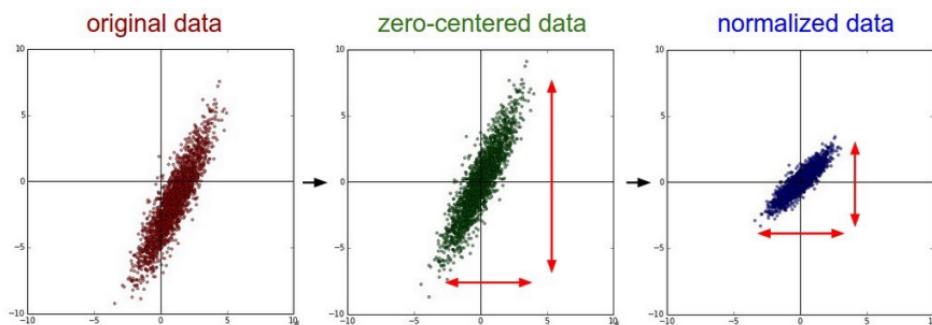


Lezione 7 - CNN - 18/11/2024

Data pre-processing

Di solito abbiamo dati che non sono normalizzati. Di solito la randomizzazione iniziale dell'algoritmo accade in un punto dove il gradiente non è alto. L'idea è quella di centrare i dati in una posizione dove il gradiente è più alto, così da avere una velocità di convergenza più alta.

- **Local mean subtraction:** subtract the mean from original data
 - $X = X - \mu$
- **Normalization:** scale original data to a specific range
 - $X = \frac{X-\mu}{\sigma}$
 - ...



L'idea è quella di centrare i dati sullo 0.

Visto che nelle immagini avremo una media di 127, e una deviazione standard anche di 10 o 20, preferiamo normalizzare i dati. In questo modo abbiamo i dati che sono molto compatti nel range -1 +1, che è preferito per le funzioni di attivazione.

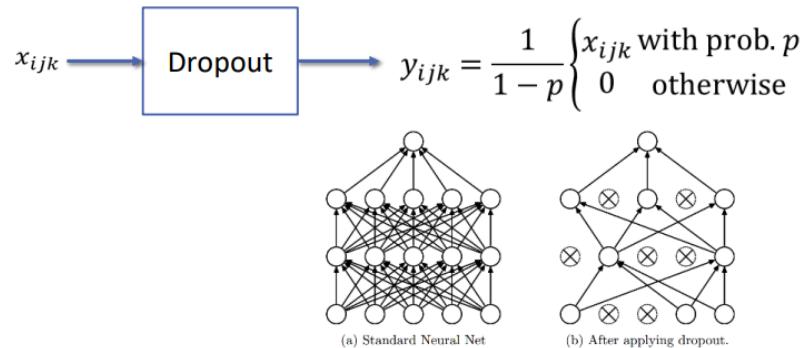
Può succedere che visto che abbiamo molti parametri nelle reti convoluzionali, potremmo avere problemi di overfitting. Si possono usare gli stessi modi usati nelle NN tradizionali, ovvero tramite:

- Weight decay (penalizza i pesi grandi)
- Dropout (applicato solo in fully-connected layers, non quelli convoluzionali).

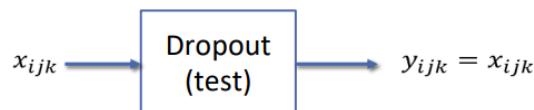
- Data augmentation

Dropout

Randomly switches off individual neurons



At test time, replaced by identity



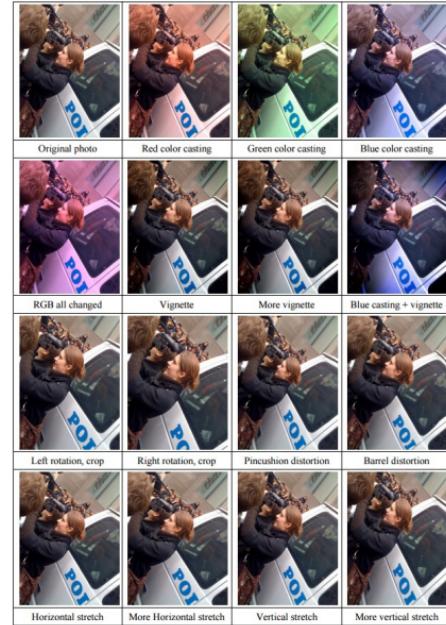
Data augmentation

Creiamo più immagini, facendo delle trasformazioni di modo che le labels siano sempre le stesse.

