratezza sul training set continua a salire mentre quella del validation set inizia a scendere dopo un certo epoch portando ad un grande divario tra le due allora significa che il modello è in **overfitting**: conviene usare regolarizzazione o ottenere più dati da utilizzare durante il training.
- Se invece l'accuratezza del training set e del testing set sono molto vicine tra loro (il divario è piccolo) allora questo è un segno di **underfitting**. Convieni effettuare il training più a lungo oppure usare un modello più grande (e complesso).



Tipicamente, si guarda alla training loss alla training e validation accuracy. Il motivo per cui si guarda alla loss è principalmente perché questa considera anche i parametri di regolarizzazione come la weight decay nella regolarizzazione L2 mentre l'accuratezza considera solo i valori finali (compara predizione e ground truth).

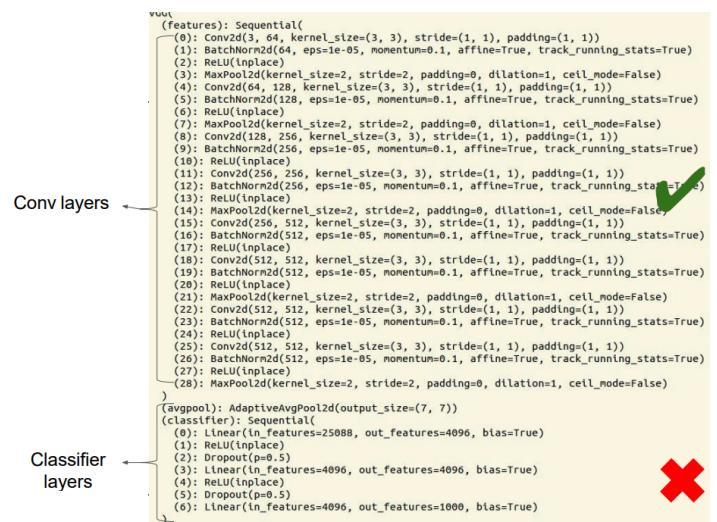
Un'idea potrebbe essere inserire all'interno di un plot le loss curves relative a diversi hyperparameters in maniera da avere una visione generale su come si comporta il modello e scegliere i parametri migliori.



4.8 Transfer Learning

Il Transfer Learning è una tecnica che consiste nell'utilizzare conoscenze acquisite da un compito o un dominio specifico per migliorare le prestazioni in un altro compito. In breve, il transfer learning permette di "trasferire" la conoscenza da una situazione all'altra, consentendo ai modelli di apprendere più efficacemente e con meno dati. Il funzionamento è il seguente:

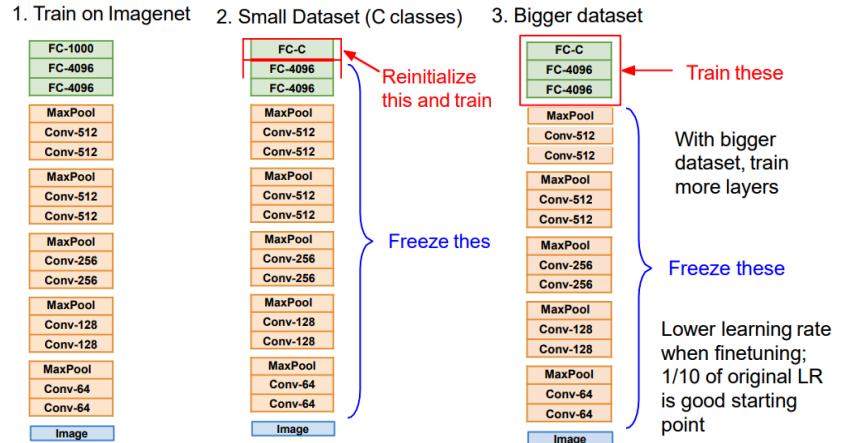
- **Pre-addestramento:** Inizialmente una rete neurale viene addestrata su un compito specifico o su un dominio di dati ricco. Questa fase di addestramento iniziale può richiedere molte risorse computazionali e una grande quantità di dati (e.g. ImageNet).
- **Estrazione delle Caratteristiche:** Dopo l'addestramento iniziale, il modello ha imparato a riconoscere caratteristiche rilevanti nei dati di input. Queste caratteristiche estratte dal modello possono essere molto informative e utili per altri compiti.
- **Trasferimento delle Conoscenze:** Invece di addestrare un nuovo modello da zero per un secondo compito, il modello pre-addestrato può essere utilizzato come punto di partenza. Le conoscenze e le caratteristiche apprese dal primo compito vengono trasferite al secondo compito, risparmiando tempo e risorse.
- **Raffinamento (Fine Tuning):** Il modello pre-addestrato viene quindi sottoposto a un processo di raffinamento o adattamento sul secondo compito o dominio specifico. Questo processo può coinvolgere l'addestramento solo di alcuni strati del modello, l'aggiunta di nuovi strati per adattare il modello o la regolazione di alcuni parametri. In questo modo, il modello può imparare a riconoscere le caratteristiche rilevanti per il secondo compito, tenendo conto di quanto appreso nel primo.



EX. Un esempio banale può essere una classificazione sul Cifar-10. Quello che potremmo fare è sfruttare AlexNet addestrato su ImageNet per provare a creare un classificatore sul Cifar-10, anziché creare un nostro modello da zero.

In generale quello che si fa nella fase di raffinamento è:

- Si mantengono i convolutional layers dal modello originale come estrattori di features;
- Si eliminano i classification layer dal modello originale e si aggiungono

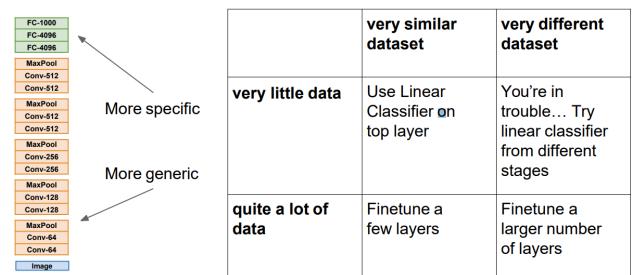


nuovi fully connected layer (es. per effettuare la classificazione sul cifar-10) che funzionano bene con il modello che vogliamo creare.

Quindi, possiamo allenare Alexnet sul dataset ImageNet. Togliamo la parte alta della rete e la inizializziamo di nuovo per fare in modo che funzioni con il modello che vogliamo creare. Infine effettuiamo di nuovo il training del layer che abbiamo inizializzato. Nota che tutti gli altri layer rimangono invariati, non dobbiamo effettuare di nuovo i training (aggiornare i pesi relativi a quei layer). Solo se il dataset è particolarmente grande, possiamo decidere di allenare più layer. Nota che conviene abbassare il learning rate quando si effettua fine tuning (circa 1/10 del learning rate originale potrebbe essere un punto d'inizio).

In generale:

- Con pochi dati e un dataset simile: Si può usare semplicemente un classificatore lineare nel layer più alto.
- Con molti dati ma un dataset simile: Si può effettuare fine tuning sugli ultimi layer.
- Con pochi dati ma dataset differenti: In questo caso la situazione è critica in quanto non avremmo abbastanza dati per effettuare di nuovo il training. Si può eventualmente provare ad effettuare fine tuning su un numero più ampio di layer.
- Con molti dati ma dataset differenti: Si può provare ad effettuare fine tuning su un numero maggiore di layer.



Il Transfer Learning con le CNN è pervasivo, ma i risultati recenti mostrano che potrebbe non essere sempre necessario. In generale:

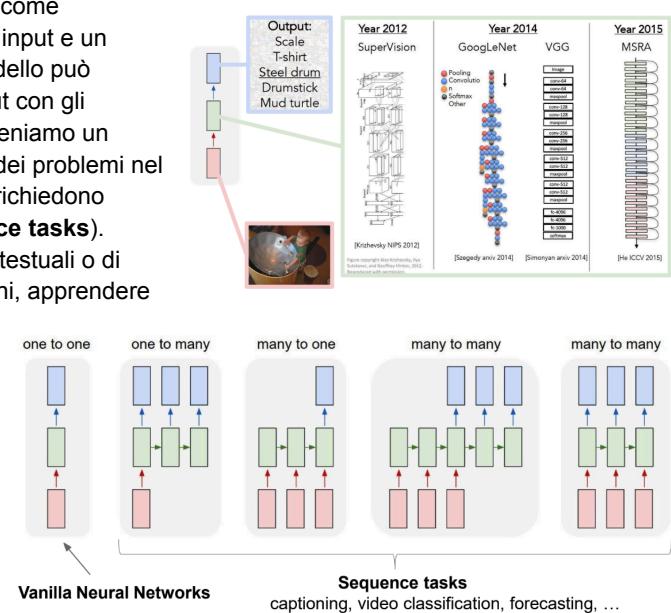
- Il pre-training di ImageNet accelera la convergenza nelle prime fasi dell'addestramento, ma non fornisce necessariamente la regolarizzazione o migliora l'accuratezza del task finale;
- In generale la normalizzazione è importante (BatchNorm o Group Norm);
- Un tempo di training sufficientemente lungo può compensare la mancanza di pretraining;
- La training da zero può essere superiore al pretraining quando il compito è (molto) diverso (es. classificazione vs. rilevamento);
- BatchNorm è importante per il training da zero, ma è generalmente rimosso dopo il pre training;
- RootResNet con BatchNorm è stato progettato specificatamente per il rilevamento.

5. Sequence Models

Prima di introdurre questo nuovo argomento, è bene ricordare come funzionano le reti neurali "vanilla". Data una certa immagine in input e un certo modello (es. SuperVision, GoogleNet, VGG, ecc..), il modello può effettuare la classificazione dell'immagine restituendo un output con gli elementi all'interno dell'immagine (quindi data un'immagine otteniamo un singolo output). C'è da riconoscere però che la maggior parte dei problemi nel mondo (es. guida autonoma, processamento dei video, ecc..) richiedono compiti che coinvolgono dati organizzati in sequenze (**sequence tasks**).

Queste sequenze possono essere costituite da dati temporali, testuali o di altra natura, e l'obiettivo è spesso quello di estrarre informazioni, apprendere da esse o compiere previsioni basate sulla struttura sequenziale dei dati. Abbiamo vari esempi di come queste sequenze possono essere strutturate:

- **One to many:** Una singola sequenza di input viene mappata in più sequenze di output. Si pensi all'**image processing** che prende in input una singola immagine (es. immagine di un cane) e ritorna una sequenza di parole in output che non è



- altro che la sua classificazione (es. "un cane marrone");
- **Many to one:** Più sequenze di input vengono mappate in una singola sequenza di output. Si pensi all'**action prediction** che analizza una sequenza di frame di un video e permette di predire la action class cioè l'azione che l'oggetto nel video sta compiendo (es. un video di un cane che salta delle aste, l'action class sarà "saltare");
 - **Many to many:** Molteplici sequenze di input vengono mappate in molteplici sequenze di output. Si pensi al **machine translation** che prende una sequenza di parole da tradurre e ritorna la sequenza di parole tradotta. Si noti che nel caso della traduzione, occorre conoscere tutto l'input prima di provare a tradurre in quanto occorre conoscere il contesto. Inoltre in alcune lingue, il costrutto delle frasi può essere differente (es. verbi a fine frase anziché all'inizio), quindi è bene avere una visione generale delle frasi che si stanno traducendo. In alcuni casi, le sequenze di input e output possono non avere la stessa lunghezza, e l'allineamento tra di loro è importante. Si pensi anche al **video captioning** che permette di prendere una sequenza di frames video e ritorna didascalia (es. il solito video del cane che salta ritorna "Un cane salta sopra un ostacolo"). Qui la predizione viene effettuata **online**, cioè la risposta viene data sulla base di quello che è stato visto fino a quel momento.

Si parla talvolta di "**Sequential Processing of Non-Sequence Data**" che si riferisce ad un approccio in cui vengono utilizzati modelli progettati per dati sequenziali anche su dati che non sono naturalmente organizzati in sequenze. Un esempio potrebbe essere la classificazione delle immagini prendendo una serie di "scorci". Ad esempio nel Riconoscimento di numeri scritti a mano (OCR), anche se i dati sono costituiti da immagini, è possibile trattare ogni cifra come una sequenza di pixel e utilizzare un modello sequenziale per identificare il numero.

Un altro caso di sequenza many to many piuttosto rappresentativa è il **forecasting**. Nel forecasting si analizzano varie sequenze di frame che rappresentano delle scene nel mondo, e predire cosa succederà (es. predire il percorso che le persone intraprenderanno in base al loro passo o dove stanno guardando - quindi date in input le precedenti posizioni nel tempo, si vogliono predire tutte le possibili predizioni future). Alcune possibili applicazioni del forecasting sono l'interazione robot-umano (es. per migliorare le prestazioni e l'efficacia delle operazioni svolte dai robot in base all'azione dell'umano), tracciamento a lungo termine attraverso le occlusioni (es. i movimenti futuri di un auto che si trova dietro un altro veicolo), sicurezza di veicoli a guida autonoma (es. possibili rischi che potrebbero verificarsi sulla strada come il possibile attraversamento improvviso di una persona), predizione di disastri meteorologici o terremoto.

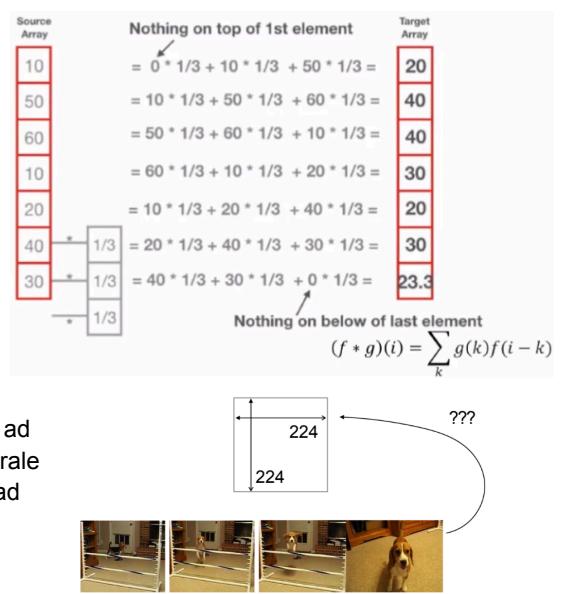
2	3	8	2	9	1	1	1	1	8
3	3	2	8	6	9	6	5	1	3
8	8	1	8	1	6	9	8	3	4
1	0	2	7	6	0	9	1	4	5
7	1	4	4	4	4	4	7	9	
3	1	8	9	3	4	2	7	2	3
6	6	1	6	3	4	3	3	9	0
8	1	0	8	3	5	1	8	3	4
9	9	1	1	3	0	5	9	5	4
1	1	8	6	9	8	3	1	1	0

5.1 Temporal Convolution Networks

Le **Temporal Convolution Networks (TCN)** sono un tipo di rete neurale basata su CNN utilizzata per l'elaborazione di dati sequenziali. Le TCN sono progettate per catturare relazioni e pattern temporali nei dati utilizzando convoluzioni (simili a come le reti neurali convoluzionali (CNN) sono utilizzate per l'elaborazione delle immagini) solo che qui viene considerata anche un'altra variabile che è il tempo. Le TCN utilizzano convoluzioni dilatate, in cui i filtri di convoluzione sono applicati a intervalli crescenti all'interno della sequenza. Ciò consente alle TCN di catturare pattern a diverse scale temporali, poiché i filtri possono coprire regioni temporali di lunghezza variabile.

Si consideri un kernel g che dato un array 1D, processa i dati in maniera sequenziale tre per volta. Questo kernel potrebbe essere visto come un metodo per predire una eventuale quarta posizione sulla base delle tre precedenti. Si noti che il kernel in questo caso è composto da valori fissi pari ad $\frac{1}{3}$ che, in questo caso, corrispondono ad uno smoothing filter. Nel caso generale avremo dei **filtri apprendibili** il cui valore è α che possono essere utilizzati, ad esempio, per fare forecasting.

Bisogna considerare alcuni problemi:



- Lunghezza finita della sequenza:** Le TCN sono progettate per elaborare **sequenze di dati di lunghezza fissa e statica** (es. AlexNet è progettato per elaborare immagini 224x224). Nel caso di sequenze arbitrarie, si possono concatenare (un numero fisso di) frame come input.
- Output a singola scelta:** L'output è una scelta singola da un elenco fisso di opzioni, tuttavia è possibile prevedere una parola alla volta o un numero fisso di parole in uscita.

Nell'esempio iniziale, abbiamo visto un filtro che prendeva tre elementi consecutivi che venivano messi in convoluzioni e generavano un output. Questo comportamento descrive appieno il grafico a destra. Nel caso delle lettere, queste vengono incorporate (embedded) in un token (il risultato è quindi un array di numeri che rappresenta una parola). Abbiamo quindi l'operazione di convoluzione che processa la parola corrente, quella precedente e quella successiva (come nell'esempio iniziale) e questo genera un certo risultato r_i . I **modelli che effettuano forecasting non usano una struttura del genere** perché è necessario per garantire che le previsioni siano basate solo su dati passati e che non includano informazioni future.

La **causal convolution** impone una direzione temporale in avanti nella computazione della convoluzione, assicurando che il modello **non "guardi" o faccia uso di dati successivi nel processo di previsione**. Quindi ipotizzando

di avere un certo input $X_T = X_0, X_1, \dots, X_{T-1}, X_T$, questo viene utilizzato per predire X_{T+1} . Ricorda che l'input di una TCN è fissato quindi per effettuare la predizione X_{T+1} si deve utilizzare un certo input fissato (es. 3 posizioni). Una volta predetto X_{T+1} , si deve shiftare l'input verso destra e continuare in questa maniera per predire la nuova posizione.

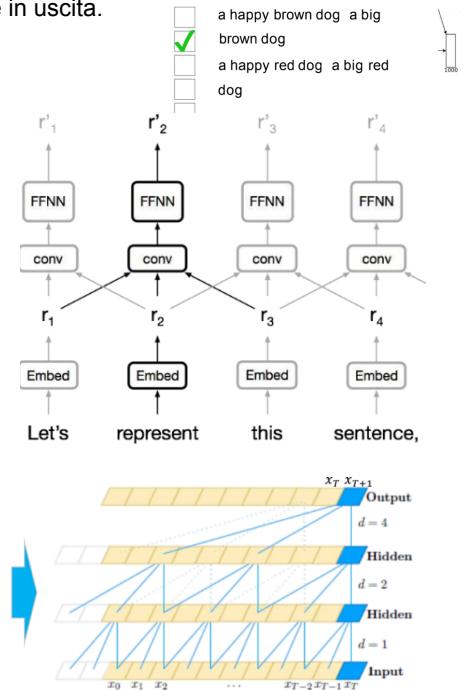
Si noti che questo tipo di architettura prevede:

- Una facile parallelizzazione:** Questo significa che i calcoli per diverse parti della sequenza temporale possono essere eseguiti contemporaneamente, sfruttando al massimo le risorse computazionali disponibili (es. sapendo che abbiamo k possibili input formato da 3 posizioni, possiamo calcolare tutte insieme in maniera parallela).
- Sfruttamento delle dipendenze locali:** Le TCN sono efficaci nel catturare relazioni locali o dipendenze a breve termine nei dati sequenziali (grazie all'operazione di convoluzione, esattamente come avviene per le singole immagini nelle CNN). Ciò è possibile grazie all'uso di convoluzioni dilatate, che consentono di esplorare regioni temporali con diverse scale di dilatazione. Le convoluzioni dilatate consentono alle TCN di acquisire informazioni a diverse distanze temporali all'interno della sequenza.

Si noti nell'esempio, come possiamo calcolare l'output di ognuna tripla utilizzando diverse scale di dilatazione.

Un problema delle TCN è che la **"distanza di dilatazione"** è **lineare con il numero di layers**. La "distanza di interazione" si riferisce a quanto lontano uno strato di una rete può influenzare uno strato successivo e questa distanza di interazione può essere influenzata da vari fattori, tra cui la dimensione dei filtri e il numero di strati nella rete. Si ricordi che la grandezza di receptive field con una grandezza dei filtri K e L layers ha dimensione $1+L^*(K-1)$ quindi all'aumentare del numero di strati e della dimensione dei filtri, la rete sarà in grado di catturare relazioni a lungo termine all'interno della sequenza in modo più efficace. Questo significa che **per catturare dipendenze a lungo termine all'interno di dati sequenziali**, come ad esempio il riconoscimento di pattern o relazioni complesse, **è spesso necessario avere una rete con un gran numero di layers** per consentire una distanza di interazione sufficientemente ampia (es. Nel caso in cui si voglia processare una sequenza di lunghezza mille, abbiamo bisogno di circa 500 layers che è un numero molto elevato. Quello che potremmo fare è aumentare la grandezza dei kernel per aumentare il receptive field ma questa è un'operazione che si vuole evitare in quanto si preferiscono kernel piccoli).

Le convoluzioni dilatate (**dilation**) consentono a un convolutional layer di avere un receptive field più ampio senza aumentare il numero di parametri o la complessità computazionale. Ciò significa che è possibile catturare dipendenze in un intervallo di tempo più ampio all'interno della sequenza in ingresso senza aggiungere ulteriori layer o filtri. Il fattore di dilatazione determina come il kernel convoluzionale campiona la sequenza in ingresso e



controlla la distanza tra i punti campionati. Modificando il **fattore di dilatazione d**, è possibile controllare efficacemente la capacità di una TCN di catturare dipendenze a breve o lungo termine nei dati. Fattori di dilatazione più piccoli catturano dipendenze a breve termine, mentre fattori di dilatazione più grandi catturano dipendenze a lungo termine. Quindi mentre prima c'era una dipendenza lineare tra output e la grandezza dell'input, con la dilatazione la **dipendenza è logaritmica** e dipende dal fattore di dilatazione.

Nota: Questo fattore di dilatazione di deve scegliere in base all'applicazione che stiamo cercando di costruire. Nel caso di un rilevatore di terremoti, che è un evento a breve termine, vorremmo evitare di perdere frame quindi occorre un fattore pari a $d=1$. Al contrario, se abbiamo un evento a lungo termine si può scegliere d più grande. Di nuovo, se non usiamo dilatazione, questo richiede l'utilizzo di molti layers.

5.2 Recurrent Neural Networks

Le **Recurrent Neural Networks (RNN)** sono un tipo di rete neurale progettata per elaborare dati sequenziali o dati che hanno una struttura temporale. Sono particolarmente adatte per problemi in cui è importante considerare l'ordine e le dipendenze tra elementi all'interno della sequenza. La caratteristica chiave delle RNN è la presenza di uno o più strati ricorrenti, noti come **unità ricorrenti** o celle, che consentono al modello di mantenere uno **stato interno** (o una memoria) che alla rete di catturare e memorizzare informazioni da osservazioni precedenti nella sequenza e di utilizzare queste informazioni per elaborare gli elementi successivi, quindi lo stato interno viene aggiornato ogni volta che una sequenza viene processata.

Un'unità ricorrente prende in ingresso l'input corrente, lo stato precedente (la memoria) e produce un output y e uno stato successivo. Questo output può essere utilizzato per fare previsioni o ulteriori elaborazioni, mentre lo stato successivo viene passato alla successiva iterazione della RNN. Questo ciclo di input, elaborazione e stato successivo si ripete per ogni elemento nella sequenza. Questo significa che il nuovo stato interno calcolato ad ogni iterazione, sarà funzione dello stato interno precedente e del nuovo input. L'output invece è funzione dello stato interno e dell'input correnti.

Possiamo elaborare una sequenza di vettori x applicando una **formula di ricorrenza** (recurrence formula) ad ogni passo temporale:

$$h_t = f_W(h_{t-1}, x_t)$$

new state
 / \ old state input vector at
 some function some time step
 with parameters W

$$y_t = f_{W_{hy}}(h_t)$$

output
 another function
 with parameters
 W_{hy}
 new state

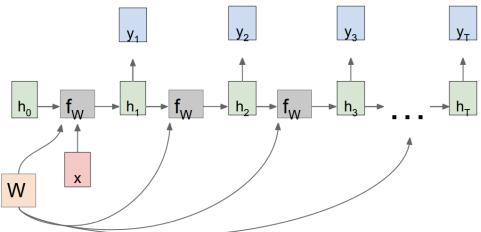
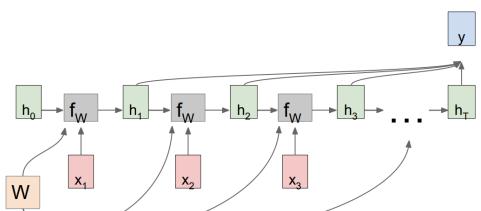
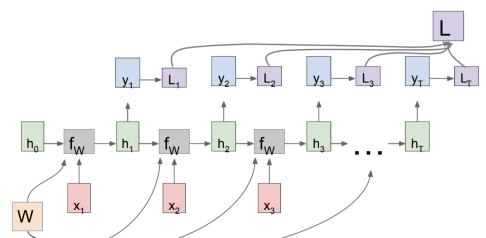
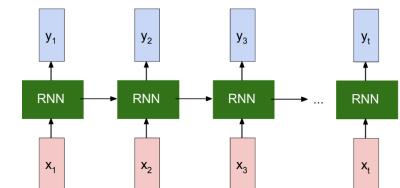
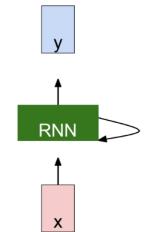
Quindi, nel **"Unrolled RNN"** abbiamo una sequenza di input x_1, x_2, \dots, x_t , una serie di stati nascosti h_1, h_2, \dots, h_t che vengono propagati nell'unità ricorrente successiva, e un numero di output y_1, y_2, \dots, y_t .

Nota: In ogni fase temporale vengono utilizzati la stessa funzione e lo stesso set di parametri (f_W). Possiamo togliere f_W dalla formula in quanto non è altro che una moltiplicazione con i parametri W e ce ne sono di due tipi: quelli che mappano da uno stato interno precedente a quello successivo e quelli che mappano dall'input allo stato interno. Quindi, ricordando di applicare non linearità (in questo caso tanh) per evitare che la moltiplicazione tra matrici sia lineare alla fine, otteniamo:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \quad y_t = W_{hy}h_t$$

Quindi, se guardiamo il grafo computazionale, si noti come ogni strato viene ottenuto effettuando una moltiplicazione tra matrici dove i pesi W utilizzati sono sempre gli stessi. In ogni strato otteniamo un output y_i a partire dal quale si può calcolare la loss L_i . La loss totale L non è altro la somma di tutte le singole loss. Nota che questa è la struttura **many to many** difatti, dati in input diversi x_i , otteniamo un numero d output y_T con T il numero di stati.

Possiamo anche definire graficamente anche la struttura **many to one** in cui otteniamo vari input x_i e abbiamo un solo output y alla fine. Possiamo avere



anche un solo input x and d output y e questo definisce la struttura **one to many**.

Nota: Si prenda la struttura one to many (l'ultima immagine in basso a destra). Ogni funzione f_W prende in input gli stessi pesi W ma solo la prima prende in input x . Per poter calcolare il nuovo stato dobbiamo necessariamente avere a disposizione un certo input anche negli f_W successivi. Possiamo utilizzare 0 ma non è ideale fare il training su una sequenza che inizialmente prende x in input e poi tutti 0. Un'altra soluzione è quella di dare in input l'output appena calcolato. Quindi per calcolare h_i utilizziamo y_{i-1} . Questa struttura permette di effettuare una predizione **autoregressiva**.

EX. $h_2 = f_W(h_1, y_1)$

Nota 2: Chiaramente per poter utilizzare una struttura del genere dobbiamo fare in modo che x e y siano della stessa tipologia. Nel caso di image captioning, dovremmo avere bisogno di tre input anziché due: La parola precedente y_{i-1} , lo stato precedente h_{i-1} e anche un altro input extra che non è altro che il contesto.

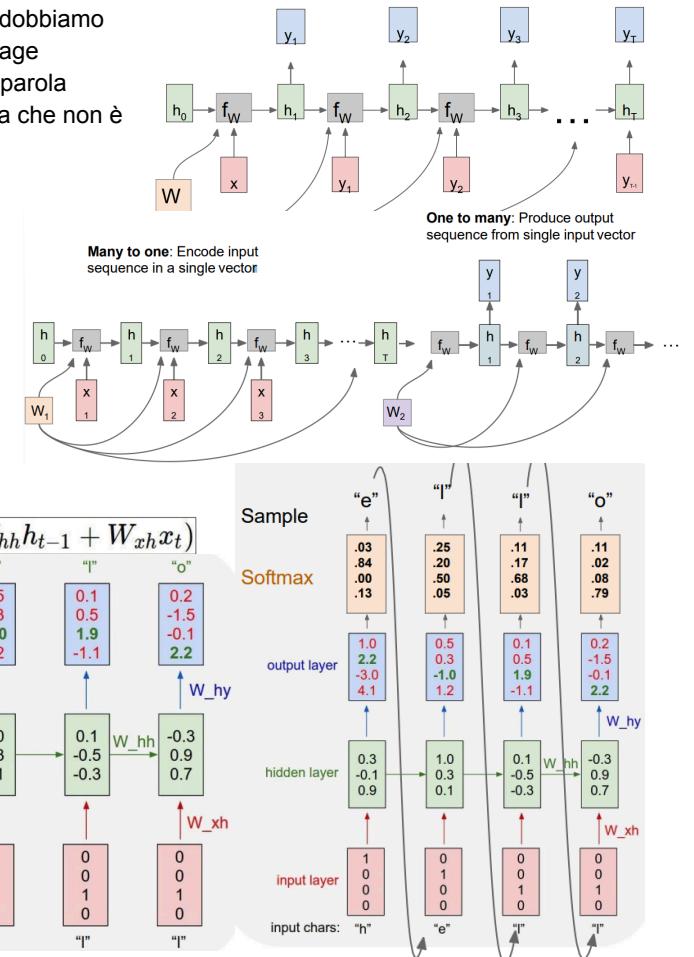
Ci sono anche altri modi per definire le RNN. Nel caso di approcci basati su **Seq2seq (sequence to sequence)** potremmo dover mescolare un approccio many-to-one e one-to-many. Si ipotizzi il caso in cui abbiamo una sequenza in input da tradurre, quello che facciamo è raggiungere un certo stato h_T che non fa altro che riassumere la sequenza di parole (codifica la sequenza in input in un singolo vettore). Una volta ottenuto questo array si effettua la sua predizione producendo la sequenza di output e questo viene effettuato dalla struttura one to many che prende in input, appunto, il vettore.

EX. Vediamo ora cosa succede su un esempio specifico. In particolare vediamo cosa succede nel training a livello di carattere in maniera da capire come scrivere le parole (**Character-level Language Model**).

In altre parole, dato un certo vocabolario in input, vogliamo capire come i caratteri si susseguono in una sequenza in maniera tale da rappresentare parole reali. Quindi, dato l'array di input $x = [h, e, l, o]$. Partendo da questo array, lo si codifica con un approccio one-hot (cioè dato l'indice i di ogni carattere nel vettore x , per ogni x_i creiamo un nuovo vettore one-hot tale che tutte le posizioni del vettore valgono 0 tranne l'indice della parola nel vettore x in input). Ora, dall'input calcoliamo l'hidden layer dove ogni elemento è dato dal processamento del precedente stato nell'hidden layer e il relativo input. Da qui, possiamo produrre l'output che è della stessa tipologia dell'input. Cioè, dato un certo carattere in input, vogliamo predire il successivo (es. nella parola "hello", dopo "h" c'è "e"). Possiamo usare quindi una funzione softmax che prende l'output e utilizza la cross entropy per capire se la lettera predetta è esattamente quella successiva (quindi la predizione viene comparata con il ground-truth ad ogni step e i pesi vengono via via aggiornati con la backpropagation). Tutto ciò avviene durante il **training**.

Durante il **testing**, si dà al modello creato una lettera alla volta e si produce la parola successiva (nota che nel softmax prendiamo sempre la lettera con il punteggio maggiore).

L'esempio visto finora viene chiamato **GT forcing (ground truth forcing)**. Questo si riferisce a una tecnica di training in cui, durante la fase di addestramento di un modello generativo, il modello riceve come input il ground truth al passo corrente e viene quindi addestrato per generare l'output successivo in modo coerente con questo input noto.



EX. Nell'esempio precedente diamo in input "hel" che è il ground-truth e questo ipotizzando di voler predire la lettera alla quarta posizione (che dovrebbe essere una l). Stiamo quindi forzando la rete a generare la lettera successiva basandosi su ciò che dovrebbe essere secondo il vocabolario. Questo processo continua passo dopo passo fino alla fine della sequenza.

Nota: Si noti che durante il training utilizziamo il GT forcing appena descritto mentre nel testing diamo in pasto una sola lettera e otteniamo una predizione di conseguenza.

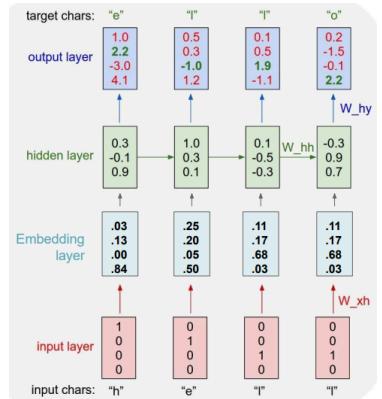
Nota: Nell'esempio precedente abbiamo descritto le parole utilizzando un vettore one-hot ma in generale non è una buona pratica in quanto accadrà che una singola parola campionerà solo una riga specifica nella matrice. Quindi, quando si apprende qual è la lettera successiva ad h, si apprende solo la prima riga

della matrice, quando si apprende qual è la lettera successiva ad l, si apprende solo la seconda riga della matrice, e così via. Quello che si fa è quindi aggiungere un "**embedding layer**" tra l'input layer a l'hidden layer.

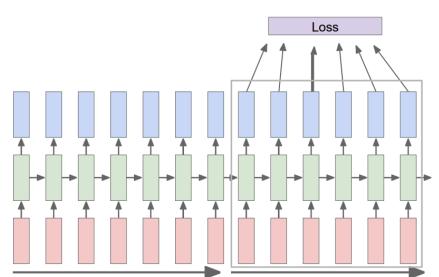
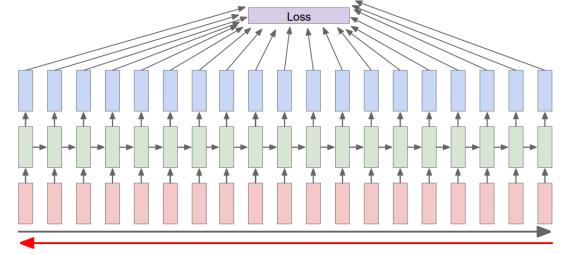
$$\begin{aligned} [w_{11} w_{12} w_{13} w_{14}] [1] &= [w_{11}] \\ [w_{21} w_{22} w_{23} w_{14}] [0] &= [w_{21}] \\ [w_{31} w_{32} w_{33} w_{14}] [0] &= [w_{31}] \\ &[0] \end{aligned}$$

Andiamo nel dettaglio di come funziona la backpropagation in sistemi del genere. La "**backpropagation through time**" (BPTT) si tratta di una variante della backpropagation utilizzata per calcolare i gradienti dei pesi di una RNN, tenendo conto dell'elaborazione sequenziale nel tempo. Il funzionamento è il seguente:

1. **Forward Pass:** Durante il training, i dati di input vengono elaborati sequenzialmente uno step alla volta. Ogni step di input viene passato attraverso la RNN, e la RNN mantiene uno "stato nascosto" (hidden layer) che tiene traccia delle informazioni elaborate fino a quel momento (come una sorta di memoria);
2. **Calcolo del Gradiente:** Dopo aver completato l'elaborazione dell'intera sequenza, si calcola la loss andando a comparare la predizione con il ground-truth. Si calcolano quindi i gradienti dei pesi della RNN utilizzando la BPTT. Inizialmente, si calcolano i gradienti per l'ultimo step temporale, proprio come in una rete feedforward. Quindi, i gradienti vengono "retropropagati" all'indietro nel tempo, passando attraverso ciascuno dei passi temporali precedenti e questo viene effettuato utilizzando la chain rule;
3. **Aggregazione dei Gradienti:** Durante la backpropagation, i gradienti calcolati in ciascun passo temporale vengono sommati insieme. Questa somma rappresenta il gradiente totale rispetto a un dato parametro della RNN, considerando tutti i passi temporali. Questo gradiente totale è utilizzato per aggiornare i pesi e i parametri del modello.
4. **Aggiornamento dei pesi:** Si utilizzano quindi i **gradienti aggregati** per aggiornare i pesi e i parametri della RNN (es. usando SGD). Questo approccio tiene conto dell'errore cumulativo lungo tutta la sequenza temporale.



Nota: Si noti che questo processo non è parallelizzabile. Difatti se prendiamo il forward pass, per ottenere ad esempio l'output del secondo stato, dobbiamo necessariamente avere quello del primo stato e così via fino all'ultima sequenza temporale. Questo può essere un problema nel caso in cui si abbiano lunghe sequenze in quanto il costo computazionale per effettuare il training è molto elevato. Esiste quindi una variante della BPTT che è la **truncated backpropagation through time**. Questa soluzione prevede di eseguire il forward pass e la backpropagation limitandosi a piccoli chunk della sequenza anziché l'intera sequenza. Quindi si prende un chunk, si effettua il forward pass e una volta arrivati all'ultima sequenza temporale nel chunk si effettua backpropagation. Quindi si prende l'output dell'ultimo stato nell'hidden layer nel chunk e si usa come input nel primo stato del chunk successivo e così via. In generale, gli stati nascosti vengono portati avanti nel tempo per sempre (chunk dopo chunk) ma si effettua backpropagation solo per un certo numero di step più piccoli dell'intera sequenza.



Nota: Si possono costruire anche **Multilayer RNN** che sono una variante delle reti neurali ricorrenti (RNN) standard in cui sono impilate **più strati di celle ricorrenti una sopra l'altra**. Questo approccio multistrato è stato sviluppato per affrontare alcune delle limitazioni delle RNN a singolo strato, tra cui la difficoltà nel catturare dipendenze a lungo termine nei dati sequenziali.

Un esempio di applicazione delle RNN sono i **language models**, cioè modelli progettati per comprendere e generare testo umano. Si pensi ad esempio alla predizione della parola (frase) successiva mentre cerchiamo qualcosa su Google. Un language model non fa altro che **predire la parola successiva tenendo in considerazione le parole precedenti**:

$P(\text{next word} \mid \text{previous words})$

EX. Vedi esempio completo sulle slide (pag.73 - slide Sequence Models)

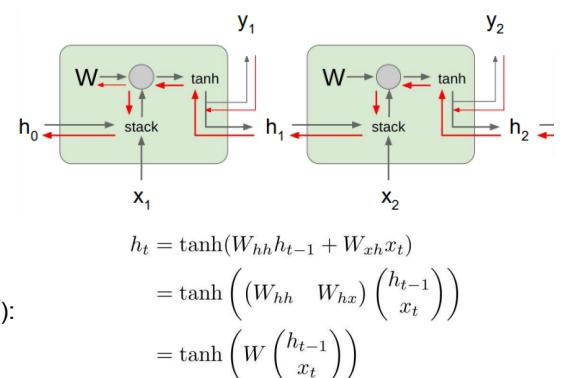
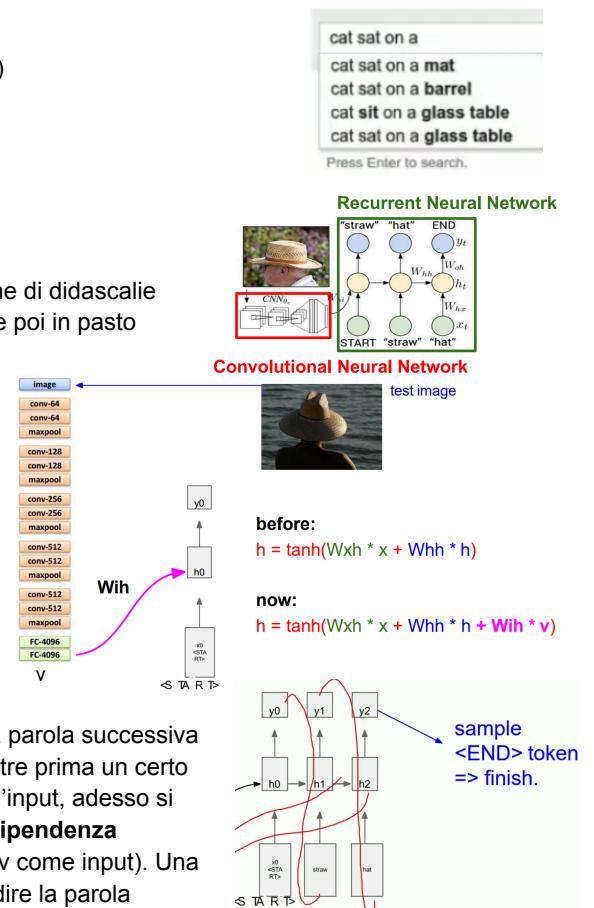
$P(\text{cat} \mid [\text{<S>}])$
 $P(\text{sat} \mid [\text{<S>}, \text{cat}])$
 $P(\text{on} \mid [\text{<S>}, \text{cat}, \text{sat}])$

Un altro esempio di utilizzo delle RNN è l'**image captioning** (generazione di didascalie per le immagini). Quello che si fa è codificare l'input con una CNN e dare poi in pasto questa codifica all'hidden layer della RNN. Questo mette quindi insieme due tipi di modelli: CNN (tipicamente pre addestrata) e RNN per predire le parole nella didascalia una dopo l'altra. Quindi:

- Estrazione delle Caratteristiche dell'Immagine: Per prima cosa, è necessario estrarre le caratteristiche rilevanti dall'immagine. Questo può essere fatto utilizzando una CNN pre addestrata. La CNN estrae le "feature map" dell'immagine. Le feature map estratte dall'immagine vengono quindi "pulite" e trasformate in un formato adatto per l'input alla RNN.
- Generazione della didascalia: La RNN viene successivamente utilizzata per generare la descrizione dell'immagine. Viene quindi effettuato sampling cioè ad ogni passo temporale, la RNN genera una parola e la utilizza come input per generare la parola successiva (questo finché non si raggiunge l'ultima sequenza). Quindi mentre prima un certo stato era il risultato dello stato precedente e una proiezione dell'input, adesso si deve **aggiungere un'altra componente che rappresenta la dipendenza dell'immagine** (quindi tutti gli strati devono anche considerare v come input). Una volta predetta la parola, questa viene aggiunta all'input per predire la parola successiva e così via.

