

Artificial Neural Networks and Deep Learning

Appunti NON UFFICIALI del corso del prof. Matteucci Matteo*

Politecnico di Milano

Fabio Ciavarella

A.A. 2019-2020

* Questo documento raccoglie gli appunti da me raccolti durante le lezioni del corso, corroborati da pezzi e immagini presi dalle slides ufficiali. Nonostante siano stati rivisti possono contenere errori oppure imprecisioni. È possibile distribuire, modificare, etc, etc questo documento, purché venga citata la fonte.

Nel caso in cui troviate errori grossolani o riuscite a spiegare un concetto in modo migliore, modificate e fate girare. Rendiamo il Poli un posto migliore :).

Tipi di apprendimento:

Apprendimento **supervisionato** (es: **classificazione**): dato un vettore di esperienza D (dati).

Dato un vettore di output desiderati, imparare a produrre il corretto output con un nuovo insieme di input. **Questa è la parte su cui si concentra il corso!**

Apprendimento non **supervisionato** (es: **clustering**): esempio: abbiamo un insieme di macchine e vogliamo creare gruppi, o clusters, sfruttando le regolarità del vettore D.

La **differenza** tra i due è che nel primo caso abbiamo già delle etichette e vogliamo che la macchina impari come assegnare le etichette agli elementi. Nel secondo caso, il clustering, diamo in pasto alla macchina dei dati, senza già dargli etichette, e la macchina in base alle somiglianze tra gli elementi impara a creare dei clusters.

Reinforcement Learning: insegnare ad un AI, agente, a stare in un ambiente sconosciuto. Producendo azioni che impattano l'ambiente e ricevendo delle ricompense, imparare a massimizzare le ricompense a lungo termine.

Hebian Learning (perceptron): (*vedi esercizio 27/09/19 sul quaderno*)

Dato un dataset formato da records e output attesi, inizializzando dei pesi a caso e scegliendo un opportuno coefficiente di apprendimento, si riesce a trovare i pesi giusti.

Le soluzioni possono essere più d'una ma ugualmente corrette, dipende dall'inizializzazione.

Ci sono due motivi per cui questo algoritmo può non convergere:

1) la soluzione non esiste

2) abbiamo η (netta, il learning rate) che è troppo grande, continuiamo a cambiare i pesi troppo in fretta e non riusciremo ad arrivare ad una soluzione. Quindi η deve essere piccola abbastanza.

Ogni iterazione è chiamata *epoca*.

Questo modo di procedere, un record alla volta e usare subito i pesi, è chiamato **ONLINE**. Perché usiamo gli ultimi pesi disponibili per computare i nuovi pesi (ricordati che nel record 2 usiamo i pesi calcolati (se son cambiati) al record 1).

Il perceptron è praticamente un classificatore lineare per il quale il confine decisionale è un iperpiano. Il bias ci serve per non dover passare per forza dall'origine degli assi.

Il perceptron funziona se possiamo separare i punti positivi e negativi con una sola retta.

Nel caso in cui ci sia bisogno di più linee l'Hebian Learning non funziona più →

→ Feed Forward Neural Network

Ci sono due importanti livelli: quello di input con I neuroni e quello di output con K neuroni. Il numero di neuroni di output rispecchia il numero di output che abbiamo bisogno. In genere noi utilizzeremo solo un output.

Quanti neuroni in ogni livello intermedio (nascosto)? Non lo sappiamo in anticipo.

Assumiamo che ogni neurone è connesso a tutti i nodi precedenti e a tutti i nodi successivi. Se così non fosse basterebbe impostare il peso a 0 (cioè c'è la freccia ma non vale un cazzo).

La cosa più importante è che il segnale va sempre dall'input all'output. E l'**output** di un neurone dipende solo dal livello precedente.

Funzioni di attivazione:

- Lineare: la usiamo in Regressione. La regressione si usa quando l'output è continuo (appartenente ad R);
- Sigmoid: output binario 0 e 1 → classificazione binaria;
- Tanh: output binario -1 e 1 → classificazione binaria.

Trovare i pesi in una rete neurale non è un'ottimizzazione lineare. Usiamo il gradiente per trovare il minimo della funzione di errore. Per non imbatterci in un minimo locale e cercare uno globale ricominciamo più volte con configurazioni iniziali diverse.

Backpropagation: il gradiente di un peso è il prodotto della derivata di un nodo per il peso dell'arco via via discorrendo fino a raggiungere il nodo di input.

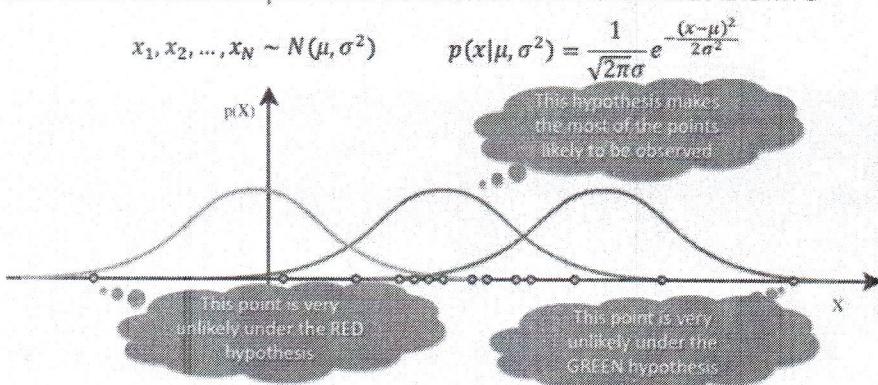
Per ogni passo, ogni aggiornamento devo prendere in esame tutti i pesi precedenti. Questi calcoli vengono fatti solitamente dalla GPU, ma per grandi dataset è inefficiente mandare tutti i dati alla gpu, riscriverli in memoria e rimandare di nuovo tutto alla gpu. Il problema è che la memoria della GPU è limitata e non ci possono entrare sempre tutti i dati.

Per questo possiamo utilizzare altri tipi di discesa col gradiente. Come quella **stocastica** o mini-batch. Il metodo stocastico è praticamente una mini informata con grandezza di dati = 1.

Funzione di errore.

Maximum Likelihood Estimation. Abbiamo una distribuzione gaussiana di cui sappiamo il sigma (la varianza). Vogliamo calcolarci la media. In pratica dove è posizionata la gaussiana nel piano cartesiano

Let's observe i.i.d. samples from a Gaussian distribution with known σ^2



In pratica vogliamo trovare *il modello per cui i dati hanno la massima probabilità (non sono sicurissimo)*.

La likelihood è la probabilità del: dato1, dato2...datoN. Se le variabili sono indipendenti la probabilità congiunta delle probabilità è il prodotto delle loro singole probabilità. Per questo motivo ci schiappa la produttoria (slide 36). Questa è la funzione che vogliamo derivare e egualare a zero.

Passando al logaritmo, la produttoria diventa sommatoria; essendo inoltre una funzione monotona, i minimi rimangono minimi e i massimi rimangono massimi allo stesso punto.

Per calcolarci il likelihood della **media** (per trovare qual è la gaussiana giusta), ci calcoliamo prima il likelihood dei dati. Dobbiamo quindi calcolare la massima probabilità, per toglierci σ^2 che è sempre negativo, cambiamo il segno della funzione e passiamo, quindi, al minimo.

$$= \underset{w}{\operatorname{argmin}} \sum_{n=1}^N (t_n - g(x_n|w))^2$$

Maximum Likelihood Estimation for Regression **GAUSSIANA**.

Maximum Likelihood Estimation for Classification **Bernoulli distribution**.

NB: La funzione che utilizziamo per la Maximum Likelihood Estimation indica quale problema vogliamo risolvere:

Regression → gaussiana

Classificazione → Bernoulli

$$\begin{aligned} & \text{Crossentropy} \\ & - \sum_{n=1}^N t_n \log g(x_n|w) + (1-t_n) \log(1-g(x_n|w)) = \\ & = \underset{w}{\operatorname{argmin}} - \sum_{n=1}^N t_n \log g(x_n|w) + (1-t_n) \log(1-g(x_n|w)) \end{aligned}$$

Come scegliere la funzione di errore?