Augment the training set
by “jittering” samples

Label preserving image
transformations

- Horizontal flip
- Random crop
- Color casting
- Geometric distortion



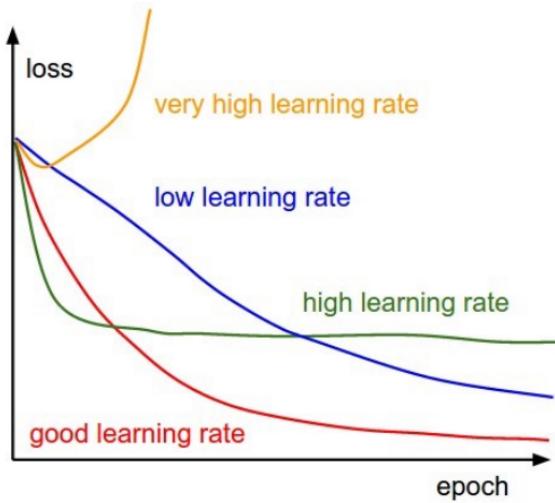
In base ai dati che abbiamo, alcune di queste trasformazioni potrebbero andare bene ed altre no. Per esempio se vogliamo riconoscere i numeri di MNIST, se ruotiamo un 6 al contrario, questo diventa un 9. Oppure se faccio il mirroring del numero 3, trovo un qualcosa che è al di fuori delle labels.

Oppure se abbiamo un fiore, se cambio il colore potrei avere un fiore diverso.

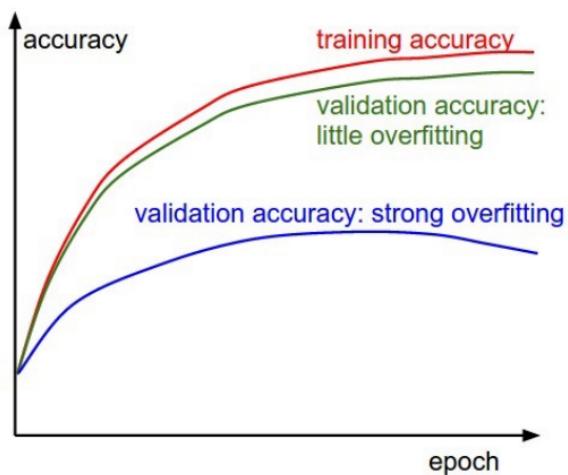
Ci sono dei tools che automaticamente capiscono quale tipo di trasformazione funziona meglio per il mio dataset.

Il controllo sull'overfitting viene fatto guardando la loss e l'accuracy.

Check the effects of learning rate on the loss trend



Track the validation/training accuracy curve trend

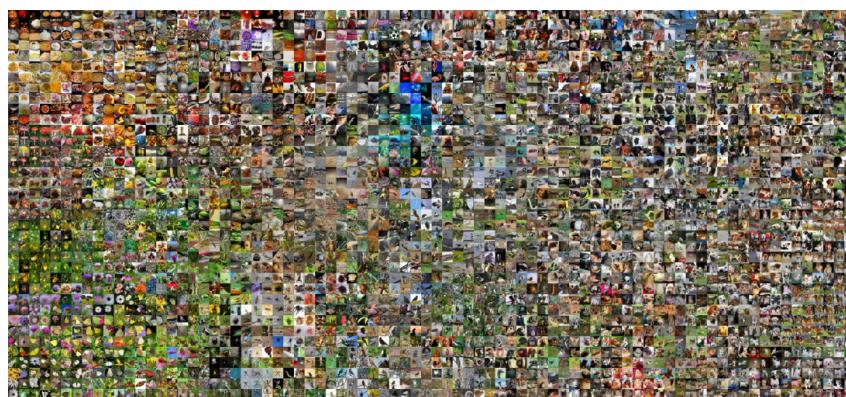


Architetture famose

Nelle reti convoluzionali di solito si inizia da un'architettura che è già stata pubblicata e testata, e poi si modifica questa. Ora vediamo alcune network famose.

Per introdurle useremo il dataset IMGENET che è una collezione di più di 40 milioni di immagini.

- 14,197,122 images
- 21,841 synsets

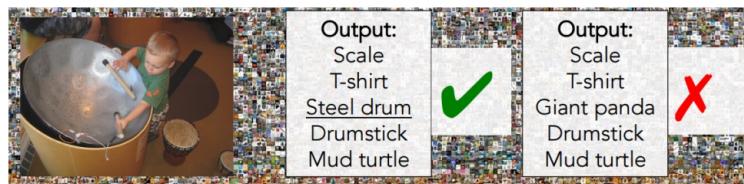


Se usiamo top-1 accuracy, prendiamo la classe con confidenza più alta come predizione e la confrontiamo al true value. Questo rende il problema difficile

Alternativamente ritorno le predizioni top 5 per accuracy, e in base a queste posso dire che l'immagine è classificata correttamente se il true value è tra questi 5 valori.