Vediamo adesso come funziona il flusso dei gradienti nelle RNN. Il termine "**Vanilla RNN Gradient Flow**" si riferisce al flusso dei gradienti nelle reti neurali ricorrenti (RNN) standard, in particolare nei modelli RNN a strato singolo, noti anche come "**vanilla RNNs**". Questo concetto riguarda come i gradienti vengono calcolati e propagati attraverso la rete durante il processo di addestramento. Ipotizziamo di avere uno strato della RNN, questo prende in input l'output dello stato precedente h_{t-1} , l'input x_t e otteniamo l'output utilizzando la solita formula e applicando non linearità (la tangente iperbolica). Si noti che possiamo concatenare i pesi W_{hh} e W_{hx} assumendo che anche i due input sono concatenati. La backpropagation da h_t a h_{t-1} viene moltiplicata per W (in particolare W_{hh}^T):

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{hx}x_t)W_{hh}$$



Assumendo ora che abbiamo un certo numero di frame (non solo uno), la derivata della loss rispetto a W non è altro che la somma di tutte le loss L_t rispetto a W:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

Se consideriamo l'ultima loss calcolata nella sequenza temporale L_T e la prima W, abbiamo che (utilizzando la chain rule) la derivata della loss totale rispetto a W non è altro che:

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_T}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Si noti che le derivate interne sono le stesse quindi possiamo sostituire tutti quei termini con una

moltiplicazione. Mettendo tutto insieme:

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \tanh'(W_{hh} h_{t-1} + W_{xh} x_t) \right) W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

Nota: Si ricorda che tanh schiaccia l'input nell'intervallo [-1, 1] quindi abbiamo che nella parte in rosso facciamo una moltiplicazione di elementi sempre minori di 1. Il prodotto quindi soffre del problema del **vanishing gradient** in cui il gradiente tenderà sempre di più a zero. Assumiamo ora che non viene applicata la funzione di attivazione. Quello che succede è che eliminiamo da derivata di tanh e rimaniamo con W_{hh}^{T-1} (un'esponenziazione della matrice W_{hh} per $T-1$ volte dove T è la lunghezza della sequenza). Nel momento in cui calcoliamo l'esponenziazione di una matrice, possiamo avere due comportamenti che dipendono dal valore più grande nella matrice:

- Se questo è maggiore di uno allora abbiamo il problema dell'**esplosione dei gradienti**. Per risolvere questo problema possiamo applicare **gradient clipping** cioè ogni volta che il gradiente supera un certo threshold, lo scaliamo al valore del threshold **evitando che la norma risulti troppo grande**.
- Se invece è minore di uno allora abbiamo il problema del **vanishing gradient**. In questo caso non possiamo fare nulla se non cambiare la struttura dell'RNN per evitare il gradient flow passi attraverso il nodo che genera il problema (dove viene effettuata la moltiplicazione tra matrici).