Usare tutta la conoscenza/assunzioni sulla distribuzione dei dati;

Sfruttare la conoscenza pregressa sul tipo di lavoro e sul modello;

Trial & Error.

How to Choose the Error Function?

We have observed different error functions so far

$$E(w) = \sum_{n=1}^N (t_n - g_1(x_n, w))^2$$

$$E(w) = - \sum_{n=1}^N t_n \log g(x_n|w) + (1-t_n) \log(1-g(x_n|w))$$

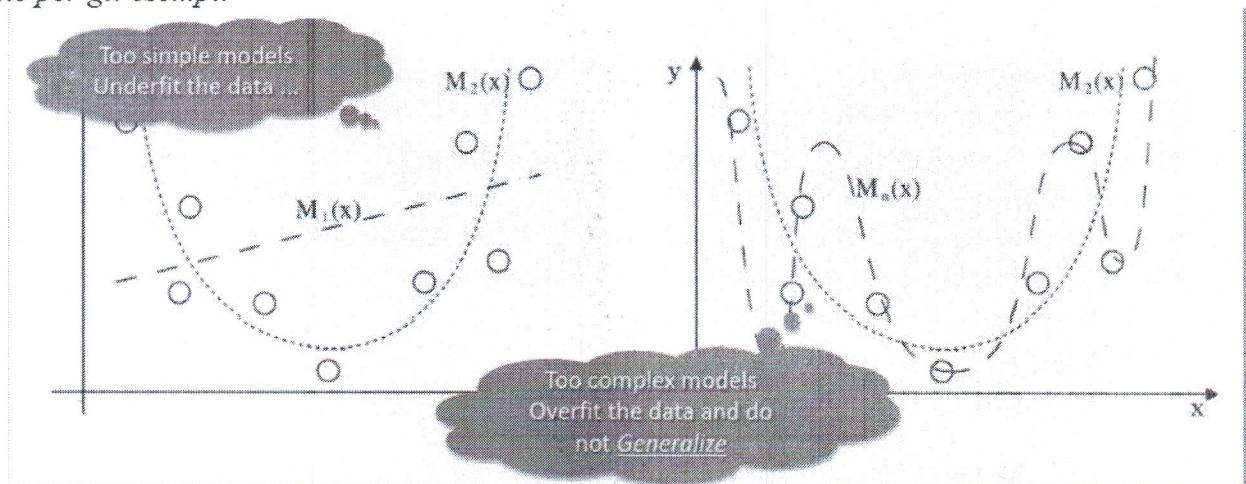
Sum of Squared Errors

Binary Crossentropy

Se usiamo lo square root error, gli errori più grandi diventano più importanti.
In alcuni casi conviene cambiare i dati invece che la funzione di errore.

Affrontare l'overfitting.

Una soluzione che approssima quella obiettivo su un insieme abbastanza grande di dati l'approssimerà anche per gli esempi.



Come misurare la generalizzazione?

Tenersi un po' di dati indipendenti per la fase di test, in modo da stimare l'errore su nuovi dati →
→ **Cross-validation**

Quando si dividono i dati per testing e training bisogna stare attenti che entrambi gli insiemi abbiano la stessa distribuzione. Per farlo, ad esempio, in classificazione, si usa il campionamento stratificato, cioè si prendono un po' di dati da ogni classe e si pucciano nell'insieme di testing ed in quello di training. **L'overfitting** consiste nel allenare troppo una rete neurale, che avrà errore ottimo in fase di training, ma si comporterà molto male in fase di test.

Tipi di cross-validation:

- Tenere buona parte dei dati per il test (si può fare solo con grandi dataset);
- LOOCV: Leave-one-out-cross-validation. Tieni 1 dato fuori ad ogni iterazione, utile quando si hanno a disposizione pochi dati. L'errore di test poi è stimato come la media degli n errori risultanti.
- K-fold: ottimo trade-off, come LOOCV ma gli insiemi sono di k cardinalità.

Come prevenire l'overfitting?

Ci sono più modi, il preferito dal prof è **l'early stopping**:

In pratica si divide il dataset in 3 insiemi: **Training & Validation** (circa 70-80% & 20-30%), Testing. Prima si fa il training, dopodiché si usa l'insieme di validazione per validare il risultato e capire se vale la pena continuare l'allenamento o meno. L'errore di validazione è praticamente una stima dell'errore che avrò in fase di test.

Come capire però quando fermarsi effettivamente?

- O continuiamo il training finché l'errore del training si stabilizza e quindi sappiamo che l'errore minimo di validazione trovato era un minimo globale;
- Oppure decidiamo che se dopo 10 epoche il valore trovato è ancora minimo, quello è un minimo soddisfacente.

Se eliminiamo il rischio di overfitting, perché non usare un sacco di neuroni?

Perché è vero che all'aumentare di neuroni in un livello l'errore di validazione tende a diminuire, ma arriva un punto in cui non conviene più farlo perché siamo costretti a fermarci troppo presto e quindi non riusciamo a minimizzare efficacemente l'errore di training. Come fare a capire qual è il numero giusto? → Trial & Error.. li provo tutti!

Hyperparameters: parametri che descrivono la struttura della rete neurale: numero di livelli, numero di neuroni per livello etc.

Regolarizzazione dei pesi.

Se abbiamo un dataset piccolo e non vogliamo privarci di dati per la validazione possiamo usare altri metodi come la regolarizzazione dei pesi.

La regolarizzazione consiste nel limitare la *libertà* del modello, basandoci su assunzioni fatte a priori sul modello stesso, con l'obiettivo sempre di ridurre l'overfitting.

Per farlo usiamo la statistica Bayesiana: in genere sappiamo che la generalizzazione migliora (e quindi l'overfitting si riduce) quando ho **pesi piccoli**. Quindi aggiungo un parametro nella funzione di errore che cerca di prevenire che i pesi crescano troppo in fretta.

$$= \operatorname{argmin}_w \underbrace{\sum_{n=1}^N (t_n - g(x_n|w))^2}_{\text{Fitting}} + \gamma \underbrace{\sum_{q=1}^Q (w_q)^2}_{\text{Regularization}}$$

Gamma serve per toglierci la varianza e vale: $\sigma^2 / \sigma^2 w$

In pratica cerco di mettere più pesi possibili a 0!

Come capire quale valore di gamma usare? Indovina un po'.. cross-validation e TRIAL & ERROR!

Un valore di gamma troppo basso produce overfitting (la regolarizzazione non ha effetto), un valore troppo alto castra troppo i pesi e la curva diventa troppo "morbida".

Questo algoritmo ci permette di poter utilizzare tutti i dati per il training.

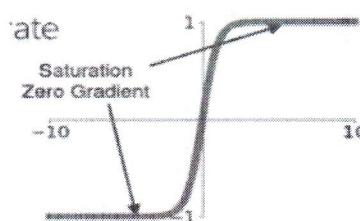
Dropout.

Ad ogni epoca spegniamo randomicamente alcuni neuroni, in questo modo gli altri sono costretti ad imparare una caratteristica indipendente e non si affidano completamente alle altre unità (co-adattamento), ad ogni epoca alleno un modello leggermente differente. Alla fine della fiera ho tanti modelli diversi e ne faccio la media. Questo metodo migliora anche la convergenza della discesa con gradiente.

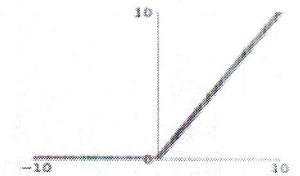
Bisogna ricordarsi che i pesi saranno tarati in base ad un ridotto numero di nodi, quindi saranno più alti. In fase di testing bisognerà usare un coefficiente per correggerlo, se spegno 1/3 dei nodi il coefficiente sarà.. 2/3.