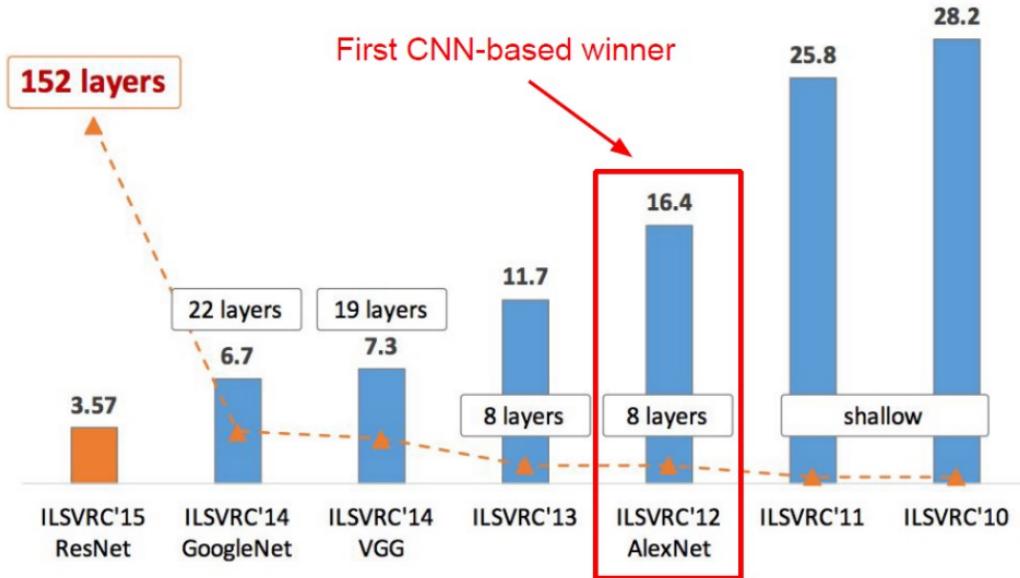
IMAGENET Large Scale Visual Recognition Challenge

The Image Classification Challenge:
1,000 object classes
1,431,167 images

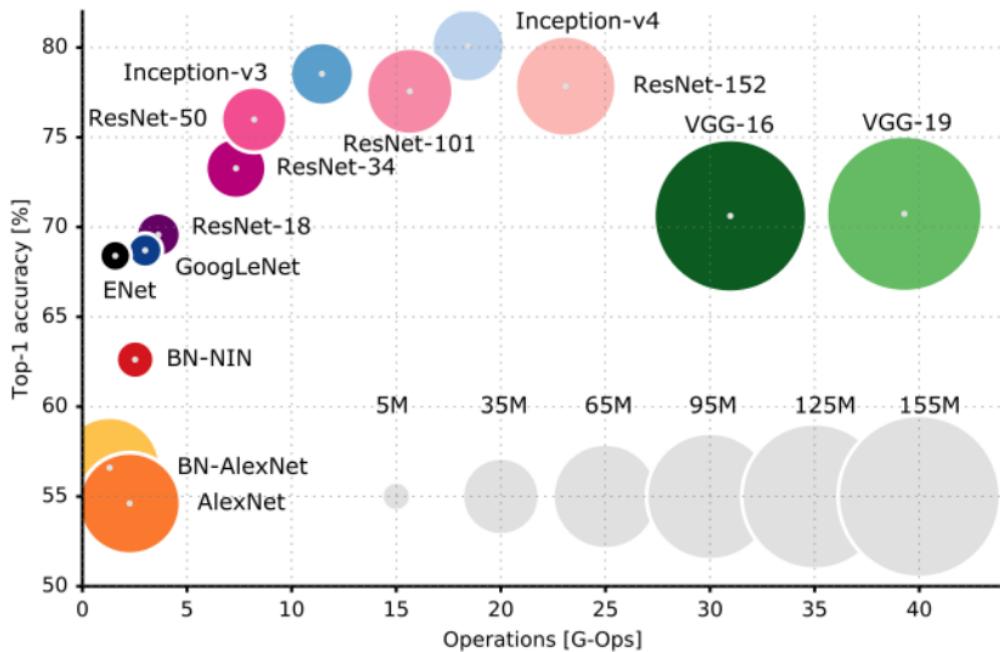


Qui a destra a sinistra abbiamo l'anno (dal 2010 al 2015) in cui è stata creata ciascuna rete. Quelle più recenti (migliori) sono CNN (la prima CNN AlexNet nel 2012 è quella da dove è iniziata la rivoluzione AI), quelle precedenti erano basate su SIFT e altro), che progressivamente hanno sempre più layers, dimezzando l'errore quasi ogni anno.