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
    grad *= (threshold / grad_norm)
```

Alcuni vantaggi delle RNN sono:

- Può elaborare qualsiasi input di lunghezza;
- Il calcolo per lo step t può (in teoria) utilizzare informazioni provenienti da molti passi indietro;
- Le dimensioni del modello non aumentano per input più lunghi;
- Stessi pesi applicati a ogni passo temporale, quindi c'è simmetria nel modo in cui vengono elaborati gli input.

Alcuni svantaggi sono:

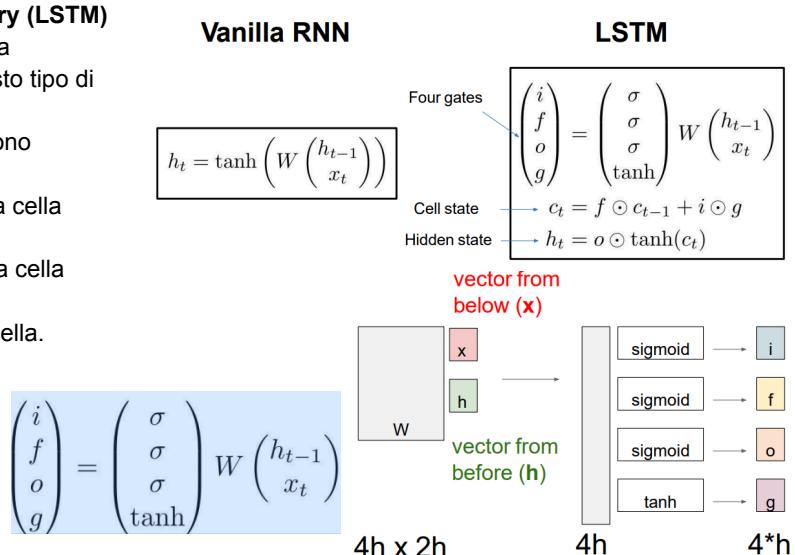
- Il calcolo ricorrente è lento (non abbiamo parallelizzazione);
- In pratica, è difficile accedere alle informazioni da molti passi indietro.

Un'alternativa alle RNN è la **Long Short-Term Memory (LSTM)**

che è una tipologia di cella ricorrente che può aiutare a combattere il problema del vanishing gradient. In questo tipo di cella abbiamo quattro gate:

- Input gate: Controlla quali informazioni vengono aggiunte alla cella;
- Forget gate: Controlla quali informazioni nella cella vengono dimenticate;
- Output gate: Controlla quali informazioni nella cella vengono utilizzate per l'output;
- Gate gate: Controlla "quanto scrivere" nella cella.

Per propagare la memoria dal primo strato al successivo, non utilizziamo h (come nella vanilla RNN) ma la **cell state** c_t che mantiene la memoria e si basa sulla cell state precedente c_{t-1} sul percorso e l'input. Abbiamo poi l'hidden state che calcola la tangente iperbolica del cell state c_t e produce un

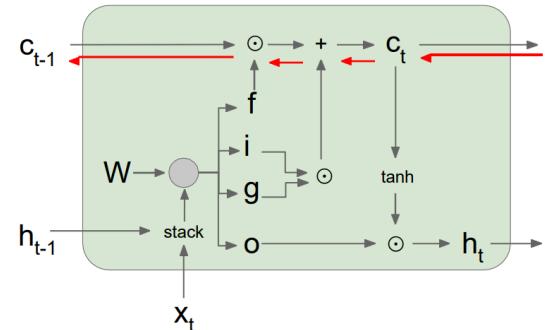


output. Quindi, lo stato nascosto contiene l'output del passo temporale corrente, mentre lo stato della cella contiene la memoria a lungo termine.

Tre delle quattro porte sono sigmoidi quindi bloccano o fanno passare il segnale (ricorda che il sigmoide ritorna un risultato che è 0 o x) mentre l'altra è tanh è corrisponde alla precedente non linearità che schiaccia l'input tra -1 e 1. Queste porte sono quindi moltiplicate per W e la concatenazione di h_{t-1} e x_t .

Vediamo quindi come funziona la backpropagation in questo tipo di architettura:

- A ogni passo temporale, la LSTM riceve un nuovo input h_{t-1} e x_t dato dallo strato precedente e l'input al passo t. L'input viene elaborato dalla input gate che determina quali informazioni sono rilevanti e dovrebbero essere aggiunte alla memoria a lungo termine. La input gate decide quali informazioni dall'input dovrebbero essere aggiunte alla memoria a lungo termine.
- La forget gate determina quali informazioni nella memoria a lungo termine dovrebbero essere dimenticate. Questo è un passaggio cruciale per garantire che la cella non accumuli informazioni obsolete. L'operazione di moltiplicazione tra la vecchia cell state e la final state determina quali informazioni precedenti vengono mantenute e quali vengono eliminate dalla memoria a lungo termine.
Nota: Ricorda che tutte queste porte sono sigmoidi quindi lasciando passare un dato o meno in base a se questo vale 0 o x.
- Si aggiorna quindi la cell state facendo una somma ponderata tra il prodotto tra la input gate e la gate gate che decide quante informazioni scrivere.
- L'output gate determina quali informazioni dalla memoria a lungo termine dovrebbero essere utilizzate per calcolare lo stato nascosto. Prendiamo quindi il cell state aggiornato c_t , vi applichiamo la funzione di attivazione tanh e ne facciamo il prodotto con la output cell generando l'input per il nuovo stato h_t nonché l'output.



La backpropagation da c_t a c_{t-1} avviene effettuando la moltiplicazione element per element (element-wise multiplication) per f non effettuando quindi alcuna moltiplicazione di matrici per W.

Si noti che:

- Il gradiente contiene il vettore di attivazioni del gate f e questo consente un migliore controllo dei valori dei gradienti, utilizzando opportuni aggiornamenti dei parametri del forget gate.
- Viene aggiunto un maggiore controllo tramite le porte f, i, g ed o apportando un miglior bilanciamento dei valori del gradiente.

LSTM risolve completamente il problema del vanishing gradient?

L'architettura LSTM rende più semplice per la RNN preservare le informazioni a lungo termine nelle sequenze temporali. Per esempio, se $f = 1$ e $i = 0$, l'informazione di quella cella viene conservata indefinitamente. Al contrario, è più difficile per la RNN vanilla apprendere una matrice di pesi ricorrente W_h che preservi le informazioni nello stato nascosto. **LSTM non garantisce che non si verifichino i problemi del vanishing/exploding gradient, ma fornisce al modello un modo più semplice per apprendere dipendenze a lungo termine.**

Si può mettere in parallelo la backpropagation tra le cell state con la mappatura dell'identità (identity mapping) in ResNet. Quindi per preservare il gradiente, si può creare una scorciatoia creando una connessione diretta tra la loss e ogni parametro di cui vogliamo calcolare il gradiente.

In between:
Highway Networks
 $g = T(x, W_T)$
 $y = g \odot H(x, W_H) + (1 - g) \odot x$

Nota: Un'altra alternativa alle RNN (oltre LSTM) sono le **Gated Recurrent Unit (GRU)**.

Per riassumere:

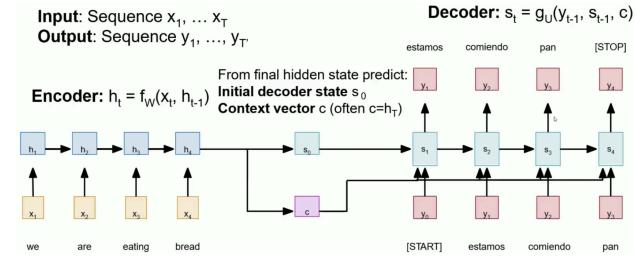
- Le RNN consentono molta flessibilità nella progettazione dell'architettura;
- Gli RNN Vanilla sono semplici ma non funzionano molto bene;
- Molto spesso si usano LSTM o GRU: Le loro interazioni aggiuntive migliorano il gradient flow;
- Il backpropagation dei gradienti nella RNN può esplodere o svanire. L'esplosione è controllata con il gradient clipping. La vanishing gradient è controllato con interazioni aggiuntive (LSTM);
- Architetture migliori/più semplici sono un tema caldo della ricerca attuale, così come nuovi paradigmi per ragionare sulle sequenze;
- È necessaria una migliore comprensione (sia teorica che empirica).

5.3 Transformer Networks

Le **transformer networks** sono un altro tipo di sequence model note per l'assenza di strati ricorrenti o LSTM nelle reti neurali e si basano invece su una struttura basata su **meccanismi di attenzione**.

Immaginiamo di avere una sequenza di input x_1, \dots, x_T e una sequenza di output y_1, \dots, y_T composte da una sequenza di parole. Queste vengono processate una dopo l'altra da un **decoder** con l'obiettivo di comprendere la sequenza di parole e tradurle in un'altra lingua. Il decoder non fa altro che elaborare le parole una dopo l'altra generando un nuovo stato h_t dove h_t è il risultato della combinazione tra l'input corrente x_t e lo stato precedente h_{t-1} .

Prima di iniziare a tradurre abbiamo bisogno di conoscere il **contesto della sequenza** (es. ricorda esempio fatto nelle sezioni precedenti) per esprimere cosa la macchina capisce dalla sequenza. Quindi possiamo immaginare una RNN che funge da **decoder** cioè che prende in input tre componenti che sono la parola corrente y_t , l'hidden state precedente s_{t-1} e l'intero contesto c (es. l'immagine nell'esempio dell'image captioning) e, ad ogni step, elabora l'hidden state s_t successivo e anche l'output y_{t+1} fino alla traduzione dell'intera sequenza.



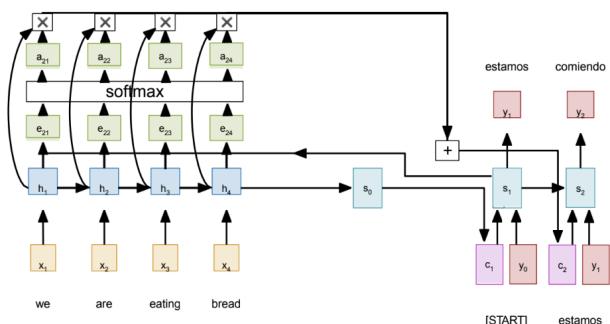
Nelle RNN tradizionali, l'elaborazione di ogni elemento della sequenza genera uno stato nascosto. Tuttavia, l'output di ogni passo temporale deve essere rappresentato da un **vettore con dimensioni fisse** dove la dimensione è solitamente stabilita prima dell'addestramento del modello. Ciascun elemento della sequenza deve essere rappresentato in questo vettore a dimensione fissa. In altre parole, non importa quanto lunga sia la sequenza iniziale, l'informazione è compressa in un vettore di dimensioni fisse. Questo può portare a una perdita di informazione, specialmente in sequenze molto lunghe, in quanto l'informazione deve essere "forzata" a passare attraverso questa dimensione fissa. La soluzione potrebbe essere quella di usare un **context vector ad ogni step del decoder** assumendo quindi di avere un contesto ogni volta che si cerca di tradurre una singola parola.

Introduciamo quindi il concetto di **attenzione**.

Ipotizziamo quindi che abbiamo degli hidden state generati a partire da una certa sequenza di parole (come nell'esempio precedente). Dall'ultimo hidden state possiamo inizializzare il **decoder state iniziale s_0** . Quello che vorremmo fare adesso, è capire quanto un certo hidden state h_t è influenzato dal precedente hidden state h_{t-1} della sequenza in input attraverso una sorta di **alignment score $e_{t,i}$** . Questo alignment ci dice quanto il decoder state corrente s_{t-1} si riferisce agli hidden state h_i con $i=1,\dots,T$ (con T la lunghezza della sequenza):

$$e_{t,i} = f_{att}(s_{t-1}, h_i) \quad (f_{att} \text{ is an MLP})$$

Partendo dagli alignment scores, possiamo normalizzare tali valori attraverso una funzione softmax che li schiaccia nell'intervallo 0, 1 (la loro somma vale 1) ottenendo gli **attention weights** che ci dicono il peso di ogni hidden state della sequenza in input:



$$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$$

Possiamo quindi calcolare il **contesto della prima parola** come **somma pesata degli attention weights $a_{t,i}$** :

$$c_t = \sum_i a_{t,i} h_i$$

A questo punto procede tutto come nel precedente esempio poiché abbiamo il contesto c_t , l'ultima parola predetta y_{t-1} e lo stato precedente s_{t-1} e li possiamo usare per calcolare lo stato successivo s_t che ci sanno input la parola successiva y_t (nota che in questo caso consideriamo l'ultima parola predetta come "Start"):

$$s_t = g_U(y_{t-1}, s_{t-1}, c_t)$$

Gli attention weights sono quindi necessari per codificare quando ogni parte dell'input è rilevante per la successiva porzione di testo che stiamo traducendo.

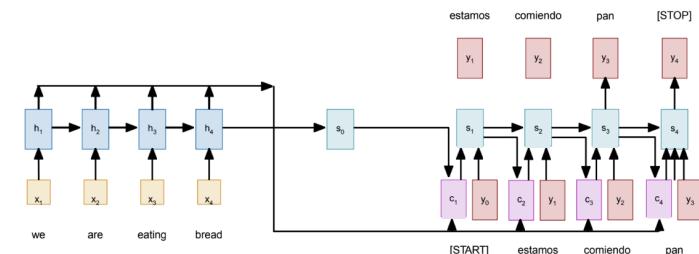
EX. Nel primo caso il picco di attenzione si potrebbe avere nel verbo cioè "estamos" = "we are" dove $a_{11}=a_{12}=0.45$ (queste corrispondono appunto a "we are") e $a_{13}=a_{14}=0.05$ (nota che la loro somma è 1).

Nota: Si noti che tutto questo è differenziabile ed inoltre l'attenzione prodotta non è supervisionata (non viene richiesto di etichettare in qualche modo le parole per generare l'attenzione prodotta).

Possiamo quindi procedere in questo con le parole successive. Possiamo quindi usare s_1 per calcolare il nuovo context vector c_2 per generare la parola successiva usando la stessa funzione di prima e continuare in questo modo finché non si predice "Stop".

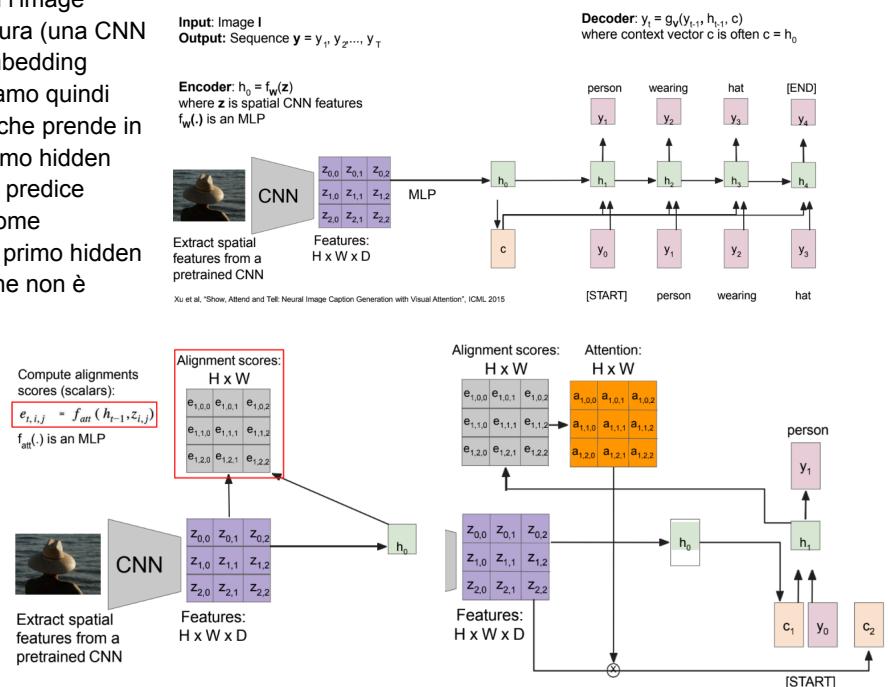
Quindi, usando un context vector differente ad ogni step del decoder:

- La sequenza di input non ostacolata dal singolo vettore;
- Ad ogni passo temporale del decoder, il context vector "guarda" diverse parti della sequenza di input.



Nota: Si noti che il decoder **non tiene conto del fatto che h_i rappresenta una sequenza ordinata di parole** difatti la gestisce come una sequenza non ordinata $\{h_i\}$ quindi otterremo la stessa architettura dell'immagine sopra anche se proviamo a cambiare a caso il set di hidden state $\{h_i\}$. Questo può essere risolto con il **positional encoding** (vedi avanti)

Ora proviamo a replicare quello visto finora con l'image captioning. Abbiamo un'immagine I e una struttura (una CNN pre addestrata) che permette di ottenere un embedding dell'immagine (una feature map $H \times W \times D$). Abbiamo quindi bisogno di un certo modello di linguaggio MLP che prende in input le feature spaziali della CNN, genera il primo hidden state h_0 e quindi parte da un certo "start" token, predice alcuni parole e termina con uno "stop" token (come nell'esempio precedente). Quindi, partendo dal primo hidden state h_0 possiamo creare un contesto c (nota che non è obbligatorio creare un contesto da h_0 , potremmo anche ottenerlo in qualche modo dall'embedding - questo è solo un esempio). Nuovamente, il contesto ha una dimensione fissa per rappresentare l'immagine intera. Quindi iniziamo a descrivere l'immagine con il **decoder** che prende in input il contesto c , l'ultima parola predetta y_{t-1} e lo stato precedente h_{t-1} producendo la parola successiva y_t . Nel primo step la parola iniziale è "start" e produciamo "person".



Continuiamo sempre con lo stesso approccio (ricorda che il contesto è sempre lo stesso!) finché non si predice "stop". Di nuovo, l'input è in qualche modo ostacolato dal context vector c che codifica l'intera immagine e questo può essere un problema nel caso in cui si vogliamo generare lunghe descrizioni.

Di nuovo, l'idea è quindi quella di utilizzare un **context vector differente ad ogni step** in maniera tale che ogni vettore possa **porre l'attenzione su diverse regioni dell'immagine**. Quello che si può fare è ottenere l'hidden state precedente del decoder e lo si va a comparare con l'embedding dell'immagine per ottenere degli **alignments score** ($H \times W$) che vengono poi normalizzati per ottenere gli **attention scores** ($H \times W$):

$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$

$$a_{t,:,:} = \text{softmax}(e_{t,:,:})$$

Ora, possiamo pesare ogni encoder (ricorda che in questo caso ogni encoder dell'embedding si concentra su una piccola porzione dell'immagine) producendo un **context vector**:

$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

Quindi, di nuovo, ogni step del decodificatore utilizza un context vector diverso che esamina parti diverse dell'immagine di input. Il decoder prende in input contesto corrente c_t , stato precedente h_{t-1} e parola precedente y_{t-1} (nel primo step "start") e produce lo stato corrente h_t nonché l'output y_t .

Di nuovo, questo processo è differenziabile ed inoltre il modello sceglie da solo gli attention weights non richiedendo un processo supervisionato.

Nota: Il punto in cui porre l'attenzione può essere soft o hard. Nel caso dell'hard attention, solo un sottoinsieme limitato di elementi di input viene selezionato per contribuire all'output, mentre gli altri vengono completamente ignorati (note che viene richiesto spesso l'uso di tecniche per capire qualche sottoinsieme dell'immagine porre l'attenzione, es. reinforcement learning). Nel soft attention, ogni elemento di input contribuisce all'output, ma con un peso diverso.

Facciamo un riassunto di quello visto finora nel caso dell'image captioning.

Abbiamo le nostre features z dell'immagine ($H \times W \times D$) e andandole a comparare con l'hidden state h del decoder (D) otteniamo gli alignment scores e_{ij} ($H \times W$).

Normalizziamo quindi gli alignment scores con una softmax ottenendo gli attention scores a ($H \times W$) e infine otteniamo il contesto effettuando una media pesata basata sugli attention score e le feature in input.

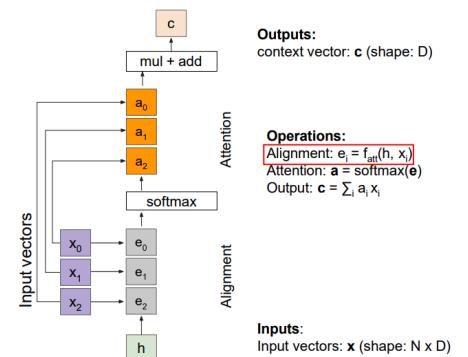
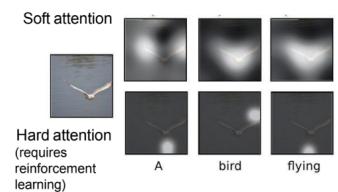
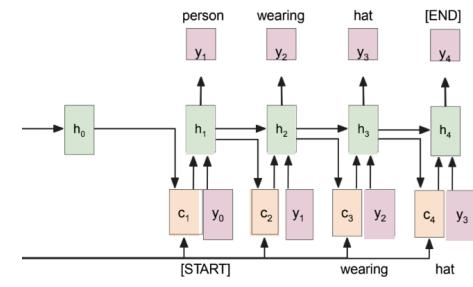
Ora, l'**operazione di attenzione è invariante per permutazione** quindi non tiene conto dell'ordinamento delle features, di conseguenza possiamo schiacciare l'immagine in maniera tale da avere $N = H \times W$ patch dell'immagine ognuna rappresentante una certa parte dell'immagine (quindi le feature in input assumono dimensione $N \times D$). Quindi, se prendiamo le varie patch nell'immagine e ne facciamo la permutazione (le mescoliamo), il risultato non cambia.

Definiamo quindi di nuovo gli alignment scores (cambiamo solo il fatto di schiacciare le features) come:

$$\text{Alignment: } e_i = f_{att}(h, x_i)$$

Prendiamo l'hidden state del decoder, l'embedding dell'immagine in input (un hidden state dell'encoder in quanto compariamo l'hidden state del decoder come ogni hidden state dell'encoder, uno alla volta) e utilizziamo un MLP che produce l'output che permette di pesare l'hidden state dell'encoder.

Nota: Ci sono svariate funzioni per calcolare gli alignment scores. Quella a cui si fa riferimento è quella additiva.



Name	Alignment score function	Citation
Content-base attention	$\text{score}(s_t, h_i) = \text{cosine}[s_t, h_i]$	Graves2014
Additive(*)	$\text{score}(s_t, h_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[s_t; h_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a s_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(s_t, h_i) = \mathbf{s}_t^\top \mathbf{W}_a h_i$ where \mathbf{W}_a is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(s_t, h_i) = \mathbf{s}_t^\top h_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(s_t, h_i) = \frac{\mathbf{s}_t^\top h_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.

(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

Spostiamoci adesso verso lo **scaled dot-product** che è la funzione tipicamente utilizzata nei transformers.

Cambiamo quindi $f_{att}(\cdot)$ a un semplice prodotto scalare (moltiplichiamo h e x_i):

Alignment: $e_i = h \cdot x_i$

Questa operazione è computazionalmente più pesante, difatti:

- Dimensioni maggiori (es. un'immagine più grande) significano più termini nella somma del prodotto scalare.
- Quindi, la varianza dei logit è maggiore. I vettori di grande magnitudo produrranno logit molto più alti.
- Quindi, la distribuzione post-softmax ha un'entropia inferiore, presupponendo che i logit siano IID.
- Infine, questi vettori di grande magnitudo causeranno il picco del softmax e assegneranno pochissimo peso a tutti gli altri (se tutti i valori hanno uno score basso allora avremo attention weights abbastanza bilanciati ma nel caso di valori più elevati allora solo pochi domineranno a causa dell'esponenziale).
- Dividi per \sqrt{D} per ridurre l'effetto dei vettori di grande magnitudo:

Gli hidden vector dei decoder diventano

quindi Alignment: $e_{i,j} = q_j \cdot x_i / \sqrt{D}$ **query vectors q** (di nuovo, questi li comparemo con ogni input x_i per

ottenere gli alignment scores che vengono normalizzati per ottenere gli attention scores a e produrre quindi gli hidden vector dell'encoder). I query vector q sono bidimensionali $M \times D$ dove M è la lunghezza dell'output che vogliamo produrre (es. la sequenza di parole nell'image captioning). Ogni query crea un nuovo **context vector y** (D) in output.

Nota inoltre che gli input vectors x_i vengono usati sia per l'allineamento che per l'operazione di attenzione.

Possiamo aggiungere più espressività al layer **aggiungendo un livello FC diverso prima di ciascuno dei due passaggi** (quando pesiamo gli input vectors e quando li prendiamo per generare gli alignment scores).

Introduciamo quindi il **general attention layer**. Aggiungiamo un fully connected layer per l'input vector (**chiave**) e un fully connected layer per effettuare la somma pesata (**valore**):

Key vectors: $k = xW_k$
Value vectors: $v = xW_v$

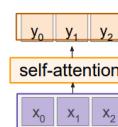
Il resto non varia se non per il fatto che occorre fare attenzione alle dimensioni. Le query vengono comparate alle chiavi k_i per generare gli alignment scores, che vengono normalizzati per ottenere gli attention scores e infine effettuiamo la somma pesata dei valori per ottenere gli output vector y .

Nota: Si noti che abbiamo **un grado di libertà maggiore** in quest'architettura poiché passiamo da un input $N \times D$ a query $q \times M \times D_k$ (M query ognuna D_k dimensionale) con k il numero di chiavi. Allo stesso tempo possiamo anche gestire la grandezza dell'output che diventa D_v con v il numero di valori.

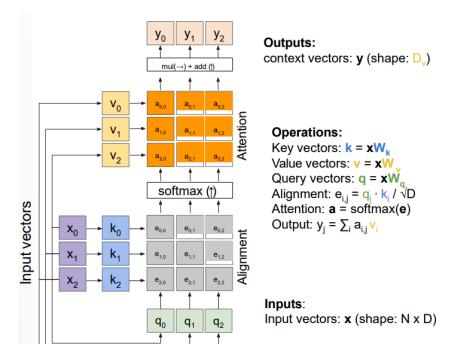
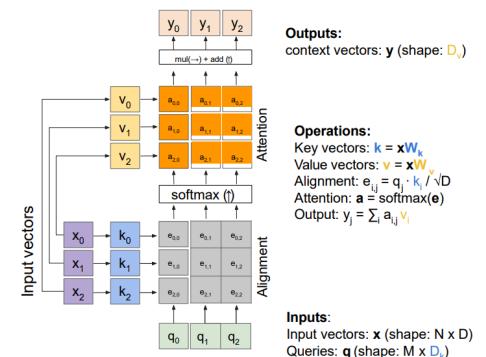
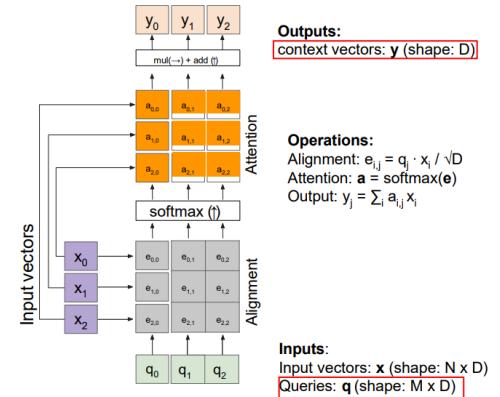
Ora, facciamo un altro step. Ricorda che gli hidden state del decoder erano funzione degli hidden state dell'encoder, cioè:

Encoder: $h_0 = f_w(z)$

Con $f_w(\cdot)$ un MLP e z le dimensioni spaziali delle features del CNN. Questo **vale anche per le query**. Possiamo calcolare i query vectors dagli input vectors definendo un "**self-attention**" layer (lo chiamiamo in questa maniera perché abbiamo gli input vectors x che in qualche modo vengono comparati con loro stessi una volta ottenute le query). Quindi quello che succede in quest'architettura finale è che andiamo una serie di input vectors x in input a una serie di context vectors y in output che usiamo per produrre le descrizioni delle immagini:



Un altro modo per vedere il funzionamento del self-attention è il seguente: Ogni parola in una frase viene incorporata (embedded) in un vettore e successivamente ogni rappresentazione della parola viene comparata l'una con l'altra. In base al risultato della comparazione sommiamo le rappresentazioni



pesate dall'output della comparazione ottenendo la rappresentazione di ogni singola parola per produrre il contesto di quella singola parola. Se si prova ad applicare quest'architettura su una vera frase quello che succede è che **certe parole saranno influenzate maggiormente da altre in base al contesto**.

EX. "It" è maggiormente influenzato da "the animal". Quindi la self-attention è una comparazione tra tutte le parole nella frase e le altre parole nella frase.

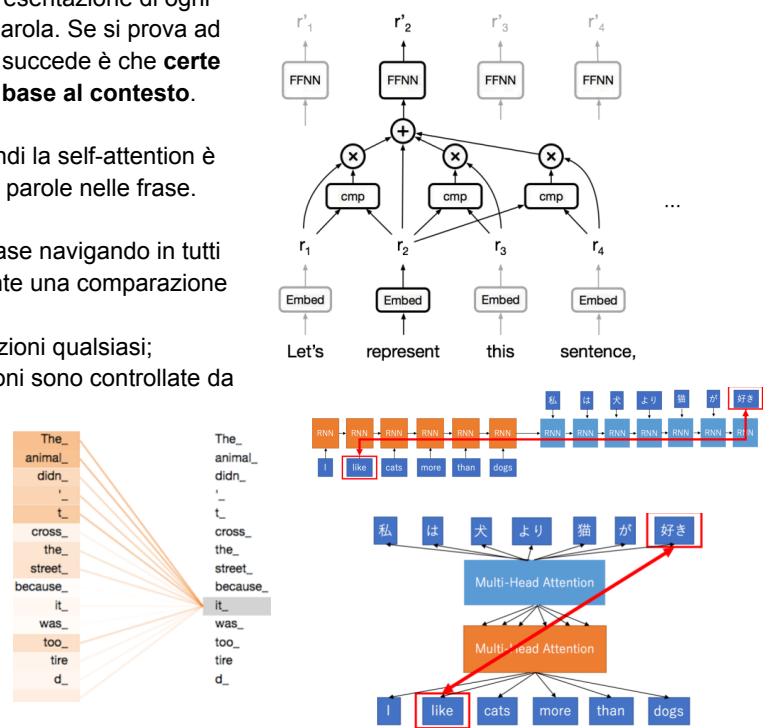
Quindi mentre nelle RNN dobbiamo analizzare tutta la frase navigando in tutti gli internal states, con l'attention abbiamo immediatamente una comparazione all-to-all diretta. Quindi, in questa architettura:

- "Lunghezza del percorso" costante tra due posizioni qualsiasi;
- Interazioni gating/moltiplicative (tutte le interazioni sono controllate da un prodotto scalare);
- Semplice da parallelizzare (per layer) (ogni parola è comparata a tutte le altre).

Alcuni costi computazionali (per layer) sono (nota come la complessità si riduce notevolmente tra RNN e Self-attention):

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n is hundreds, *d*=512 or 1024, *k* conv kernel size, *r* neighborhood reduced attention



Si introducono adesso la **Positional Encoding** e la **Masked-self attention**.

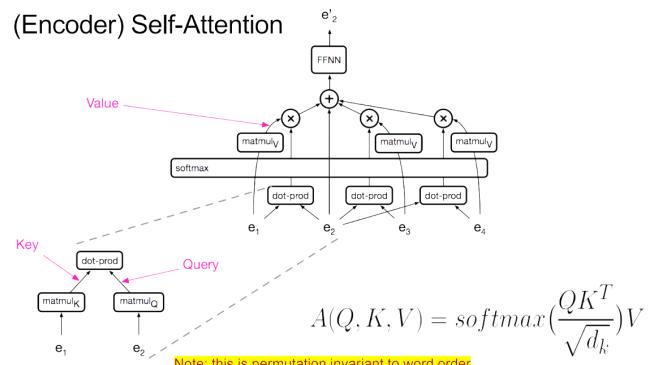
L'immagine a destra rappresenta la encoder self-attention vista finora. Quindi da una sequenza e_1, \dots, e_n in input viene effettuato il prodotto scalare (dot product) all-to-all. Si noti che in questo esempio ci si concentra sull'input e_2 che viene comparato con tutti gli altri, quindi si effettua il prodotto scalare tra e_2 e tutti gli altri elementi, si calcola la softmax (ottenendo l'attention scores) e quindi effettuiamo la somma pesata basandosi sulla comparazione degli elementi ottenendo quindi l'output e'_2 . In seguito all'applicazione di una FFNN (fully connected neural network - una MLP).

Si noti che **l'ordine non ha importanza** in questa architettura. I self-attention layers non tengono conto dell'ordine dell'input difatti diciamo che questo è **invariante alla permutazione**. Se non cambiamo l'ordine degli elementi nella sequenza di input, l'output finale comunque non cambierà.

Nota: Questo non avviene, ad esempio, nelle CNN dove abbiamo un filtro che scorre sull'immagine dove ad ogni posizione otteniamo un output che dipende dalla porzione di immagine che stiamo analizzando. Quindi se cambiamo l'ordine di qualche elemento nella sequenza (diciamo i pixel dell'immagine) anche l'output cambia. Anche nelle RNN abbiamo una nozione di ordine in quanto dobbiamo scorrere la sequenza un elemento per volta fino alla sua completa elaborazione. Al contrario, nel self-attention layer visto finora questo non si verifica in quanto effettuiamo un semplice prodotto scalare all-to-all.

Come possiamo codificare sequenze ordinate come il linguaggio o le feature di un'immagine ordinate spazialmente?

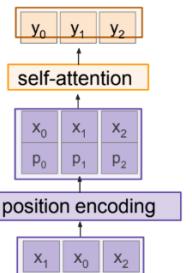
Si può fare introducendo il **positional encoding**. L'idea è quella di concatenare/aggiungere una codifica posizionale (positional encoding) speciale p_j ad ogni vettore in input x_j . Questa funzione **pos: $N \rightarrow \mathbb{R}^d$** mappa un numero naturale (l'ordine in una sequenza) ad una codifica (embedding) d -dimensionale a valori reali (quindi codifichiamo la



posizione j nel vettore con un vettore d -dimensionale), dove $\mathbf{p}_j = \text{pos}(j)$. Quello che vorremmo ottenere da questa funzione è:

- Dovrebbe produrre una **codifica univoca** per ogni passaggio temporale (posizione della parola in una frase). Quindi in base alla posizione nella sequenza dovremmo ottenere una codifica diversa;
- La **distanza tra due passaggi temporali** qualsiasi dovrebbe essere coerente tra frasi di diversa lunghezza. Quindi vorremmo poter codificare che, ad esempio, due parole si trovano ad una distanza d l'una dell'altra altra;
- Il nostro modello dovrebbe essere **generalizzato a frasi più lunghe** senza alcuno sforzo. I suoi valori dovrebbero essere limitati;
- Deve essere deterministico (non vogliamo randomicità nella generazione della codifica).

Senza l'encoding posizionale, il modello Transformer non sarebbe in grado di distinguere l'ordine dei dati di input, poiché il meccanismo di attenzione da solo non contiene informazioni sulla posizione.



Cosa possiamo implementare questa funzione pos(.)?

Abbiamo due opzioni:

- Si può apprendere una **lookup table**. Assumendo di voler T posizioni diverse, possiamo creare una lookup table con $T \times d$ parametri dove i **parametri T vengono appresi dalla rete**. Quindi per ogni numero naturale $t \in [0, T]$, si guarda nella lookup table, si prende la t -esima riga che contiene il vettore d -dimensionale (la codifica) e questa sarà la codifica posizionale che verrà poi concatenata all'input.
- Si può progettare una **funzione fissa** con i desiderata. Vogliamo qualcosa che generalizzi anche sequenze più lunghe tale che la distanza relativa tra i passaggi temporali (distanza nella sequenza) sia preservata. Quello che si può fare è calcolare il seno e coseno dei diversi periodi ω_k dove:

$$\omega_k = \frac{1}{10000^{2k/d}}$$

Si noti che se $k=1$ questa funzione è sempre molto vicina all'uno (d è generalmente uguale a 512 oppure 256) e questo significa che si ha un minimo nel periodo di questa funzione (ogni 2π di ripete). Se si incrementa k , diciamo $k=256$, abbiamo che il periodo della funzione è $10k/2\pi$, cioè la funzione si ripete ogni $\sim 10,600$ passaggi temporali. Quindi, in generale, con una k piccola la funzione si ripete molto velocemente (e questo avviene nella prima parte del vettore $p(t)$). Ricorda che stiamo parlando di seno e coseno quindi stiamo affermando che la funzione fluttua velocemente nell'intervallo $[-1, 1]$. Se incrementiamo k , la funzione si ripete molto lentamente (e questo avviene nell'ultima parte del vettore $p(t)$).

$$p(t) = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \vdots \\ \sin(\omega_{d/2} \cdot t) \\ \cos(\omega_{d/2} \cdot t) \end{bmatrix}_d$$

Come fa questa combinazione di seni e coseni a rappresentare una posizione/ordine?

Supponiamo di voler rappresentare un numero in formato binario. È possibile individuare la velocità di variazione tra bit diversi. Il bit meno significativo si alterna su ogni numero, il secondo bit più basso ruota ogni due numeri e così via. Usare valori binari sarebbe uno spreco di spazio nel mondo dei float. Quindi, possiamo usare le funzioni sinusoidali che si ripetono (equivalgono a bit alternati) e le possiamo utilizzare per sequenza di lunghezza arbitraria perché si ripetono (es. arrivati a 15, possiamo ricominciare da 0).

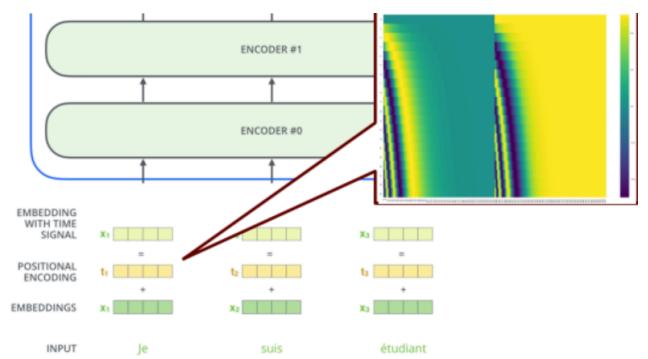
0 :	0 0 0 0	8 :	1 0 0 0
1 :	0 0 0 1	9 :	1 0 0 1
2 :	0 0 1 0	10 :	1 0 1 0
3 :	0 0 1 1	11 :	1 0 1 1
4 :	0 1 0 0	12 :	1 1 0 0
5 :	0 1 0 1	13 :	1 1 0 1
6 :	0 1 1 0	14 :	1 1 1 0
7 :	0 1 1 1	15 :	1 1 1 1

Possiamo visualizzare questo modello in un altro modo.

Immaginiamo di avere una sequenza di parole dove ogni parola nella sequenza ci restituisce un certo embedding x_i . Per ogni embedding x_i concateniamo un positional encoding t_i e la loro somma ci restituisce un embedding con il segnale temporale (*with time signal*) x_i , che diamo quindi in pasto all'encoder. Si noti come all'inizio ($k=1$) la funzione si alterna velocemente mentre dopo ($k=256$) la funzione si alterna lentamente.

Quindi → Stiamo inserendo un timestamp t nella sequenza in input x per fare in modo di preservare l'ordine delle parole.

EX. Nella frase “Marco gioca a palla”, otteniamo l'embedding di ogni parola e codifichiamo ogni parola per ottenere il positional



encoding (Marco = 0001, gioca = 0010, ecc..). Sommando embedding e positional encoding sappiamo qual è l'ordine delle parole nella sequenza (quindi, ad esempio, sapremo che Marco nella prima posizione).

Quindi, vediamo adesso l'intera architettura di un **transformer** definita nel 2017 nel paper "Attention Is All You Need". In questa architettura abbiamo un **encoder** (sulla sinistra) e un **decoder** (sulla destra).

Nell'**encoder** quello che facciamo è **concatenare (add)** un **positional encoding** con il vettore di input.

Dopodiché abbiamo la parte di self-attention dove, ipotizzando che ci concentriamo sulla seconda parola x_2 , compariamo (cmp) la parola x_2 ="represent" con tutte le altre parole nel vettore di input. Effettuiamo quindi un softmax a partire dalle comparazioni. Usiamo poi i valori del softmax per calcolare la somma pesata con il vettore di input. Si usa una FFNN (feed-forward network) che non è altro che una MLP con la quale otteniamo l'output e'_2 .

Quest'ultima operazione prende il nome di **position-wise feed-forward**. Questa operazione la dobbiamo ripetere per ogni layer quindi avremo una sequenza di self-attention e feed-forward fino al termine dell'elaborazione della sequenza di input al termine del quale otteniamo un **output** che non è altro che una **rappresentazione dell'input** (es. nella machine translation, un possibile output potrebbe essere una riassunto dell'input che usiamo come memoria per effettuare poi la traduzione).

Una volta ottenuto l'output dell'encoder, lo possiamo usare nel **decoder**. Nel caso nel machine translation, partiamo con uno "start" token e da qui iniziamo

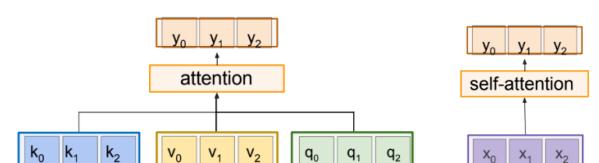
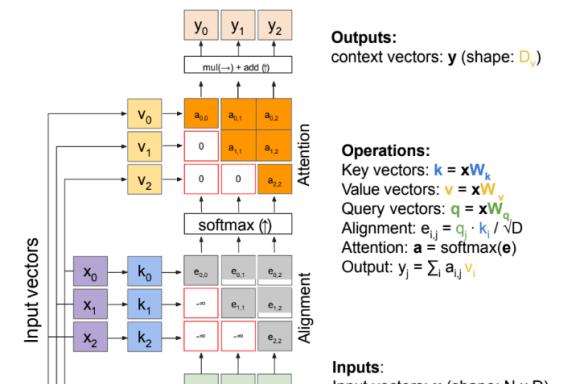
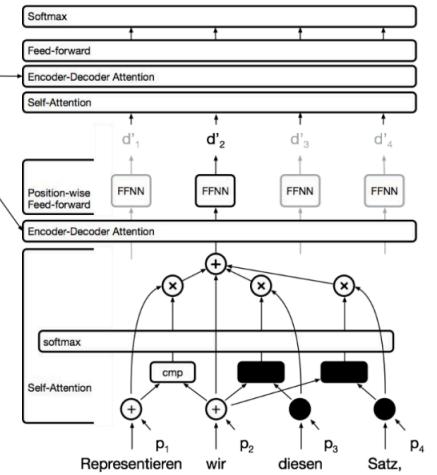
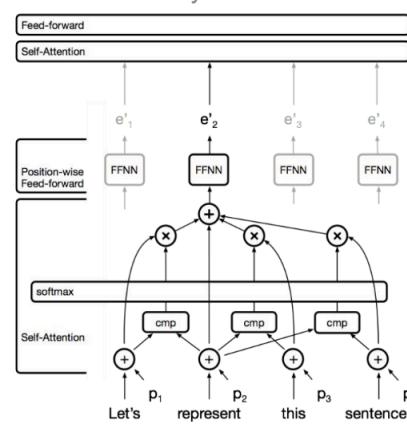
a produrre una parola dopo l'altra in maniera **autoregressiva**. Nota che nella parte di decoder teniamo conto nel positional encoding. La parte di **self-attention** viene utilizzata per garantire una certa consistenza con quello che stiamo cercando di predire. Per quanto riguarda l'encoder e decoder, il loro ruolo non cambia molto. La differenza sostanziale è che le queries (le parole che stiamo traducendo) arrivano dal decoder. Usiamo quindi la memoria (keys e values) proveniente dall'encoder per produrre l'output del decoder. Questo lo dobbiamo iterare fino alla fine della sequenza (diciamo finché non abbiamo uno "stop" token) alternando self-attention, encoder-decoder attention e feed-forward per una serie di layer. Alla fine produciamo una parola, e partendo da questa produciamo tutte le altre in maniera autoregressiva.

Nota: Si noti che nell'immagine della parte del decoder, stiamo cercando di tradurre la seconda parola. Tutte le parole dopo quella non vengono considerate (non le consideriamo nella comparazione). **Se impostiamo il risultato della comparazione a -inf, otteniamo 0 nella softmax.** Questo permette di mantenere una certa consistenza rispetto a cosa è stato prodotto fino a quel momento. Quindi, applicando la **masked self-attention layer**, si impedisce ai **vettori di guardare i vettori futuri** e questo lo si può fare **impostando manualmente gli alignment scores su -inf**. Effettivamente, il decoding avviene parola per parola e vogliamo avere una consistenza sulla base di quello che abbiamo già prodotto.

SUM. Facciamo un recap sulla **differenza tra general attention e self-attention**. Nella general attention, i dati di input sono trattati come due insiemi separati. Questo significa che il modello calcola l'attenzione tra gli elementi di due insiemi diversi, ad esempio, tra parole in una frase e concetti in una domanda. Quindi compariamo keys k, values v e queries q in arrivo da diverse sequenze. Nella

The Transformer

"Attention is All you Need"



self-attention (o auto-attention), i dati di input sono trattati come un unico insieme. Il modello calcola l'attenzione tra gli elementi all'interno dello stesso insieme, ad esempio, tra le parole in una frase. Quindi compariamo elementi di una sequenza con elementi della sequenza e una somma pesata degli elementi della sequenza (di conseguenza, keys k e values v vengono dalla stessa sequenza di input (memoria o input di un encoder) mentre le query q vengono dalla stessa sequenza se usiamo un encoder, mentre vengono da un decoding vogliamo query dinamiche generate dall'output del decoder che tiene conto di ciò che è stato prodotto fino a quel momento).

Prima ci siamo concentrati su come funziona l'encoder self-attention. Vediamo adesso nel dettaglio come funziona il **decoder self-attention**. Ipotizzando di concentrarsi sull'elemento d_2 , il decoder self-attention effettua il prodotto scalare (dot product) tra d_2 e tutti gli elementi già prodotti (nota che mascheriamo gli elementi futuri). Siccome è una self-attention, query e keys arrivano dalla stessa sequenza di input. Otteniamo quindi la softmax, facciamo la somma pesata prendendo softmax e values e otteniamo l'output.

Nota: Nei transformers non abbiamo uno stato interno (una struttura tipo le RNN) quindi effettuiamo una comparazione all-to-all che diamo in pasto ad un MLP nella fase di feed-forward. Questo significa che questa sequenza è parallelizzabile perché possiamo prendere l'intera sequenza, passarla alla transformer network e poi effettuiamo self-attention e MLP in parallelo (ricorda che abbiamo un loop di self-attention/MLP per l'elaborazione dell'intera sequenza).

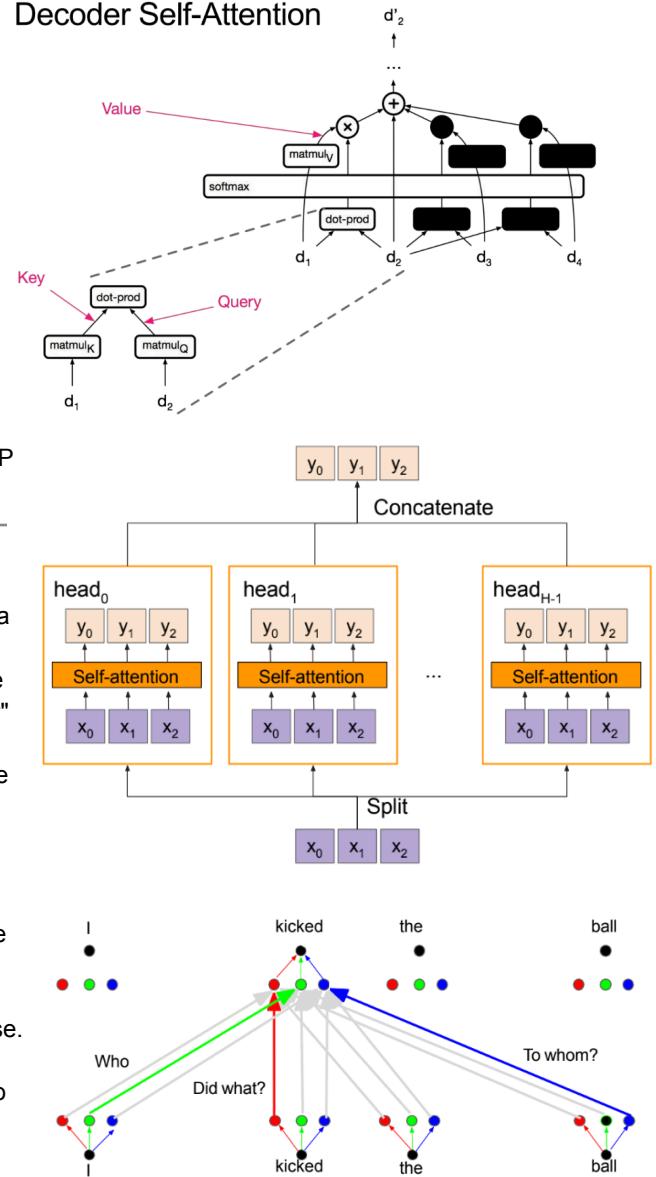
Ora, possiamo estendere la nozione di self-attenzione. Di nuovo, la self-attention è una tecnica che consente al modello di calcolare l'attenzione tra tutte le coppie di parole o unità all'interno della stessa sequenza di dati di input. Questo significa che il modello può considerare le relazioni tra tutte le parti della sequenza per catturare informazioni complesse. L'idea del "**multi-head self attention layer**" è quella di **eseguire la self-attention non una sola volta, ma multiple volte in parallelo**. Ogni "head" di self-attention è un canale indipendente che apprende relazioni diverse tra le unità nella sequenza. Ogni "head" calcola quindi attention scores diversi per ciascuna coppia di unità nella sequenza, consentendo al modello di catturare relazioni di diversi tipi e livelli di astrazione. Gli output di tutte le "head" sono poi concatenate per produrre l'output finale. Tale aggregazione aiuta il modello a cogliere informazioni più complete.

EX. Potremmo voler ottenere informazioni più complesse in una frase. L'assunzione del multi-head self-attention è che abbiamo una self-attention specifica per ogni ruolo e relazione che si ha all'interno della frase. Quindi quando ci focalizziamo ad esempio su "kicked", abbiamo varie self-attention che si focalizzano su chi ha compiuto l'azione ("I"), l'azione compiuta ("kicked") e l'oggetto dell'azione ("ball"). La probabilità di queste parole (diciamo l'attenzione che gli diamo) sarà maggiore e viene assegnata in parallelo.

Ho bisogno di una struttura del genere nelle CNN?

No, perché si ha una disposizione rigida degli elementi del kernel quindi nell'esempio precedente apprenderemmo I=Who, Kicked=Did What? e the=To whom?. In generale, le CNN sono particolarmente adatte per catturare pattern spaziali in dati come immagini, dove la posizione relativa dei pixel o dei pixel nelle vicinanze è importante. I modelli Transformer sono progettati per catturare relazioni a lungo raggio in sequenze di dati, come il testo, dove le parole o

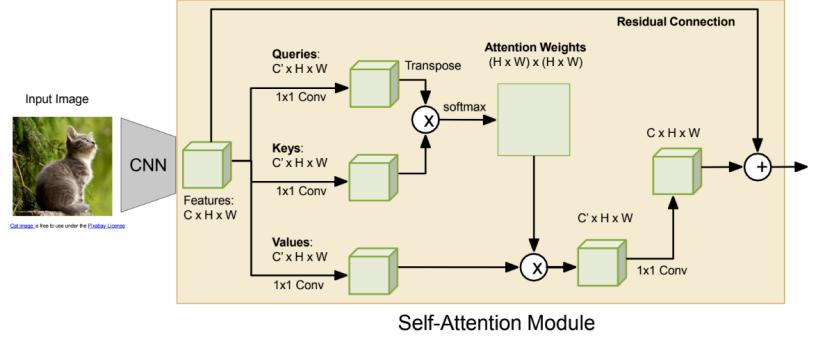
Decoder Self-Attention



le unità possono essere correlate anche se sono distanti tra loro nella sequenza (effettuiamo una comparazione all-to-all delle parole nella sequenza) ed è per questo che questa struttura ha più senso nei transformers.

EX. CNN with self-attention:

Consideriamo che dobbiamo implementare il task di image captioning. Usiamo una CNN per elaborare le features dell'immagine (quindi dall'immagine otteniamo C activation maps $C \times H \times W$ dove C è il numero di canali). Dalle activation maps estriamo queries, keys e values. Applichiamo la self-attention comparando queries e keys. Vi facciamo la softmax e otteniamo gli attention weights (Nota che la comparazione è del tipo $N \times N$ nel senso che essendo una self-attention comparemo gli input vectors con se stessi quindi $(H \times W) \times (H \times W)$). Facciamo quindi la somma pesata per ottenere l'output. Nota che usiamo una convoluzione con un kernel 1×1 per ottenere la dimensionalità che vogliamo. Una volta ottenuto l'output lo possiamo sommare alla feature map iniziale e il risultato lo si può considerare come un'altra activation map e prende il nome di **residual connection**. L'intera operazione (dall'input al calcolo della residual connection) prende il nome di **self-attention module**.



Vediamo comunque funziona nel dettaglio.

Immaginiamo di avere un'immagine I in input. L'output che si vuole ottenere è una sequenza $y = y_1, y_2, \dots, y_T$ che descriva l'immagine e questa descrizione la possiamo ottenere grazie ad una CNN che ci dà in output le activation maps ($H \times W \times D$). Possiamo prendere i pixel dell'activation maps, si possono comparare l'un l'altro, si può ottenere una somma pesata di questi pixel per ottenere un contesto c (l'embedding). Query, keys e values sono tutti ottenuti dallo stesso embedding. Il contesto c in output tiene conto della varie parti dell'immagine e anche come queste sono relazionate l'una all'altra. Una volta ottenuto il contesto, lo usiamo nel decoder che è anch'esso un transformer e si basa sulla self-attention, mask, encoder-decoder attention e MLP. Dobbiamo iterare questa operazione per ogni parola nella sequenza (quindi dallo start token, prediciamo la prima parola e così via fino ad incontrare uno stop token). Per il training possiamo parallelizzare il processo con la multi-head self attention layer.

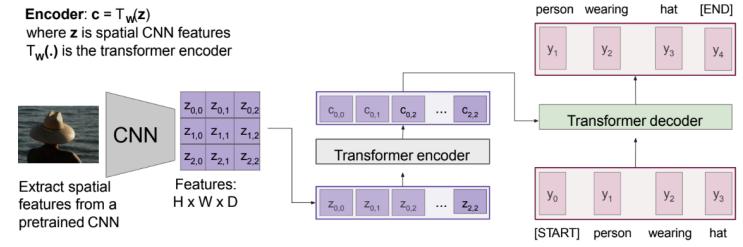
Quindi, la parte di encoding prevede di prendere una frase completa, passarla all'encoder e produrre la memoria (il contesto) nonché K e V . Possiamo poi combinare Q dell'output con K e V per produrre l'output in maniera autoregressiva.

Andiamo ancora di più nel dettaglio nell'**encoder block**. Abbiamo l'input embedding e da questo vogliamo ottenere il contesto. Possiamo notare come il transformer encoder è formato da **N encoder blocks** collegati in serie per ottenere la codifica dell'input (In Vaswani et al. $N = 6$, $D_q = 512$). Vediamo cosa accade all'interno di un singolo blocco:

1. Dato un certo input x , l'input dell'encoder block è una sequenza di dati, ad esempio una sequenza di parole in un testo. Ogni elemento della sequenza è rappresentato come

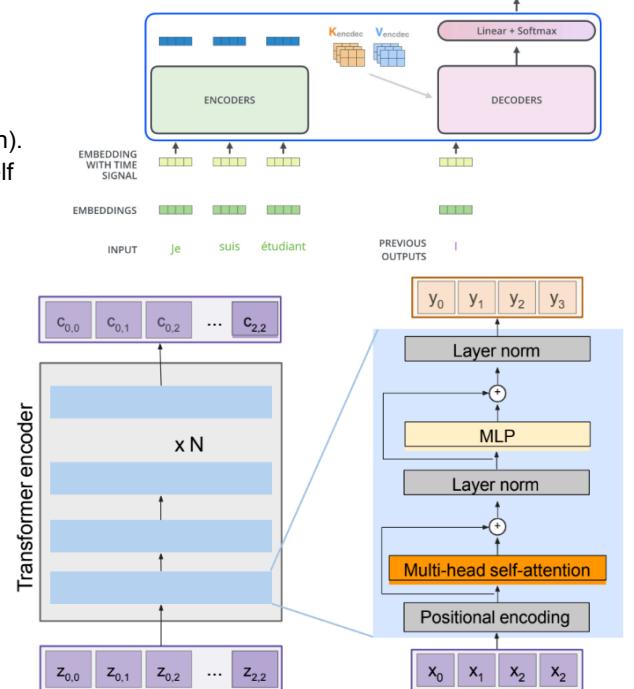
Input: Image I
Output: Sequence $y = y_1, y_2, \dots, y_T$

Decoder: $y_t = T_D(y_{0:t-1}, c)$
 where $T_D(\cdot)$ is the transformer decoder



Decoding time step: 1 2 3 4 5 6

OUTPUT



un vettore (ad esempio, l'embedding della parola) che cattura le informazioni sulla parola stessa.

Applichiamo quindi il **positional encoding** per mantenere l'ordine delle parole nella sequenza.

2. Aggiungiamo poi uno strato di **multi-head self-attention** (su tutti i vettori). Questo strato calcola l'attenzione tra tutti gli elementi della sequenza, consentendo al modello di catturare relazioni a lungo raggio tra le parole. L'output di questo strato è una versione pesata dell'input, che tiene conto delle relazioni tra le parole. L'output della "multi-head self-attention layer" viene sommato all'input originale attraverso una "**residual connection**". Questo significa che le informazioni originali non vengono completamente sovrascritte, ma piuttosto integrate con le nuove informazioni ottenute dall'attenzione.
3. A questa applichiamo la **layer normalization** ad ogni vettore in maniera individuale per normalizzare i dati e garantire stabilità;
4. Dopo l'aggiornamento della residual connection, l'output passa attraverso uno o più **strati feedforward**. Assumiamo di usare una MLP anche questa applicata ad ogni vettore in maniera individuale. L'output dei strati feedforward viene nuovamente sommato all'input originale tramite una "residual connection".
5. Applichiamo di nuovo layer normalization e otteniamo l'output.

Questi passaggi si ripetono per ciascun elemento della sequenza, in modo che il blocco dell'encoder possa catturare le caratteristiche rilevanti in tutta la sequenza. L'output dell'encoder block è quindi utilizzato come input per il prossimo encoder block o per altri compiti nel modello, come il decodificatore nei modelli di traduzione automatica.

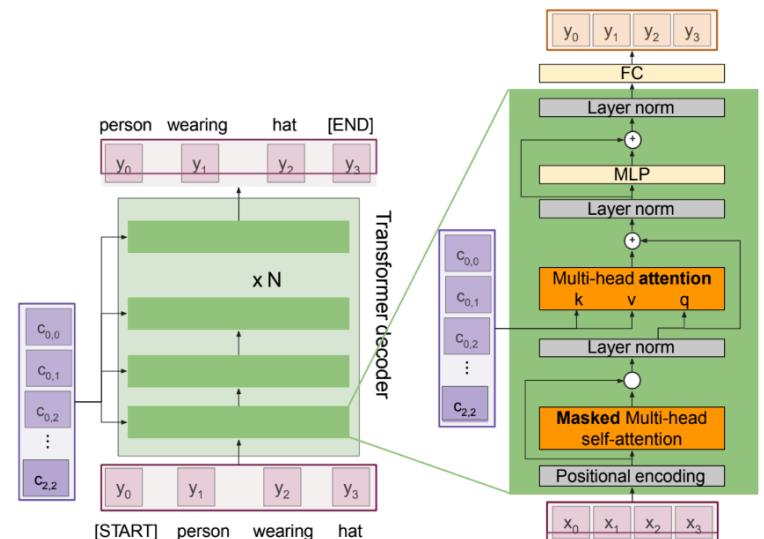
Quindi, per riassumere:

- In input abbiamo un insieme di vettori x , in output un insieme di vettori y ;
- La self-attention è l'unica interazione che abbiamo tra i vettori;
- La layer normalization e MLP operano indipendentemente su ogni vettore;
- Questo approccio è scalabile, parallelizzabile ma consuma molta memoria.

Passiamo ora al **decoder block**. Anche qui abbiamo **N**

decoder blocks che sono collegati in serie per generare l'output finale. Vediamo cosa succede all'interno di un singolo blocco:

1. Il decoder riceve due tipi di input:
 - Encoding delle informazioni di Input: Queste sono le rappresentazioni delle parole o delle unità dell'input originale, spesso prodotte dal blocco di encoding del Transformer.
 - Output parziale generato finora: Questo è l'output parziale generato fino a quel momento. È una sequenza di parole o unità già generate come parte dell'output.
2. Applichiamo la **masked multi-head self-attention layer** che calcola l'attenzione tra le parole o le unità nell'output parziale generato finora. La **maschera (mask)** è applicata per garantire che il modello possa vedere solo le parole generate fino a quel momento, evitando così di guardare avanti nel testo. L'attenzione tra le parole nell'output parziale aiuta il modello a capire come le parti già generate influenzano la generazione dell'output successivo.
3. L'output della "masked multi-head self-attention layer" è sommato all'input originale attraverso una "**residual connection**". Applichiamo **layer normalization**.
4. Il decoder calcola un altro strato di "**multi-head self-attention**" ma questa volta considera anche le informazioni di input. Questo consente al modello di catturare relazioni tra le parole nell'output parziale e quelle nell'input. Nota che keys k e values v arrivano dal contesto calcolato nell'encoder mentre q arriva dal decoder. Nota che per l'image captioning, questo è il passo in inseriamo le features dell'immagine nel decoder;



5. L'output del passaggio di "multi-head self-attention rispetto alle informazioni di input" è nuovamente sommato all'input originale tramite una "residual connection". Applichiamo di nuovo Layer Normalization;
6. L'output delle operazioni di attenzione viene passato attraverso uno o più **strati feedforward**, simili a quelli nel blocco del codificatore (quindi MLP). Questi strati aiutano a catturare relazioni complesse tra le parole o le unità nell'output parziale.
7. L'output dei strati feedforward è nuovamente sommato all'input originale attraverso una "residual connection". Applichiamo Layer Normalization.
8. L'output del decoder passa attraverso uno **strato lineare finale** seguito da una **funzione di softmax** per ottenere una distribuzione di probabilità sul vocabolario delle possibili parole o unità di output. Una parola o un'unità vengono campionati da questa distribuzione per diventare la parte successiva dell'output parziale.
9. La parola o l'unità campionata viene aggiunta all'output parziale, in modo da poter essere considerata nelle iterazioni successive.
10. I passaggi da 2 a 9 vengono ripetuti fino a quando il modello raggiunge una condizione di arresto (come la generazione di un simbolo di fine sequenza - uno stop token).

Questo processo iterativo consente al decoder di costruire l'output finale un'unità alla volta, utilizzando l'attenzione rispetto alle informazioni di input e all'output parziale generato finora per catturare relazioni complesse e produrre output coerenti e ben formati.

Quindi, per riassumere:

- In input abbiamo un insieme di vettori x e una serie di context vectors c , in output un insieme di vettori y ;
- La masked self-attention interagisce con gli input passati;
- La multi-head attention block non è la self-attention. Interviene sugli output dell'encoder;
- Questo approccio è scalabile, parallelizzabile ma consuma molta memoria.

Alcune ottimizzazioni:

- Ottimizzatore ADAM con una learning rate warm up (warm up + exponential decay);
- Dropout durante il training ad ogni layer appena prima di aggiungere il residual component;
- Layer normalization;
- Attention dropout;
- Checkpoint-averaging;
- Label smoothing;
- Auto-regressive decoding con beam search e length biasing.

Nota: Comparazione tra RNN e Transformers:

- RNN:
 - (+) Gli LSTM funzionano abbastanza bene per sequenze lunghe;
 - (-) Si aspetta una sequenza ordinata di input;
 - (-) Calcolo sequenziale: i successivi stati nascosti possono essere calcolati solo dopo che quelli precedenti sono stati completati (non vi è parallelizzazione).
- Transformers:
 - (+) Bravo nelle sequenze lunghe. Ogni calcolo dell'attenzione esamina tutti gli input;
 - (+) Può operare su insiemi non ordinati o sequenze ordinate con positional encodings;
 - (+) Calcolo parallelo: Tutti gli alignments e attention scores per tutti gli input possono essere eseguiti in parallelo;
 - (-) Richiede molta memoria: gli $N \times M$ alignment e attention scalers devono essere calcolati ed archiviati per una single self-attention head (ma le GPU stanno diventando sempre più grandi e migliori).

SUM. Aggiungere l'**attenzione** agli RNN consente loro di "prestare attenzione" a diverse parti dell'input in ogni fase temporale. Lo **general attention layer** è un nuovo tipo di strato che può essere utilizzato per progettare nuove architetture di reti neurali. I **transformers** sono un tipo di livello che utilizza la **self-attention** e la layer normalization.

- È altamente scalabile e altamente parallelizzabile;
- Training più rapido, modelli più grandi, prestazioni migliori nelle attività visive e linguistiche;

- Stanno rapidamente sostituendo gli RNN, gli LSTM e potrebbero (?) persino sostituire le convoluzioni.

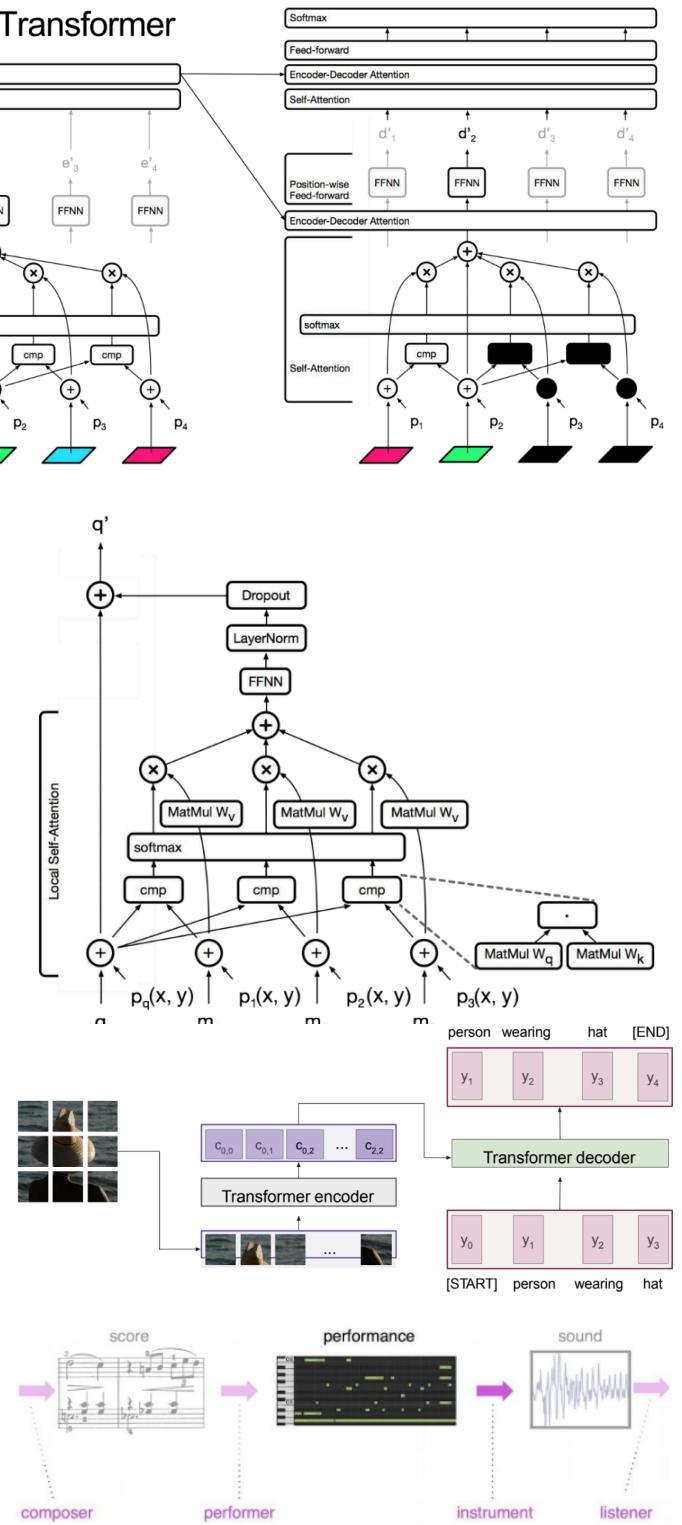
La "self-similarity" nelle immagini si riferisce alla presenza di strutture o pattern all'interno di un'immagine che sono simili o ripetuti a diverse scale o posizioni all'interno della stessa immagine. Questi pattern possono essere identificati tramite l'analisi delle correlazioni tra regioni diverse dell'immagine o attraverso l'uso di metodi di elaborazione delle immagini. Possiamo sfruttare i cosiddetti "image transformers" per identificare la self-similarity nelle immagini. L'identificazione avviene in questo modo: L'immagine di input può essere suddivisa in patch o regioni, simili all'uso dei patch nei Vision Transformers. Ogni patch è rappresentato come un vettore. Viene aggiunta un'informazione sulla posizione a ciascun patch, in modo che il modello sia consapevole delle posizioni relative dei patch nell'immagine. Quindi mentre normalmente il positional encoding prende in input un numero naturale (la posizione in una sequenza), qui dobbiamo considerare le coordinate (x, y) che ci consentono di identificare una posizione nell'immagine. L'immagine, insieme alle informazioni sulla posizione, viene alimentata in una rete basata

sull'architettura Transformer. Questa rete può includere multi-head self-attention e strati feedforward. Durante il processo di attenzione, il modello può rilevare relazioni a lungo raggio tra i patch, identificando regioni simili o ripetitive nell'immagine. L'output del modello può essere utilizzato per vari compiti, come la segmentazione delle regioni simili o l'identificazione di pattern.

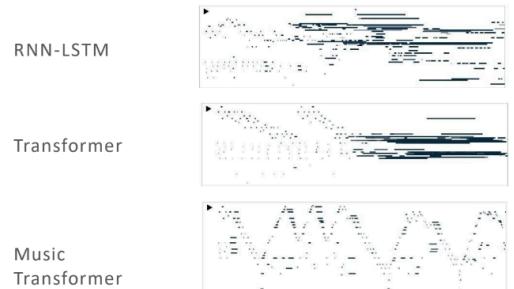
Nota: Nell'image captioning ottenevamo le activation maps applicando una CNN sulle immagini e poi davamo in pasto le activation maps all'encoder. Quello che possiamo fare però è **dare subito l'immagine ad un transformer**. Questo tipo di transformers prende il nome di **Vision Transformers**:

- L'immagine di input viene suddivisa in patches, e ciascun patch viene considerato come un vettore o una sequenza di dati. Questi patch costituiscono l'input alla rete. Ogni patch viene posizionato in una sequenza bidimensionale, in modo che la posizione relativa dei patch nell'immagine sia conservata. Questo preserva le informazioni sulla struttura spaziale dell'immagine;
- Poiché le reti neurali non hanno una comprensione intrinseca delle posizioni, viene aggiunta un'informazione sulla posizione a ciascun patch. Questo può essere fatto attraverso il "positional encoding", che fornisce alle patch informazioni sulla loro posizione nell'immagine;
- La sequenza di patch, insieme alle informazioni sulla posizione, viene passata attraverso una serie di "Transformer encoder blocks". L'output dell'encoder è una sequenza di vettori, ognuno dei quali rappresenta una determinata regione dell'immagine.

The Image Transformer



Introduciamo adesso la **relative self-attention** mostrando come può essere utilizzata nella **generazione della musica**. Nella musica utilizziamo un pentagramma per rappresentarla. Un performer può suonare la musica nel pentagramma utilizzando un certo strumento e questo genera un suono. Dato un certo punto nel pentagramma, sappiamo cosa è stato suonato fino a quel momento quindi vogliamo continuare a suonare da quel punto in poi. Si noti come questo sia molto simile ad un language model in cui osserviamo una frase fino ad una certa parola t , e vogliamo produrre la parola successiva $t+1$. Ora, dato un certo motivo, vogliamo predire la musica futura e per conseguire questo task possiamo utilizzare vari approcci come lo RNN-LSTM o un Transformer. Si può notare nell'immagine a destra come le RNN non si comportano molto bene in quanto a mano a mano che si analizza la sequenza, questa non è in grado di memorizzare e processare lunghe dipendenze e dopo un certo punto non ci permette più di generare nuova musica. I transformers si comportano un po' meglio ma dopo un po' ricadiamo nello stesso problema.



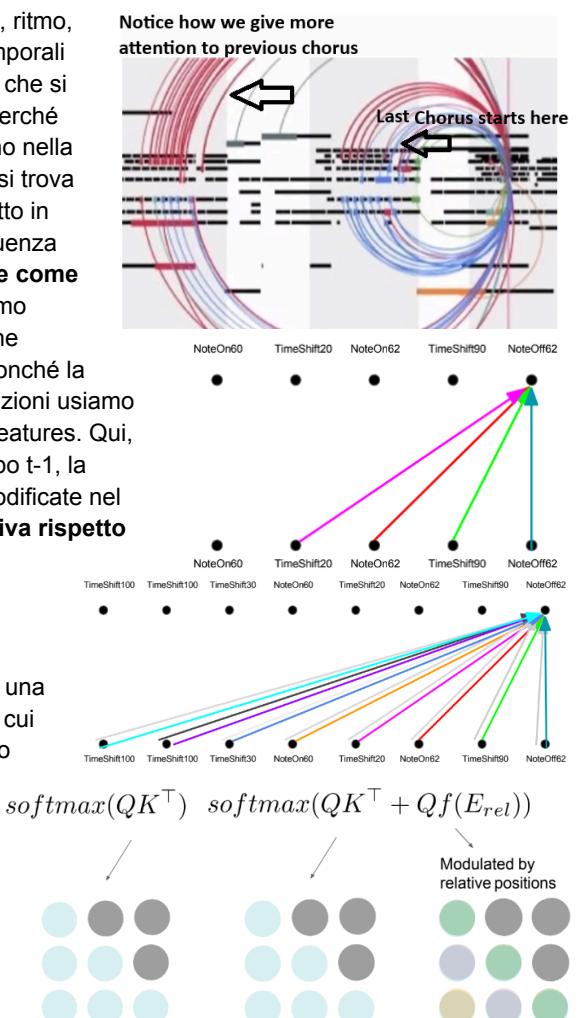
Questo significa che tali modelli, utilizzati così come sono, **non sono in grado di catturare le relazioni musicali nei dati nel caso di lunghe sequenze**. Si possono quindi utilizzare i music transformers che sono dei modelli basati sui transformer standard ma che si comportano bene con le applicazioni di generazione musicale. La **self-similarity** (autosimilarità) nella musica è un concetto che fa riferimento alla presenza di **motivi o pattern musicali** che si ripetono o si riflettono all'interno di una composizione musicale.

Questi motivi o pattern possono essere simili o identici in termini di melodia, ritmo, armonia o struttura, e questa similarità può manifestarsi a diverse scale temporali all'interno di una composizione. Possiamo ad esempio identificare il chorus che si ripete più volte in una sequenza musicale. Il self-similarity ci è quindi utile perché guardando al passato, possiamo cogliere parti della sequenza che ci aiutano nella generazione di musica (note successive). Ad esempio, nel momento in cui si trova nel chorus, quello che si può fare è generare nuova musica così com'è. Detto in altro modo, nella generazione di musica diamo attenzione a parti della sequenza nel passato per predire musica nel futuro. Finora abbiamo visto l'**attenzione come una semplice somma pesata** in cui comparavamo, la nota corrente (diciamo quella che stiamo cercando di predire) con tutte le note passate. Ricorda che utilizziamo positional encoding quindi sappiamo qual è l'ordine delle note nonché la distanza tra di queste. Ora, ricorda che abbiamo già visto che nelle convoluzioni usiamo dei kernel che guardano a posizioni fisse dell'input per ottenere le relative features. Qui, ad esempio, se ci troviamo nel tempo t troviamo la relazione verde nel tempo $t-1$, la relazione rossa in $t-2$ e così via (quindi le relazioni sono in qualche modo codificate nel kernel stesso). Diciamo quindi le **convoluzioni usano un'attenzione relativa rispetto alla posizione**.

Possiamo mettere insieme l'attenzione relativa rispetto alla posizione della convolution e la multihead attention?

La risposta è sì. Possiamo pensare a questo meccanismo come l'utilizzo di una sorta di spostamento relativo (relative shift) del positional encoding tale per cui possiamo identificare pattern di attenzione nel passato. Quindi, nell'esempio precedente, potremmo voler generare la prossima nota tenendo conto delle relazioni immediatamente precedenti ($t-1$, $t-2$, ecc..) ma anche spostarsi nel tempo in maniera tale da poter identificare eventuali pattern nel passato.

Quindi, ripartiamo dalla formulazione di attenzione che non è altro che la softmax della comparazione tra queries e keys. Quindi, ipotizzando che ci troviamo al tempo t , guardiamo indietro (non al futuro, ricorda che l'attenzione è causale) per predire le note successive. Modifichiamo la precedente formula in modo da aggiungere la possibilità di effettuare uno **shift**. Quindi alla precedente attenzione sommiamo uno shift cioè una funzione che prende in input delle **matrici di posizionamento relativo** e questa viene



combinata alle queries. Quindi è come se la rete adesso è in grado di imparare un offset con il quale possiamo dire al modello che nel momento in cui predice un'unità successiva, possiamo porre l'attenzione alla sequenza passata che si trova a distanza "offset" dall'unità che stiamo cercando di predire (es. nell'esempio del chorus, se capiamo che questo si sta ripetendo, possiamo tornare indietro nella sequenza dando maggiore attenzione ai precedenti chorus).

Più nel dettaglio, la **Relative Self-Attention** (attenzione relativa) è una variante dell'operazione della self-attention utilizzata nei modelli Transformer e nelle loro varianti. L'attenzione relativa è stata progettata per affrontare alcune limitazioni della self-attention, in particolare quando si tratta di modelli con sequenze lunghe.

Nella self-attention tradizionale, le interazioni tra i token (ad esempio, parole in un testo o patch in un'immagine) sono calcolate in base a relazioni fisse e simmetriche tra di essi. Questo significa che ogni token "guarda" tutti gli altri token nello stesso modo, indipendentemente dalla posizione o dalla distanza tra di essi. Questo può portare a problemi quando si trattano sequenze molto lunghe, poiché richiede calcoli onerosi in termini di memoria e computazione. La relative self-attention, invece, introduce una componente di posizionamento relativo nel calcolo dell'attenzione. Questo consente al modello di considerare la relazione tra i token in base alla loro posizione relativa nella sequenza. In altre parole, l'attenzione non è più completamente simmetrica, ma tiene conto della distanza e della posizione relativa tra i token. Questo può essere utile per catturare relazioni a lungo raggio in sequenze.

L'implementazione pratica della relative attention comporta l'uso di **matrici di posizionamento relativo** per calcolare l'attenzione tra i token. Queste matrici consentono al modello di ponderare in modo diverso le relazioni tra i token a seconda della loro posizione relativa nella sequenza.

Per riassumere la self-attention:

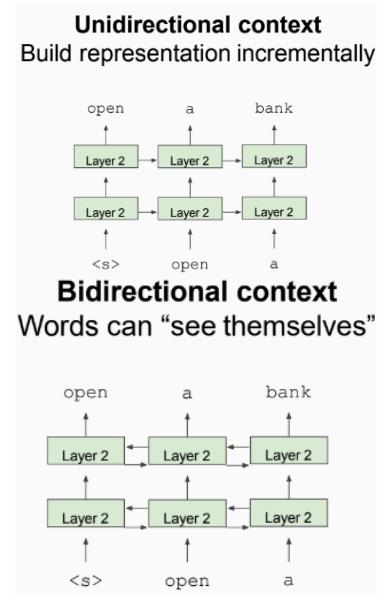
- La lunghezza del percorso è costante tra due posizioni qualsiasi;
- La memoria illimitata: Siccome possiamo confrontare gli elementi con ogni elemento del passato, significa che questi li dobbiamo salvare in memoria;
- Banale da parallelizzare (per layer);
- Può essere usata per modellare la self-similarity;
- La relative attention fornisce tempismo espressivo (viene introdotto lo shift nel tempo), **equivarianza** (la parte relativa della self-attention cambia in maniera proporzionale rispetto allo shift che applichiamo) e si estende naturalmente ai grafici.

BERT (acronimo di "Bidirectional Encoder Representations from Transformers") è un modello di linguaggio basato su transformer che ha rivoluzionato il campo del Natural Language Processing (NLP). È stato presentato da Google Research nel 2018 (circa due anni dopo i transformers) ed è noto per la sua capacità di comprendere e generare il linguaggio naturale in modo eccezionale. Una delle innovazioni chiave di BERT è la sua **capacità di comprensione bidirezionale del testo**. A differenza dei modelli precedenti che trattavano il testo in modo unidirezionale (da sinistra a destra o viceversa), BERT considera il contesto sia precedente che successivo a una determinata parola. Questa bidirezionalità consente a BERT di catturare relazioni e significati più complessi all'interno del testo.

I motivi per cui i language models precedenti a BERT erano unidirezionali sono:

- La direzionalità è necessaria per generare una distribuzione di probabilità ben formata (in altre parole, vogliamo capire qual è la probabilità di generare una parola sulla base delle precedenti e nel caso in cui si possa guardare anche alle parole future questo approccio non va bene).
- Le parole possono "vedersi" in un codificatore bidirezionale (se si può guardare anche nel futuro, non sappiamo più come allenare il modello perché questo apprenderà la parola futura memorizzandola e basta).

L'idea di BERT è quella di **mascherare il k% delle parole in input** e quindi **effettuare la predizione sulle parole mascherate**. Nota che tipicamente si usa k=15%. Un mascheramento troppo piccolo potrebbe richiedere un training computazionalmente troppo dispendioso (Mascherare solo poche parole



the man went to the [MASK] to buy a [MASK] of milk
store gallon

richiede che il modello si basi su un contesto più ampio per fare previsioni accurate sulle parole nascoste. Questo richiede più tempo di addestramento perché il modello deve essere esposto a un numero significativo di contesti diversi ed inoltre può portare ad overfitting). Allo stesso tempo, un mascheramento troppo grande porterebbe a non avere abbastanza contesto per effettuare delle previsioni accurate.

Contextual representation si riferisce alle rappresentazioni delle parole all'interno di una frase che tengono conto del contesto circostante. Immaginiamo di avere un encoder a cui diamo in pasto una serie di tokens (parole). Ne mascheriamo una (in modo che gli altri tokens non possano vederla). L'obiettivo è predire quel token mascherato sulla base del contesto. Quindi, per allenare BERT il modello è esposto a testi con alcune parole mascherate e deve prevedere quale sia la parola mascherata. Tuttavia, c'è un problema noto con il "**Masked LM**": le parole che vengono mascherate durante il pre-addestramento non vengono mai viste durante la fase di "fine-tuning" del modello per compiti specifici e questo fa sì che l'accuratezza della fase di test si abbassi. Per risolvere questo problema, BERT utilizza una strategia di mascheratura specifica: Invece di sostituire tutte le parole con "[MASK]" 100% delle volte, BERT sostituisce le parole in modi diversi:

- 80% delle volte, sostituisce una parola con "[MASK]". Ad esempio, "went to the store" potrebbe diventare "went to the [MASK]".
- 10% delle volte, sostituisce una parola con una parola casuale. Ad esempio, "went to the store" potrebbe diventare "went to the running".
- 10% delle volte, mantiene la stessa parola. Ad esempio, "went to the store" rimane "went to the store".

Questa strategia consente al modello di vedere parole reali durante il pre-addestramento, rendendolo più robusto quando deve fare previsioni su testi reali durante la fase di fine-tuning.

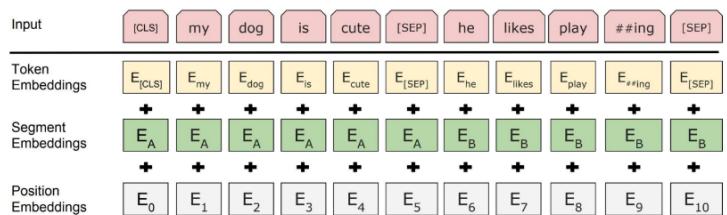
Un altro metodo utilizzato nella fase di pre-addestramento di BERT prevede di capire qual è la **relazione le tra frasi**. Praticamente, il modello riceve due frasi consecutive: "Frase A" e "Frase B". Deve decidere se "Frase B" è una continuazione logica e coerente di "Frase A" o se è una frase casuale estratta casualmente da un testo. Questo tipo di addestramento è noto come **next sentence prediction** (predizione della frase successiva).

Sentence A = The man went to the store.
Sentence B = He bought a gallon of milk.
Label = IsNextSentence

Sentence A = The man went to the store.
Sentence B = Penguins are flightless.
Label = NotNextSentence

Quindi, in BERT l'input viene rappresentato in questo modo:

- Il testo di input viene suddiviso in token, dove un token può essere una parola, una sottoparola di una parola o un simbolo speciale come il "[CLS]" token per l'inizio della sequenza e il "[SEP]" token (per la fine). Ogni token è identificato con un ID univoco nel vocabolario del modello.
- Ciascun token viene sostituito con il suo embedding vettoriale. BERT utilizza un embedding di token pre addestrato basato su grandi quantità di testo. Gli embeddings di token catturano le rappresentazioni semantiche delle parole e dei token.
- Per tener conto dell'ordine delle parole nella sequenza, vengono aggiunti embeddings di posizione a ciascun token. Questi embeddings catturano la posizione relativa di ciascun token nella sequenza.
- BERT è in grado di gestire coppie di frasi, quindi deve capire quali token appartengono a quale frase. Per farlo, assegna embeddings di segmento diversi a ciascun token in base alla frase di appartenenza. Ad esempio, il testo "[CLS] My dog is cute [SEP] He likes playing [SEP]" riceverà embeddings di segmento diversi per le due frasi.



BERT viene allenato in due fasi principali:

- Pre-addestramento: In questa fase, BERT è allenato su un corpus di testo molto ampio e generale, come l'intero dump di Wikipedia o tutto il testo di Twitter. Durante il pre-addestramento, il modello impara a catturare rappresentazioni linguistiche ricche e contestuali. Questo è reso possibile dalla sua capacità di prevedere parole mancanti in una sequenza di testo, utilizzando un'architettura di trasformatori

bidirezionale. BERT prevede parole mancanti sia a sinistra che a destra di una data parola, il che gli consente di catturare il contesto sia precedente che successivo.

- Fine-tuning: Dopo il pre-addestramento, BERT può essere utilizzato in compiti specifici di NLP. Ad esempio, può essere adattato per il riconoscimento delle entità, il sentiment analysis, il part-of-speech tagging e molti altri compiti. Durante la fase di fine-tuning, il modello viene adattato per il compito specifico utilizzando un set di dati etichettato. Ad esempio, per il sentiment analysis, verrà addestrato su un set di recensioni di film con etichette di sentimento (positivo/negativo/neutro). Il fine-tuning comporta l'aggiornamento dei pesi del modello in modo che sia in grado di svolgere il compito specifico in base ai dati di allenamento.

Nota: Quando andiamo ad allenare il modello, la loss function prende in considerazione solo le predizioni sui valori mascherati e ignora le predizioni sulle parole non mascherate. Diciamo che qui l'obiettivo non è quello di predire la parola successiva bensì la stessa parola mascherata. Una volta allenato il modello, possiamo utilizzarlo (attraverso il fine tuning) per altri scopi quali il machine translation o, in generale, la predizione di una parola successiva. BERT è quindi un chiaro esempio di transfer learning.

Alcuni dettagli sul modello:

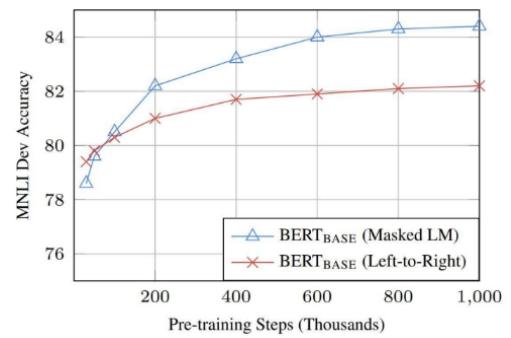
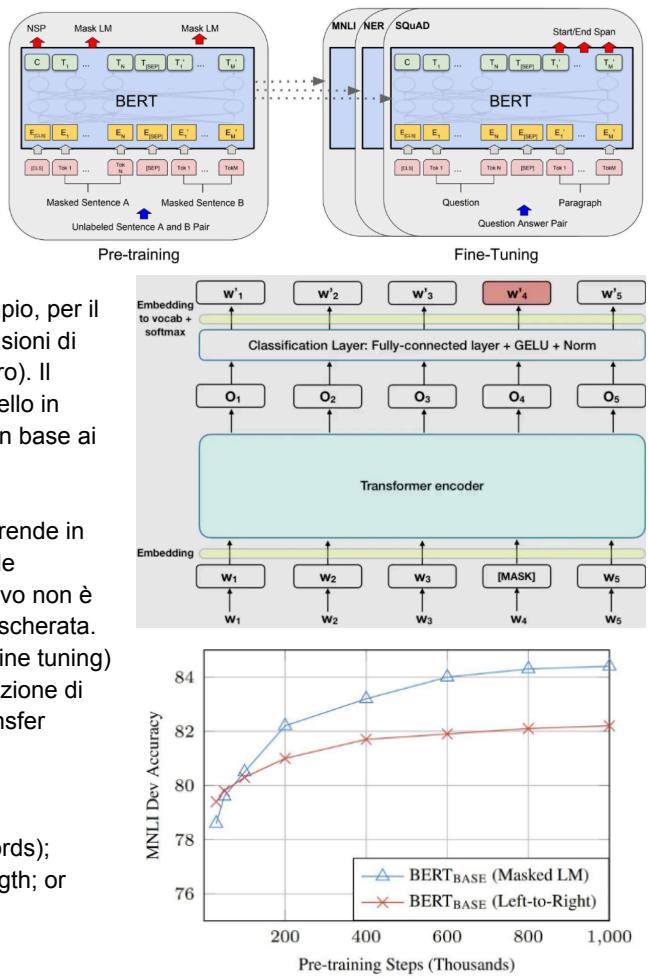
- Data: Wikipedia (2.5B words) + BookCorpus (800M words);
- Batch Size: 131,072 words (1024 sequences * 128 length; or 256 sequences * 512 length);
- Training Time: 1M steps (~40 epochs);
- Optimizer: AdamW, 1e-4 learning rate, linear decay;
- BERT-Base: 12-layer, 768-hidden, 12-head;
- BERT-Large: 24-layer, 1024-hidden, 16-head;
- Trained on 4x4 or 8x8 TPU slices for 4 days.

Nota: Il Masked LM impiega leggermente più tempo per convergere perché prevediamo solo il 15% anziché il 100%, ma i risultati assoluti sono molto migliori quasi immediatamente. Inoltre è stato visto che utilizzare modelli più grandi, aiuta a raggiungere performance più elevate. È stato fatto in test in cui passare da 110 milioni -> 340 milioni di parametri aiuta anche su set di dati con 3.600 esempi etichettati.

SQuAD 2.0 (Stanford Question Answering Dataset) è una versione successiva di SQuAD, un popolare set di dati utilizzato per valutare e testare modelli di domanda-risposta nell'ambito dell'elaborazione del linguaggio naturale (NLP). SQuAD 2.0 è stato introdotto per affrontare alcune limitazioni della versione originale. A differenza di SQuAD 1.1, in cui ogni contesto aveva almeno una risposta possibile, SQuAD 2.0 include domande per cui non esiste una risposta nel contesto fornito. Pertanto, è richiesto ai modelli di riconoscere quando una domanda non può essere risposta con le informazioni disponibili. L'obiettivo di SQuAD 2.0 è fornire un set di dati più realistico che riflette meglio la complessità del compito di rispondere alle domande utilizzando un contesto dato. Ciò favorisce lo sviluppo di modelli di NLP più sofisticati che possono gestire scenari in cui una risposta potrebbe non essere chiaramente definita o potrebbe mancare del tutto nel contesto.

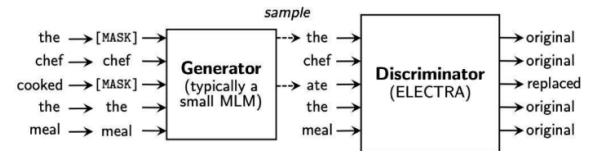
Esistono alcune variazioni di BERT:

- **RoBERTa** (Robustly optimized BERT approach) è una variante avanzata di BERT. In RoBERTa, la mascheratura dinamica delle parole durante il pre-addestramento è stata sostituita da una mascheratura



statica. Ciò significa che durante il pre-addestramento, le stesse parole vengono sempre mascherate, migliorando la consistenza dell'allenamento. RoBERTa utilizza batch size più grandi e learning rates più elevati rispetto a BERT, il che contribuisce a migliorare le prestazioni del modello. RoBERTa utilizza sequenze di testo più lunghe durante il pre-addestramento rispetto a BERT, consentendo al modello di catturare meglio il contesto più ampio.

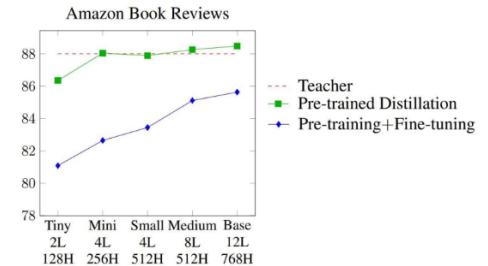
- **ELECTRA:** Un altro modello basato su BERT. La caratteristica distintiva di ELECTRA è la sua procedura di addestramento contrastivo. A differenza di BERT, che usa il task di mascheramento delle parole (prevedendo parole mascherate), ELECTRA sostituisce alcune parole con token "rumorosi" (parole generate casualmente) e addestra il modello per prevedere se ogni token è originale o rumoroso. ELECTRA segue un approccio di **framework generatore-discriminatore**. Un modello, noto come generatore, sostituisce alcune parole con token rumorosi, mentre un modello discriminatore cerca di distinguere tra token originali e rumorosi. L'obiettivo è che il generatore diventi così bravo da ingannare il discriminatore, creando rappresentazioni migliori.



Nota: BERT e altri modelli linguistici pre-addestrati sono estremamente grandi e costosi. In che modo le aziende li stanno applicando ai servizi di produzione a bassa latenza? La risposta è la **distillation** (o model compression). La distillation (distillazione della conoscenza), è un processo in cui un modello più grande e complesso ("teacher") viene utilizzato per allenare un modello più piccolo e più semplice ("student") trasferendogli le sue conoscenze.

- **Addestra il "Teacher":** Si inizia ad addestrare un modello più grande (Teacher) utilizzando tecniche avanzate di pre-addestramento (ad esempio, pre-addestramento su un grande corpus di dati con language models seguite da un fine-tuning su dati specifici di un certo compito. L'obiettivo è ottenere un modello con massima accuratezza sul compito di interesse).
- **Etichettatura di Esempi non Etichettati con il Teacher:** Il modello Teacher viene quindi utilizzato per etichettare un gran numero di esempi non etichettati. Ciò significa che il modello Teacher genera previsioni su questi esempi, creando etichette artificiali basate sulla sua conoscenza.
- **Allenamento dello "Student":** Si procede ad addestrare un modello più piccolo (Student), che è significativamente più compatto rispetto al modello Teacher (ad esempio, 50 volte più piccolo). Il modello Student è addestrato per imitare le previsioni del modello Teacher, utilizzando le etichette artificiali create precedentemente. La loss function del modello Student può essere la Mean Square Error o la Cross Entropy.

Alcuni test hanno mostrato che la tecnica di distillation ha performance migliori rispetto ad applicare pre-training e fine-tuning ad un modello più piccolo.

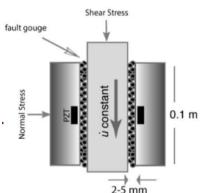


SUM. Per riassumere: I language models bidirezionali pre-addestrati funzionano incredibilmente bene (BERT) tuttavia questi sono tipicamente estremamente costosi. I miglioramenti (sfortunatamente) sembrano provenire principalmente da modelli ancora più costosi e da più dati. Il problema dell'inferenza/servizio viene per lo più "risolto" attraverso la distillation.

5.4 Graph Neural Networks

Prima di introdurre i graph neural networks, vediamo alcuni casi studio.

Abbiamo già visto che **"Auto-regressive forecasting"** è una tecnica di previsione che coinvolge la previsione di valori futuri basandosi su una sequenza di valori passati. Vediamo come possiamo applicarla alla **previsione dei terremoti**. Tipicamente negli esperimenti controllati (nei laboratori) che provano a simulare un terremoto si usa un apparato a doppio taglio (double-shear apparatus) che registra lo stress di taglio (shear stress) e le emissioni acustiche. Questo apparato prevede di spingere un cilindro che provoca un certo attrito provocando un certo stress che viene prima o poi rilasciato. Questo permette di simulare accumulo di stress lungo una faglia che porta poi al movimento della placca tettonica.

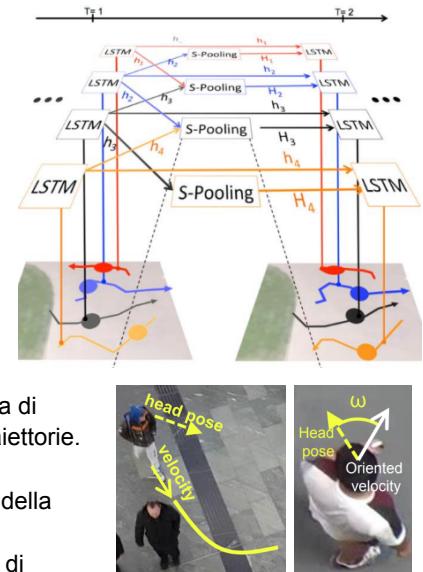
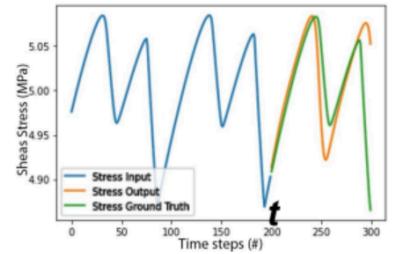
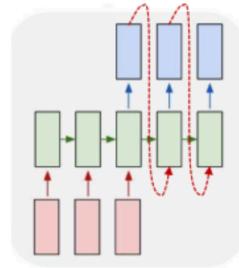


In un contesto autoregressivo, quello che si può fare nel caso di un terremoto è utilizzare le osservazioni passate per prevedere i valori futuri di una variabile. Ad esempio, se si vuole prevedere lo shear stress al tempo $t+1$, si utilizzano i valori di shear stress ai tempi precedenti t , $t-1$, ..., $t-p$ (dove p è la lunghezza della finestra temporale passata utilizzata per la previsione). Quindi, in base alle osservazioni dello stress precedenti, quello che si fa è cercare di capire quando avverrà il terremoto più forte. Questa tecnica prevede alcuni vantaggi:

- **Self-Supervised:** Non occorre dare etichette ma è il modello che prende da solo in input i dati (la sequenza di shear stress) e predice l'elemento successivo.
- **Side-Steps Labeling Ambiguity and Supervision Collapse:** Questa tecnica permette di evitare ambiguità nell'etichettatura e problemi di "supervision collapse". La supervision collapse può verificarsi quando l'uso di etichette esterne porta il modello a memorizzare le etichette anziché apprendere i pattern nei dati (praticamente il rischio è quello di non imparare nulla). L'auto-regressione può mitigare questi problemi concentrandosi sulla previsione basata su dati temporali senza etichette esterne.
- **Predicts Beyond the Next Failure:** Un altro vantaggio è la capacità di prevedere oltre il prossimo evento di interesse, come il superamento di una soglia critica di shear stress. Ciò significa che il modello può essere in grado di anticipare situazioni di criticità oltre il prossimo passo temporale.

In generale, è stato visto che con eventi aperiodici lenti e veloci, LSTM si comporta abbastanza bene anche con un piccolo quantitativo di dati. I transformers funzionano meglio ma solo con un grande quantitativo di dati per il training ottenibili grazie ad un pre-addestramento con dati sintetici (praticamente i TF richiedono un certo quantitativo di dati per il training per cui si possono usare dati generati sinteticamente che, nei test effettuati, funzionano bene su test reali).

Un altro caso studio è la **previsione della traiettoria delle persone** (People Trajectory Forecasting), quindi data

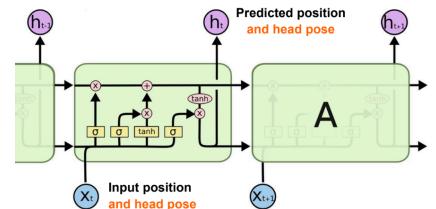


Nota: In generale è stato visto che il Trajectory Avoidance (LTA) model performa peggio di LSTM [CVPR16] e S-LSTM [CVPR16].

L'utilizzo congiunto della velocità delle persone e della posa della testa può essere una strategia efficace per la previsione del movimento delle persone. La velocità delle persone fornisce informazioni cruciali sulla direzione e sulla velocità con cui un individuo si sta muovendo. La posa della testa può fornire ulteriori indizi sulla direzione in cui una persona sta guardando o sul suo orientamento rispetto agli altri. Ad esempio, se una persona sta guardando in una direzione specifica, potrebbe indicare una possibile intenzione di muoversi in quella direzione. **Unendo la**