Lezione 24/10/19 - Training tricks and Vanishing gradient

Usando la formula della discesa con gradiente devo moltiplicare per la derivata di ogni neurone che incontro. Se la derivata è tra 0 e 1, i pesi vicini all'input saranno cambiati pochissimo! ($0,5^{\text{numero neuroni}} \approx 0$), al contrario se la derivata fosse più grande di 1 i pesi sarebbero cambiati esponenzialmente.



Nel caso della sigmoide, per esempio, il gradiente della coda sinistra o destra è molto, molto basso (la pendenza è quasi nulla). Le funzioni di sigmoide e Tangente iperbolica, quindi, saturano → il gradiente svanisce.



Per questo motivo hanno introdotto la **ReLU**.

È molto più veloce a convergere e non è lineare. La derivata è 0 o 0 o 1, quindi è molto più facile da calcolare (basta vedere se è maggiore o minore di 0). Disattivando automaticamente una parte di neuroni è come se implicitamente applicasse il dropout.

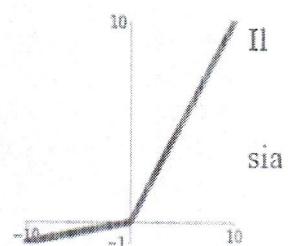
Tra gli svantaggi abbiamo che non è centrata nello zero, quindi se non vogliamo un output che non sia strettamente positivo, non possiamo usarla e dobbiamo usare la sigmoide.

Dying neurons: se per qualche motivo un neurone è disattivato, l'unico modo per riattivarlo sarebbe di modificare il peso che gli entra. Il problema è che l'unico modo per modificare quel peso è usare il gradiente, che però di neurone spento è..0! Quindi il neurone rimane morto.

Il problema non è che un neurone muoia, perché può anche essere che non serva. Il problema è quando succede troppo presto. Succede soprattutto per learning rate molto alti, che appunto spostano il gradiente nella parte piatta della ReLU.

Come fixare questo problema? Usiamo la **Leaky ReLU**, che permette, nel caso si andati troppo a sinistra, di, piano piano, risalire.

La maggior parte delle volte si usa **ReLU o Leaky ReLU**.



Weight initialization (slide 25, minuto 28).

- Zero è un'idea terribile.
- Numeri grandi: potrebbe portare all'explosive gradient. Ma c'è un altro motivo: se abbiamo per esempio una sigmoide e il peso è molto alto, ci troviamo nella parte molto destra della sigmoide. Il gradiente in questa parte (la pendenza) è molto basso, quindi se ci troviamo dal lato sbagliato e vogliamo andare dall'altra parte ci mettiamo **un sacco di tempo**.
- Distribuzione gaussiana con media 0 e varianza 0,01, più grande è la varianza più tempo impieghiamo per convergere. Funziona decentemente soprattutto per piccole reti. Nelle reti grandi c'è il problema che moltiplicando il gradiente passo passo, esso diventa troppo piccolo. Allo stesso modo se il gradiente è troppo grande, passo passo diventa troppo grande.

Xavier initialization. (slide 26, minuto 34 e qualcosa).

Perché i dati in input devono avere media 0? Non è obbligatoria ma è stato osservato che dati con media 0 sono più stabili nel training, è una common practice.

Ogni neurone amplifica N volte, dove N è il numero di inputs.

Quindi la varianza dell'output è uguale alla varianza dell'input, moltiplicata n volte.
Se volessimo la varianza dell'input essere uguale a quella dell'output deve valere che:

$$N * \text{Var}(w_j) = 1$$

Xavier quindi propone di inizializzare i pesi a:

$$w \sim N(0, 1/n_{in})$$

Questo è stato dimostrato essere più veloce nel training e più stabile.

Glorot & Bengio hanno trovato che:

$$n_{out} * \text{Var}(w_j) = 1$$

Quindi propongono:

$$w \sim N(0, 2/n_{in} + n_{out})$$

He dice che Xavier avrebbe ragione se tutti i ReLu fossero attivi (cioè nella parte destra della funzione), ma non è sempre così. Quindi propone:

$$w \sim N(0, 2/n_{in})$$

Gradient descent (slide 29, minuto 51 e qualcosa).

Ci sono un sacco di possibili variazioni, che sono più veloci a trovare la soluzione e trovano anche un diverso minimo. Il più semplice è il **Nesterov Accelerated gradient**.

Normalmente, col momentum, aggiungiamo al gradiente un momentum, un'inerzia.

L'idea è che puoi migliorare questo processo, migliorare la stima, dividendolo in due steps. Prima applico il momentum, cioè vado dove porta il momentum, che indica la direzione, dopodiché stimo in **quel punto** il gradiente, così hai una miglior stima della funzione, in genere è più efficiente.

Altri algoritmi cercano di adattare il learning rate. Questo perché il gradiente diminuisce man mano che arriviamo all'input. L'idea è che ogni livello dovrebbe avere il suo learning rate, questo processo è praticamente impossibile. Ci sono degli algoritmi (AdaGrad e Adam) che cercano di farlo: per esempio se ci accorgiamo che il gradiente si muove troppo in fretta, probabilmente il learning rate è troppo alto. Tutte sono funzioni di primo grado, al più velocizzano il Gradient Descent.

Ci sono anche quelle di secondo grado che utilizzano la curvatura della funzione. Creano la parabola e saltano al vertice, moooooolto più veloce.

Il problema è che dobbiamo farci la matrice hessiana, che ha grandezza quadratica rispetto al numero di parametri, quindi la maggior parte delle volte non entra in memoria.

Batch Normalization (slide 32, minuto 01:04)

Le reti convergono più in fretta se vengono normalizzate, cioè hanno media zero e varianza unitaria (in pratica stanno in un quadrato nel piano cartesiano).

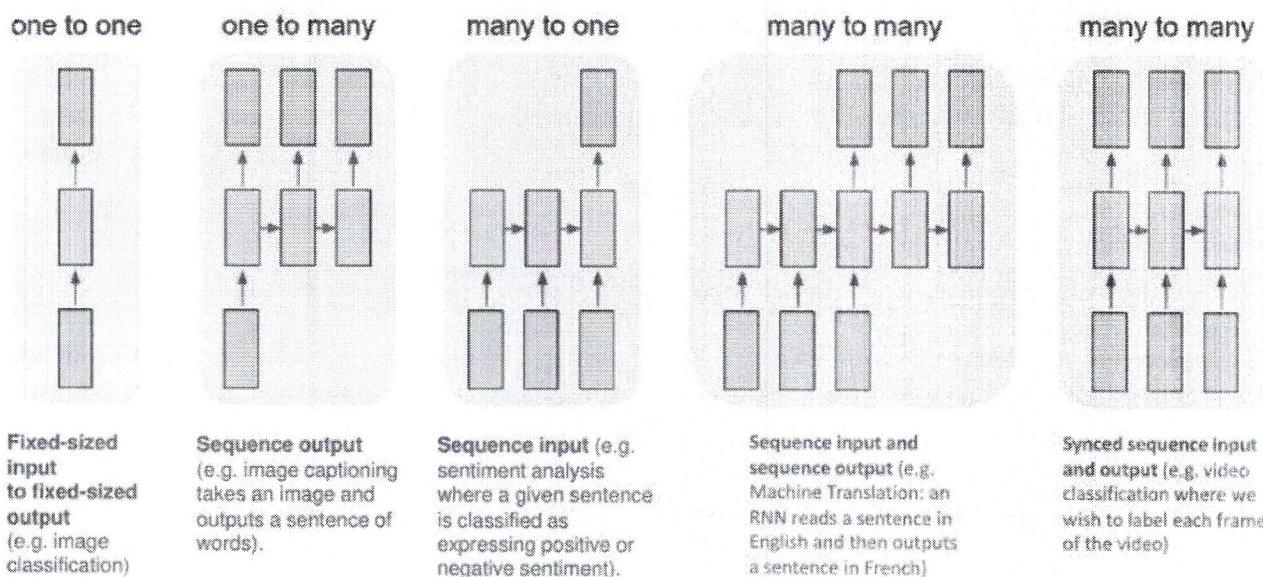
Ci è utile che ciò accada anche nei livelli intermedi e quindi aggiungiamo un livello di batch normalization. La normalizzazione è differenziabile. La media e la deviazione standard è calcolata per ogni minibatch.