velocità delle persone e la posa della testa, il modello può apprendere relazioni complesse tra questi fattori e prevedere la traiettoria futura delle persone. Ad esempio, se una persona si sta muovendo velocemente in una direzione e la sua testa è orientata in quella stessa direzione, potrebbe suggerire un movimento continuo in quella direzione.

Una proposta di implementazione di questo metodo viene data nella **MX-LSTM** in cui, data una determinata iterazione della LSTM, come input x_t si prendono posizioni in input e posa della testa. Queste vengono elaborate dalla LSTM e in output h_t viene data la posizione predetta e la posa della testa. Per calcolare la loss si prende la distanza euclidea tra le due posizioni. Nota che oltre alla posizione nel futuro, possiamo anche predire una distribuzione gaussiana (media e deviazione standard). Questo ci permette di misurare la probabilità della nostra predizione rispetto alla distribuzione data dal ground-truth. Questo permette di avere meno incertezza nella predizione e quindi ottenere predizioni più accurate.

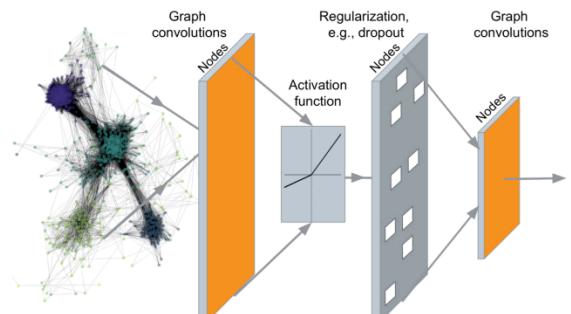


Prendendo la MX-LSTM e unendola con la social pooling (social pooling basata sulla posizione della testa) sono state raggiunte delle ottime performance. Questo approccio è chiamato S-MX-LSTM. Nota che negli approcci precedenti, l'errore era strettamente legato alla velocità della persona infatti, più questa procedeva lentamente maggiore era l'errore. Aggiungendo la posa della testa, la dipendenza dalla velocità viene in qualche modo annullata e l'errore rimane stabile indipendentemente dalla velocità della persona.

Nota: Un altro approccio più recente prevede di modellare ogni persona con una transformer network e, nonostante la difficoltà di modellazione della parte sociale, apporta delle performance molto più elevate rispetto agli LSTM.

Ci sono dei dati che non possono essere modellati da array 1D o 2D. Si pensi ad esempio alla posa delle persone: Ogni posa è formata da varie giunture ciascuna rappresentata da una coordinata tridimensionale, e ogni giuntura è collegata. Vedendo la posa in questo modo è facile notare come questa struttura possa essere rappresentata da un grafo. In generale, utilizzando i grafi possiamo definire delle strutture in cui non c'è nessun ordinamento dei nodi o punto di riferimento. Se i dati non seguono un ordinamento fisso o non hanno un punto di riferimento fisso, potrebbero richiedere approcci di modellazione più flessibili e i grafi consentono di rappresentare relazioni arbitrarie tra i nodi senza una sequenza fissa. Inoltre i grafi sono spesso dinamici e possono avere caratteristiche multimodali.

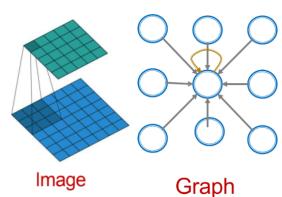
Vorremmo poter codificare un grafo in maniera tale che possiamo poi effettuare una predizione su questo. Possiamo quindi definire le **Graph Convolutional Networks (GCN)** che sono un tipo di modello progettato per gestire dati rappresentati sotto forma di grafi. Questi modelli sono particolarmente adatti per affrontare problemi in cui le relazioni tra gli elementi sono fondamentali per la comprensione del sistema. In un grafo, gli elementi sono rappresentati come nodi e le relazioni tra di essi sono rappresentate come archi. Ogni nodo può avere attributi associati, e gli archi possono avere pesi o attributi specifici. Più in dettaglio, assumiamo di avere un **grafo G** con **V** l'insieme dei vertici (dei nodi), **A** una matrice di adiacenza binaria che definisce come i vertici sono collegati tra di loro (es.



nel caso della posa questa ci direbbe come sono collegate le giunture nello spazio (connessione $j_1^t \rightarrow j_2^t$) nelle righe, e come cambia la posizione delle singole giunture nel tempo (connessione $j_1^t \rightarrow j_1^{t+1}$) nelle colonne - quindi considerando 26 giunture e 8 frames, A ha dimensionalità 26x8) e $X \in \mathbb{R}^{m \times |V|}$ una matrice di caratteristiche dei nodi (matrix of node features) che ci specifica gli attributi di ogni nodo nel grafo (es. nella posa la posizione (x, y, z) di ogni giuntura - quindi $m=3$ dimensioni e $|V|=26$ giunture). Specifichiamo con v un nodo in V e $N(v)$ l'insieme dei vicini di v (es. nella posa i vicini di una giuntura v sono le giunture direttamente connesse al passo t oppure nuova posizione della stessa giuntura al passo $t+1$).

Nota: Quando intendiamo che la matrice di adiacenza A è binaria, si intende che abbiamo un 1 se esiste un collegamento tra un nodo n_i e un altro n_j , 0 altrimenti.

Nota: La differenza principale tra CNN e GCN è che la prima viene utilizzata principalmente per dati strutturati in forma di griglia, come immagini e video quindi sono progettate per catturare



relazioni spaziali e gerarchie di feature in dati bidimensionali. La seconda è progettata specificamente per dati rappresentati come grafi poiché i grafi sono una struttura di dati più generale rispetto alle griglie e possono rappresentare relazioni più complesse tra gli elementi. In ogni caso, la struttura e l'idea dietro le GCN è simile a quella delle CNN. Difatti ogni nodo (diciamo pixel in questo caso) è incastrato in una struttura rigida (l'immagine) e quando effettuiamo la convoluzione passiamo un kernel sull'immagine ottenendo un valore nonché una feature relativa alla porzione di input esaminata. Quindi ci muoviamo di layer in layer per ottenere un'immagine astratta e rappresentativa dell'input. Nel caso si cerca di aggregare le informazioni dei nodi vicini e poi trasformare l'informazione ottenuta e spostarsi al layer successivo. Quindi, di nuovo, l'idea è quella di prendere l'informazione dei vicini e combinarla:

- Si trasformano i "messaggi" h_i dai vicini: $W_i h_i$
- Si sommano insieme (si aggregano): $\sum_i W_i h_i$. Questo è l'attributo che viene passato al layer successivo.

C'è però una differenza sostanziale in questa trasformazione. Nelle CNN usiamo un kernel che ha una struttura rigida quindi, dato un pixel, applichiamo la convoluzione sapendo che il risultato è chiaramente il contributo che si trovano, ad esempio, a destra e sinistra dove questi hanno i loro coefficienti. Nelle GCN non abbiamo il concetto di ordinamento e quindi non possiamo, ad esempio, distinguere i pixel a destra e a sinistra.

Quindi l'idea dei Graph Convolutional Networks si rifà al concetto di trasformazione/aggregazione anticipato in precedenza. **I vicini di un nodo ci definiscono il grafo computazionale** quindi una volta identificati i nodi le informazioni vengono propagate lungo i collegamenti del grafo. Le **informazioni propagate vengono quindi trasformate** mediante operazioni di trasformazione che tengono conto delle caratteristiche del nodo stesso e infine vengono **aggregate**. L'idea chiave è di **generare degli embeddings dei nodi** basandosi sul **vicinato locale della rete** (local network neighborhoods).

ES. Immaginiamo di avere il grafo a destra e consideriamo il nodo A. Vogliamo capire quali informazioni i vicini di A e i successivi vicini possono propagare. Quindi A riceve informazioni da B, C e D (i suoi diretti vicini). La loro informazione viene processata, trasformata e infine aggregata. Questi hanno loro stessi dei vicini. Nel caso di B, questo riceve informazioni da A e C. Di nuovo, le loro informazioni vengono processate, trasformate e aggregate. Questo processo è eseguito iterativamente su più layer (**layer GCN**) per catturare informazioni da nodi a distanze variabili. Nel caso in cui vogliamo che un messaggio arrivi da E a D abbiamo bisogno di 3 layers ($E \rightarrow C, C \rightarrow A, A \rightarrow D$).

Quindi, **ogni nodo ci rappresenta un grafo computazionale in base al vicinato**.

ES. Possiamo fare le stesse considerazioni dell'esempio precedente per ogni altro nodo dove questo riceve le informazioni dagli altri nodi nel grado a vari layers.

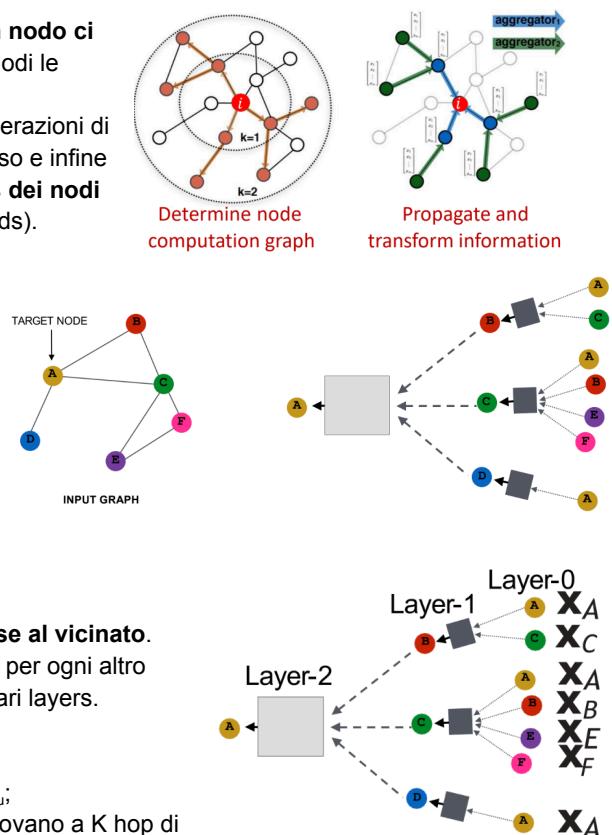
Il modello può avere una **profondità arbitraria**:

- I nodi hanno embeddings su ogni layer;
- L'embedding allo 0-layer del nodo u è la sua input feature x_u ;
- L'embedding al k-layer ottiene informazioni da nodi che si trovano a K hop di distanza (quindi A riceve informazioni da A, C e D a distanza 1 ma anche anche da tutti quelli a distanza 2).

I blocchi, in cui viene applicata trasformazione e aggregazione, sono le **reti neurali** in cui apprendiamo. Le distinzioni principali riguardano il modo in cui i diversi approcci aggregano le informazioni tra i livelli. L'approccio più basico prevede di:

- Fare la media delle informazioni provenienti dal vicinato;
- Applicare una rete neurale.

ES. Nel caso di A, si aggregano le informazioni (gli attributi) di B, C e D. Si fa la media x e infine applichiamo la trasformazione calcolando xW .



Cerchiamo di formalizzare questo primo approccio. Sia h_v^0 il 0-layer embedding cioè un embedding equivalente alle features dei nodi (es. x_F nel nodo F nell'esempio precedente). Questo è l'input dell'encoder:

$$h_v^0 = x_v$$

Siano:

- W_k : Matrice dei pesi per l'aggregazione dei vicini;
- B_k : Matrice dei pesi per trasformare il vettore nascosto del singolo nodo;
- $h_v^{(l)}$: La hidden representation del nodo v al layer l .

Quello che facciamo adesso è trasformare gli embedding layer per layer (layer-wise) in maniera ricorsiva. Quindi nel layer $l+1$ prendiamo l'embedding di v al layer l precedente ($h_v^{(l)}$) e moltiplichiamolo per B_l che è una trasformazione applicata ai nodi stessi. Consideriamo poi la media degli embeddings dei vicini u del nodo v : Quindi siano ($h_u^{(l)}$) gli attributi dei vicini u di v al layer precedente, li sommiamo e li dividiamo per la cardinalità dei vicini di v e otteniamo la media. Applichiamo quindi la **trasformazione** moltiplicando la media calcolata con W_l (si tratta di un semplice MLP).

Siccome vogliamo concatenare $L-1$ layers, dobbiamo aggiungere non linearità (es. ReLU):

$$h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$$

Non Linearity (e.g. ReLU) Average of neighbor's previous layer embeddings Total number of layers

embedding of v at layer l

Alla fine otteniamo z_v (output dell'encoder) corrisponde all'embedding di ogni nodo all'ultimo layer L :

$$z_v = h_v^{(L)}$$

Ora vogliamo allenare il modello che genera questi embedding (quindi vogliamo generare W_l e B_l ottimali) e per fare questo abbiamo prima di tutto bisogno di una **loss function sugli embeddings**. Quello che si può fare è calcolare gli embeddings per ogni nodo v fino al layer L e poi applicare SGD per allenare i pesi.

Molte aggregazioni possono essere eseguite in modo efficiente mediante operazioni su matrici (sparse).

Quindi, sia $H^{(l)}$ la concatenazione delle varie hidden representation dei nodi v al layer l :

$$H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^T$$

Nota che calcoliamo la trasposta della matrice e questo ci suggerisce che ogni elemento di $H^{(l)}$ si comporterà come la riga nelle operazioni tra matrici. Ora si ricorda che $A_{v,:}$ vale 1 se c'è un collegamento, 0 altrimenti. Quindi definiamo:

$$\sum_{u \in N(v)} h_u^{(l)} = A_{v,:} H^{(l)}$$

Come la somma delle hidden representation dei vicini u di v . Nota che questa la possiamo definire anche come la moltiplicazione tra $H^{(l)}$ e A_v in quanto consideriamo solo gli elementi $h_u^{(l)}$ di $H^{(l)}$ che sono vicini di v cioè tali per cui $A_{v,:}=1$. Ora, sia D la matrice diagonale dove in D abbiamo il numero di vicini per ogni nodo:

$$D_{v,v} = \text{Deg}(v) = |N(v)|$$

In $D_{v,v}$ abbiamo il numero di vicini di v nonché il grado $\text{Deg}(v)$. Anche l'inverso D^{-1} di una matrice diagonale D è una matrice diagonale, quindi:

$$D_{v,v}^{-1} = 1/|N(v)|$$

Possiamo quindi riscrivere la media nella formula precedente come:

$$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \rightarrow H^{(l+1)} = D^{-1} A H^{(l)}$$

Dove D^{-1} (è il denominatore che ci permette di normalizzare) che viene moltiplicato alla somma dei vicini $A H^{(l)}$.

Possiamo quindi riscrivere l'intera formula ricorsiva come:

$$H^{(l+1)} = \sigma(\tilde{A} H^{(l)} W_l^T + H^{(l)} B_l^T)$$

Con $\tilde{A} = D^{-1}A$. Ora, la parte rossa (sx) riguarda l'aggregazione dei vicini (**neighborhood aggregation**) mentre la parte blu (dx) è riguarda la trasformazione del nodo stesso (**self transformation**). In pratica, ciò implica che è possibile utilizzare una moltiplicazione efficiente di matrici sparse (\tilde{A} è sparsa).

Nota: Non tutti i GNN possono essere espressi in forma matriciale, quando la funzione di aggregazione è complessa (più avanti vedremo un caso).

Adesso, vogliamo semplificare ulteriormente il numero di parametri apprendibili dalla rete quindi cerchiamo di rimuovere B_i . Quello che possiamo integrare è $B_i=1$ dentro $A'=A+I$, cioè quello che vogliamo fare è fare in modo che **B_i e W_i condividano gli stessi parametri**. Applicando quello che abbiamo appena detto, si ottiene:

$$H^{(l+1)} = \sigma(\tilde{A} H^{(l)} W_i^T) \quad \text{where} \quad \tilde{A} = D^{-1}A'$$

Questa semplice formula include sia la trasformazione (moltiplicazione tra ogni nodo v e W) e l'aggregazione che in questo caso è semplicemente la media dei vicini.

Come facciamo ad allenare una GCN (GNN)?

Abbiamo visto prima che l'output della rete è l'embedding del nodo finale z_v che è funzione del grafo in input con tutte le aggregazioni che vengono fatte al suo interno. In maniera tale da apprendere i parametri del grafo, dobbiamo stabilire un certo obiettivo che dipende dal task che vogliamo compiere:

- **Supervised setting:** Se abbiamo le etichette per i nodi cioè abbiamo il ground-truth, allora possiamo andare a comparare output e ground-truth per generare una loss. L'obiettivo è **minimizzare la loss**:

$$\min_{\Theta} \mathcal{L}(y, f(z_v))$$
 dove y è il ground-truth. Per misurare la loss si può usare la L2 distance se y è un numero reale oppure la cross entropy se y è categorico.
- **Unsupervised setting:** In questo caso non abbiamo etichette per i nodi quindi possiamo usare la **struttura del grafo come una sorta di supervisione**. Ad esempio, se sappiamo che ci deve essere una certa connettività tra vari nodi possiamo controllare come e se avviene questa connettività. Anche il forecasting prevede unsupervised setting (in particolare self-supervision) dove non abbiamo etichette e utilizziamo dati passati per predire dati futuri (ad esempio, dato una certa sequenza in input potremo usare la prima parte per predire la seconda e quindi in questo modo abbiamo anche il ground truth).

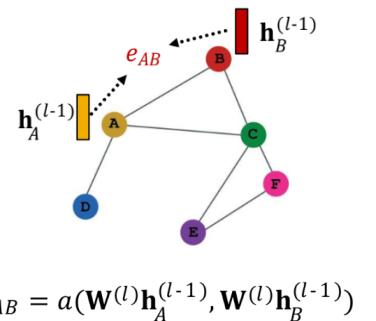
Finora abbiamo usato le matrici di adiacenza e una semplice aggregazione tra vicini. Possiamo fare meglio di così? Quello che potremmo fare è stimare i termini α_{vu} in una matrice di affinità basandosi sugli attributi dei nodi. Questo tipo di grafi prende il nome di **Graph Attention Networks (GAT)**. L'obiettivo di questo approccio è quello di **assegnare un'importanza arbitraria a diversi vicini di ciascun nodo** nel grafo. L'idea fondamentale è di calcolare l'embedding $h^{(l)}$ di ogni nodo nel grafo seguendo una strategia di attenzione. In altre parole, i nodi dirigono la loro attenzione verso i messaggi provenienti dai loro vicini, specificando implicitamente pesi diversi per nodi diversi all'interno di un vicinato.

Quindi, assumiamo che il termine α_{vu} sia calcolato come il sottoprodotto (byproduct) di un **meccanismo di attenzione α** dove α calcola i **coefficienti di attenzione e_{vu}** tra coppie di nodi u, v basandosi sui loro messaggi:

$$e_{vu} = \alpha(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)})$$

Quindi la funzione α (attenzione) guarda ai messaggi aggregati e trasformati dei nodi u e v , e sulla base di questi messaggi crea un peso che è il coefficiente di attenzione e_{vu} . Il coefficiente di attenzione indica quindi l'importanza del messaggio di u per il nodo v . I messaggi inviati dai vicini u di v vengono pesati in base ai loro pesi calcolati. Questo significa che alcuni vicini possono avere un contributo maggiore nella creazione del nuovo embedding del nodo rispetto ad altri. Per fare questa cosa possiamo prima di tutto **normalizzare i coefficienti di attenzione e_{vu}** usando una **softmax** (in maniera che questi sommino ad 1) ottenendo i final attention weight α_{vu} :

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$



$$e_{AB} = \alpha(W^{(l)} h_A^{(l-1)}, W^{(l)} h_B^{(l-1)})$$

Infine calcoliamo la **somma pesata** basandosi sul final attention weight appena calcolato:

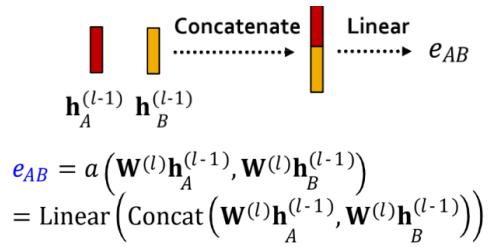
$$\mathbf{h}_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Nota: L'attenzione utilizzata qui è del tipo additivo (non la dot-product come nei transformers).

L'approccio descritto è **agnostico rispetto alla scelta di a** , il che significa che è possibile utilizzare diverse implementazioni di funzioni di attenzione in base alle esigenze del modello.

EX: Un esempio comune è utilizzare un semplice strato lineare (Linear layer) come funzione di attenzione. Questo strato lineare ha pesi (trainable parameters) che vengono appresi durante il processo di addestramento. Quindi, i pesi della funzione di attenzione sono trattati come parametri addestrabili che vengono ottimizzati insieme agli altri parametri della rete neurale. I pesi della funzione di attenzione sono considerati parametri addestrabili e vengono ottimizzati durante il processo di addestramento insieme agli altri parametri della rete neurale.

I parametri della funzione di attenzione a sono appresi congiuntamente agli altri parametri della rete neurale, come ad esempio i pesi $\mathbf{W}^{(l)}$.



Nota: Possiamo anche usare una **multi-head attention** che permette di **stabilizzare il processo di apprendimento dei meccanismi di attenzione**. L'approccio funziona in questo modo:

- Creazione di repliche di attention scores: Vengono create diverse **repliche (head) del meccanismo di attenzione**, ognuna con un set **diverso di parametri appresi durante l'addestramento**. Ogni head opera indipendentemente e apprende rappresentazioni differenti delle relazioni tra i nodi:

$$\begin{aligned} \mathbf{h}_v^{(l)}[1] &= \sigma(\sum_{u \in N(v)} \alpha_v^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}) \\ \mathbf{h}_v^{(l)}[2] &= \sigma(\sum_{u \in N(v)} \alpha_v^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}) \\ \mathbf{h}_v^{(l)}[3] &= \sigma(\sum_{u \in N(v)} \alpha_v^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}) \end{aligned}$$

- Calcolo degli attention scores multipli: Ciascuna head calcola i suoi propri attention scores. Questi punteggi riflettono l'importanza relativa dei vicini rispetto al nodo di partenza e sono ottenuti applicando la funzione di attenzione appropriata a ciascuna testa.
- Aggregazione degli output delle head: Gli output delle diverse head vengono quindi aggregati per ottenere l'output finale del attention layer. Le due modalità comuni di aggregazione sono:
 - Concatenazione: Gli output delle head vengono concatenati lungo una nuova dimensione.
 - Somma: Gli output delle head vengono sommati.

$$\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$$

Quali sono i benefici del meccanismo di attenzione?

- Importanza Differenziata: Questo approccio consente di specificare implicitamente valori di importanza diversi (α_{vu}) per i diversi vicini di ciascun nodo. Questo significa che il modello può attribuire pesi differenti alle relazioni con i vicini, consentendo una maggiore flessibilità nell'apprendimento delle rappresentazioni.
- Efficienza Computazionale: Il calcolo dei coefficienti di attenzione può essere parallelizzato su tutti gli archi del grafo. L'aggregazione può essere parallelizzata su tutti i nodi del grafo.
- Efficienza di Archiviazione: Le operazioni su matrici sparse non richiedono più di $O(V+E)$ voci da archiviare. Abbiamo inoltre un numero fisso di parametri, indipendentemente dalle dimensioni del grafo. Questo perché impariamo una sola funzione per layer che ci calcola l'attenzione dati gli attributi di due nodi.
- Localizzato: L'attenzione si concentra solo sui vicini locali di ciascun nodo, rendendo l'approccio più adatto per grafi di grandi dimensioni.
- Capacità Induttiva: È un meccanismo condiviso tra gli archi (**edge-wise mechanism**), il che significa che non dipende dalla struttura globale del grafo. Questo favorisce la capacità induttiva del modello, consentendo di generalizzare bene a nuovi dati o grafi.

Nella pratica, un punto di partenza per definire una layer di una GNN è dato dall'esempio a destra.

Ora, partendo da questo singolo layer, per costruire una GNN possiamo impilare uno o più layer in maniera sequenziale. Chiaramente più sono i layer, maggiore è il processamento dell'informazione derivante dai vicini.

Vediamo adesso nel dettaglio come possiamo costruire una GNN. Come detto poco fa l'approccio standard prevede di impilare layer GNN in maniera sequenziale. Quindi data la feature di nodo x_v come input, possiamo passare questo input di layer in layer fino ad ottenere l'embedding del nodo $h_v^{(L)}$ dove L è il numero di layer GNN nella rete. Purtroppo non possiamo impilare troppi layer in quanto potremmo incorrere nel **problema della sovrassaturazione**

(**over-smoothing**). Il problema è che quando si applica ripetutamente l'aggregazione delle informazioni attraverso i vicini nei diversi strati di una GNN, gli embedding dei nodi tendono a **convergere verso un unico valore comune**. Questo può essere problematico perché la differenziazione degli embedding dei nodi è fondamentale per il corretto funzionamento della GNN. Perché questo avviene?

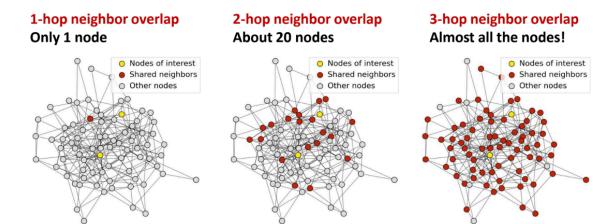
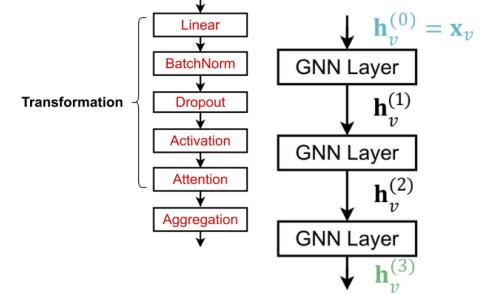
Ricorda che chiamiamo **receptive field** l'insieme dei nodi che determinano l'**embedding di un certo nodo**. In una GNN con K-layer, ogni nodo ha un receptive field basato su nodi a **distanza K** dal nodo stesso (**K-hop neighborhood**). Si può notare come i vicini condivisi tra i vari crescano rapidamente quando aumentiamo il numero di hop (numero di livelli GNN). Ora, l'embedding di un nodo è direttamente collegato al suo receptive field quindi se due nodi hanno receptive fields altamente sovrapposti, i loro embeddings saranno molto simili. In contrasto con le reti neurali in altri domini, come le reti neurali convoluzionali (CNN) per la classificazione delle immagini, l'aggiunta di ulteriori layer GNN non è sempre vantaggiosa. Invece di seguire l'approccio di aggiungere più strati, ci sono alcune considerazioni importanti da prendere in considerazione:

- Analisi del Campo Recettivo Necessario: Prima di decidere quanti layer GNN aggiungere, è essenziale analizzare il receptive field necessario per risolvere il problema specifico. Ad esempio, questo può essere fatto calcolando il diametro del grafo, cioè la massima distanza tra due nodi nel grafo.
- Determinare il numero di layer GNN (L): Impostare il numero di strati GNN (L) in modo che sia **leggermente superiore al receptive field desiderato**. Non è consigliabile impostare L in modo eccessivamente grande.

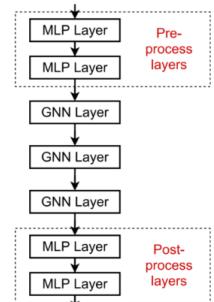
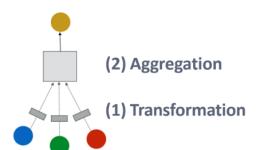
Quindi, come possiamo incrementare l'espressività di un GNN anche se ha pochi layer?

Sono possibili diverse soluzioni:

1. Per rendere una Graph Neural Network (GNN) più espressiva, una soluzione che consiste nell'**aumentare la potenza espressiva all'interno di ciascun strato GNN**. Nelle implementazioni precedenti, ogni funzione di trasformazione o aggregazione includeva solo un singolo strato lineare. Tuttavia, è possibile rendere più profonda la rete neurale all'interno di ciascuna funzione di aggregazione o trasformazione introducendo più strati lineari o persino reti neurali profonde all'interno di ciascuna funzione di trasformazione o aggregazione (es. se necessario, ogni box potrebbe includere 3-layer MLP).
2. E' possibile **aggiungere strati che non passano messaggi**. Ciò significa che una GNN non deve contenere esclusivamente layer GNN, è possibile introdurre anche fully connected layers (MLP) che operano su ciascun nodo prima e/o dopo i layer GNN. Questi layer extra possono essere denominati:
 - Pre-processing layers:** Questi layer sono importanti quando è necessario codificare le caratteristiche del nodo in modo più avanzato prima di passarle attraverso gli strati GNN. Ad esempio, se i nodi rappresentano immagini o testo, l'uso di strati MLP prima di applicare gli strati GNN può aiutare a catturare rappresentazioni più ricche.
 - Post-processing layers:** Questi strati sono importanti quando è necessario eseguire ragionamenti o trasformazioni avanzate sulle rappresentazioni dei nodi ottenute dagli strati GNN. Ad esempio, in task di classificazione di grafi, è possibile aggiungere strati MLP dopo gli strati GNN per ottenere una rappresentazione finale dei nodi.



Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem



Come ci si può comportare se la rete richiede comunque molti layer GNN?

In questo caso è possibile migliorare la capacità della rete aggiungendo **skip connections** nelle GNN. L'idea qui è di aggiungere una connessione diretta (una **shortcut**) dal layer in input al risultato del layer corrente. In questo modo, l'informazione originale x viene mantenuta e può contribuire direttamente all'output finale. Questo può essere particolarmente utile per preservare informazioni importanti che potrebbero altrimenti essere smussate durante il passaggio attraverso diversi layer GNN. Quindi:

- Prima di aggiungere shortcut: Output= $F(x)$;
- Dopo l'aggiunta di shortcut: Output= $F(x)+x$.

Questo permette quindi di incrementare l'impatto dei layer precedenti sull'embedding del nodo finale.

L'intuizione di base è che l'aggiunta di skip connections genera una varietà di percorsi attraverso la rete, consentendo alla rete di apprendere sia da percorsi più superficiali che da percorsi più profondi. Questa varietà di percorsi crea una sorta di **insieme di modelli**, ognuno dei quali può contribuire in modo unico all'apprendimento della rete. Quindi, assumendo di avere N skip connections nella rete (un percorso alternativo nella rete che può eventualmente anche includere o saltare determinati layer), abbiamo **2^N possibili percorsi**. Ogni percorso rappresenta una combinazione diversa di moduli inclusi o saltati. Quindi, automaticamente, otteniamo una **miscela di GNN più superficiali e GNN più profonde**. Le GNN più profonde sono quelle che soffrono di over-smoothing, quelle più superficiali mantengono il potere discriminativo del grafo.

$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)} \right)$$

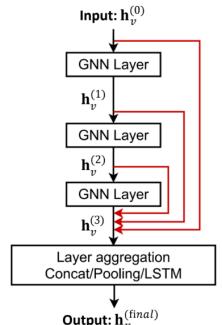
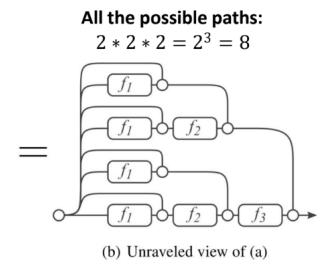
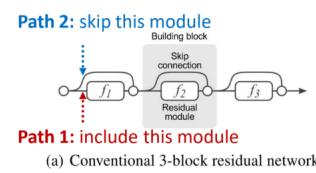
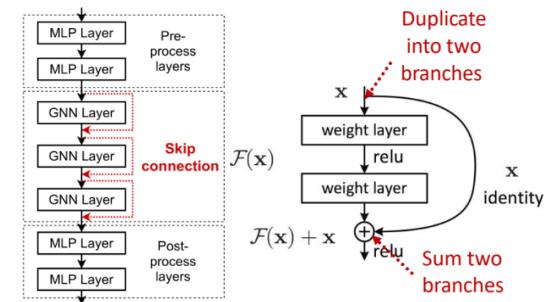
$\mathbf{F}(x) \quad + \quad x$

Abbiamo un'altra possibile opzione che implica che le **skip connections passino direttamente all'ultimo strato della rete**. Invece di attraversare tutti gli strati intermedi, le informazioni vengono trasmesse direttamente all'ultimo strato per l'aggregazione finale. In pratica, ciò significa che l'output di uno strato viene aggiunto direttamente all'output finale senza passare attraverso gli altri strati. Questo può essere utile quando si ritiene che le informazioni da uno specifico strato siano particolarmente rilevanti o quando si vuole semplificare il percorso delle informazioni attraverso la rete.

Vediamo adesso un caso studio. Ipotizziamo che abbiamo un certa posa composta rappresentata con V articolazioni (joints). Il nostro obiettivo è predire le coordinate future dei V joints. Immaginiamo di avere T frames (**motion history**) che raggruppiamo in un tensore $X_{in} = [X_1, X_2, \dots, X_T]$ dove in ogni frame rappresentiamo le varie coordinate x, y, z dei V joints (quindi $X_i \in \mathbb{R}^{3 \times V}$): Questo è il nostro input. Vogliamo predire i futuri K frames $X_{out} = [X_{T+1}, X_{T+2}, \dots, X_{T+K}]$: Questo è il nostro output. Si ricorda che questo è un esempio di self-supervision in quanto non abbiamo etichette (ad esempio, dato una certa sequenza in input potremo usare la prima parte per predire la seconda e quindi in questo modo abbiamo anche il ground truth). Ci sono stati vari approcci per risolvere il problema e tipicamente quello che si fa è **codificare le relazioni temporali e spaziali (relazioni tra le articolazioni) in maniera separata usando diversi framework**. Una possibile architettura è la seguente:

- **GCN-only Encoder**: Si può usare un Grafo Convolutional Network (GCN) come strumento principale per codificare le relazioni tra le articolazioni. Ogni articolazione in ogni frame osservato viene trattata come un nodo nel grafo. Gli archi del grafo vengono utilizzati per codificare le relazioni cinematiche nel corpo (kinematic tree).
- **Temporal Convolutional Network Decoder (TCN)**: Una volta che abbiamo la codifica dei nodi nel grado possiamo usare una semplice TCN per predire i fotogrammi futuri. Utilizzando convoluzioni 1×1 , il TCN può modellare le relazioni temporali e spaziali tra le articolazioni per predire i fotogrammi futuri (K frames successivi) sulla base dei fotogrammi osservati (T fotogrammi precedenti). In questo caso non teniamo conto della relazione tra i nodi, ma la **predizione viene fatta su ogni singola articolazione**.

Ora, è stato proposto un framework per codificare queste due informazioni in grado.



Dato un grafo $G=(V, E)$ i nodi sono dei tipo $x_{vt} \in \mathbb{R}^3$ con $v=1..V$ e $t=1..T$ con V il numero di articolazioni e T i frames, mentre gli archi sono del tipo $(x_{vt}, x_{vt}) \in \mathbb{R}^{VT \times VT}$ e ci codificano le relazioni spazio temporali. La matrice di adiacenza $A^{st} \in \mathbb{R}^{VT \times VT}$. Un layer GCN può essere codificato in questo modo:

$$W^{(l)} \in \mathbb{R}^{C^{(l)} \times C^{(l+1)}} \quad C^{(1)} = 3, \quad \mathcal{H}^{(1)} = \mathcal{X}_{in} \quad \mathcal{H}^{(l+1)} = \sigma(A^{st-(l)} \mathcal{H}^{(l)} W^{(l)})$$

Rispetto agli standard GCN, si noti come qui abbiamo una sorta di ordinamento che ci permette di definire la posizione fisica delle varie articolazioni in un corpo. Quindi, il vantaggio di questa struttura è che nella matrice di adiacenza A , ogni posizione a_{ij} rappresenta una relazione tra nodo (un'articolazione) ad un certo periodo di tempo (frame) e un altro nodo in un certo periodo di tempo.

Il problema del "cross-talk" spazio-temporale si riferisce alla difficoltà di separare chiaramente le informazioni spaziali da quelle temporali in un modello che deve gestire sia relazioni spaziali che temporali. L'approccio **STS-GCN** (Space-Time Separable Graph Convolutional Network) propone una strategia di factorization per affrontare questo problema in ciascun livello (layer) della rete GCN. Si propone di separare spazio e tempo attraverso una convoluzione grafica fattorizzata. In particolare, si fa riferimento a questa factorization come $A_{st} = A_s A_t$, dove:

- A_s : Rappresenta la matrice di adiacenza per le relazioni tra le articolazioni in un dato frame.
- A_t : Rappresenta la matrice di adiacenza per le relazioni temporali per una data articolazione.

La fattorizzazione proposta consente di affrontare separatamente le relazioni tra articolazioni in uno specifico frame (A_s) e le relazioni temporali per una specifica articolazione (A_t). Possiamo anche introdurre il concetto di attenzione all'interno di questa rete.

6. Detection and Segmentation

Finora abbiamo affrontato il problema dell'image classification, quindi abbiamo una certa immagine che processiamo una deep neural network (es. Alex Net) ottenendo l'embedding sul quale possiamo calcolare un punteggio usando, ad esempio, la softmax e questo ci permette di capire qual è la classe dell'oggetto.

In generale, ci sono svariati problemi che possono essere risolti:

- **Classificazione** (classification): L'obiettivo è assegnare un'etichetta o una classe a un'immagine. Ad esempio, riconoscere se un'immagine contiene un cane o un gatto.
- **Segmentazione Semantica** (semantic segmentation): L'obiettivo è assegnare etichette semantiche a ciascun pixel di un'immagine, dividendo l'immagine in regioni che rappresentano oggetti o parti di oggetti. Ad esempio, assegnare una classe (come "strada", "auto", "persona") a ciascun pixel.
- **Rilevamento Oggetti** (object detection): L'obiettivo è individuare e classificare gli oggetti all'interno di un'immagine, solitamente fornendo anche le coordinate del rettangolo delimitatore (bounding box) per ciascun oggetto rilevato. Ad esempio, rilevare auto in un'immagine stradale.
- **Segmentazione di Istanza** (instance segmentation): Simile alla segmentazione semantica, ma con l'aggiunta di distinguere le diverse istanze di oggetti della stessa classe. Ad esempio, se ci sono due persone in un'immagine, la segmentazione di istanza dovrebbe separarle.

6.1. Semantic Segmentation

Andiamo adesso nel dettaglio con la Semantic Segmentation. Prima di tutto definiamo il concetto di "THINGS" e "STUFF" che è spesso utilizzato per distinguere tra oggetti individuabili (entità distinte, come persone, gatti, cavalli, ecc.) e sfondi o elementi ambientali più omogenei (come strade, erba, cielo, ecc.). In generale:

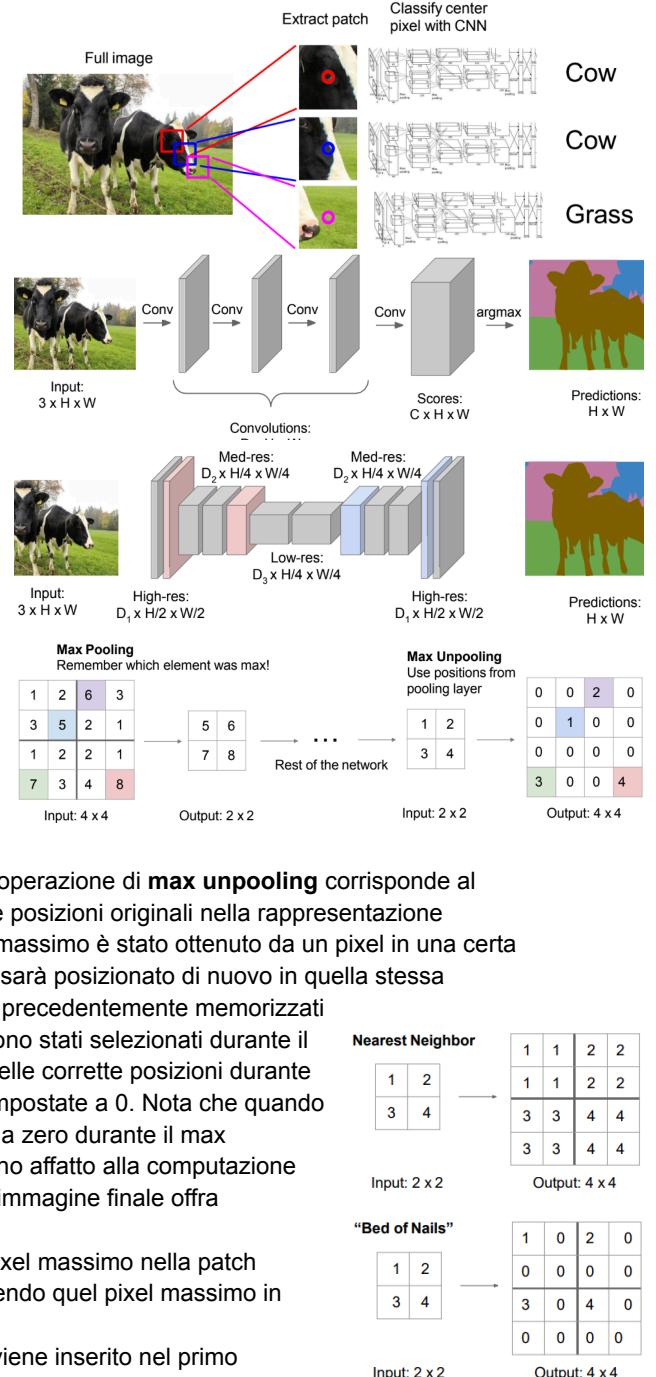
- Things: Include entità distinte e ben definite come persone, animali, veicoli, ecc. Gli oggetti hanno generalmente forme chiare e ben definite. Gli oggetti possono avere identità distinte, e il riconoscimento delle istanze è spesso un obiettivo.
- Stuff: Include elementi più amorfi e sfondi, come strade, cieli, erba, ecc. Spesso non hanno forme chiare e possono occupare aree estese. Non si cerca di distinguere le diverse istanze di "STUFF". Ad esempio, non si cerca di distinguere una parte specifica dell'erba da un'altra. L'analisi può essere eseguita a livello di pixel, concentrando sulla "texture" o sulle caratteristiche locali.

La **Semantic Segmentation** è un compito di visione artificiale in cui l'obiettivo è assegnare una categoria (etichetta semantica) a ciascun pixel di un'immagine. A differenza della instance segmentation, non si cerca di distinguere tra diverse istanze di oggetti della stessa classe, ma piuttosto di **assegnare una categoria generica a ogni pixel**.

EX. Ad esempio, considera un'immagine di una scena che contiene il cielo, un gatto e l'erba. Nella segmentazione semantica, ogni pixel nell'immagine verrebbe etichettato con una delle categorie come "Cielo", "Gatto" o "Erba". Il risultato finale è un'immagine dove ogni pixel è associato a una categoria semantica.

Un primo approccio potrebbe essere quello di selezionare delle patch dell'immagine, prendere il pixel centrale della patch e classificarlo per capire a quale classe appartiene. Il problema è che è estremamente inefficiente infatti non riutilizza le features condivise tra patch sovrapposte (cosa che invece viene fatta nelle CNN) quindi si può usare un approccio migliore che prevede le CNN. Data un'immagine $3 \times H \times W$ (con 3 il numero di canali RGB) possiamo usare una serie di convolutional layers $D \times H \times W$ che elaborano l'immagine e alla fine otteniamo delle activation maps $C \times H \times W$ con C il numero di classi, che poi utilizziamo per effettuare la previsione. Ora, qui il problema è che le immagini in input possono essere molto grandi così come il numero di classi C , cosa che porta ad avere activation maps $C \times H \times W$ molto grandi. Un approccio migliore prevede di progettare una rete costituita principalmente da **strati convoluzionali**, consentendo sia il **downsampling** che l'**upsampling all'interno della rete**. Quindi il **downsampling** viene impiegato per diminuire le dimensionalità e la risoluzione dell'immagine in input e questo lo possiamo fare semplicemente applicando **pooling** o **convoluzione con stride**. Questo downsampling aiuta a catturare informazioni di contesto più ampie ma con meno dettagli. Per tornare alla risoluzione originale dell'immagine, si possono utilizzare strati di **upsampling**. L'upsampling può essere ottenuto tramite tecniche come:

- **Max Unpooling:** Durante la fase di pooling, viene effettuata una riduzione delle dimensioni (downsampling) dell'immagine o della rappresentazione. L'operazione di pooling, ad esempio il max pooling, comporta la selezione del valore massimo in una finestra di pooling. L'operazione di **max unpooling** corrisponde al "disfare" il max pooling, cioè, restituire i valori massimi alle posizioni originali nella rappresentazione precedente al pooling. Se durante il max pooling il valore massimo è stato ottenuto da un pixel in una certa posizione, durante il max unpooling, quel valore massimo sarà posizionato di nuovo in quella stessa posizione. Di solito, questa operazione è guidata da indici precedentemente memorizzati durante il max pooling. Gli indici indicano quali elementi sono stati selezionati durante il max pooling, e vengono utilizzati per posizionare i valori nelle corrette posizioni durante il max unpooling. Le altre posizioni sono semplicemente impostate a 0. Nota che quando faremo poi la convoluzione i pixel che sono stati impostati a zero durante il max unpooling contribuiranno minimamente o non contribuiranno affatto alla computazione del valore di output della convoluzione. Questo fa sì che l'immagine finale offra un'informazione parziale soprattutto vicino ai bordi.
- **Nearest Neighbor:** In questo caso, una volta ottenuto il pixel massimo nella patch durante il pooling, nell'unpooling si ripopola la patch inserendo quel pixel massimo in tutta la patch (e non solo nella vecchia posizione);
- **Bed of Nails:** Come il max unpooling ma in bit maggiore viene inserito nel primo elemento della patch.
- **Transpose Convolution:** A differenza di una convoluzione standard, che riduce le dimensioni dello spazio di input (ipotizzando uno stride maggiore di 1), la **transpose convolution aumenta le dimensioni dello spazio di input**. Nella transpose convolution, invece di applicare un filtro a piccola scala ad ogni regione dell'input, si applica un filtro più ampio che si sovrappone su diverse regioni dell'input. Questo può essere



visualizzato come l'operazione inversa di una convoluzione. Assumiamo di applicare una 3x3 transpose convolution con stride=2 e pad=1. Per ogni pixel dell'immagine in input, il kernel si muove di due pixel nell'immagine di output (grazie allo **stride che ci definisce il ratio tra il movimento dell'input e dell'output**). L'obiettivo principale è "riempire" gli zeri durante il max unpooling, ripristinando la risoluzione dell'immagine.

EX. Immaginiamo di avere una semplice immagine composta da un'immagine rappresentata da un array 1D. Immaginiamo che ci troviamo sull'input a quindi quello che facciamo è applicare un filtro che si produce un output ax, ay, az. Adesso ci spostiamo nella prossima posizione dell'input e compiamo la stessa operazione ottenendo bx, by, bz. **L'output contiene copie del filtro ponderato dall'input, dove l'input viene sommato dove ci sono sovrapposizioni nell'output.** Adesso, la convoluzione può essere espressa come una moltiplicazione tra matrici e, introducendo uno stride=2 possiamo notare come questo equivale a saltare dalla prima alla terza riga:

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=1, padding=1

$$\vec{x} * \vec{a} = X\vec{a}$$

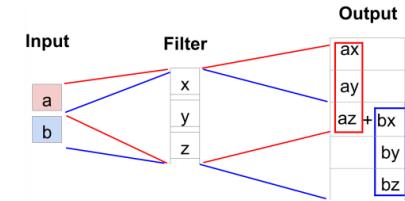
$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

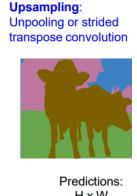
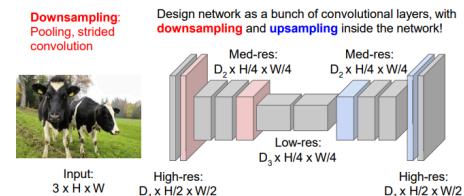
$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

Example: 1D conv transpose, kernel size=3, stride=2



La transpose convolution (il risultato che otteniamo nell'esempio precedente) si ottiene facendo la convoluzione della matrice X dell'input.

Nota: In generale si preferisce la transpose convolution perché dotata di parametri che possono essere addestrati durante il processo di apprendimento della rete neurale. Questo significa che la rete può apprendere schemi complessi di upsampling in base ai dati di addestramento. D'altra parte, il max unpooling è una tecnica di upsampling senza parametri; quindi, non c'è apprendimento diretto degli schemi di upsampling.



Per quanto riguarda la valutazione di un modello di segmentazione semantica esistono varie metriche che si possono utilizzare:

- **Accuracy (Precision):** Questa metrica rappresenta la percentuale di pixel correttamente classificati rispetto al totale dei pixel nell'immagine. Tuttavia, in caso di **sbilanciamento delle classi**, l'accuracy potrebbe non essere una metrica informativa, poiché un modello potrebbe ottenere un'accuracy elevata prediligendo semplicemente la classe maggioritaria.
- **Intersection over Union (IoU):** Si calcola come l'area dell'intersezione di previsione e ground-truth divisa per l'area dell'unione tra la previsione e la ground-truth. L'IoU è particolarmente utile quando le classi sono sbilanciate.
- **Per-Class Accuracy:** In questo caso l'accuracy viene calcolata separatamente per ciascuna classe e alla fine si effettua la media. Può essere utile identificare le classi specifiche per cui il modello potrebbe avere difficoltà.

La segmentazione semantica presenta diverse sfide nella raccolta dei dati e nell'annotazione:

- Localizzazione Precisa: L'annotazione di una localizzazione precisa per ogni pixel può richiedere molto tempo ed inoltre il processo può essere soggetto a errori.
- Cattura delle Sottocategorie: Etichettare ogni possibile sottocategoria in una scena può portare a una distribuzione sbilanciata delle classi, con alcune classi ben rappresentate e altre sottocategorie meno frequenti. La soluzione potrebbe essere quella di annotare solo alcune classi di oggetti (diciamo quelle che danno il contributo maggiore nell'immagine) e etichettare le altre con "altro".

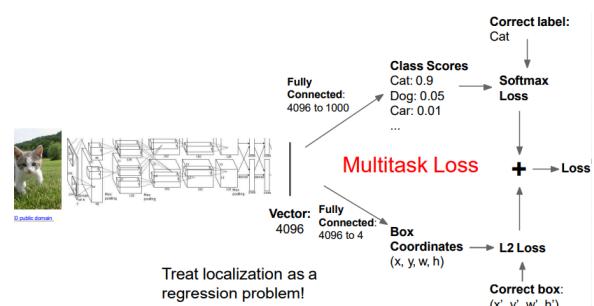
Ci sono alcuni metodi che aiutano nella fase di annotazione dei dati:

- **Propagazione delle Etichette** (label propagation): In alcuni casi, vengono utilizzati approcci di propagazione delle etichette, in cui le annotazioni vengono propagate in modo semi-automatico attraverso gli strumenti di annotazione. Questo può ridurre il carico di lavoro umano ma potrebbe introdurre errori.
- **Strumenti di Annotazione Semi-automatici** (Semi-automatic labelling annotation tools): Alcuni strumenti, come **Fluid Annotation**, cercano di semplificare il processo di annotazione attraverso approcci semi-automatici, migliorando l'efficienza della raccolta dati.

6.2. Object Detection

L'object detection è una tecnologia che si occupa di identificare e localizzare oggetti (multipli) in un'immagine o in un video. L'obiettivo è **individuare la presenza di oggetti specifici e determinare la loro posizione all'interno dell'immagine**. Gli algoritmi di object detection analizzano un'immagine e generano **bounding box** intorno agli oggetti identificati, spesso associando anche un'etichetta che indica la classe dell'oggetto (ad esempio, "persona", "auto", "cane", ecc.). Un modo per vedere l'object detection è una sorta di finalizzazione dell'operazione di classificazione, quindi possiamo utilizzare una CNN che prende un'immagine in input e ci restituisce due tipi di output:

- **Classificazione**: Viene assegnata un'etichetta all'oggetto individuato. Ad esempio, se l'immagine contiene un gatto, la rete dovrebbe classificarlo come "gatto" dandogli il punteggio più alto (es. utilizzando la Cross Entropy loss per l'addestramento del modello e la softmax per ottenere la predizione).
- **Localizzazione**: Questa parte implica la definizione di una bounding box intorno all'oggetto rilevato. Invece di trattare la localizzazione come una classificazione di pixel, viene trattata come un **problema di regressione**. Questo significa che la rete apprende a predire direttamente i parametri della bounding box, come le coordinate del rettangolo e la sua dimensione (es. utilizzando la L2 loss tra predizione e ground-truth per addestrare il modello).



Una volta ottenute le **due loss possiamo sommarle tra di loro per ottenere la loss finale** e questo ci permette di addestrare la rete risolvendo i due problemi in maniera congiunta.

Nota: L'utilizzo di due loss può rendere necessaria l'assegnazione di un'importanza. Quello che si può fare è introdurre un parametro λ apprendibile dalla rete che permette di scalare le due loss.

Nota: Chiaramente si può usare transfer learning per allenare la CNN che ci restituisce l'output.

Ora, l'approccio appena descritto funziona bene con un solo oggetto da identificare ma questo **diventa problematico nel caso in cui ci siano più oggetti nell'immagine** principalmente perché lo stesso modello dovrebbe restituirci più output (quindi più bounding boxes degli oggetti da identificare con le relative etichette). La soluzione potrebbe essere quella di **applicare una CNN a varie parti (crop) dell'immagine** che ci permette di classificare ogni crop come un oggetto o come il background (nel caso in cui non venga identificato nessun oggetto). Questo approccio è problematico in quanto **dobbiamo considerare una quantità esagerata di posizioni, scale e proporzioni (aspect ratio)** e questo richiede di applicare la CNN su numerosi cropping dell'immagine (richiede troppa potenza computazionale).

Possiamo ridurre il numero di finestre possibili utilizzando un approccio di **Selective Search**. Selective Search è un algoritmo utilizzato per generare **proposte di regioni** (region proposals) in un'immagine. L'obiettivo di questa fase è utilizzare un insieme di strategie per individuare regioni dell'immagine che sono "**blobby**" cioè raggruppamenti di pixel simili in termini di colore, texture, forma e dimensione. Una volta individuate le regioni omogenee, questo genera una serie di proposte di regioni che potrebbero contenere oggetti e che vengono considerate come candidati per la successiva fase di classificazione e localizzazione. Una delle caratteristiche distintive di Selective Search è la sua relativa velocità di esecuzione. Può generare un gran numero di proposte di regioni, ad esempio, 2000 regioni, in un breve periodo di tempo, anche su una CPU. Questa velocità è essenziale per rendere praticabile l'object detection in tempo reale o in scenari in cui è necessaria una risposta rapida.

Come facciamo ad individuare questi raggruppamenti di pixel?

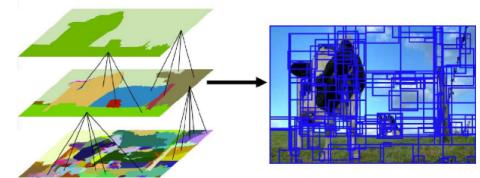
Quello che vorremmo ottenere è:

- **Recall alta:** Vogliamo fare in modo di non perderci alcun oggetto in quanto non può essere recuperato successivamente. La recall viene calcolata come la proporzione degli oggetti che sono coperti da qualche bounding box con un overlap maggiore di 0.5. Quindi una proposta può essere considerata solo se questa condizione di overlapping è valida.
- La posizione grossolana (coarse location) dell'oggetto è sufficiente per il riconoscimento (non è necessario ottenere una delineazione dell'oggetto accurata ma bastano degli bounding box);
- L'operazione di calcolo dei raggruppamenti deve essere veloce.

Ora, le immagini sono intrinsecamente gerarchiche cioè possono contenere oggetti a diverse scale quindi **la segmentazione a scala singola non è abbastanza** e si deve adottare un **raggruppamento gerarchico**

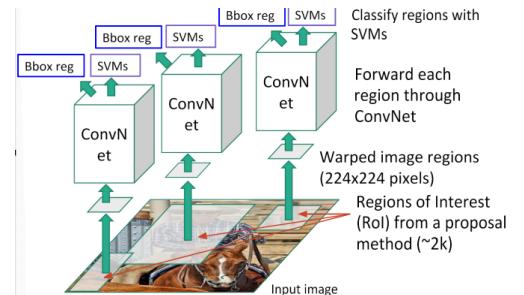
(hierarchical grouping) basato su colore. Quindi, l'approccio funziona in questa maniera:

- Over-Segmentazione: L'immagine di input viene suddivisa in segmenti omogenei basati su criteri come colore, texture, intensità, e altri attributi visivi. Questa fase produce regioni in cui i pixel sono simili tra loro.
- Raggruppamento gerarchico: I segmenti ottenuti vengono successivamente raggruppati gerarchicamente in segmenti più grandi. Questo processo tiene conto della **similarità tra i segmenti** (formula sulle slide) e li combina in raggruppamenti più ampi (fondendoli). L'idea è quella di catturare varie scale di dettaglio nell'immagine.
- Generazione di proposte: Le proposte risultanti dalla fase di fusione vengono ordinate in base a una metrica di diversificazione che tiene conto della varietà di contenuti al loro interno. In questo modo, l'algoritmo cerca di garantire che le proposte siano rappresentative delle varie parti dell'immagine.



R-CNN (Region-based Convolutional Neural Network) è un tipo di architettura di rete neurale convoluzionale utilizzata per l'object detection. La pipeline R-CNN include principalmente quattro fasi:

1. **Generazione delle proposte ed estrazione delle Regioni di interesse (RoI):** Vengono generate (circa 2k) proposte di regioni che sono candidate per contenere oggetti. **Sulla base delle proposte vengono estratte le Regioni di Interesse (RoI)** dall'immagine originale che vengono utilizzate come input per la successiva fase. Questi sono ritagli dell'immagine originale (ricavate dalla proposta) con una dimensione sulla base del modello CNN che andremo ad utilizzare (es. 224x224 per AlexNet).
2. **Estrazione delle Caratteristiche (Feature Extraction):** Ciascuna RoI estratta viene sottoposta a una rete neurale convoluzionale pre-addestrata (es. AlexNet). Questa rete è utilizzata per estrarre caratteristiche significative dalla regione di interesse.
3. **Classificazione e Regressione:** Le caratteristiche estratte vengono utilizzate per la **classificazione dell'oggetto presente nella RoI** (es. utilizzando un qualsiasi modello di ML come le SVMs) e per la **regressione delle coordinate della bounding box** che delimita l'oggetto (la predizione in questo caso è un oggetto (dx, dy, dw, dh) che ci rappresenta posizione e grandezza della bounding box). Quindi, la rete determina la classe dell'oggetto e raffina la sua posizione.



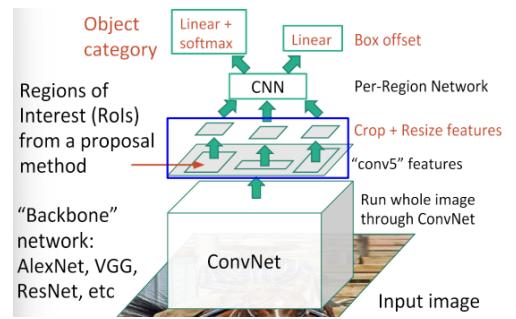
Purtroppo questo approccio è molto lento infatti abbiamo bisogno di 2k forward passes indipendenti per ciascuna RoI in ogni immagine. Ci sono state quindi ulteriori proposte come le **Fast R-CNN** dove l'idea è quella di **processare l'immagine prima di effettuare l'operazione di cropping delle RoI** (praticamente scambiamo la fase di convoluzione e cropping).

Nota: Data un'immagine 224x224, in AlexNet dopo i vari convolutional layer otteniamo una feature map che è 6x6 risultato dell'immagine originale (quindi c'è una corrispondenza tra i due). In particolare, ogni pixel nell'output 6x6 corrisponde a 16 pixel dell'immagine originale. Assumiamo di avere un'immagine originale più grande da cui estraiamo varie patch di dimensioni 224x224 pixel utilizzando una finestra scorrevole con uno spostamento di 16 pixel tra una patch e l'altra. Questo significa che la finestra scorrevole si sposterà di 16 pixel ogni volta che estraiamo una nuova patch. Se l'immagine originale è più grande della patch estratta, possiamo aspettarci una sovrapposizione tra le patch adiacenti. Supponiamo che l'immagine originale sia grande abbastanza da consentire l'estrazione di diverse patch senza uscire dai suoi confini. Quindi, per ciascuna di queste patch di dimensioni 224 x 224 pixel, possiamo applicare una rete neurale convoluzionale (come AlexNet) per ottenere una feature map di

dimensioni 6x6, con uno spostamento di 1 pixel tra i pixel adiacenti nella feature map. In altre parole, la finestra scorrevole con uno spostamento di 16 pixel dell'immagine originale dovrebbe generare patch sovrapposte, e applicare la rete neurale convoluzionale a ciascuna di queste patch produce feature map sovrapposte con uno spostamento di 1 pixel. Il vantaggio è che le computazioni svolte sulla parte sovrapposta vengono condivise con la convoluzione. Quindi anziché calcolare la convoluzione in ogni RoI, si può calcolare una volta per tutte in modo da condividere la computazione è l'output.

Quindi, introduciamo adesso le **Fast R-CNN**:

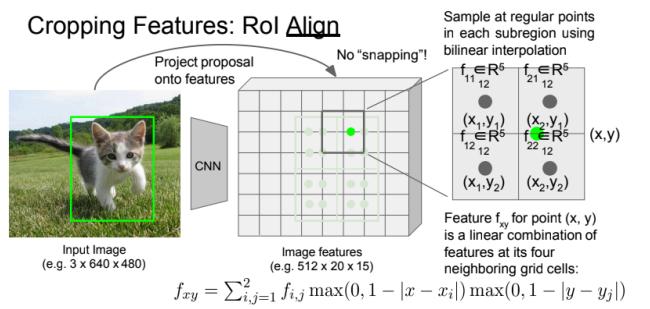
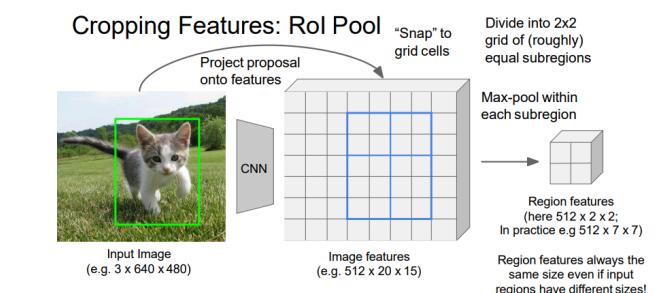
1. Prendiamo l'immagine originale e la **utilizziamo come input di una "backbone network"** che non è altro che una ConvNet (es. AlexNet, VGG, etc..) che ci **estrae la feature map** nonché le caratteristiche dell'immagine.
2. A partire dalla feature map calcolata, **si estraggono le regioni in interesse (RoI)** sulla base di metodo che ci ritorna delle proposte. E' qui che avviene il **cropping e il ridimensionamento delle regioni dell'immagine su cui identificare l'oggetto** (ricorda che facciamo resize perché la CNN che utilizziamo potrebbe richiedere dimensioni fissate per input).
3. Una volta ottenuta una regione (il cropping) dell'immagine di dimensione fissata, questa viene sottoposta a una CNN che esegue la classificazione dell'oggetto presente nella regione (utilizzando uno strato lineare seguito da softmax) e restituisce anche un box offset (utilizzando uno strato lineare) che indica la posizione della bounding box dell'oggetto all'interno della regione.



Adesso, andiamo ancora di più nel dettaglio di come avviene il **cropping delle features** introducendo l'operazione di **RoI Pooling**. L'idea è di "ritagliare" le feature map generate da una rete CNN in corrispondenza delle Regioni di Interesse (RoI) in modo che queste regioni siano rappresentate da feature map di dimensioni fisse. Diciamo che l'obiettivo è quello di ritagliare l'immagine facendo in modo di centrare l'oggetto il più possibile. L'input per il RoI Pooling è la feature map ottenuta dall'ultimo strato convoluzionale della rete di base. In particolare:

1. Le Regioni di Interesse (RoI) vengono mappate sulla feature map generata dalla rete CNN. Le RoI possono avere dimensioni diverse, ma il **RoI Pooling forza la loro rappresentazione in una griglia di celle** (questo mapping viene fatto mediante un'approssimazione).
2. **Ogni RoI viene suddivisa in una griglia 2x2 di sottoregioni approssimativamente uguali.** Questa divisione in sottoregioni consente una maggiore flessibilità nel catturare dettagli locali all'interno della RoI.
3. Per ciascuna delle sottoregioni 2x2, viene applicata **un'operazione di max-pooling**. In altre parole, il valore massimo all'interno di ciascuna sottoregione viene mantenuto, mentre gli altri valori vengono scartati.
4. Il risultato di questo processo è un set di valori, uno per ciascuna sottoregione 2x2, che rappresentano la RoI sulla feature map originale. Questi valori servono come **rappresentazione fissa della RoI, indipendentemente dalle sue dimensioni o posizione**. Questi valori sono spesso utilizzati come input per le fasi successive della rete neurale, come la classificazione e la regressione per l'object detection.

Ora, quando le dimensioni delle RoI non si allineano perfettamente con la griglia della feature map, può verificarsi questo **problema di allineamento impreciso delle features** (region features slightly misaligned). Per risolvere questo problema si può adottare il **RoI Align** anziché il max-pooling (RoI pooling). Quindi:



1. Inizialmente, le Regioni di Interesse vengono mappate sulla feature map ottenuta dall'ultimo strato convoluzionale della rete di base. A differenza del RoI Pooling, che suddivide la RoI in celle discrete, RoI Align suddivide ogni RoI in un certo numero di sottoregioni. Queste sottoregioni sono definite in termini di coordinate continue (e non discrete), il che consente un allineamento più flessibile. Per ottenere i valori esatti all'interno di ciascun bin, RoI Align utilizza l'**interpolazione bilineare** cioè viene usata interpolazione per calcolare i **valori basati sui pesi attribuiti ai quattro punti più vicini nell'immagine** (i quattro vicini nella griglia di celle).
2. In ciascun bin, l'operazione di pooling viene eseguita basandosi sui valori ottenuti dall'interpolazione bilineare. Ciò significa che ogni sottoregione contribuisce con una certa quantità di informazione alla rappresentazione finale, evitando il problema di perdita di dettagli durante l'approssimazione.
3. Il risultato finale del RoI Align è una rappresentazione dettagliata e allineata della RoI sulla feature map originale.

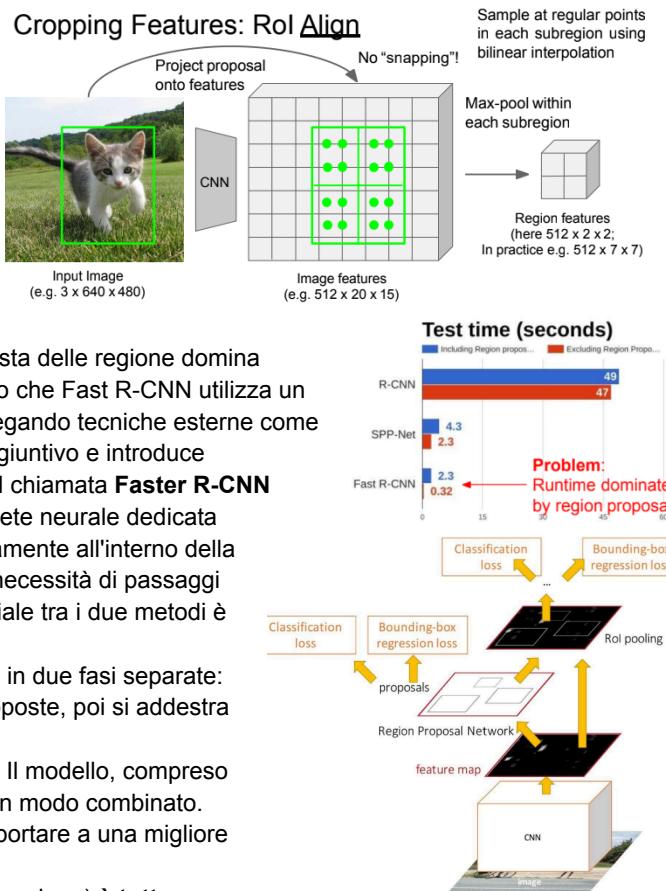
Ora, in R-CNN la parte di proposta delle regioni richiede un tempo minimo rispetto al tempo totale necessario per ottenere una predizione durante il test time. In Fast R-CNN riusciamo ad abbassare notevolmente il tempo necessario per ottenere una predizione ma qui, il tempo necessario ad ottenere la proposta delle regioni domina notevolmente rispetto al tempo totale. Il problema nasce dal fatto che Fast R-CNN utilizza un approccio separato per generare proposte di regioni (RoI), impiegando tecniche esterne come l'algoritmo Selective Search. Questo comporta un passaggio aggiuntivo e introduce complessità al sistema. Una versione successiva di Fast R-CNN chiamata **Faster R-CNN** introduce il concetto di **Region Proposal Network (RPN)**, una rete neurale dedicata che può generare automaticamente le proposte di regioni direttamente all'interno della rete. Questo rende l'intero processo più efficiente ed elimina la necessità di passaggi esterni per la generazione delle proposte. La differenza sostanziale tra i due metodi è la seguente:

- Con Fast R-CNN, l'addestramento del modello avviene in due fasi separate: prima si addestra la rete per la classificazione delle proposte, poi si addestra la rete finale per l'object detection.
- Faster R-CNN permette un apprendimento end-to-end. Il modello, compreso il RPN e il classificatore dell'oggetto, viene addestrato in modo combinato. Questo semplifica il processo di addestramento e può portare a una migliore convergenza del modello.

Per il resto (crop features per ogni proposta e la relativa classificazione) **è tutto uguale a Fast R-CNN**.

Quindi, più in dettaglio Faster R-CNN funziona in questa maniera:

1. Backbone Network: L'immagine di input viene passata attraverso una backbone network che non è altro che una rete neurale convoluzionale (CNN) pre addestrata come VGG o ResNet. Questa rete estrae le caratteristiche dell'immagine e produce una feature map.
2. Region Proposal Network (RPN): La feature map ottenuta dalla rete di base viene utilizzata come input per il RPN. Il **RPN è una rete neurale che analizza la feature map per generare proposte di regioni (RoI) che potrebbero contenere oggetti**. Il RPN utilizza convoluzioni per esaminare le caratteristiche dell'immagine e produce un insieme di proposte di RoI, ognuna associata a un punteggio di proposta che indica la probabilità di contenere un oggetto.
3. RoI Pooling o RoI Align: Le proposte di RoI generate dal RPN vengono utilizzate per estrarre regioni di interesse dalla feature map. Viene quindi applicato RoI Pooling grazie al quale ogni RoI viene suddivisa in celle di dimensioni fisse, e per ciascuna cella viene eseguita un'operazione di max-pooling. Successivamente si effettua anche RoI Align dove si utilizza l'interpolazione bilineare per ottenere valori esatti all'interno delle celle, migliorando la precisione rispetto a RoI Pooling.



4. Classificazione e Regressione degli Oggetti: Le regioni estratte vengono sottoposte a due percorsi distinti: Un classificatore per determinare la classe dell'oggetto contenuto nella RoI e un regressore per predire le coordinate della bounding box dell'oggetto. Il classificatore produce un punteggio per ciascuna classe possibile e questo permette di effettuare la predizione (es. softmax), mentre il regressore fornisce aggiustamenti alle coordinate della bounding box iniziale proposta dal RPN.

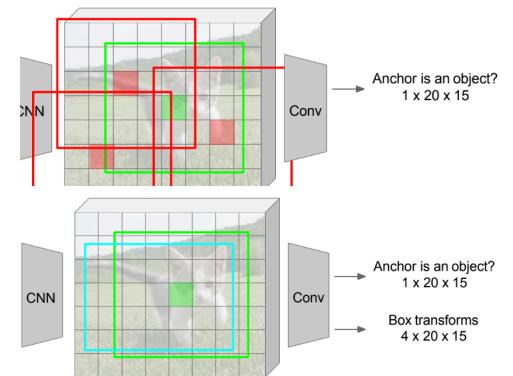
Nota: La prima fase (punti 1 e 2) vengono effettuati una volta per immagine mentre la seconda fase (punti 3 e 4) vengono effettuati una volta per regione calcolata.

Adesso andiamo nel dettaglio di come funziona la **Region Proposal Network (RPN)**:

- Un **anchor box** è una bounding box predefinita con dimensioni e rapporto d'aspetto specifici che viene centrata su un determinato pixel dell'immagine. L'anchor box viene posizionata in diverse posizioni sulla feature map per fungere da riferimento per potenziali oggetti. In ogni posizione spaziale sulla feature map, viene effettuata una previsione per determinare se l'anchor box centrata in quella posizione contiene un oggetto o meno (utilizzando un certo **object score**). Questa previsione viene spesso eseguita utilizzando la logistic regression per pixel, dove ogni pixel nella feature map corrisponde a una posizione nell'immagine di input.
ES. Assumendo che la feature map sia 512x20x15, ogni anchor box centrata in un pixel ha dimensioni 1x20x15 e ci dice se può contenere un oggetto o meno.
- Siano le **positive boxes** le anchor boxes che vengono classificate come contenenti un oggetto (foreground). Oltre a prevedere se contengono un oggetto o meno, si prevede anche una box transform che cerca calcolare i parametri che descrivono come l'anchor box dovrebbe essere trasformata per allinearsi meglio alla bounding box del ground-truth.
ES. Questa box transform ha dimensioni 4x20x15 dove 4 sono le coordinate (dw, dy, dz, dy) che ci descrivono la posizione e grandezza della box nell'immagine.
- Nella pratica, invece di avere una sola anchor box in ogni punto della feature map, **vengono utilizzate K diverse anchor boxes** Kx20x15 con dimensioni e rapporti d'aspetto differenti (ad esempio, potrebbero essere definite anchor boxes di diverse altezze e larghezze) sulla base delle quali vengono generate le proposte di regioni (RoI). Utilizzando diverse anchor boxes, il modello è in grado di adattarsi a oggetti di varie dimensioni e forme. Ad esempio, anchor boxes più grandi possono essere più adatte per oggetti più grandi nell'immagine, mentre anchor boxes più piccole sono adatte per oggetti più piccoli. Come prima possiamo calcolare le box transforms per allineare meglio di anchor boxes che in questo caso saranno 4Kx20x15.
- Tutti i possibili Kx20x15 boxes calcolati, si ordinano in base al object score e si prendono solo i primi ~300 che corrispondono a quello con la maggiore probabilità di contenere un oggetto.

Quindi **si effettua il training su 4 valori utilizzando 4 losses in maniera congiunta:**

- **RPN Classify Object/Not Object:** La prima loss si riferisce alla classificazione binaria dell'oggetto da parte della Region Proposal Network (RPN) in cui si predice se un'ancora contiene un oggetto o meno.
- **RPN Regress Box Coordinates:** La seconda loss riguarda la regressione delle previsioni della bounding box (box coordinates) nella RPN che migliora la precisione delle proposte di regioni (RoI) generando aggiustamenti alle coordinate delle anchor boxes in modo da allineare meglio con le bounding boxes reali degli oggetti.
- **Final Classification Score (Object Classes):** La terza loss riguarda la classificazione finale degli oggetti. Questo si verifica dopo che le regioni proposte sono state selezionate e sottoposte a RoI pooling o RoI align per ottenere le feature di una regione.
- **Final Box Coordinates:** La quarta loss è associata alla regressione finale delle coordinate della bounding box. Dopo la fase di classificazione, viene eseguita una regressione aggiuntiva per raffinare ulteriormente le coordinate della bounding box predetta, cercando di adattarla meglio alla posizione reale dell'oggetto.



Ora, è possibile fare un ulteriore step in avanti usando tecniche come YOLO (You Only Look Once), **SSD** (Single Shot Multibox Detector) e RetinaNet che **cercano di rilevare gli oggetti in un'unica fase** piuttosto che in più fasi:

- Sia l'immagine in input $3 \times H \times W$, l'immagine di input viene divisa in una griglia con celle, ad esempio, 7×7 . Ogni cella della griglia contiene un insieme di base boxes (anchor boxes) che sono centrati in quella cella. Il numero di base boxes è denotato da B (es. $B=3$).
- All'interno di ogni cella della griglia, il modello effettua previsioni per ogni **base boxes**. Per ogni base boxes, vengono previsti:
 - Cinque numeri per la regressione: $(dx, dy, dh, dw, confidence)$ dove dx e dy rappresentano gli spostamenti rispetto al centro della cella, dh e dw rappresentano l'altezza e la larghezza della bounding box rispetto alle dimensioni della cella e la Confidence indica la probabilità che la bounding box contenga un oggetto.
 - Punteggi predetti per ogni classe, inclusa la classe di background.

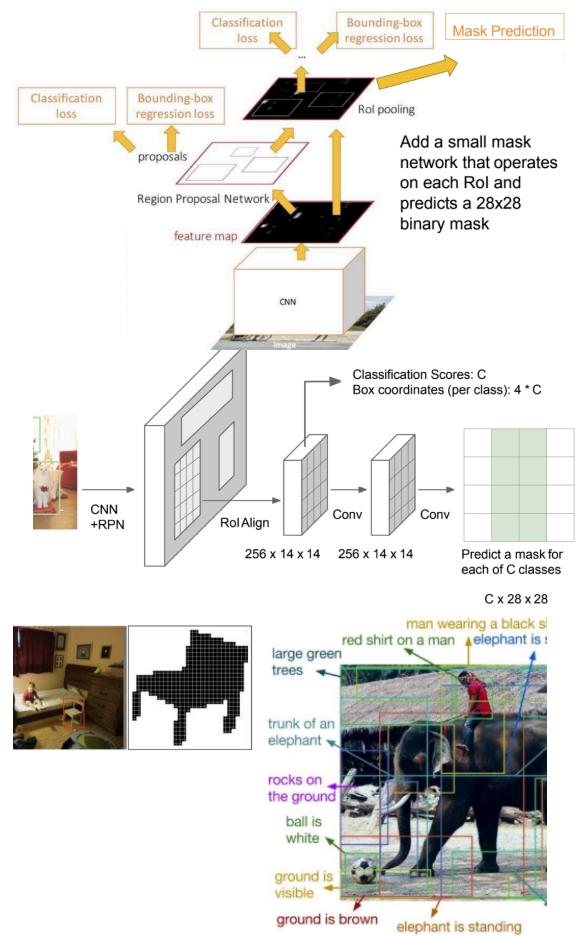
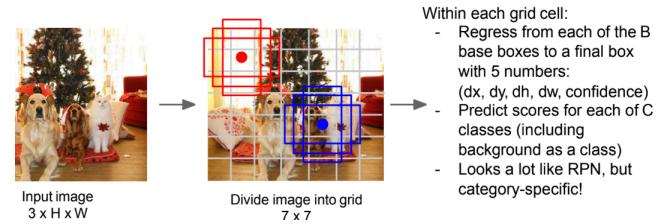
Nell'esempio precedente l'input sarebbe $7 \times 7 \times (5B+C)$.

Nota: SSD è molto più veloce di Faster R-CNN ma è meno accurato quindi occorre valutare se si preferisce accuratezza o tempo di esecuzione. Nella Faster R-CNN facciamo previsioni solo sulle regioni positive che dovrebbero contenere quasi certamente un oggetto. In SSD invece classifichiamo ogni cella della griglia. In generale, in scenari in cui alcune classi sono rappresentate da un numero limitato di esempi nel dataset, SSD potrebbe avere problemi a gestire queste classi meno frequenti. Questo potrebbe tendere a "sovraprendere" le classi più frequenti nel dataset di addestramento (es. background), a discapito delle classi meno frequenti. Ciò potrebbe comportare una mancanza di generalizzazione alle classi meno rappresentate quando il modello è esposto a nuovi dati.

6.3. Instance Segmentation

L'**instance segmentation** si occupa di riconoscere, separare e assegnare un'etichetta unica a ogni istanza individuale di oggetti all'interno di un'immagine. A differenza della semantic segmentation, che assegna una singola etichetta di classe a ciascun pixel nell'immagine, la **instance segmentation** si occupa di distinguere gli oggetti individuati, fornendo una maschera precisa per ciascuna istanza. Quindi non solo vogliamo assegnare una bounding box ad ogni istanza di oggetto come nel caso dell'object detection ma vogliamo proprio capire qual è la sua posizione precisa nell'immagine.

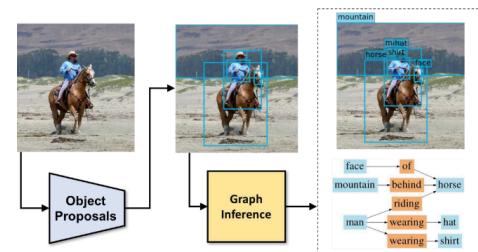
Mask R-CNN (Mask Region-based Convolutional Neural Network) è un modello che estende l'architettura di Faster R-CNN aggiungendo un componente per la **generazione di maschere pixel-per-pixel**. Quindi oltre ad effettuare class prediction and bounding box regression, dopo Roi Align, le feature map vengono inviate a un **mask head**, che è una rete neurale dedicata per la generazione di maschere pixel-per-pixel. Questa rete elabora le feature map della regione di interesse (Roi) e produce una maschera per ciascuna classe prevista (alla fine si prende solo la maschera della classe predetta dal classificatore). Per allenare la mask head a predire una maschera in maniera più accurata possibile, abbiamo bisogno del ground truth che non è altro che una maschera, quindi vogliamo ridurre la differenza tra le due maschere il più possibile. Ricorda che una maschera non è altro che un'immagine composta da soli 0 e 1 che ci dice dove si trova l'oggetto nell'immagine.



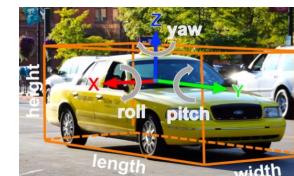
Nota: Possiamo usare Mask R-CNN anche pose estimation. L'idea è quella di calcolare una maschera diversa per ogni parte del corpo (joint). Quindi possiamo considerare ogni joint come fossero ciascuno un canale diverso e otteniamo la maschera per ogni canale utilizzando la mask head e queste ci determinano la posizione di ogni singolo joint nel corpo.

Finora abbiamo spiegato l'object detection di immagini 2D ma la ricerca ha permesso di andare oltre. Un esempio è la **Dense Captioning** che mira ad effettuare Object Detection e Captioning. Questa è una tecnica che si occupa della generazione di descrizioni testuali per regioni specifiche di un'immagine. A differenza dell'image captioning che produce una sola descrizione per l'intera immagine, la dense captioning fornisce descrizioni dettagliate per diverse regioni o oggetti all'interno dell'immagine. Questa prevede l'utilizzo di una backbone network che genera le features e a partire da queste viene applicato un localization layer che funge da Object Detector. Quindi una volta ottenute le RoI, si tenta di assegnare una descrizione alla regione in base alla relazione degli elementi al suo interno. Nel paper "DenseCap" del 2016 veniva utilizzato un LSTM per fare ciò ma oggi si potrebbe sostituire quell'architettura con, ad esempio, un trasformer. La stessa idea è poi stata applicata anche ai video (**Dense Video Captioning**) dove vari frame vengono processati per generare una didascalia di quello che sta accadendo nella sequenza.

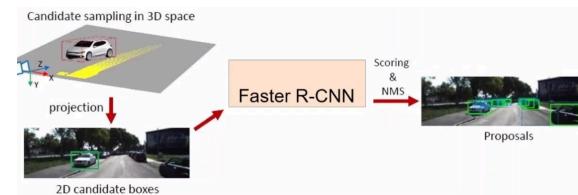
Scene graph prediction involves predicting the structure of a scene in terms of objects and their relationships. L'obiettivo è generare un scene graph, che è una rappresentazione basata su grafi degli oggetti in un'immagine e delle relazioni tra di essi, dove ogni nodo nel grafo corrisponde a un oggetto e ogni arco rappresenta una relazione tra gli oggetti. Quindi si identificano prima gli oggetti nella scena (nodi) attraverso una certa object proposal, poi si prevedono le relazioni tra di essi e questo permette di creare lo scene graph.



La **3D Object Detection** si occupa di individuare e localizzare oggetti tridimensionali in uno spazio tridimensionale. Mentre in un'immagine tridimensionale dobbiamo cercare di identificare l'oggetto utilizzando delle semplici bounding box (x, y, w, h) che ci specificano la posizione e la dimensione dell'oggetto nell'immagine, quindi dobbiamo considerare bounding box orientate (x: coordinata lungo l'asse orizzontale, y: coordinata lungo l'asse verticale, z: coordinata lungo l'asse della profondità, w: larghezza, h: altezza, l: lunghezza, r: roll, p: pitch, y: yaw) che considerano anche profondità e la modellazione tridimensionale. Nel caso della Object Detection possiamo considerare bounding box semplificate che non considerano roll e pitch.



Uno dei possibili modi per effettuare **3D Object Detection** prevede l'utilizzo di **Faster R-CNN** con l'unica differenza che oltre alle proposte di bounding box 2D vengono generate anche

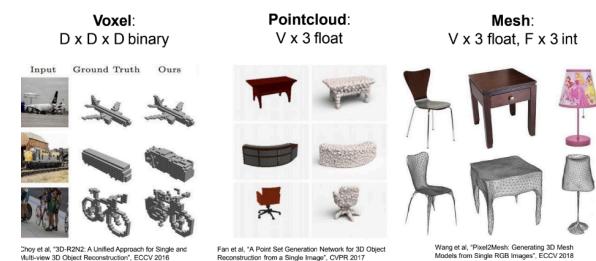


proposte di bounding box in 3D dove ogni proposta 3D include informazioni sulla posizione dell'oggetto nello spazio 3D (x, y, z), le dimensioni (lunghezza, larghezza, altezza), l'orientamento e un punteggio di classe. Quindi:

- La rete è addestrata per eseguire una regressione dei parametri di bounding box in 3D per ogni proposta candidata. Ciò comporta la predizione degli aggiustamenti necessari per la proposta iniziale di bounding box 3D per adattarsi meglio all'oggetto reale nella scena.
- Insieme alla regressione dei parametri di bounding box in 3D, il modello predice un punteggio di classe per ogni proposta, indicando la probabilità che la proposta appartenga a una particolare classe di oggetti (ad esempio, auto, pedone, ecc.).

Per quanto riguarda la **3D shape prediction** (previsione della forma di un oggetto), sono stati proposti vari approcci nel tempo:

- **Voxel:** Metodo che prevede l'utilizzo di quadrati per rappresentare l'oggetto tridimensionale. Quindi assumiamo che lo spazio 3D sia un volume Dx DxD binario, all'interno di questo volume possiamo marcare l'oggetto inserendo 1 nella mappa Dx DxD dove si trova l'oggetto, 0 altrimenti.



ES. Nel caso di un modello tridimensionale di un aereo. Possiamo marcare con 1 le regioni dello spazio in cui si trova l'aereo, 0 altrimenti.

Il problema di questo approccio è che richiede di codificare tutto il volume, anche quello in cui non troviamo l'oggetto. Vorremmo quindi poter eliminare gli 0 nel volume DxDxD.

- **Pointcloud:** Metodo che prevede di rappresentare l'oggetto mediante una mappa Vx3 che specifica l'uso di vertici ognuno del quali è rappresentato da coordinate tridimensionali. Nella pratica è come se rappresentiamo un oggetto mediante una serie di punti (ecco perché point cloud). Questo metodo permette di memorizzare solo l'informazione su qual è la forma dell'oggetto ma i punti non hanno volume (quindi ipotizzando che per identificare un oggetto usiamo un raggio di luce, questo passa attraverso l'oggetto, non c'è modo di capire quando questo raggio incontra l'oggetto in quanto i punti usati per rappresentarlo non hanno volume).
- **Mesh:** Metodo che prevede di rappresentare l'oggetto tramite una griglia (mesh) con la quale possiamo identificare anche l'orientazione e la curvatura dell'oggetto stesso. Quindi oltre ad avere l'insieme di punti Vx3 che ci definiscono la forma dell'oggetto, abbiamo anche una sorta di triangolazione Fx3 tra questi che ci definisce un volume cioè cerca di riempire lo spazio tra i punti delineando quella che è la superficie dell'oggetto. Quindi:
 - Vx3 float: "V" rappresenta il numero di vertici (vertices) nella mesh. "x3" indica che ciascun vertice è definito da tre coordinate float (floating-point) rappresentanti le sue coordinate spaziali tridimensionali (x, y, z).