Si è visto che la batch normalization migliora il flusso del gradiente, possiamo usare learning rates più alti (e quindi apprendere più velocemente). Alle volte, invece, è stato visto che alcune reti funzionano meglio senza batch normalization.

Recurrent Neural Network

Una RNN è una classe di reti neurali dove le connessioni tra i nodi formano un grafo direzionale su una sequenza temporale. In pratica ciò permette di esibire un comportamento dinamico a livello temporale. Rispetto agli FFN, le RNN utilizzano il loro stato interno (memoria) per processare sequenze di input.

Sequential Data Problems



Credits: Andrej Karpathy

In genere, per evitare di dover computare il gradiente su sequenze troppo lunghe, la backpropagation è troncata.

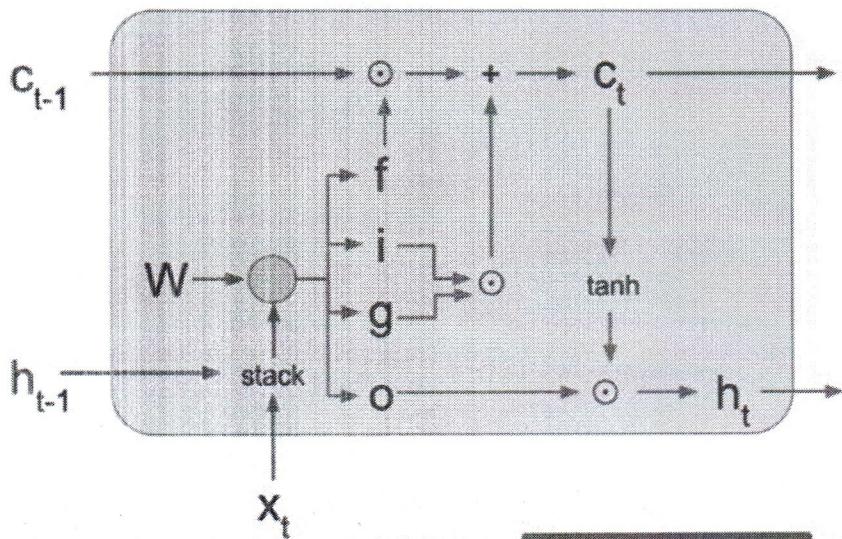
Un modello più avanzato: **attenzione**. La rete CNN invece che produrre un singolo vettore a rappresentare tutta l'immagine, produce una griglia di vettori in cui ognuno riassume una particolare posizione dell'immagine. Dopodiché la rete impara dove porre l'attenzione quando analizza l'immagine.

Problema grave: backpropagando attraverso le celle di memoria, le celle più vecchie devono passare attraverso tutte quelle precedenti e si finisce a moltiplicare il gradiente per gli stessi numeri della matrice dei pesi. In questo modo il gradiente要么 esplode要么 diventa zero (exploding and vanishing gradient problem). Una roba che si usa è il *clipping*, se la norma L2 diventa più grande di una determinata soglia, si divide per due e si continua.

Per il vanishing gradient invece bisogna cambiare l'architettura della RNN. Usiamo le **Long Short Term Memory** che risolvono entrambi i problemi.

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

In pratica si mantengono due stati nascosti di memoria, uno quello normale $h(t)$ (long term?) e l'altro, $c(t)$ (short term?), lo stato di cella. È un vettore che viene tenuto all'interno della cella e non viene esposto.

Ci sono dei gates:

i sta per input gate, dice quanto input vogliamo nella cella (sigmoid)

f sta per forget gate, quanto vogliamo dimenticare quanto sta nella cella di memoria (sigmoid)

o sta per output gate, quanto vogliamo rivelarci al mondo esterno (sigmoid)

g sta per gate gate, quanto vogliamo scrivere nella cella di memoria (tanh)

Insomma possiamo ad ogni time step, ricordarci o dimenticarci lo stato, oppure incrementare e decrementare ogni elemento.

Tentativo di spiegazione della cella di memoria. Prende in input lo stato precedente della cella e quello nascosto (c e h ($t-1$)) più l'input corrente (x). Questo h e x vengono stackati e moltiplicati per la matrice W per formare i 4 gates. Il forget gate viene moltiplicato elemento per elemento per $c(t-1)$ e il risultato sommato per la moltiplicazione elemento per elemento di i e g . Il risultato è $c(t)$, quindi lo stato di cella precedente.

Non solo! Questo C_t viene fatto passare attraverso una tanh e moltiplicato elemento per elemento con o . Il risultato è $h(t)$.

Cosa accade quindi alla backpropagation? Il gradiente passa solo per $c(t)$ e viene moltiplicato solo per il forget gate. Inoltre il forget gate cambia da cella a cella e quindi non abbiamo più i problemi di gradiente che avevamo prima. La forget gate ha valori da 0 a 1 (esce da una sigmoide) e quindi è meglio.

Modello Seq2Seq

(spiegato molto bene qui <https://distill.pub/2016/augmented-rnns/#neural-turing-machines>)

Un modello sequence-to-sequence è un modello che prende in ingresso una sequenza di oggetti (parole, lettere, caratteristiche di immagini) e manda in output un'altra sequenza di oggetti.

Segue l'architettura classica di encoder-decoder: L'encoder processa ogni oggetto della sequenza di input e compila le informazioni in un vettore, chiamato **contesto**. Dopo aver terminato il processamento dell'intera sequenza l'encoder manda il contesto al decoder che comincia a produrre la sequenza di output oggetto per oggetto.

La grandezza del vettore contesto è praticamente il numero di unità nascoste dell'encoder RNN.

Il modello base di seq2seq si rivelò inefficace nel caso di lunghe sequenze perché l'encoder invia al decoder solo l'ultimo hidden state (frutto del processamento di tutti i precedenti). È qui che entra in gioco il concetto di **attenzione**. L'encoder invia al decoder tutti gli hidden states. Il decoder, poi, impara a quale oggetto della sequenza deve porre più attenzione per produrre il corretto output.

Il decoder sceglie l'output, tra tutte quelli possibili, usando il softmax.

In genere rappresentiamo le parole utilizzando bigrammi e trigrammi, che sono meno di tutte le parole presenti nel dizionario. Alle volte abbiamo bisogno di caratteri speciali, come il carattere di inizio computazione, di inizio e fine stringa, di parola sconosciuta (che accelera la computazione nel caso di parole presenti poche volte nel dizionario o non presenti affatto) e di padding, che servono per fare in modo che tutti i vettori abbiano la stessa lunghezza (le batch devono essere della stessa dimensione).

Ci può essere utile per capire il senso delle frasi, per esempio negli audio o in presenza di ironia, fare una passata da sinistra verso dietro ed una al contrario.

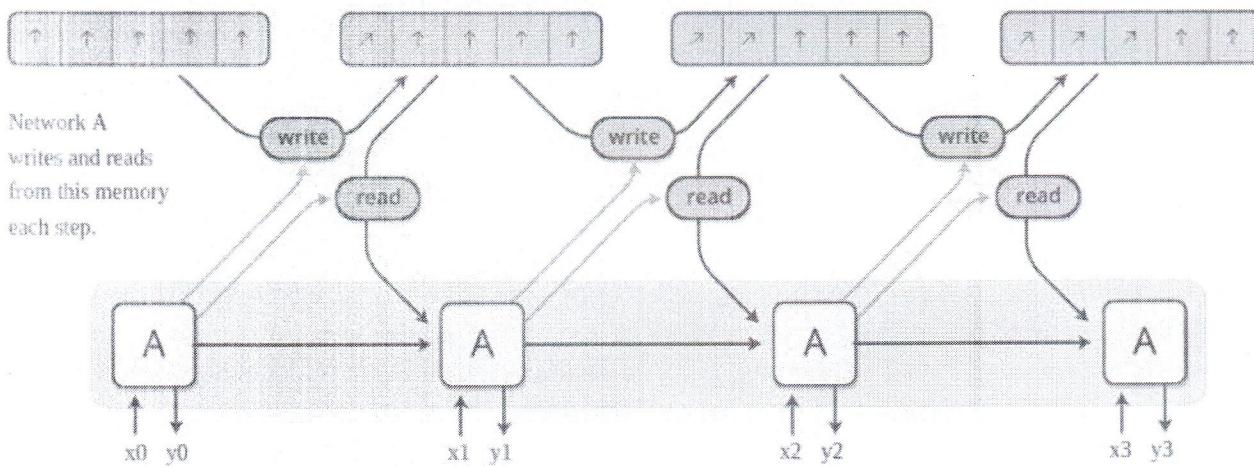
Sono stati create più estensioni alla RNN di base. Alcuni esempi interessanti:

- **Neural Turing Machines:** esse hanno una memoria esterna da cui possono leggere e scrivere. La RNN impara da dove leggere e scrivere,. In realtà la rete dà in output un vettore di attenzione che indica quanto, da ogni cella, si vuole leggere o scrivere. Quindi la lettura non è altro che una media pesata, in base all'attenzione, di tutte le celle di memoria. Stessa cosa per la scrittura. Ciò si rende necessario per rendere la funzione di attenzione differenziabile.

La NTM usa l'attenzione in 2 modi diversi:

- basata sui contenuti: per trovare in memoria quello che si sta cercando;
- basata sulla posizione: per permettere al modello di spostarsi nella memoria ed effettuare loops.

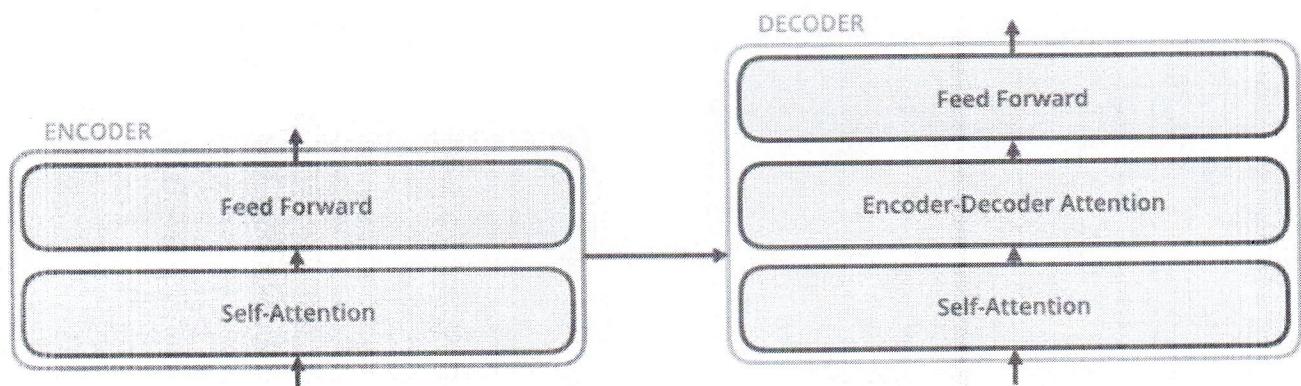
La NTM può imparare a scrivere in memoria lunghe sequenze, a ciclare e ripetere sequenze o a ordinare dei numeri.



- **Attentional Interfaces:** le reti neurali possono imparare a concentrarsi su un sottoinsieme delle informazioni che vengono date loro. Per rendere l'attenzione differenziabile usiamo lo stesso trucco delle NTM viste sopra, cioè poniamo l'attenzione su tutto ma a diverso grado. Si può usare questa rete per la traduzione, per il riconoscimento vocale, per i chatbot etc.

Transformer: la struttura è la medesima: encoder-decoder; tuttavia si impilano un certo numero di encoders (6 nel paper) e lo stesso numero di decoders. Gli encoders sono identici ma non condividono i pesi. Ogni encoder è diviso in 2 sottolivelli: un livello di “auto-attenzione” che aiuta l’encoder a guardare ad altre parole della sequenza di input mentre codifica la specifica parola, un livello di Feed Forward NN che riceve in ingresso il risultato del livello di attenzione.

Il decoder ha un ulteriore livello in mezzo ai 2: un livello di attenzione che aiuta il decoder a concentrarsi sulle parti rilevanti della sequenza in ingresso.



Il concetto di “auto-attenzione” è molto semplice da comprendere. Immagina di voler tradurre la frase:

“The animal didn't cross the street because it was too tired”

Per cosa sta "it" per "street" o "animal"? Il livello di auto-attenzione associa "it" a "animal". Nel decoder l'attenzione funziona in modo leggermente diverso, ossia gli è permesso di prestare attenzione solo a parole in posizioni precedenti.
Il transformer ha più "teste" di attenzione, che si concentrano su parti differenti della sequenza di input. Inoltre il transformer riesce a tenere conto della posizione delle parole nella sequenza. Per farlo aggiunge ad ogni embedding dell'input un vettore. Questo vettore aiuta il modello ad imparare un pattern che gli permette di determinare la posizione assoluta della parola nella sequenza, oppure permette di conoscere la distanza tra due parole.

L'ultimo livello è composto da un livello Lineare e un livello Softmax. Il livello lineare altro non è che una semplice FCNN che proietta il vettore prodotto dai decoders in un vettore della grandezza del nostro dizionario.

Il softmax poi si occupa di trasformare i punteggi in probabilità (tutti positivi e la cui somma è 1). La cella con più alta probabilità viene scelta come parola associata all'output corrente.

Word embedding

L'embedding indica una tecnica per mappare una parola o una frase, dal suo spazio d'origine di grande dimensione in uno spazio vettoriale numerico più piccolo. Inoltre le parole di significato simile formano dei clusters. È molto importante perché a differenza di un'immagine o di un audio, le parole sono sparse e con grande dimensionalità (pensa ad una codifica one-hot dell'intero dizionario..).

Il concetto dietro è che ogni parola è identificata dal contesto in cui è usata. Due parole sono simili se le altre parole che vengono usate nel contesto sono uguali (o simili).

Uno dei metodi utilizzati per il word embedding è **Word2Vec**. È un metodo statistico per imparare efficacemente l'embedding di una parola da una parte di testo. Usa un modello meno profondo che fornisce prestazioni migliore di 1000 volte. Questo permette di allenare il modello su un dataset molto più ampio. Inoltre riesce a capire la sintattica e la semantica del linguaggio, per esempio capisce che "re" è il maschile di "regina".

Ci sono due modelli che possono essere utilizzati:

Continuous Bag-of-Words, che predice la parola corrente in base al contesto;

Continuous Skip-Gram, che predice il contesto in base alla parola corrente.

Sbominatura da schiaffarci dentro:

Word embedding finding a space where you can represent words. An embedding is a representation. Le parole sono complesse. Con regole per definire il loro significato. Siccome una parola è un oggetto in un dizionario, la rappresentazione di una frase, per esempio di due parole, è un punto in uno spazio

di 2 dimensioni, molto sparsa. C'è la maledizione della dimensionalità: la complessità esponenziale in termini di dati per stimare una distribuzione multidimensionale.