 - Fx3 int: "F" rappresenta il numero di facce (faces) nella mesh. "x3" indica che ogni faccia è costituita da tre indici interi (int) che corrispondono ai vertici che la compongono.

7. Visual Search

Ipotizziamo di avere una certa immagine che rappresenta un certo oggetto, la **Visual Search** può utilizzare tali immagini come input per ottenere risultati pertinenti nonché per cercare l'oggetto all'interno delle altre immagini nel dataset.

Nota: Si noti che questa operazione non è necessariamente una classificazione. Se includiamo l'oggetto nel training set allora ipotizzando di avere diversi oggetti nel training set, durante il test possiamo dare nuovamente l'immagine dell'oggetto e il modello ci restituirà una certa classe. Quindi l'obiettivo è che il modello apprenda a riconoscere i tratti distintivi di ciascuna categoria e possa assegnare correttamente un'etichetta a nuove immagini non viste durante il training. La visual search è più orientata alla ricerca di somiglianze visive tra diverse immagini, consentendo agli utenti di trovare corrispondenze o informazioni simili a un'immagine di interesse. Quindi usiamo un algoritmo che, dato un certo training set, apprende le informazioni necessarie per trovare le corrispondenze visive tra più immagini (potenzialmente usando dataset diversi). Quindi **l'inferenza viene effettuata su oggetti mai visti** (altrimenti sarebbe una semplice classificazione).

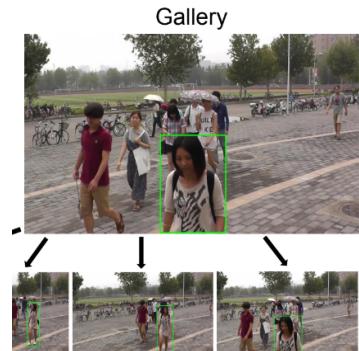
ES. Diamo un set di monumenti (senza il Colosseo) ed addestriamo un modello su questo set. Ora, prendiamo un nuovo dataset di monumenti e usiamo l'immagine del Colosseo come query usando l'algoritmo (già addestrato) per trovare le corrispondenze visive tra la query e le immagini nel nuovo dataset.

La visual search è spesso utilizzata in vari contesti, tra cui:

- E-commerce: Gli utenti possono cercare prodotti online caricando un'immagine del prodotto desiderato, consentendo loro di trovare articoli simili o identici.
- Ricerca di informazioni: Gli utenti possono cercare informazioni su oggetti sconosciuti o luoghi semplicemente fornendo un'immagine (es. data un'immagine del monumento otteniamo informazioni e posizione).
- Identificazione di oggetti: La visual search può essere utilizzata per identificare oggetti, animali, piante o altre entità presenti in un'immagine (es. fotografando l'immagine di un libro, otteniamo sue informazioni).

Ora, bisogna tenere in considerazione il fatto che comprendere il significato di un'immagine spesso dipende dal contesto o dall'applicazione quindi **potremmo avere diverse**

interpretazioni. L'interpretazione di una query spesso dipende dall'applicazione specifica o dal contesto in cui viene utilizzata difatti la stessa query potrebbe avere significati diversi.



Per affrontare l'**ambiguità intrinseca** (inherent ambiguity), i sistemi possono essere progettati per inserire informazioni precedenti o sfruttare il training. Ciò implica l'incorporazione di conoscenze acquisite da dati precedenti o training set per migliorare la capacità del modello di comprendere e interpretare le query.

Per quanto riguarda le persone, la Visual Search tipicamente si applica al **trovare una cerca da una certa galleria di immagini (rilevamento + re-identificazione)**. Quindi, ipotizzando abbiamo varie immagini di una certa persona nella galleria, possiamo ipotizzare di avere un video nel quale la persona appare e da qui possiamo re-identificarla. Nel flusso video chiaramente dobbiamo prima rilevare tutte le persone e poi possiamo procedere alla re-identificazione di quella che ci interessa. Nota che la differenza tra identificazione e re-identificazione sta nel fatto che l'identificazione implica che la persona sia già stata vista è inclusa nel dataset, effettuiamo una semplice classificazione), nella re-identificazione non è vista prima.



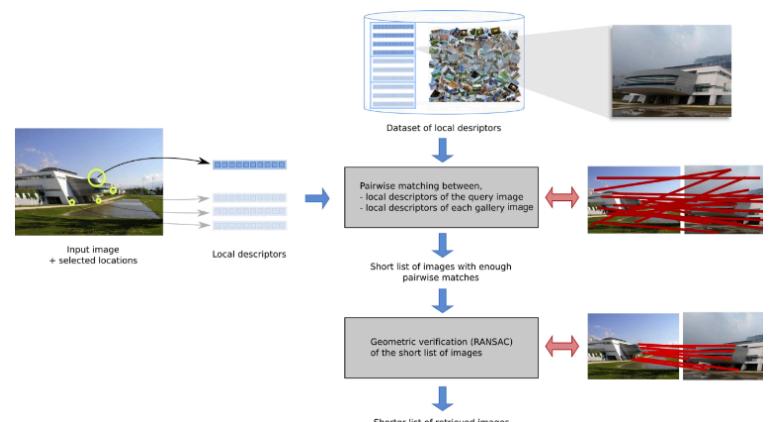
Nota: Potremmo pensare di applicare multitask quindi allenare un modello che ci permetta di effettuare sia il task di rilevazione che quello di re-identificazione. In realtà si noti che questi task molto diversi infatti il primo ha come scopo quello di identificare feature generiche per la rilevazioni di persone generiche mentre il secondo richiede di identificare una persona specifica. Si lavora sempre sulle persone ma i parametri condivisi potrebbero non essere molti quindi potrebbe essere migliore allenarli separatamente.

7.1. Object Search

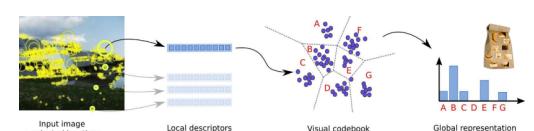
La **Object Search** si concentra sull'individuazione e sul recupero di oggetti specifici all'interno di un ambiente o di un insieme di dati. Con il tempo sono stati sviluppati vari metodi per effettuare questo tipo di ricerca:

- Early methods: I primi metodi prevedevano di trovare una sorta di descrittore globale delle immagini attraverso, ad esempio, l'uso di color histograms. Una volta trovato l'istogramma delle immagini da confrontare si usava una certa misura di somiglianza per capire quanto queste fossero uguali. Il problema di questo approccio è quando dovevamo riconoscere molti oggetti diversi tra loro ma con un color histogram simile. Inoltre questo approccio era fortemente influenzato da problemi come l'occlusione, l'illuminazione e la scala che modificavano fortemente il risultato dell'istogramma.

- Local representations (**matching-based methods**): Un altro approccio prevedeva l'utilizzo delle **local representations** cioè punti nell'immagine che permettono di rappresentare facilmente porzioni specifiche o regioni di un'immagine. Quindi si cerca di catturare dettagli locali piuttosto che l'immagine per intero. Per effettuare questa operazione si usavano dei **local descriptors** cioè degli algoritmi progettati per essere **invarianti** rispetto a determinate trasformazioni (come rotazioni, scalature o illuminazione) e allo stesso tempo **discriminativi**, consentendo di distinguere tra diverse regioni (i punti selezionati saranno diversi tra di loro).



La selezione delle posizioni in cui calcolare i descrittori è importante (**repeatability**) ed esistono vari metodi applicabili (es. Harris, Hessian, ecc...). Ora, come facciamo ad applicare questo approccio: ipotizziamo di avere una certa immagine, estraiamo una serie di local descriptors e li andiamo a confrontare con quelli di una galleria di immagini. Andiamo quindi ad estrarre la lista di immagini più simili ed applichiamo una **verificazione geometrica (RANSAC)** su questo set di immagini che ci consente di estrarre solamente i match che sono consistenti in seguito ad una certa trasformazione geometrica come la rotazione e la traslazione (es. gli oggetti rigidi come il palazzo non si possono deformare) quindi otteniamo solo i local descriptors ottimali (**inliers**) e scartiamo gli altri (**outliers**). Sulla



base di questi scartiamo ulteriormente altre immagini (quelle con meno inliers). Il problema di questo approccio è che il numero di stime per ogni coppia di immagini e le trasformazioni geometriche da applicare richiedono troppo tempo.

- **Global representations (aggregation methods):** Inizialmente, vengono estratti local descriptors da diverse regioni di un'immagine che vengono quindi convertiti in "parole visive" attraverso l'uso di un "visual codebook" (dizionario visivo). Ogni immagine viene rappresentata come un istogramma che conta quante volte ciascuna "parola visiva" (o codice) appare nei local descriptors estratti dall'immagine. In altre parole, l'istogramma rappresenta la distribuzione delle parole visive nell'immagine.

Nota: Prima di procedere agli approcci più recenti, un dataset usato per l'object search è l'Oxford dataset composto da circa 5k immagini. L'approccio usato per la performance evaluation è la Mean Average Precision (mAP).

SUM. Quindi, i metodi basati sulle local representations (**matching-based methods**) hanno un'accuratezza maggiore ma richiedono un alto costo computazionale per effettuare tutti i matching e le operazioni di verifica geometrica. I metodi basati sulle global representations (**aggregation methods**) sono molto più veloci ma restituiscono anche un'accuratezza minore.

Vediamo adesso un approccio basato su deep learning. I primi approcci di deep learning coinvolgevano il **pre addestramento di reti per compiti di classificazione generale** e l'utilizzo di tali reti come estrattori di caratteristiche. Questo permetteva di ottenere rappresentazioni compatte grazie all'uso delle features map ed inoltre permetteva di effettuare una previsione in maniera veloce durante la fase di test. Questo approccio però prevede alcune limitazioni:

- **Generalizzazione intra-classe:** Le reti neurali pre addestrate per classificare oggetti in categorie generiche potrebbero non essere ottimali per compiti di object search specifici in cui è richiesta una comprensione più fine delle caratteristiche degli oggetti.
- **Bassa risoluzione e rapporto di aspetto distorto:** Le immagini utilizzate per addestrare le reti neurali iniziali potrebbero avere bassa risoluzione o aspect ratios distorti, il che potrebbe influire negativamente sulla qualità delle rappresentazioni apprese.
- In generale, utilizzare direttamente le rappresentazioni apprese da reti neurali pre addestrate come caratteristiche per compiti specifici potrebbe portare a risultati deludenti.

Sono stati quindi definiti nuovi approcci per cercare di superare queste limitazioni e migliorare i risultati. Ora, ipotizziamo di partire da un certo dataset di monumenti famosi composto da circa 200k immagini rappresentanti circa 600 attrazioni. Sono state applicate tecniche di pulizia a questo dataset e il risultato ha portato il dataset ad avere 40k immagini con annotazioni sul bounding box che ci specifica la posizione del monumento nell'immagine.

Una possibile architettura è il **R-MAC Descriptor** (Regional Maximum Activation of Convolutions). Funziona in questo modo:

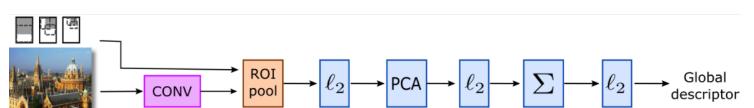
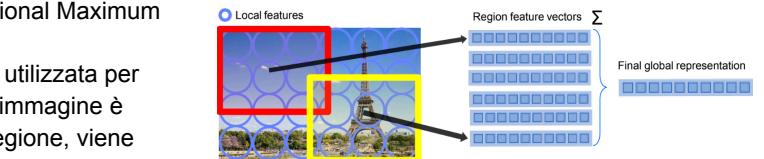
- Una rete neurale convoluzionale (CNN) viene utilizzata per estrarre caratteristiche locali dall'immagine. L'immagine è suddivisa in regioni sovrapposte, e per ogni regione, viene eseguito il max pooling sulle caratteristiche estratte. Il max pooling cattura la massima attivazione presente in ciascuna regione, enfatizzando le parti salienti dell'immagine. Alle rappresentazioni risultanti da ciascuna regione viene quindi applicata una L2 normalization. Questo passo è utile per garantire che le rappresentazioni siano indipendenti dalla scala e facilitare la comparazione tra diverse immagini.
- Le rappresentazioni normalizzate delle diverse regioni dell'immagine vengono concatenate per formare la rappresentazione finale del R-MAC descriptor.

Alcuni vantaggi di questo approccio sono:

- Nessuna distorsione del rapporto di aspetto.
- Questo approccio è in grado di codificare immagini ad alta risoluzione.
- Velocità di confronto rapida grazie all'uso del prodotto scalare.

Alcune osservazioni:

- **Le fasi di aggregazione**, come il max pooling regionale e la normalizzazione L2,



possono essere integrate direttamente all'interno della struttura della CNN. Questo significa che il processo di aggregazione può essere considerato come parte del flusso di apprendimento durante l'addestramento della rete.

- Ogni passo del processo, incluso l'aggregazione, è **differenziabile**.

In sintesi, possiamo progettare la rete in modo da eseguire automaticamente le operazioni di aggregazione durante il forward pass e che l'intero modello può essere addestrato in maniera congiunta consentendo così di apprendere efficientemente un descrittore R-MAC per compiti specifici. Questo ci consente di mantenere i dettagli delle immagini.

Il **training per il recupero delle informazioni (retrieval)** prevede l'apprendimento di un modello che può classificare efficacemente **l'importanza relativa delle immagini rispetto a una query**. Questo approccio è stato definito nel paper "Learning to rank" che è stato via via espanso. In generale, l'obiettivo è consentire ai modelli di classificare automaticamente le immagini in base alla loro rilevanza per una determinata query calcolando una **ranking loss**. Per addestrare un modello in questo modo viene utilizzata una **3-stream Siamese network**:

- La struttura base di una rete siamese coinvolge due rami identici (o "gemelli") che condividono gli stessi pesi e architettura. Nella variante "3-stream", vengono utilizzati tre rami distinti invece di due. Ne usiamo 3 perché in questo caso teniamo in considerazione la query, un'immagine rilevante che ci consente di identificare l'oggetto e un'immagine non rilevante.
- Elaboriamo le tre immagini utilizzando la stessa architettura e pesi e alla fine generiamo una **triplet loss** calcolata in questa maniera:

$$L_v(q, d^+, d^-) = \frac{1}{2} \max(0, m + \|q - d^+\|^2 - \|q - d^-\|^2)$$

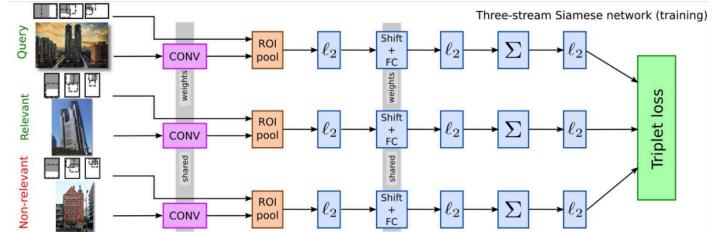
Dove q è la query mentre d^+ e d^- sono rispettivamente immagine rilevante e non rilevante mentre m è un margine nonché una costante che stabilisce la differenza minima desiderata tra la query e l'immagine rilevante e la query e l'immagine non rilevante. Sarà questa la loss utilizzata per l'addestramento del modello.

SUM. Di seguito un riassunto di quello visto finora:

- **Features ottenute da ImageNet non sono buone per la discriminazione intra-classe:** Le caratteristiche estratte da ImageNet potrebbero non essere ottimali per la discriminazione fine all'interno delle classi, specialmente in contesti specifici come il riconoscimento di luoghi. La soluzione a questo problema potrebbe essere quella di addestrare il modello su un dataset di luoghi pulito automaticamente per garantire la qualità delle annotazioni.
- **Architetture tradizionali lavorano su piccoli ritagli dell'immagine e a bassa risoluzione:** Le architetture convenzionali potrebbero operare su regioni ridotte delle immagini a risoluzioni più basse, limitando la capacità di preservare dettagli importanti. Dobbiamo quindi scegliere opportune architetture che **preservano i dettagli dell'immagine**.
- **Reti solitamente addestrate per la classificazione:** Le reti neurali sono spesso addestrate per compiti di classificazione generale, il che potrebbe non essere ottimale per applicazioni specifiche come l'object search. Una soluzione potrebbe essere quella di adottare una **ranking loss** che prevede di ordinare correttamente gli esempi in base alla loro rilevanza.

7.2. Person Search

Vediamo finalmente come applicare quello visto in precedenza per la ricerca delle persone. La **person re-identification** consiste nel riconoscere e identificare la stessa persona in diverse immagini acquisite da telecamere diverse. La person re-identification è formulata come un **compito di recupero**, in cui l'obiettivo è recuperare correttamente l'immagine della stessa persona tra diverse telecamere. Questo richiede la capacità di riconoscere le caratteristiche distintive di una persona indipendentemente da variabili come illuminazione,



angolazione e condizioni ambientali. Si assume che le **immagini utilizzate per la person re-identification siano ritagliate da un rilevatore di persone**. Ciò significa che l'input del sistema è costituito da immagini che isolano chiaramente la figura umana, facilitando così il riconoscimento della persona.

L'algoritmo deve essere in grado di **identificare correttamente persone che non sono state osservate durante la fase di training**, basandosi su caratteristiche comuni o distintive. Per fare ciò, il set di identificatori (ID) di addestramento deve essere diverso da quello di test.

Per effettuare la re-identificazione si può usare il Siamese framework visto in precedenza utilizzando un approccio del genere:

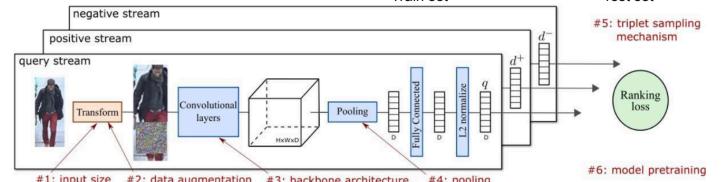
- Data una certa immagine in input (#1) (**possibilmente senza distorsioni**) si effettua una trasformazione (#2) dell'immagine in maniera da aumentare la difficoltà e la robustezza durante il training. Si pensare ad esempio di applicare **data augmentation** con **cut-out** dell'immagine con dimensioni crescenti;
- A questo punto, si passa l'immagine trasformata ad una CNN per estrarre una feature map (#3). Si può usare una **backbone network pre-addestrata** (es. su ImageNet) (#6);
- Si applica pooling (#4) (tipicamente max-pooling), dopodiché di applica uno strato lineare per la classificazione e all'output viene applicata la L2 normalization;
- Applichiamo la Siamese framework, quindi eseguiamo la stessa operazione su tre immagini (query, positive stream e negative stream) (#5) utilizzando gli stessi pesi e architettura e otteniamo una Ranking loss finale che contribuisce a migliorare la capacità del modello di discriminare tra diverse identità, rendendo più facile il processo di recupero delle persone.

Quali regioni contribuiscono maggiormente per effettuare il matching tra coppie di immagini?

Grad-CAM (Gradient-weighted Class Activation Mapping), una tecnica utilizzata per visualizzare quali regioni di un'immagine contribuiscono maggiormente all'attivazione di determinate feature. Nel contesto delle coppie di immagini accoppiate, Grad-CAM può essere utilizzato per **individuare le regioni specifiche di ciascuna immagine che contribuiscono maggiormente alla somiglianza tra le due immagini**. Grad-CAM può essere applicato a ciascuna dimensione, evidenziando le regioni delle immagini che sono rilevanti per l'attivazione di quella dimensione specifica. Questo può fornire una comprensione più dettagliata di quali aspetti visivi contribuiscono maggiormente alla misura di similarità. Ad esempio, per coppie di immagini possiamo selezionare le **top-5 dimensioni** che contribuiscono maggiormente alla somiglianza tra le immagini.

Con **Person Search** intendiamo **Rilevazione + Re-Identificazione**. Quindi data una certa immagine, l'obiettivo è identificare le persone al suo interno e capire se tra queste persone c'è quella che stiamo cercando (la query). Ci sono vari approcci che possiamo utilizzare:

- **Approccio base**: Ho un'immagine con delle persone, la elaboro con Faster R-CNN per estrarre le ROI con le persone. Data una query (la persona che sto cercando) la passo ad un RE-ID network che si occupa di confrontare immagine della galleria e query per identificare la persona.
- **OIM**: La query e l'immagine vengono passate a una Faster R-CNN. Quindi le immagini vengono passate a una backbone network che le elabora. Da qui si usa RPN e per ogni ROI otteniamo le due losses (object/not object e regression losses) che ci permettono di correggere le predizioni sulla region proposal. Applichiamo ROI align sulle ROI e le passiamo ad un classification layer che ci effettua le predizioni finali (classificazione della



#6: model pretraining

#5: triplet sampling mechanism

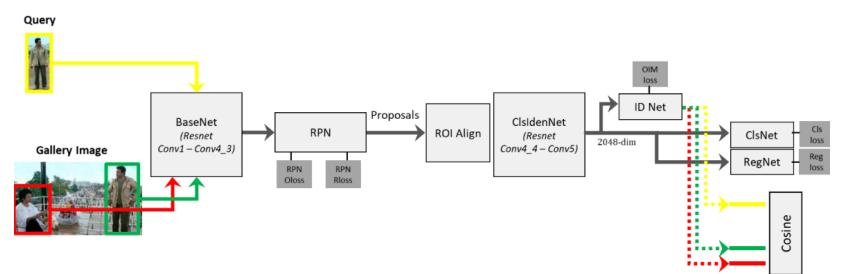
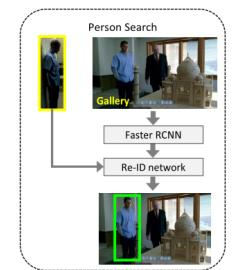
#4: pooling

#3: backbone architecture

#2: data augmentation

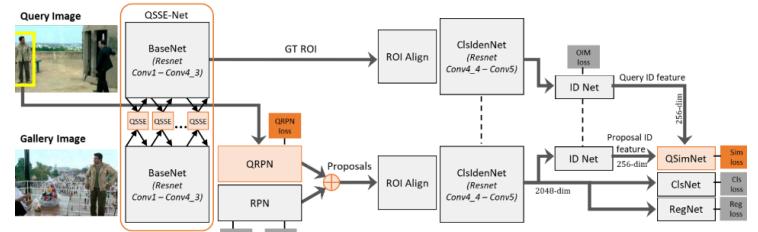
#1: input size

negative stream
positive stream
query stream



classe, in questo caso se la ROI contiene o meno una persona e regressione della bounding box). In più, il risultato della classificazione lo passiamo ad un **identification network** che ci restituisce un ID per le persone nella scena (sia l'immagine della galleria e la query). ID simili (questa somiglianza ci è data, ad esempio, da una **cosine distance**) significano che molto probabilità c'è una somiglianza maggiore. Chiaramente una volta ottenuti gli ID li possiamo salvare in una tabella velocemente per effettuare una **online instance matching**.

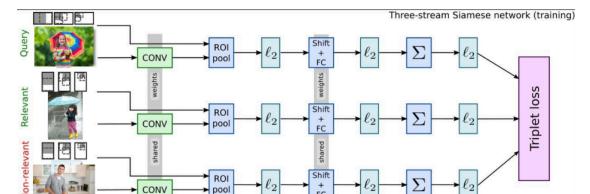
- **QEEPS** (Query-guided End-to-End Person Search): In sintesi, l'obiettivo è utilizzare l'intera immagine con la persona da identificare anziché il solo cropping in maniera da usare anche le informazioni nella scena. Oltre ad usare Faster R-CNN si utilizza una Siamese Faster R-CNN dove l'intera architettura ci ritorna tre valori: QSimNet (apprendimento metrico delle distanze dei vettori re-id), QRPN (Proposte di persone specifiche per query) e QSSE (Guida del contesto globale alla rete di base).



7.3. Semantic Search

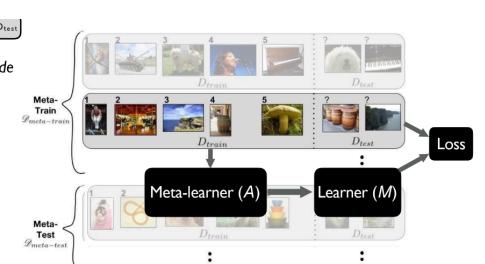
La **Semantic Search** si riferisce a un tipo di ricerca che tiene conto del significato e del contesto delle parole chiave utilizzate, oltre alla loro corrispondenza letterale. In altre parole, la ricerca semantica mira a comprendere l'intento dell'utente dietro una determinata query e a restituire risultati più pertinenti basati sulla comprensione del significato semantico delle parole. Quindi, data un'immagine di "padre e figlia che giocano a pallone" vogliamo restituire da una certa galleria di immagini quelle che sono **semanticamente simili**, cioè i risultati pertinenti. Ora data una certa immagine, potremmo ricevere due immagini pertinenti in output ma non sappiamo quale è realmente la più pertinente (es. un uomo che cammina con un ombrello ci restituisce una persona che cammina sulla strada e una persona con un ombrello in un campo ma quale delle due è più pertinente?). Possiamo usare delle **annotazioni umane** per indicare qual è la preferenza delle persone e **rimuovere questa ambiguità**. Si chiede quindi di annotare le immagini e **usare le annotazioni per il training** (sulla base di 3 immagini - query e due possibili immagini come nell'esempio a destra). Durante **l'inferenza si toglie il punteggio di uno degli annotatori dall'agreement score**. Ora, fare questa operazione con gli umani è estremamente dispendioso quindi sono stati creati dei metodi per generare delle annotazioni delle immagini in maniera automatica come **tf-idf**. Quindi si possono creare delle **didascalie (caption)** che descrivono l'immagine e si vanno ad estrarre due immagini (oltre la query). La prima immagine è un'immagine con un embedding semantico simile (esempio positivo), la seconda è un'immagine con un embedding semantico differente (esempio negativo). Questo permette di creare un "**semantic embedding space**". Possiamo quindi usare una 3-stream siamese network che calcola una ranking loss:

$$L_v(q, d^+, d^-) = \frac{1}{2} \max(0, m - (\phi_q^T \phi_+ + \phi_q^T \phi_-))$$



7.4. Meta-Learning

Il **Meta Learning** si concentra sull'abilità di un modello di **apprendere come apprendere**. In altre parole, il meta learning riguarda la capacità di un algoritmo di adattarsi e **migliorare le sue prestazioni su nuovi compiti, basandosi sull'esperienza acquisita da compiti di apprendimento precedenti**. Il **few-shot learning** riguarda l'addestramento di un modello per eseguire bene compiti per i quali ci sono pochissimi dati etichettati. Possiamo affrontare questo problema utilizzando il meta learning:



- **Meta training:** Un algoritmo A viene addestrato su una varietà di compiti, ognuno con il proprio piccolo set di dati. Per ogni compito di meta-training l'algoritmo A prende questo piccolo set di dati e produce parametri θ di un modello M che si comportano bene su quel compito.
- **Meta testing:** Si valutano le prestazioni dell'algoritmo appreso A su nuovi compiti (compiti di meta-testing) per i quali c'è un dataset limitato.

L'intero processo è end-to-end perché l'algoritmo A viene appreso in modo che ottimizzi la sua capacità di adattarsi rapidamente ed efficacemente a nuovi compiti con piccoli set di dati.

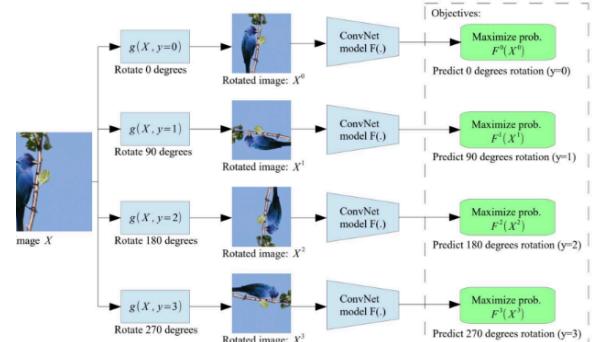
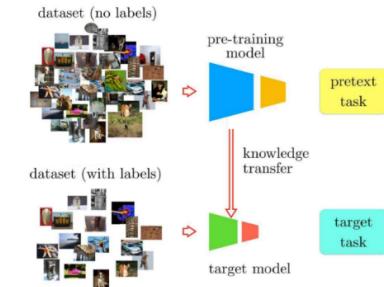
8. Self Supervised Learning

Supervised learning è l'approccio tipicamente adottato nel deep learning. Questo prevede l'utilizzo di etichette (label) che possono essere utilizzate come ground-truth per l'allenamento di un modello. Purtroppo questo approccio prevede alcune limitazioni:

- **Dati annotati limitati ed eccessivamente costosi:** Gli algoritmi di deep learning spesso richiedono una grande quantità di dati annotati per un addestramento efficace e ottenere questi dati può essere difficile e costoso (soprattutto se effettuato da un essere umano). La tendenza all'uso di architetture profonde e ampie amplifica ulteriormente la necessità di una quantità considerevole di dati. Ultimamente gli sforzi sono distribuiti nella creazione di modelli pre addestrati ampi e generali per svariate attività (ad esempio, il modello Transformer nell'NLP) contribuendo alla richiesta di dataset diversificati ed estesi.
- **Collaudo della Supervisione:** Durante l'addestramento, le reti neurali possono concentrarsi eccessivamente su specifici pattern delle immagini che distinguono ciascuna classe, portando a rappresentazioni fortemente raggruppate nel feature space. Questo focus può causare il cosiddetto **collaudo della supervisione** (supervision collapse), dove la rete potrebbe trascurare altre similarità tra le immagini. Il problema nasce dal fatto che **per campioni al di fuori del training set**, la rete potrebbe **enfatizzare pattern spurii** provenienti dal training set stesso, che potrebbero non essere indicatori affidabili dell'appartenenza a una classe e questo collasso può ostacolare la capacità del modello di fare associazioni corrette.

Quindi, è possibile effettuare il training di un modello su un set possibilmente infinito di dati e per ogni possibile task? E' qui che viene in aiuto il **Self Supervised Learning (SSL)** dove l'idea generale è produrre automaticamente alcune etichette per risolvere dei task generici e poi trasferire la conoscenza acquisita su un task secondario (in cui abbiamo le etichette) per la sua valutazione. In particolare:

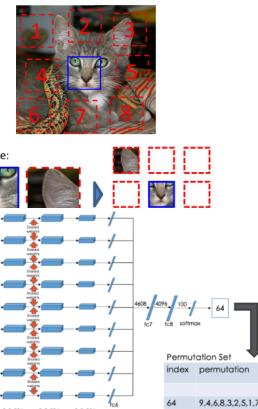
1. **Doppio Addestramento (Pretext e Downstream):**
 - Compito di Pretesto (Pretext Task):** Si tratta di un compito molto facile, senza applicazioni dirette al tuo obiettivo finale, ma che consente di apprendere qualcosa di molto generale. L'idea è che il modello affronti un compito che non richieda etichette target specifiche per la tua applicazione finale, ma che comunque insegni al modello rappresentazioni utili e generalizzabili.
 - Compito Principale (Downstream Task):** Questo è l'obiettivo finale. Per questo compito, le etichette target specifiche (ad esempio, classi ID per la classificazione delle immagini o descrizioni di immagini per il sottotitolaggio) di solito non sono disponibili in quantità sufficienti su larga scala. In sintesi, possiamo effettuare fine tuning sul modello generale addestrato su una quantità di dati generici, sul nostro modello target che verrà addestrato su un set di dati etichetti molto più piccolo e, soprattutto, specifici per il nostro problema.
2. **Etichette Relative al Compito da Risolvere:** Quando si affrontano compiti specifici si ha bisogno di etichette direttamente collegate al nostro obiettivo. Per addestrare un modello generico, è necessario introdurre un **compito generico** che non richieda etichette specifiche del tuo compito finale cioè deve utilizzare **etichette "libere" o generiche**, che non siano legate direttamente all'applicazione finale. La scelta del pretext task



dipende da come è possibile ottenere etichette "libere" o generiche in modo efficace.

Quali sono i possibili pretext task?

- **Rotazione:** Si effettua una trasformazione o rimozione di qualcosa dall'input (ad esempio, l'orientamento di un'immagine) e si lascia che il modello cerchi di 'ricostruire' o predire ciò che è stato rimosso o trasformato. Quello che si fa è **rimuovere l'orientamento dell'immagine e il modello deve imparare a prevedere l'angolo di rotazione corretto**. Si presume che questo approccio possa aiutare ad estrarre una conoscenza generale dell'immagine poiché si cerca di creare un modello possa riconoscere correttamente la rotazione di un oggetto grazie all'uso di un "**buon senso visivo**" (**visual commonsense**) di come l'oggetto dovrebbe apparire senza perturbazioni.
ES. Data un'immagine di un cane ruotata di 270°, il visual commonsense si riferisce al fatto che tipicamente un cane non si trova a quella rotazione ma a 0°.
Il modello di Rotation Classification, in breve, funziona in questo modo: data un'immagine, si ruota per un certo numero di gradi (es. 80°, 180°, ..) e, ad ogni immagine risultante, viene associato l'angolo di rotazione applicato come etichetta. Durante l'addestramento, il modello regola i suoi pesi in modo che possa fare previsioni accurate sull'angolo di rotazione. Addestrando il modello a prevedere l'angolo di rotazione, si costringe il modello a **imparare caratteristiche che sono invarianti alla rotazione**. Il significato semantico dell'immagine rimane lo stesso anche quando cambia l'orientamento, e il modello, per eseguire con successo la previsione della rotazione, deve concentrarsi sulle caratteristiche che sono stabili attraverso diverse orientazioni.
- **Patch Location:** Vengono estratte diverse patch dall'immagine, ma **l'informazione sull'ordine relativo delle patch viene rimossa**. Il modello deve imparare a **prevedere la posizione relativa di ciascuna patch rispetto alle altre**. Durante l'addestramento, il modello regola i suoi pesi in modo che possa fare previsioni accurate sulla posizione relativa delle patch nell'immagine quindi data un'immagine in input con patch in posizioni casuali, prevede con successo la disposizione relativa di tali patch.
- **Jigsaw Puzzles:** L'immagine originale viene suddivisa in patch, e **l'informazione sull'ordine specifico di queste patch viene alterata o rimossa**. Il modello deve imparare a **prevedere la disposizione corretta delle patch nell'immagine**, anche quando l'ordine originale è stato alterato. Il modello deve imparare a prevedere la disposizione corretta delle patch nell'immagine e quindi, data un'immagine in input con patch disposte in ordine casuale, questo deve prevedere la disposizione originale delle patch.

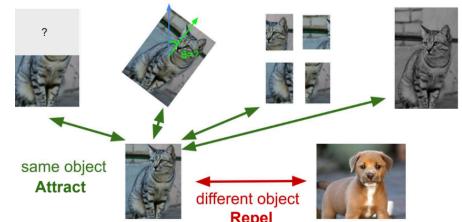


Ci sono altri metodi come l'**Inpainting** in cui rimuoviamo pixel dall'immagine e il modo deve apprendere a ricostruire l'immagine, oppure il Colorizing in cui si rimuovono i colori e l'obiettivo è colorare nuovamente l'immagine. In generale, l'obiettivo è quello di **trasformare o rimuovere qualcosa dall'input e lasciare che il modello cerchi di 'ricostruire' o prevedere ciò che è stato rimosso o trasformato**. Nota che in generale, non ci interessa la performance specifica di questi compiti di pretesto, ma piuttosto quanto siano utili le caratteristiche apprese per compiti successivi (classificazione, rilevamento, segmentazione).

Introduciamo adesso un altro pretext task che ha portato alla definizione vera e propria di Self Supervised Learning che è il **Contrastive Learning**. Ci sono dei problemi con i metodi visti in precedenza. In generale:

- Creare pretext tasks efficienti è un'operazione complessa;
- Le rappresentazioni apprese potrebbero essere non proprio generali ma strettamente legate al pretext task specifico.

L'idea è quindi quella di **usare un pretext task più generico** apprendendo più di una singola trasformazione (ad esempio tutti i metodi visti in precedenza) ed **attraendo immagini rappresentanti lo stesso oggetto e respingendo oggetti differenti**. Quindi, data un'immagine x , denotiamo gli esempi positivi x^+ (quelli che contengono lo stesso oggetto) e quelli negativi x^- (quelli che non contengono lo



stesso oggetto). Possiamo formalizzare la Contrastive Learning con questa formula:

$$\text{score}(f(x), f(x^+)) >> \text{score}(f(x), f(x^-))$$

Praticamente vogliamo apprendere una funzione che ci restituisce un score altro per le coppie positive (x, x^+) e un punteggio basso per le coppie negative (x, x') dove lo score viene calcolato da una semplice similarity function. Più in dettaglio, dato 1 esempio positivo x^+ e N-1 esempi negativi, possiamo **calcolare la loss** come:

$$L = -\mathbb{E}_X \left[\log \frac{\exp(s(f(x), f(x^+)))}{\exp(s(f(x), f(x^+))) + \sum_{j=1}^{N-1} \exp(s(f(x), f(x_j^-)))} \right]$$

Questo tipo di loss prende il nome di **InfoNCE loss** e non è altro che una Cross Entropy Loss per un classificatore softmax a N-strade (cioè consideriamo N samples).

Ora, ci sono due modelli che cercano di implementare il Contrastive Learning e uno di questi è il **SimCLR**. L'idea è quella di partire da un'immagine x , si applicano due trasformazioni t e t' a questa immagine (es. rotazione e inpainting) ottenendo delle "augmented views" x_i, x_j dell'immagine. Otteniamo quindi un encoder $f()$ (es. un ViT o una CNN) che, applicato sulle due immagini ci restituisce le loro rappresentazioni. A queste applichiamo un **projector head $g()$** che trasforma le rappresentazioni nelle embeddings su cui applicare l'operazione di similarità. Quindi, assumendo di usare la cosine similarity, vogliamo aumentare la similarità tra le immagini trasformate della stessa istanza e diminuire la similarità con tutte le altre immagini (istanze diverse). La InfoNCE Loss finale sarà:

$$L = -\mathbb{E}_X \left[\log \frac{\exp(s(f(x), f(x^+)))}{\exp(s(f(x), f(x^+))) + \sum_{j=1}^{N-1} \exp(s(f(x), f(x_j^-)))} \right]$$

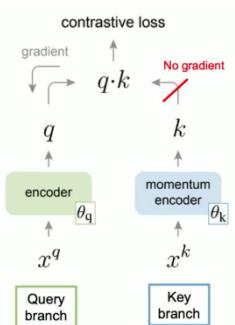
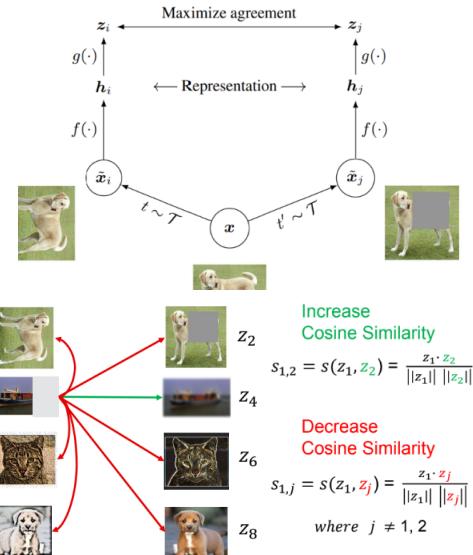
Quindi, una volta allenato questo modello, per creare il downstream task possiamo:

1. Lasciare solo l'encoder $f()$ usato per allenare il pretext task;
2. Aggiungere uno strato lineare \mathbf{l} per l'operazione di classificazione downstream task;
3. Allenare nuovamente il modello risultante sullo stesso dataset (freezando i parametri dell'encoder $f()$). Per quanto riguarda il dataset possiamo decidere di usare solo una piccola percentuale del dataset (1, 10%) oppure l'interno dataset.

In sintesi:

- SimCLR usa una **forte data augmentation**, che può includere trasformazioni come la traslazione, la rotazione e la regolazione del contrasto, per creare coppie di campioni simili e negativi a partire da un'unica immagine.
- Solo quando la **dimensione del batch è sufficientemente grande**, la loss function può coprire una raccolta sufficientemente diversificata di campioni negativi. Questo consente al modello di apprendere una rappresentazione significativa per distinguere diversi esempi.
- Il **projection head $g()$** è un componente aggiuntivo nel SimCLR. L'obiettivo di SimCLR potrebbe comportare la perdita di informazioni utili per downstream tasks. Lo spazio di rappresentazione z è addestrato per essere invarianto rispetto alle trasformazioni dei dati quindi utilizzando il Projection Head, SimCLR mira a preservare più informazioni nello spazio di rappresentazione.

Un altro modello è **MoCo** (Momentum Contrast) in cui la differenza principale con SimCLR è quella di rimpiazzare l'encoder nella parte destra con un **momentum encoder**. Quindi abbiamo supponiamo di avere una query x^q e un esempio positivo x^k , abbiamo quindi due branch che li elaborano, la **query branch** (a sx) e la **key branch** (a dx). A sinistra si prende in input la query x^q e aggiorniamo l'encoder calcolando i gradienti e quindi effettuando la backpropagation tenendo in considerazione solo la query. A destra utilizziamo un approccio diverso che prende in input delle

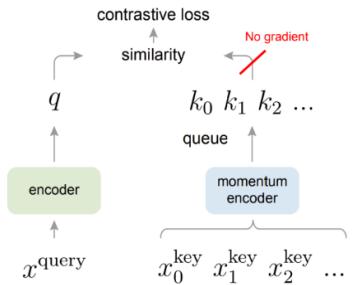


keys x^k e si tiene conto del momentum cioè i parametri del momentum encoder vengono **lentamente aggiornati** con una **update rule**:

$$\theta_k \leftarrow m\theta_k + (1 - m)\theta_q$$

Cioè i parametri Θ_k vengono aggiornati prendendo gli stessi parametri Θ_k precedenti e i nuovi parametri Θ_q della query branch. Questo aggiornamento lento della key branch consente una maggiore stabilità durante il training. Chiaramente questo permette anche di accorciare i tempi in quanto dobbiamo effettuare backpropagation solo nella query branch (e non nella key branch).

Vediamo ora cosa succede con gli esempi negativi. Le rappresentazioni delle immagini target vengono memorizzate nella **Memory Bank** durante l'addestramento e vengono utilizzate come esempi negativi per le immagini query. Ipotizziamo di avere una certa query x^q e un insieme di esempi negativi x^k_j nella Memory bank. Possiamo rappresentare le varie keys (esempi negativi) in una queue. Calcoliamo quindi la rappresentazione delle keys e sulla base di queste le andiamo a confrontare con la query e a calcolare la contrastive loss. MoCo utilizza la dinamica di momentum per aggiornare le rappresentazioni target nella Memory Bank. Praticamente l'idea è mantenere una versione aggiornata e in evoluzione delle rappresentazioni passate, migliorando così la varietà degli esempi negativi disponibili. Questo permette di **utilizzare molti esempi negativi nel training** senza la necessità di utilizzare batch size estremamente grandi e questo fa sì che il modello si possa addestrare anche su sistemi limitati.