Come codifichiamo il testo? Il word embedding è iniziato con N-gram. È praticamente un modo per rappresentare un linguaggio. In pratica cerca di capire la parola successiva in base alle N parole precedenti, insomma la probabilità condizionata di una parola in base alle N precedenti. La complessità è N^N . Come funziona? Semplicemente guardando tutto il testo e calcolandosi le probabilità. Il problema di questo modello è che N non può essere troppo grande perché diventa computazionalmente troppo complesso. Servono anche un sacco di dati per avere una buona accuratezza, almeno 100 volte il numero di slots (n parole $\wedge N$). Più testo abbiamo, più parole ci sono e più aumenta la complessità. Per cui si è riusciti ad arrivare al massimo ad $N = 3$. Quindi il modello non riesce a prendere un contesto abbastanza grande.

Come usiamo l'autoencoder? Cerchiamo di comprimere le parole in uno spazio più piccolo dove però, due parole con semantica simile dovrebbero essere trattate come effettivamente come simili, quindi se decomprimendo usiamo una parola per un'altra non c'è problema.

Bengio: l'idea era di usare un deep autoencoder per costruire un modello di linguaggio non lineare e continuo. Cerca di capire qual è la parola successiva in base alle n-1 precedenti. Cerca di trasformare il modello di linguaggio come un problema di classificazione con una NN. In pratica comprimono il dizionario di V parole, con dimensione di input ($n-1 * |V|$), in un vettore reale tra le 500 e le 1000 e, in base a questo, cercano di predire la parola successiva. 24% improvement on n-gram. Il problema è che non lo si può usare su grandi corpus a causa della complessità e non funzionava troppo bene con parole rare.

Computer vision - Boracchi

(non ho seguito le lezioni, sono appunti presi da Coursera + slides di Stanford su cui è basato il corso).

Classificazione delle immagini.

Un primo approccio potrebbe essere quello di considerare la distanza pixel-pixel tra l'immagine test e quella del training set. L'etichetta da assegnare sarebbe quella delle K immagini del training set più vicine.

L'approccio non è adeguato per le immagini per 2 motivi:

1. la somiglianza tra i pixel è un concetto diverso dalla somiglianza percepita.
2. l'algoritmo è molto veloce nella parte di training ma molto lento in fase di testing, che è il contrario di quello che vorremmo.

Classificatore lineare.

Si crea un vettore $K(x)$, di dimensione L (numero di etichette). $[K(x)]_i$ rappresenta il punteggio dell'immagine x di appartenere alla classe i . Si divide quindi l'immagine per colonne e per ogni colonna si moltiplica per i pesi W e si somma il bias b . Il risultato è appunto il vettore $K(x)$. Il classificatore assegna come etichetta quella che corrisponde al punteggio massimo del vettore.

In pratica il classificatore crea un template, un modello, che rappresenta al meglio che può l'etichetta. I classificatori lineari sono i livelli più importanti di una rete neurale. Un singolo livello però non basta.

Convolutional Neural Networks.

Le immagini diventano troppo grandi. Se abbiamo un'immagine 1000x1000, 3 colori, vuol dire che abbiamo 3 milioni di nodi input e che quindi i pesi entranti nel primo livello della rete neurale sono addirittura 3 MILIARDI! Per questo utilizziamo le Convolutional Neural Network.

Una CNN è formata da tanti livelli, di diverso tipo:

- Livelli di convoluzione: usano i filtri per trovare delle caratteristiche nelle immagini;
- Livelli di pooling: downsampling, servono per ridurre la dimensione del volume e stabilizzare la funzione;
- Livelli fully-connected.

Un filtro è una matrice più piccola che utilizziamo per convolvere la matrice di input; ogni filtro serve per catturare una determinata caratteristica dell'immagine. La matrice iniziale viene divisa in tante sottomatrici della grandezza di quella di filtro, ogni cella in posizione (x,y) della sottomatrice viene moltiplicata con la cella nella stessa posizione (x,y) della matrice di filtro; dopodiché si somma tutto ed il risultato costituisce una cella della matrice destinazione, che, ovviamente, sarà più piccola. Da una matrice 6x6 con un filtro 3x3, viene fuori una matrice 4x4. $(n - f + 1) * (n - f + 1)$

La profondità rimane la stessa. Possiamo usare più filtri in un unico livello.

(In realtà quella che stiamo facendo è un'operazione di cross-correlazione, per la convoluzione dovremmo ribaltare la matrice originale. Lo dico, che poi i matematici si arrabbiano).

Un filtro può essere utilizzato per creare un edge-detector, per esempio verticale o orizzontale.

I numeri da mettere nel filtro possono essere considerati come parametri, pesi, e imparati direttamente dal modello. Possono imparare magari anche spigoli a diversa angolazione.

Ci sono due problemi:

- 1) magari non vogliamo che l'immagine diventi troppo piccola andando avanti con i filtri;
- 2) i pixel sui limiti dell'immagine vengono considerati meno, perché ci sono meno convoluzioni che li includono.

Padding: aggiungere un contorno all'immagine per risolvere entrambi i problemi. Generalmente tutte le celle vengono riempite con 0. Padding = 1, significa aggiungere 1 pixel per lato. ($n + 2p - f + 1$). Quanto usare il padding? Valid convolution → no padding; Same → pad size che fa in modo che output size = input size → $p = (f-1)/2$. Per convenzione f è solitamente dispari.

Strided convolutions: significa saltare un pezzo della matrice di input. Uno stride di 2 significa che se prima avevamo il filtro centrato nella colonna 2, dopo non va alla 3, ma alla 4. Quindi la matrice di output è più piccola: $(nxn) * (fxf)$ p padding, s stride → $[(n+2p-f)/s] + 1$. Nel caso in cui non sia un intero si prende solo la parte intera.

Pooling layers (supersampling): serve per ridurre l'input e stabilizzare la funzione. Se vogliamo fare un max pooling di una matrice $4x4$ in una $2x2$, dividiamo la matrice in 4 quadrati e prendiamo il max di ognuno di essi. In pratica $f = 2$ e $s = 2$. A cosa serve? Se becco una caratteristica che mi interessa, che si traduce in un numero molto alto in uno dei quadrati, prendendo il max conservo questa caratteristica ma riduco la matrice. Nel livello di pooling non c'è nessun parametro da imparare! In genere col max pooling non usiamo padding.

Fully connected layer: connette ogni neurone di un livello ad ogni neurone dell'altro livello. (stesso principio del multi-layer perceptron PLP).

Calcolo del numero di parametri: $f=5$, 3 canali, 8 filtri → $(5*5*3+1)*8$. 1 è il bias.

Due proprietà delle CNN:

Weight sharing: tutti i neuroni alla stessa profondità usano gli stessi pesi e lo stesso bias, in questo modo si riduce il numero di parametri della CNN. L'assunzione è che se è utile computare una caratteristica in una posizione allora è utile probabilmente anche in un'altra parte di un'immagine.

Sparsità di connessioni: in ogni livello, ogni valore di output dipende solo da un piccolo numero di input.

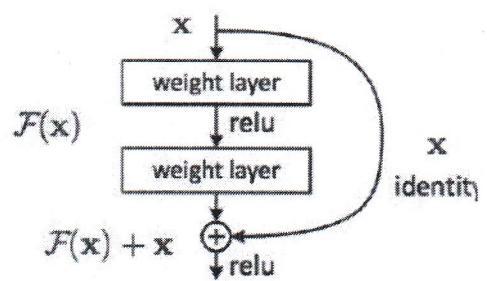
Data Augmentation: se abbiamo poche immagini e vogliamo comunque avere una buona generalizzazione, modifichiamo alcune immagini o con trasformazioni geometriche, cioè ruotandole, scalandole o capovolgendole, oppure con trasformazioni fotometriche, ossia cambiando l'illuminazione, il contrasto o aggiungendo rumore etc. In questo modo il CNN pensa che ci siano più immagini e si adatta anche ad immagini diverse dal "solito".

Alcune reti neurali famose:

- **AlexNet** (2012) introduce la ReLu.
- **VGG16** (2014): variante dell'AlexNet più profonda e con filtri molto più piccoli (3x3). 3 filtri 3x3 hanno lo stesso campo ricettivo di un filtro 7x7 ma con più non linearità e meno parametri.
- **GoogLeNet** (2014): si basa sull'idea che una caratteristica può presentarsi in un'immagine in dimensioni diverse. Usa quindi più filtri di grandezza differente e li concatena. Per evitare l'aumento esponenziale della profondità si aggiungono dei "livelli di bottleneck" formati da convoluzioni 1x1 che servono a diminuire la profondità ad ogni livello e quindi ridurre il costo computazionale.
- **ResNet** (2015): rivoluzione della profondità: 152 livelli.

Avevano notato che una rete più profonda performa peggio di una meno profonda, nonostante non ci fosse overfitting. Hanno pensato che questo fosse un problema di ottimizzazione, perché, per costruzione, doveva performare almeno uguale. Reti più profonde sono più difficili da allenare per il problema della scomparsa del gradiente.

Funziona in questo modo: il risultato della convoluzione viene sommato con la stessa funzione di input (identità). Quindi invece di calcolare di volta in volta la funzione che vogliamo imparare $H(x)$, la dividiamo in $F(x) + x$. x è quello che abbiamo già imparato, l'input, mentre $F(x)$ è quello che impariamo da questo livello, è chiamato residuo (qualcosa da aggiungere, appunto, all'identità x). $H(x) = F(x) + x$. In pratica ci calcoliamo il delta.



Transfer learning: usare un modello già allenato per la ricerca delle features e cambiare solo la parte di FC per adattarle al problema in questione. In realtà ci sono due opzioni:

- Solo la parte di FC viene allenata: freezare i pesi della parte convolutiva e riallenare l'intera rete. Utile quando ci sono pochi dati ed il modello già allenato funziona per il problema che abbiamo.
- Tutto il CNN è riallentato ma i livelli convolutivi sono inizializzati con i pesi del modello già allenato. Utile quando abbiamo tanti dati o non ci aspettiamo che il modello preimpostato vada bene.

Ci sono 4 tipi di task di vision:

La **classificazione** che abbiamo già affrontato, l'**instance segmentation** che è una generalizzazione della semantic segmentation che riconosce le diverse istanze di uno stesso oggetto.

Nella **Semantic Segmentation** non ci sono oggetti, solo pixels. In input l'immagine ed in output le decisioni della categoria di ogni pixel dell'immagine. **Non divide tra istanze** dello stesso oggetto.
Approcci:

- Sliding windows: croppare l'immagine in pezzi e dare un'etichetta per il pixel centrale. Molto molto inefficiente. Dobbiamo provare tantissime combinazioni.

- **f-CCN - Fully Convolutional Neural networks**

Il CNN funziona solo con immagini di dimensioni fisse e prestabilite. Il problema è la parte di fully connection che ha bisogno di un input fisso. I filtri invece vanno bene con qualsiasi dimensione.

Ma siccome il livello completamente connesso è lineare possiamo esprimere con una convoluzione!

Se N sono gli input al livello FC e L sono gli outputs il livello convolutivo avrà L filtri di dimensione $1 \times 1 \times N$.

Invece che classificare gli oggetti dell'immagine abbiamo una heatmap in cui ogni livello di profondità ci dice la probabilità che un determinato pixel (in realtà del campo recettivo di ogni pixel) appartenga ad una determinata categoria (heatmap).

Usare direttamente questo metodo per la segmentazione produce un **risultato molto grossolano**.

Ci sono **due problemi** quando facciamo segmentazione: per capire il contenuto di un'immagine bisogna andare in profondità in modo da ottenere un'informazione che appartiene a molti pixel, d'altra parte per avere una stima accurata, quindi scontornare bene le gli oggetti, dobbiamo tenere l'immagine alla più alta risoluzione possibile.

Facciamo downsampling all'inizio (ricerca delle features) e **upsampling** verso la fine (ottenere contorni definiti).

Come si upsampla? Con un procedimento chiamato **unpooling**:

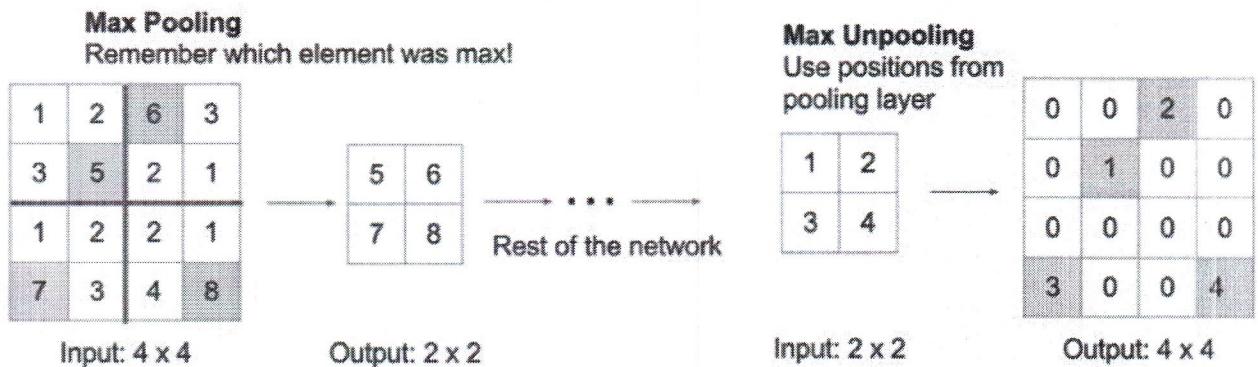
- **Nearest Neighbor unpooling**, in pratica prendiamo una cella e la duplichiamo formando un quadrato di 4 celle.
- **Bed of nails**: lo stesso quadrato invece che duplicato riempito di zeri a parte la cella iniziale.

Nearest Neighbor																	
<table border="1"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4	→												
1	2																
3	4																
	<table border="1"><tr><td>1</td><td>1</td><td>2</td><td>2</td></tr><tr><td>1</td><td>1</td><td>2</td><td>2</td></tr><tr><td>3</td><td>3</td><td>4</td><td>4</td></tr><tr><td>3</td><td>3</td><td>4</td><td>4</td></tr></table>	1	1	2	2	1	1	2	2	3	3	4	4	3	3	4	4
1	1	2	2														
1	1	2	2														
3	3	4	4														
3	3	4	4														
Input: 2 × 2																	
Output: 4 × 4																	

"Bed of Nails"																	
<table border="1"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4	→												
1	2																
3	4																
	<table border="1"><tr><td>1</td><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>0</td><td>4</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	2	0	0	0	0	0	3	0	4	0	0	0	0	0
1	0	2	0														
0	0	0	0														
3	0	4	0														
0	0	0	0														
Input: 2 × 2																	
Output: 4 × 4																	

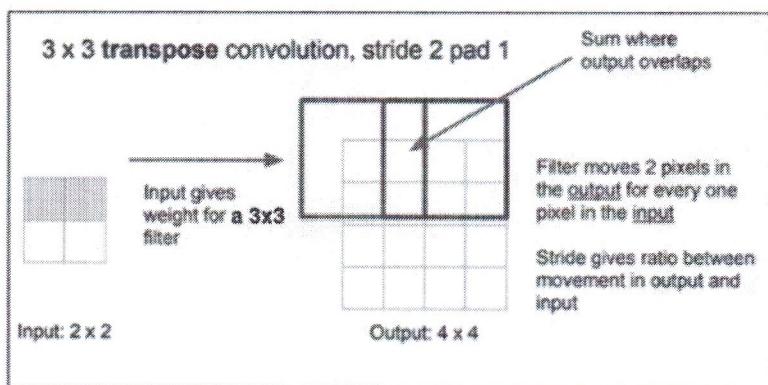
- **Max unpooling:** sfruttiamo la simmetria della rete. Ci ricordiamo le posizioni dei massimi nel pooling iniziale e lo sfruttiamo per l'unpooling. Bed of nails ma mettiamo il dato nella posizione che ci siamo ricordati.