Esiste anche una versione alternativa di MoCo chiamata **MoCo v2** che è un'idea ibrida tra MoCo e SimCLR. Quindi prende l'uso della non-linear projection head (non presente in MoCo) e la forte data augmentation, mentre da MoCo prende la queue di esempi negativi e l'utilizzo del momentum encoder.

Un altro modello è **BYOL** (Bootstrap Your Own Latent). La principale differenza con MoCo v2 è l'**assenza di campionamento negativo** durante l'addestramento quindi non è necessario campionare negativi per creare coppie di immagini contrastive, ed inoltre viene **rimossa completamente la coda** utilizzata per elaborare via via gli esempi negativi. Nella branch di sinistra viene aggiunto un ulteriore elemento dopo la projector head (ricorda che questa viene aggiunta in MoCo v2) che è la **predictor head p**. Definiamo:

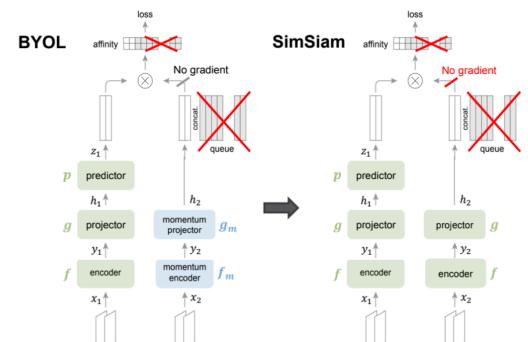
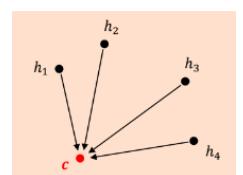
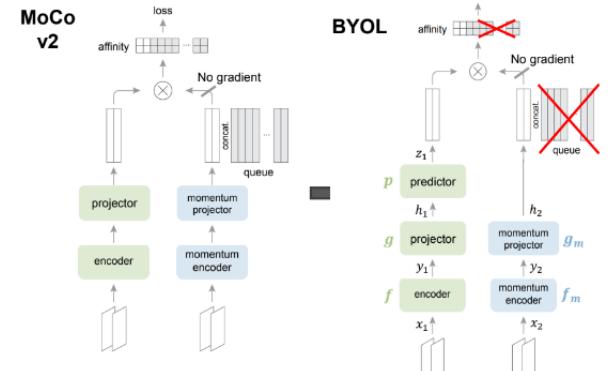
$y_1 = \text{f}(x_1)$, $y_2 = \text{f}_m(x_2)$ are the *representations*

$h_1 = \text{g}(y_1)$, $h_2 = \text{g}_m(y_2)$ are the *embeddings*

$z_1 = \text{p}(h_1)$ is the *prediction*

Ricorda che vogliamo attrarre gli esempi positivi e respingere quelli negativi, quindi se consideriamo un serie di punti h_i in uno spazio 2D dove ogni punto rappresenta un certo esempio, allora vogliamo fare in modo che alla fine gli esempi positivi sono vicini tra loro e quelli negativi sono lontani. Senza esempi negativi, alla fine avremo che tutti i punti sono molti vicini tra loro e in generale non vogliamo questo perché altrimenti apprenderemo delle rappresentazioni generiche. La situazione peggiore che si può verificare è che questi punti si trovano inizialmente in un certo punto nello spazio e questi **collassano verso lo stesso punto c** (supervision collapse).

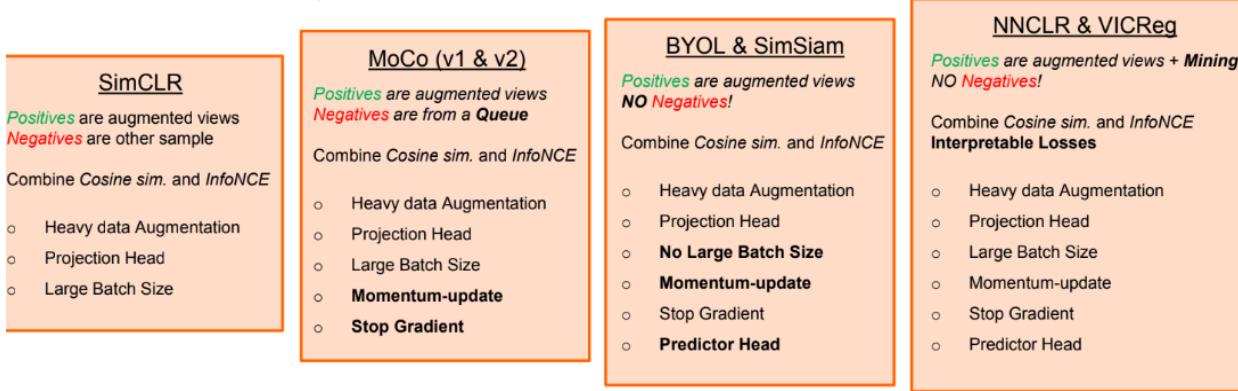
Per effettuare il training senza esempi negativi evitando il problema del supervision collapse, BYOL usa la **predictor head p** e l'**aggiornamento dell'encoder f_m e del projector g_m con il momentum**. Due visualizzazioni della stessa immagine x vengono create applicando trasformazioni di data augmentation. Aggiungendo il predictor head, la struttura è asimmetrica quindi se si compara la versione più fine dell'immagine z_1 data dalla branch di sinistra a partire da x_1 e l'immagine h_2 data dalla branch di destra a partire da x_2 , queste due immagini saranno necessariamente differenti.



Un'altra alternativa è **SimSiam** (Exploring Simple Siamese Representation Learning) che è molto simile a BYOL (quindi si utilizza predictor head e non si usa il campionamento negativo) ma **non prevede l'aggiornamento dell'encoder e del projector head con il momentum** e nella branch di destra si applica **stopping gradient** cioè non si effettua backpropagation sulla branch di destra (quindi entrambe le branch vengono aggiornate allo stesso modo).

Per riassumere:

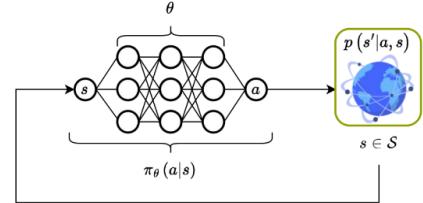
- BYOL rimuove la necessità di effettuare il training con il campionamento negativo;
- SimSiam rimuove anche l'aggiornamento con il momentum (introducendo lo stopping gradient);
- In entrambi i modelli viene utilizzata la forte data augmentation;
- In entrambi i modelli vengono utilizzate predictor head e projector head;
- In SimSiam è possibile ottenere performance abbastanza elevate anche senza l'uso di batch size grandi.



9. Social Navigation

Nel precedenti capitoli è stato introdotto il **Reinforcement Learning**. Si ricorda che gli ingredienti di questo metodo di apprendimento prevedono, dato un certo stato s , l'esecuzione di una certa azione in un ambiente che porta alla modifica dello stato s nell'ambiente. In particolare, sia **S** lo spazio degli stati, una rappresentazione continua o discreta del mondo, **A** è lo spazio di azione del robot (continuo o discreto) allora possiamo rappresentare con $p(s'|a, s)$ è la **probabilità (di transizione) di terminare nello stato s' eseguendo l'azione a nello stato s** . $\pi_\theta(a | s)$ è una **policy** che ci dice quale azione compiere nello stesso s . L'obiettivo è quello di massimizzare la ricompensa quindi dobbiamo **apprendere una policy** che ci permetta sempre di scegliere l'azione per cui si ottiene la massima ricompensa. $r(s, a)$ è una reward function che ci dice quale stato e azione sono migliori. Consideriamo adesso un esperimento in cui vogliamo calcolare la **probabilità di eseguire una traiettoria** composta da **coppie stato-azione**. La probabilità totale è data da:

$$p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t) \quad \text{Markov chain on } (s, a)$$



Come possiamo vedere questa equivale alla probabilità di iniziale in uno stato iniziale s_1 (che è una probabilità sconosciuta) moltiplicata per la probabilità di effettuare un'azione a_t dato uno stato s_t (valore che conosciamo grazie alla policy) per probabilità di transizione di terminare nello stato s_{t+1} dato s_t e a_t (non conosciamo questo valore), questo per tutti i passi $t \in T$ della traiettoria. Quindi vogliamo calcolare i gradienti senza conoscere la probabilità dello stato iniziale e la probabilità di transizione. Pertanto emarginiamo la distribuzione di tutte le possibili coppie stato-azione:

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

$J(\theta)$

L'insieme ottimale di parametri per la policy è quello che **massimizza il valore atteso della reward function** sotto la distribuzione della traiettoria. Quindi l'obiettivo è quello di calcolare $J(\Theta)$ è per fare ciò possiamo cercare di simulare un grande numero di traiettorie (esperimenti) in maniera da rimuovere la probabilità (il valore atteso):

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

Ora, non vogliamo solo valutare l'obiettivo ma vogliamo anche addestrare il modello per ottenere la policy migliore e per fare ciò dobbiamo calcolare il gradiente della policy. Di nuovo, dobbiamo calcolare il gradiente di qualcosa che non conosciamo (stato iniziale e probabilità di transizione) quindi dobbiamo rimuoverli dal valore atteso. Ora, possiamo espandere la formula per calcolare $J(\Theta)$ (il valore atteso) con un integrale della probabilità della traiettoria e la reward:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} [r(\tau)] = \int p_\theta(\tau) r(\tau) d\tau = \sum_{t=1}^T r(s_t, a_t)$$

Dobbiamo calcolare il gradiente quindi, essendo questo lineare, lo possiamo portare dentro l'integrale:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = E_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) r(\tau)] \\ \nabla_\theta \left[\log p(s_1) + \sum_{t=1}^T \log \pi_\theta(a_t | s_t) + \log p(s_{t+1} | s_t, a_t) \right]$$

Effettuiamo quindi varie trasformazioni sul gradiente e alla fine colllassiamo il valore atteso. Nota che il gradiente trasforma il prodotto che avevamo nella precedenti formule, in una somma. Siccome stiamo calcolando il gradiente rispetto a Θ , i due logaritmi interni varranno 0 e li possiamo togliere rimanendo solo con la somma finale.

Sostituendo quindi la $p_\theta(\tau)$ appena espresso, otteniamo:

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \right) \left(\sum_{t=1}^T r(s_t, a_t) \right) \right] \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) \right) \left(\sum_{t=1}^T r(s_{i,t}, a_{i,t}) \right)$$

Rimuovendo il valore atteso come prima. Possiamo quindi effettuare il training della policy usando SGD:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Un algoritmo noto per il training dei sistemi di Embodied AI è **REINFORCE** (1992):

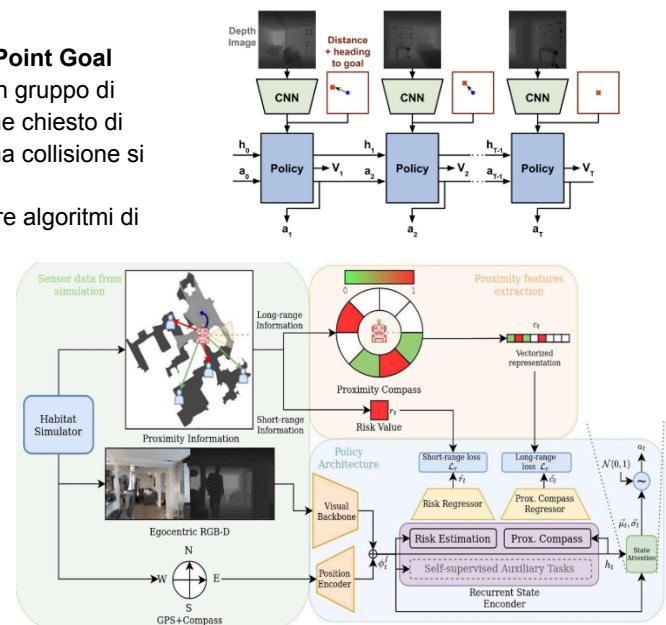
REINFORCE algorithm:

- 1. sample $\{\tau^i\}$ from $\pi_\theta(a_t | s_t)$ (run the policy)
- 2. $\nabla_\theta J(\theta) \approx \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_t^i | s_t^i)) (\sum_t r(s_t^i, a_t^i))$
- 3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Nel capitolo sull'Embodied AI è stato già spiegato come funziona la **Point Goal Navigation**. La **Social Navigation** funziona in modo molto simile: Un gruppo di persone viene posizionata randomicamente in un ambiente e gli viene chiesto di muoversi verso una destinazione e di tornare indietro. Se avviene una collisione si termina l'esperienza.

In generale, ci sono delle linee guida che bisogna seguire per valutare algoritmi di **Social Robot Navigation**.

Il modello base per la Point Goal Navigation funziona in questo modo: Data un'immagine ottenuta ad esempio dalla fotocamera dell'agente da cui vengono estratte le features utilizzando una CNN, e data le coordinate GPS che dicono qual è la distanza dall'obiettivo queste vengono date alla policy che calcola la nuova azione da intraprendere in maniera ricorsiva. Ora, questa architettura non sfrutta features sociali quindi è stata modificata in modo da introdurre dei **proximity tasks** che calcolano il **rischio di collisione** con una persona (è quindi un valore a basso range che vale 0 se ci si trova ad almeno k metri della persona, 1 altrimenti), e



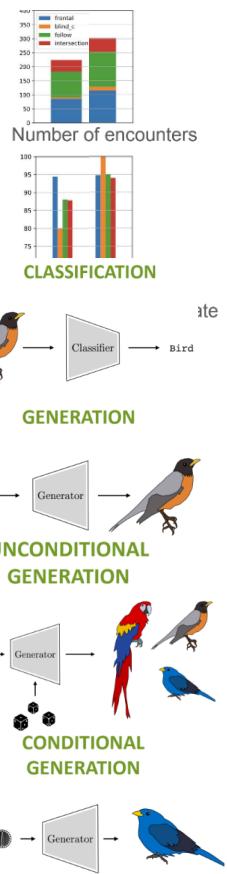
una **bussola di prossimità** che è un valore che calcola la probabilità di collisione sulla base del settore nell'ambiente (quindi è un valore ad ampio raggio che dice all'agente dove è sicuro andare per non collidere). Questi due valori vengono quindi introdotti nell'architettura e usati per il training.

Alcune **metriche di valutazione** note sono:

- Successo: Percentuale di test riusciti;
- SPL: Misura l'ottimalità del percorso intrapreso dall'agente. Negli episodi falliti il suo valore è 0;
- Collisione umana: Percentuale di episodi terminati a causa di una collisione con un essere umano.

Successo e SPL non sono abbastanza per valutare il social navigation quindi è stato sviluppato un'ulteriore metodo chiamato **Encounter Based Evaluation** con l'obiettivo di capire come e quando avvengono interazioni con una persona. Quindi la classificazione degli incontri avviene in base ai dati di traiettoria, visibilità e direzione dell'agente. Sono emersi due tipo di comportamenti:

- **Encounter elusion** (basso numero di incontri, ESR medio) (Baseline)
- **Encounter avoidance** (alto numero di incontri, ESR alto) (FBK's solution)



10. Generative Models

Un **Generative Model** è un tipo di modello progettato per **generare nuovi punti dati che assomigliano a un insieme di dati di allenamento specifico**. Ora, nella classificazione quello che succede è il modello è tipicamente addestrato per categorizzare i dati di input in classi predefinite (quindi data un'immagine di un uccello, il modello ci ritorna la sua classe). Nella **generazione** il modello è addestrato per generare nuovi dati che sono simili ai dati di addestramento (quindi data la classe "uccello", il modello ci ritorna l'immagine di un uccello). Ora, il **rumore** è una **variabile latente** del nostro modello e, in generale, vogliamo imparare una funzione che genera dati da rumore. Abbiamo due scenari:

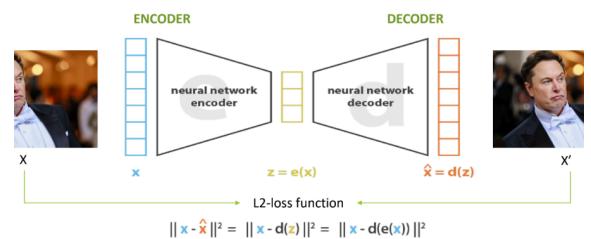
- **Generazione Incondizionata:** La generazione incondizionata comporta la generazione di nuovi dati senza condizionamenti specifici in input. Il modello impara a generare campioni di dati senza informazioni aggiuntive.
- **Generazione Condizionata:** La generazione condizionata comporta la generazione di nuovi dati in base a condizioni o input specifici. Il modello è addestrato per generare campioni di dati che sono condizionati da un contesto o informazioni specifiche.

Quindi data la classe "uccello" possiamo specificare una condizione per la generazione (es. il colore) oppure il modello genererà l'immagine delle classi richiesta con condizioni randomiche. Le **variabili latenti organizzano i dati** quindi data una certa immagine generata (es. immagine un vulcano) possiamo via via generare altre immagini con altre condizioni (es. cambiare l'ora del giorno e la luminosità della scena).

Prima di addentrarci in cos'è un Auto Encoder, facciamo un riassunto sul **Unsupervised Learning**: Quello che accade in questo tipo di setting è che non abbiamo delle etichette e quindi si cerca di apprendere pattern o strutture nei dati. Non si hanno dei target o degli output da predire, per cui l'algoritmo cerca di identificare relazioni nascoste tra i vari dati. Qui, per riassumere:

- Dati non etichettati: Gli algoritmi di Unsupervised Learning lavorano con dati che non sono etichettati;
- Scoperta di modelli: Gli algoritmi di Unsupervised Learning mirano a scoprire strutture sottostanti, relazioni o modelli nei dati.
- Pattern nel data space: I metodi di Unsupervised Learning si concentrano sull'identificazione di pattern, regolarità o tendenze all'interno dello spazio dei dati senza l'uso di etichette predefinite.

Un **Autoencoder** è un tipo di rete che apprende una codifica di dati di input attraverso l'encoder e cerca di ricostruire i dati originali dall'encoding attraverso il decoder. L'obiettivo è ridurre la dimensione dei dati di input (compressione) in una rappresentazione più compatta, e successivamente, ricostruire i dati originali (decompressione) utilizzando questa rappresentazione compatta. In questo modo, l'auto encoder cerca di catturare le caratteristiche più rilevanti dei dati.



L'autoencoder è addestrato in modo **non supervisionato**, il che significa che **non richiede etichette per i dati di addestramento**. L'obiettivo principale dell'addestramento è massimizzare la precisione della ricostruzione, in modo che le caratteristiche apprese possano essere utilizzate per generare una buona approssimazione dei dati originali. Quindi, data l'immagine x questa viene passata all'encoder che calcola la codifica $z = e(x)$ e successivamente si passa al decoder che ricostruisce l'immagine $\hat{x} = d(z)$. Una volta ottenuta \hat{x} calcoliamo la **L2-loss** come:

$$\|x - \hat{x}\|^2 = \|x - d(z)\|^2 = \|x - d(e(x))\|^2$$

Quindi quello che si fa è:

1. L'encoder mappa l'immagine originale in uno spazio latente: $\Phi: X \rightarrow F$;
2. Il decoder mappa lo spazio latente ad un output: $\psi: F \rightarrow X$;
3. Il processo di addestramento minimizza l'errore di ricostruzione, cercando di ricreare i dati originali dopo una compressione non lineare generalizzata nello spazio latente:

$$\phi, \psi = \arg \min_{\phi, \psi} \|X - (\psi \circ \phi)X\|^2$$

Un **Variational Autoencoder (VAE)** è strutturalmente simile a un autoencoder standard infatti ha ancora un encoder e un decoder, ma ha una differenza fondamentale: introduce il concetto di **probabilità nella rappresentazione degli spazi latenti**. Nel VAE, la differenza principale rispetto all'autoencoder tradizionale è che invece di avere un singolo punto nello spazio latente per ogni input, si ha una distribuzione di probabilità e questa cattura l'incertezza o la variabilità nella rappresentazione latente. Quindi, mentre **l'autoencoder tradizionale fornisce una rappresentazione deterministica dello spazio latente**, il

VAE fornisce una rappresentazione probabilistica cioè non produce un singolo vettore latente, ma una distribuzione di probabilità su tutto lo spazio latente. La loss function utilizzata in un Variational Autoencoder è composta da due parti principali: la loss di ricostruzione e la **divergenza di Kullback-Leibler (KL divergence)**. Queste due componenti riflettono gli obiettivi fondamentali del VAE, che sono sia ricostruire fedelmente i dati di input sia garantire che la distribuzione nello spazio latente seguia una forma specifica, tipicamente una distribuzione normale (gaussiana):

$$\|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

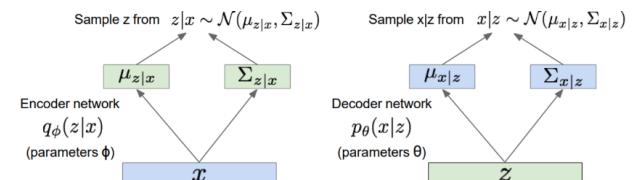
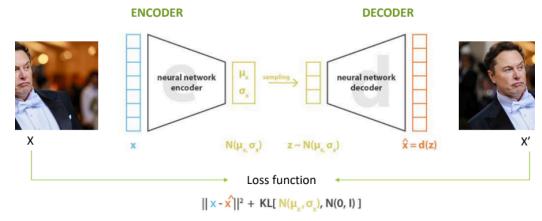
Quindi assumiamo che vi siano due distribuzioni principali:

1. Si assume che la rappresentazione latente z sia campionata da una distribuzione a priori $p(z)$. Questa distribuzione rappresenta le caratteristiche latenti che saranno utilizzate per generare i dati.
2. Si assume che i dati x siano campionati dalla distribuzione di verosimiglianza condizionale (conditional likelihood) $p(x|z)$. Questa distribuzione rappresenta la probabilità di generare i dati osservati x dato un certo valore della rappresentazione latente z .

Ridefinendo l'Autoencoder considerando la versione probabilistica, otteniamo:

1. Il decoder probabilistico è definito dalla distribuzione condizionale $p(x|z)$. In termini matematici, la probabilità di generare x dato z è rappresentata come $p(x|z)$.
2. L'encoder probabilistico è definito dalla distribuzione $q(z|x)$, che rappresenta la probabilità di ottenere una certa rappresentazione latente z dato un'osservazione x . Questa distribuzione cattura l'incertezza nell'associazione tra x e z .
3. Le rappresentazioni codificate z nello spazio latente sono assunte seguendo la distribuzione a priori $p(z)$. Questo garantisce che le rappresentazioni latenti siano coerenti con le caratteristiche latenti predefinite dalla distribuzione prior $p(z)$.

Nota che siccome stiamo modellando la generazione probabilistica dei dati, anche encoder e decoder sono probabilistici. In generale, quello che vogliamo fare nella formula in basso è massimizzare il limite inferiore cioè la probabilità dell'input originale di essere ricostruito:



$$\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\
&= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\
&= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z) q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)}) q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\
&= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\
&= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \underbrace{D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)} + \underbrace{D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))}_{>0} \\
\text{Training: Maximize the Lower Bound, i.e. the likelihood of original input being reconstructed} \\
\theta^*, \phi^* &= \arg \max_{\theta, \phi} \sum_{i=1}^N \mathcal{L}(x^{(i)}, \theta, \phi) \quad \text{Tractable lower bound which we can take gradient of and optimize!} \quad p(z|x) \text{ intractable, can't compute this KL. But we know divergence always } \geq 0
\end{aligned}$$

Quindi:

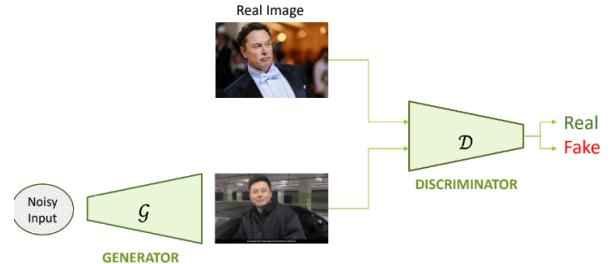
- L'autoencoder comprime i dati in input in uno **spazio latente fisso**. Sono facili da allenare in quanto non dobbiamo individuare una distribuzione probabilistica dello spazio latente e la ricostruzione potrebbe essere più precisa rispetto ai VAE.
- I Variational Autoencoders apprendono una **distribuzione probabilistica dello spazio latente**. Questi offrono degli spazi latenti continui e strutturati che permettono la generazione di nuove immagini sulla base di certa distribuzioni appresa.

Introduciamo adesso i **Generative Adversarial Networks (GANs)**.

L'idea principale dietro le GAN è addestrare contemporaneamente un **generatore** e un **discriminatore** attraverso un processo avversario.

Quindi:

- Il generatore** mira a produrre campioni di dati sintetici minimizzando la capacità del discriminatore di differenziare tra campioni reali e generati.
- Il discriminatore** mira a differenziare tra dati reali e falsi (generati). Quindi l'obiettivo è quello di migliorare la sua capacità di classificare accuratamente i dati imparando da campioni reali e generati.



Le GAN operano in modo avversario, dove il generatore e il discriminatore sono impegnati a giocare l'uno contro l'altro. Il generatore cerca di creare dati così realistici che il discriminatore non possa distinguere tra campioni reali e generati. Allo stesso tempo, il discriminatore cerca di diventare competente nel distinguere tra dati reali e generati. Il processo è iterativo, con sia il generatore che il discriminatore che migliorano nel tempo. Possiamo quindi effettuare il training in maniera congiunta utilizzando un **minmax game** dove il discriminatore vuole massimizzare l'obiettivo in modo tale che $D(x)$ sia vicino a 1 (reale) e $D(G(z))$ sia vicino a 0 (falso) mentre il generatore vuole minimizzare l'obiettivo in modo tale che $D(G(z))$ sia vicino a 1 (il discriminatore viene ingannato nel pensare che $G(z)$ generato sia reale):

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log (1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Discriminator output for real data x Discriminator output for generated fake data G(z)

Alcuni vantaggi dei GAN sono:

- Generazione di dati realistici: I GAN eccellono nel generare dati sintetici altamente realistici che assomigliano molto ai dati di addestramento.
- Risultati diversi: Offrono la possibilità di produrre risultati diversi e nuovi, favorendo la creatività e l'esplorazione nella generazione dei dati.
- Apprendimento non supervisionato: I GAN operano in modo non supervisionato, eliminando la necessità di dati di addestramento etichettati in alcuni scenari.

Alcuni svantaggi sono:

- Instabilità del training: I GAN possono essere difficili da addestrare a causa del loro delicato equilibrio tra generatore e discriminatore.
- Complessità della valutazione: Valutare le prestazioni e la qualità dei campioni generati dal GAN può essere soggettivo e impegnativo.

- Elaborazione intensiva: Il training dei GAN può essere computazionalmente costoso e richiedere risorse significative, in particolare per output ad alta risoluzione o set di dati complessi.

Il Denoising Diffusion Probabilistic Model (DDPM) è un modello generativo progettato per generare campioni di alta qualità in modo probabilistico. L'idea centrale di DDPM si basa su un **processo di diffusione**, in cui un rumore viene aggiunto ai dati in modo iterativo. Il modello mira a imparare la distribuzione di probabilità dei dati ad ogni passo del processo di diffusione. Durante l'addestramento, il modello viene esposto a versioni rumorose dei dati di addestramento dove l'obiettivo è imparare i parametri del processo di diffusione in modo che, quando il rumore viene gradualmente rimosso, il modello possa generare campioni realistici che corrispondono alla distribuzione dei dati di addestramento. La parte di **denoising** comporta la rimozione del rumore aggiunto in modo iterativo, quindi il modello impara a togliere il rimuovere dai dati catturando la struttura sottostante e le dipendenze nei dati.

Da un punto di vista matematico:

- Forward Pass: Si applica il processo di diffusione iterativo cioè, ad ogni passo, il rumore si diffonde nel dato x_t . Supponendo di avere x_{t-1} , applichiamo il rumore ad x_t nel seguente modo:

$$q(x_t | x_{t-1}) = \mathcal{N}(\sqrt{1 - \beta_t} x_{t-1}, \beta_t^2)$$

Dove β è un parametro che rappresenta la correlazione tra il dato precedente e attuale. Dopo un numero desiderato di passi di diffusione, si ottiene il dato finale x_T che è una versione rumorosa del dato originale.

- Reverse Process: Per generare campioni dal modello, si parte da un dato rumoroso x_T e si applica il processo inverso. Il parametro σ rappresenta la varianza dei dati precedenti:

$$p_\theta(x_{t-1} | x_t) = \mathcal{N}(f_\theta(x_t, t), \sigma^2)$$

Il modello deve apprendere p_θ , quindi si procede iterativamente all'indietro attraverso i passi di inversione fino a ottenere x_0 , il campione di rumore iniziale.

Alcuni vantaggi sono:

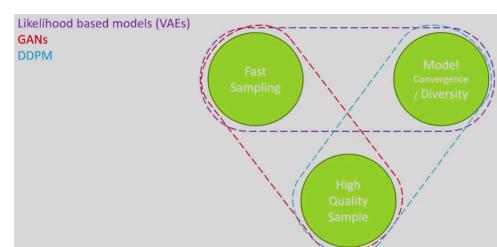
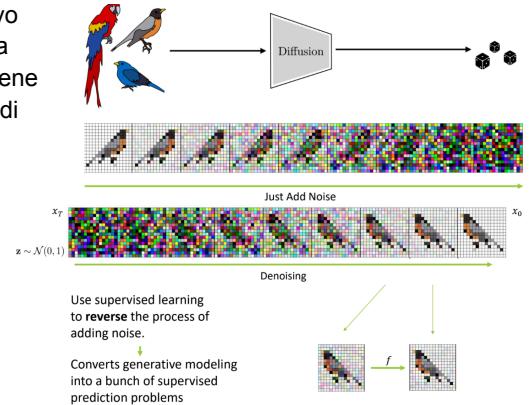
- Modellazione probabilistica esplicita: DDPM modella esplicitamente la distribuzione probabilistica dei dati, consentendo un controllo più preciso sul processo di generazione.
- Formazione stabile: Il training non prevede componenti antagonisti o architetture complesse, spesso determinando una convergenza più stabile durante la formazione.
- Versatilità: Può generare campioni in sequenza su più passaggi, acquisendo dipendenze complesse all'interno della distribuzione dei dati.

Alcuni svantaggi sono:

- Complessità computazionale: I DDPM possono essere computazionalmente pesanti, soprattutto durante il processo di campionamento a causa dell'iteratività passaggi coinvolte nel processo di diffusione, rendendoli più lenti rispetto ad altri modelli generativi.
- Sfide di formazione: Il training potrebbe richiedere una maggiore quantità di dati e metodi di ottimizzazione più complessi rispetto a modelli generativi più semplici, che possono rendere il training più impegnativo e dispendioso in termini di risorse.
- Requisiti di memoria: La memoria richiesta potrebbe essere più elevata rispetto ad altri modelli, soprattutto quando si gestiscono dati ad alta dimensione, a causa della necessità di memorizzare passaggi di diffusione intermedi.

Alcuni differenze tra i vari metodi sono:

- Approccio di modellazione distinto: A differenza dei GAN, che utilizzano uno schema di contrastive learning, e degli autoencoders, che si concentrano sulla codifica e decodifica delle strutture, DDPM si basa su una modellazione probabilistica esplicita attraverso processi di diffusione.
- Complessità della formazione: I DDPM possono avere requisiti computazionali inferiori rispetto ai GAN (in particolare durante il training) a causa dell'assenza di componenti contraddittori.



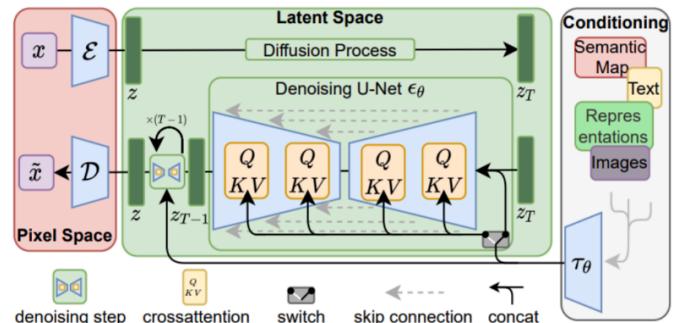
- Metodo di generazione dei campioni: DDPM genera campioni applicando iterativamente rumore ai dati, in contrasto con i GAN che imparano a produrre campioni attraverso una rete di generatori alimentata con rumore casuale.

Il **Latent Diffusion Model (LDM)** è un modello che sfrutta la decomposizione del processo di formazione delle immagini in un'applicazione sequenziale di autoencoder per la denoising e modelli di diffusione (DMs), ottenendo risultati di sintesi all'avanguardia su dati di immagini e oltre. Alcune caratteristiche sono:

- Decomposizione del Processo di Formazione delle Immagini: Il LDM decomponete il training delle immagini in una sequenza di applicazioni di autoencoder per la denoising e modelli di diffusione.
- Controllo del Processo di Generazione dell'Immagine: La formulazione del modello consente un meccanismo guida per controllare il processo di generazione delle immagini senza necessità di re-training.
- Utilizzo dello Spazio Latente: Per consentire l'addestramento del DM con risorse computazionali limitate, il modello viene applicato nello **spazio latente** di autoencoder preaddestrati. Questo approccio consente di raggiungere un punto quasi ottimale tra **riduzione della complessità** e **conservazione dei dettagli**, migliorando notevolmente la fedeltà visiva.
- Introduzione di Strati di Cross-Attention: Aggiungendo strati di **cross-attention** all'architettura del modello, i DM vengono trasformati in generatori potenti e flessibili per **input di condizionamento generale** come testo o bounding boxes, rendendo possibile la sintesi ad alta risoluzione in modo convoluzionale.
- Riduzione dei Requisiti Computazionali: Gli LDM riducono significativamente i requisiti computazionali rispetto ai DM basati su pixel, mantenendo nel contempo la qualità e la flessibilità del modello.

L'architettura è la seguente:

- Reconstruction (Pixel Space): Il processo di ricostruzione nello spazio dei pixel coinvolge due componenti principali:
 - Encoder: L'encoder comprime i dati di input in una rappresentazione latente a dimensionalità inferiore modellata come una distribuzione gaussiana o un'altra distribuzione.
 - Decoder: La rete decoder tenta di invertire il processo di compressione eseguito dall'encoder. Il suo obiettivo è ricostruire i dati originali a partire dalla rappresentazione latente, cercando di preservare il più possibile le informazioni originali.
- Diffusion Process (Latent Space): Nel processo di diffusione nello spazio latente, si verifica una diffusione graduale della rappresentazione latente attraverso i seguenti passaggi:
 - Aggiunta di Rumore: Ad ogni passo del processo di diffusione, del rumore viene aggiunto allo spazio latente. Questo processo è progettato per far diffondere gradualmente la rappresentazione latente su tutta la varietà dei dati.
 - Denoising U-Net: La rete Denoising U-Net cerca di recuperare le informazioni originali, pulite, a partire dai dati rumorosi generati durante il processo di diffusione nello spazio latente. Questo è un passaggio critico per mantenere la qualità dell'informazione originale.
- Manipolazione (Conditioning): Per la manipolazione del modello mediante il condizionamento, si effettuano le seguenti operazioni:
 - Generazione Condizionata: Il modello può generare output condizionati su attributi specifici, come etichette di classe, descrizioni di testo o altre caratteristiche.
 - Rappresentazione Latente della Condizione: Una rete è utilizzata per ottenere una rappresentazione latente della condizione, ovvero delle informazioni di condizionamento. Questa rappresentazione è poi utilizzata durante il processo di generazione per influenzare la produzione di output.



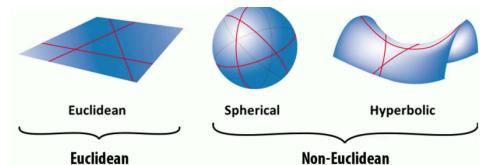
Ecco alcuni vantaggi chiave della diffusione latente rispetto al DDPM:

- Maggiore stabilità nella formazione: Questi modelli potrebbero impiegare tecniche che affrontano i problemi di convergenza, portando a dinamiche di formazione più stabili e affidabili.
- Migliore qualità e diversità del campione: La capacità di navigare nello spazio latente in modo più efficace può portare alla produzione di risultati più vari e realistici.
- Rappresentazione efficiente dello spazio latente: La diffusione latente spesso apprende una rappresentazione dello spazio latente più efficiente e informativa rispetto al DDPM.
- Multimodalità: I modelli di diffusione latente possono possedere una multimodalità avanzata, consentendo la generazione di diversi output corrispondenti a varie modalità all'interno della distribuzione dei dati.

11. Hyperbolic Neural Network

Finora abbiamo lavorato su NN che processano dati su uno spazio euclideo ma può capitare anche di dover lavorare su **dati che si trovano su uno spazio non-euclideo quali dati sferici o iperbolici**.

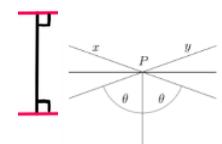
La geometria iperbolica rappresenta la geometria naturale delle gerarchie e qui la crescita è esponenziale (contrariamente allo spazio euclideo in cui la crescita è lineare). Per questo motivo le gerarchie e lo spazio euclideo non si adattano bene. Ciò ci consente di incorporare dati gerarchici con una distorsione minima.



La ricerca ha visto che questa codifica gerarchica dei dati (soprattutto in problemi di NLP che hanno a che fare con testo) permettono di catturare sia le relazioni locali che globali nelle strutture di testo e quindi di aumentare il livello di dettaglio catturato da questi modelli. Questo tipo di codifica non ha riguardato solo l'NLP ma anche la medicina (es. codifica delle molecole) e la computer vision. In particolar modo, per quanto riguarda la computer vision dobbiamo tenere conto del fatto che le collezioni di immagini sono tipicamente gerarchiche così come la semantica (es. "Apparecchio elettrico" ha come figli "Installare una lampadina" o "installare un ventilatore" che a loro volta avranno dei figli semanticamente simili).

Lo hyperbolic learning non è ancora lo standard per una serie di motivi (es. deep learning tools come PyTorch sono euclidei, i computer lavorano meglio con dati euclidei, ecc.). Ora, cos'è la geometria iperbolica? Introduciamo prima di tutto la geometria euclidea. Gli assiomi euclidei sono:

1. Una linea può essere tracciata da due punti qualsiasi;
2. Qualsiasi linea retta può essere estesa indefinitamente;
3. Esiste un cerchio per ogni centro e raggio;
4. Tutti gli angoli retti sono congruenti;
5. Data una retta e un punto esterno ad essa, esiste esattamente una retta passante per il punto dato che non interseca la retta data (postulato delle parallele).



La geometria iperbolica sostituisce il quinto assioma: Dato un punto P e una retta l che non passa per P, esiste più di una retta passante per P che non incontrano l (il postulato delle parallele nella geometria iperbolica). Quindi, data una linea l e un punto P, ci sono infinite linee passanti per P parallele a l. Alcune proprietà della geometria iperbolica sono che la somma degli angoli del triangolo è meno di 180°, la circonferenza del cerchio di raggio r è maggiore di $2\pi r$ e così via.

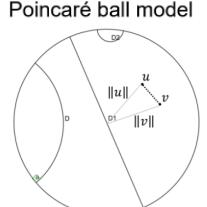


Ci sono vari modelli che sono stati studiati nella geometria iperbolica e ognuno di questi ha una sua equazione e la sua rete neurale. Uno dei modelli è il **Poincaré ball Model**. Quello che si fa è prendere tutti i punti nello spazio euclideo (chiaramente infinito) e schiacciarli all'interno di un cerchio con raggio 1:

$$\mathbb{B}^n = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\| < 1\}$$

Le linee rette (geodetiche) sono i diametri e gli archi circolari perpendicolari al confine. La distanza di Poincaré tra u e v dipende anche dai raggi iperbolici di u e v:

$$d_{\mathbb{B}^n}(u, v) = \text{arcosh} \left(1 + 2 \frac{\|u - v\|^2}{(1 - \|u\|^2)(1 - \|v\|^2)} \right)$$



Quindi, assumendo che v sia l'origine, la distanza tra u e v cresce esponenzialmente (piuttosto che linearmente nello spazio euclideo). Per andare dallo spazio euclideo a quello iperbolico, possiamo mappare ogni punto utilizzando la seguente equazione:

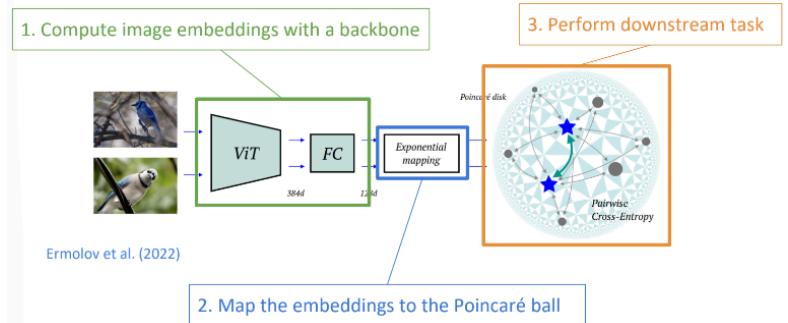
$$\tilde{h} = \text{Exp}_0^c(\tilde{x}) = \tanh(\sqrt{c} \|\tilde{x}\|) \frac{\tilde{x}}{\sqrt{c} \|\tilde{x}\|} \quad (\text{curvature } c)$$

Gli angoli tra due punti sono preservati ma per calcolare la distanza occorre utilizzare la formula vista sopra. Quindi:

- Il volume e le distanze in crescita esponenziale consentono di incorporare gerarchie con bassa distorsione.
- La crescita esponenziale della distanza di Poincaré **rende il raggio iperbolico $\|\mathbf{h}\|$ un metodo per calcolare l'incertezza** (maggiore è la distanza, maggiore è l'incertezza che sappiamo cresce esponenzialmente).

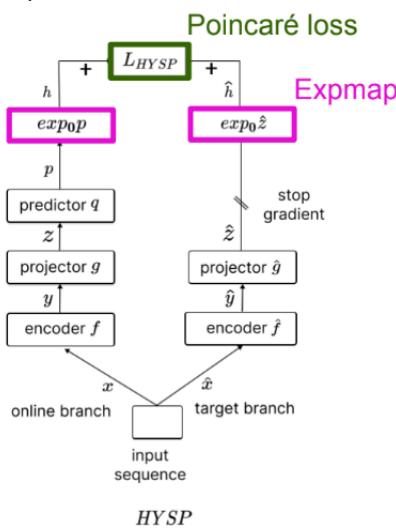
Possiamo utilizzare questo meccanismo per valutare il modello. Se il modello non è certo di una predizione prenderà una penalità minore rispetto a essere certi e sbagliare (in questo caso gli verrà assegnata una penalità esponenziale).

Per andare nello spazio iperbolico si può calcolare l'embedding di un'immagine con una backbone network (es. ViT + FC) e una volta che abbiamo la codifica, possiamo mapparla utilizzando le formule viste in precedenza nonché la Poincaré ball. Questa operazione prevede di calcolare una **Exponential map**. Una volta fatto ciò si può effettuare il downstream task calcolando la loss (o l'incertezza) applicando una MLR (Multinomial Logistic Regression) per fare la predizione nello spazio iperbolico.



Nota: Esiste una libreria Python chiamata "hypil" che permette di gestire tutto il processo appena descritto.

Un possibile **architettura basata su SSL** che permette di calcolare la **Poincaré Loss** è la seguente:



Quindi la rete apprende ad assegnare una incertezza maggiore agli elementi che sono più distanti tra loro, e una incertezza minore agli esempi che sono vicini tra loro.

Per riassumere: La distanza di Poincaré o perdita di Poincaré è un concetto utilizzato nell'ambito dell'apprendimento automatico, in particolare nel contesto della geometria iperbolica. Viene spesso impiegata quando si tratta di dati che mostrano strutture gerarchiche o ad albero. La distanza di Poincaré misura la dissimilarità tra i punti nello spazio iperbolico.

Nella geometria iperbolica, a differenza di quella euclidea, le distanze non sono costanti nello spazio. Invece, crescono in modo esponenziale man mano che ci si allontana da un punto di riferimento. La distanza di Poincaré cattura questa proprietà ed è comunemente utilizzata per l'incorporamento di dati gerarchici in uno spazio iperbolico.

La perdita di Poincaré è una funzione matematica che quantifica la differenza tra le relazioni gerarchiche previste e quelle effettive nello spazio iperbolico. È progettata per ridurre al minimo la distorsione della struttura gerarchica durante l'addestramento di modelli di apprendimento automatico.

La funzione di perdita è tipicamente definita sulla base della distanza di Poincaré tra i punti nello spazio iperbolico. Minimizzando questa perdita durante l'addestramento, il modello mira a imparare rappresentazioni che rispettano le relazioni gerarchiche presenti nei dati.