In-Network upsampling: “Max Unpooling”



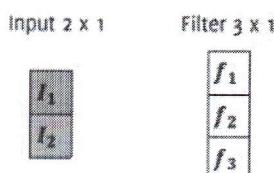
- **Transpose convolution:** in pratica l'input fornisce dei pesi che usiamo per moltiplicare i valori del filtro e scriverli nell'output. Se l'output si sovrappone sommiamo. L'idea è quella che la rete neurale "impara" a fare l'upsampling.

Transpose Convolution



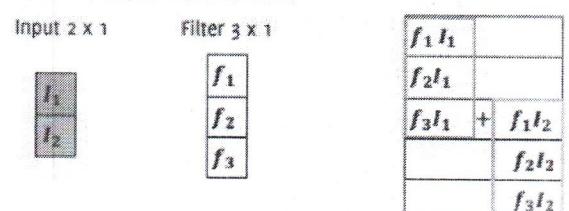
Transpose Convolution

Transpose convolution with stride 1



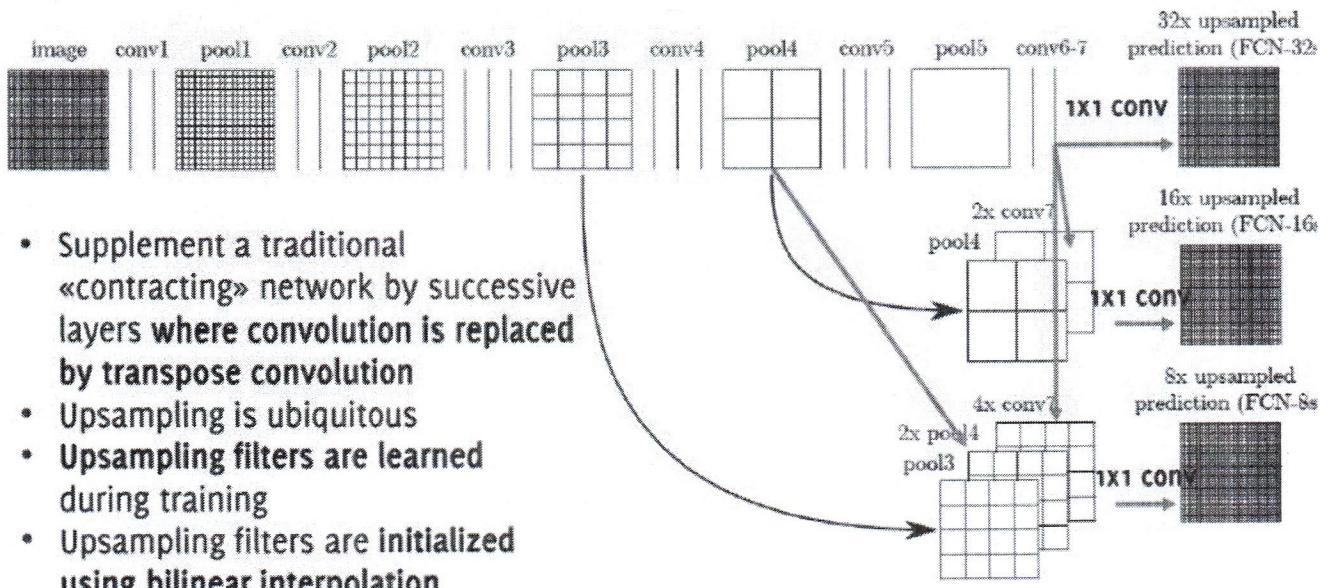
Transpose Convolution

Transpose convolution with stride 2



Ultima soluzione: saltare delle connessioni. Si creano 3 reti, la prima alla risoluzione più bassa, i pesi sono usati per inizializzare la seconda a risoluzione più alta e stessa cosa per la terza.

Solution: Skip Connections!



Global Averaging Pooling Layers (GAP)

Invece di usare un livello FC alla fine della rete, il GAP computa la media di ogni feature map.

L'idea di fondo è che i livelli FC hanno tanti parametri e quindi tendono a overfittare.

I GAP non hanno parametri da ottimizzare, quindi overfittano meno. Sono più robusti alle trasformazioni spaziali delle immagini e possono essere usati con immagini di dimensioni differenti. La classificazione è effettuata da un livello softMax alla fine del GAP (attenzione, il numero delle feature maps deve essere uguale a quello delle classi in output).

Localizzazione.

La localizzazione consiste nel trovare tutte le istanze delle classi in un'immagine.

Un primo approccio può essere quello di effettuare una classificazione per ogni regione dell'immagine. Il problema è che questo approccio è computazionalmente molto costoso. Perché le immagini possono essere in qualsiasi dimensione e forma.

Così si possono usare delle tecniche di computer vision che, per ogni immagine, producono ~2K regioni di interesse, dove è probabile che si trovi un oggetto.

R-CNN funziona proprio così. Le regioni di interesse sono ridimensionate perché la rete neurale prende in ingresso immagini della stessa dimensione. Per ogni regione di interesse si decide se fa parte

di una delle classi oppure del background. Inoltre predice 4 numeri di offset che servono per spostare le regioni di interesse, nel caso in cui le regioni non fossero in posizione perfetta.

Anche questo è molto lento. Per questo è nato **Fast R-CNN**.

L'idea è la stessa, però l'immagine viene prima fatta passare attraverso un CNN, e le regioni di interesse vengono prese dalla feature map generata. In questo modo la convoluzione, che è costosa, viene fatta una sola volta.

Il bottleneck in questo modo diventa la computazione delle regioni di interesse. **Faster R-CNN** risolve il problema facendo computare le regioni di interesse direttamente al CNN.

Modelli generativi

L'obiettivo è, date delle immagini in ingresso, generare nuove immagini che siano simili a quelle di partenza (stessa distribuzione).

Possiamo utilizzarlo per colorare, per creare nuove immagini e per la super-resolution.

Un autoencoder è una rete neurale che impara a copiare il suo input nel suo output. Ha all'interno un livello nascosto che descrive un codice usato per rappresentare l'input. È formato da due parti: l'encoder mappa l'input nel codice ed il decoder mappa il codice ad una ricostruzione dell'input originale (come se fosse una compressione - decompressione). L'errore quindi è quello di ricostruzione, ossia quanto l'output differisce dall'input; errore che cerchiamo di minimizzare.

L'autoencoder cerca inoltre di mantenere la rappresentazione sparsa, cioè cerca di settare quanti più pesi possibili a 0.

Gli autoencoder non servono per generare dati ma è un metodo di apprendimento non supervisionato per imparare alcune caratteristiche da dati non etichettati.

I pesi trovati dall'encoder possono essere usati per inizializzare un classificatore, questo riduce il rischio di overfitting, soprattutto nel caso di piccolo dataset.

La sfida più grande è definire una loss adatta a determinare se l'output sia un'immagine realistica o no. Per questo nascono i **Generative Adversarial Networks**.

In pratica si allena una coppia di reti neurali che hanno funzioni differenti e "competono" come in una sorta di gioco con due giocatori. Il **generatore** cerca di produrre immagini realistiche avendo in input del rumore randomico, cercando di truffare il **discriminatore**. Il secondo cerca di capire se l'immagine in ingresso è reale o no. Si allenano entrambe e alla fine si tiene solo G.

Il training avviene con i soliti strumenti: backpropagation e dropout.

Prima si allena il discriminatore, che non è altro se un classificatore. Dopodiché si allena il generatore che manda delle immagini fake al discriminatore. Esso classifica le immagini e calcola la loss di classificazione. Questa loss viene backprogata attraverso discriminatore e generatore per ottenere i gradienti ed essi vengono usati per cambiare **solo** i pesi del generatore.

Il training delle due parti può anche essere alternato.

Il training è abbastanza instabile. Questo accade perché più migliora il generatore, meno il discriminatore riesce a classificare correttamente le immagini. In questo modo il suo responso diventa randomico e quindi dà feedback randomici al generatore, che comincia ad allenarsi su responsi poco attendibili.