Imagenet classification error



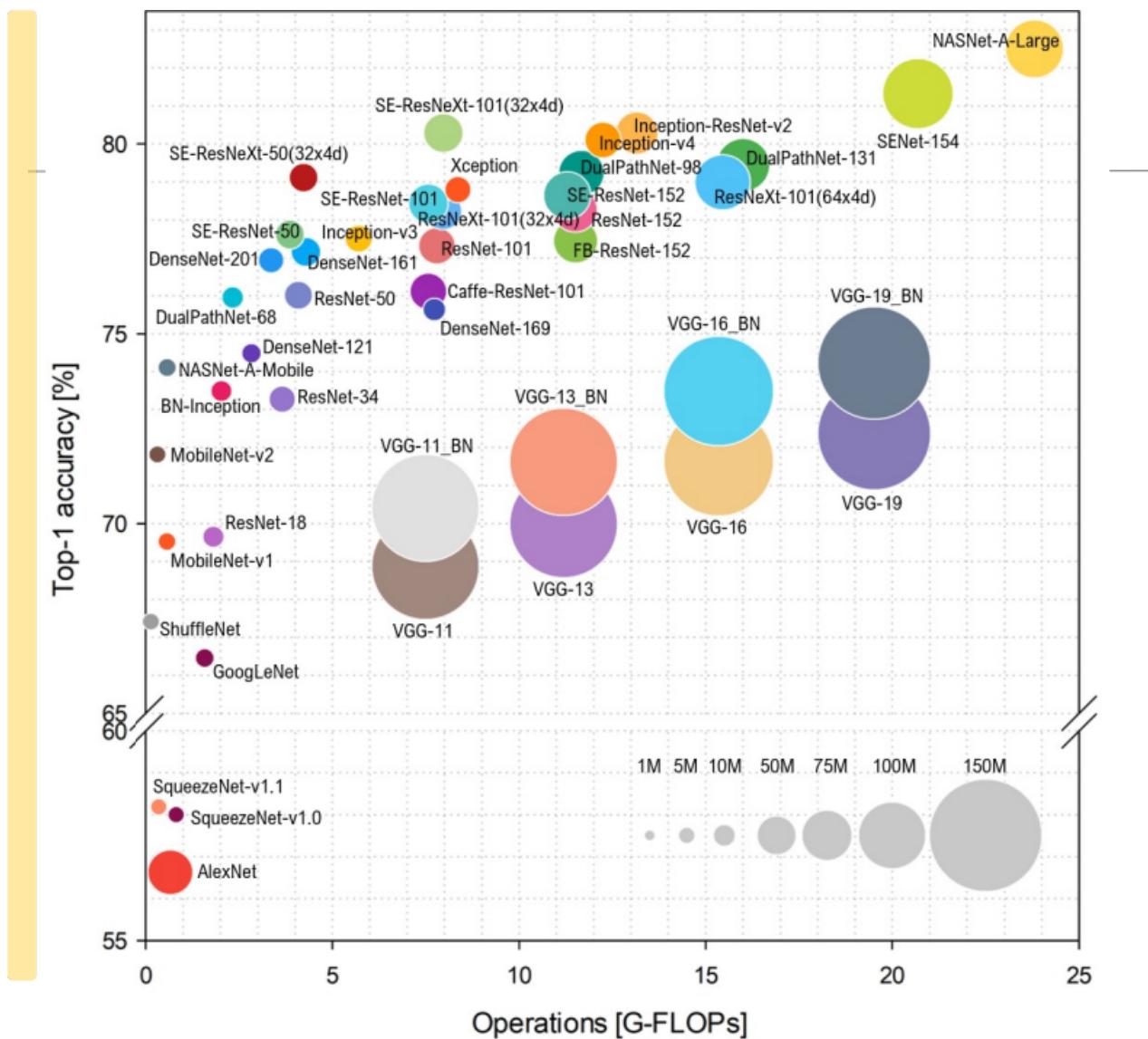
In base all'accuracy a cui puntiamo e la grandezza della rete (quanta memoria possiamo usare, dipende dal device che dobbiamo usare), possiamo seguire questo schema per raggiungere il tradeoff migliore possibile:



Il cerchio rappresenta il numero di parametri del modello.

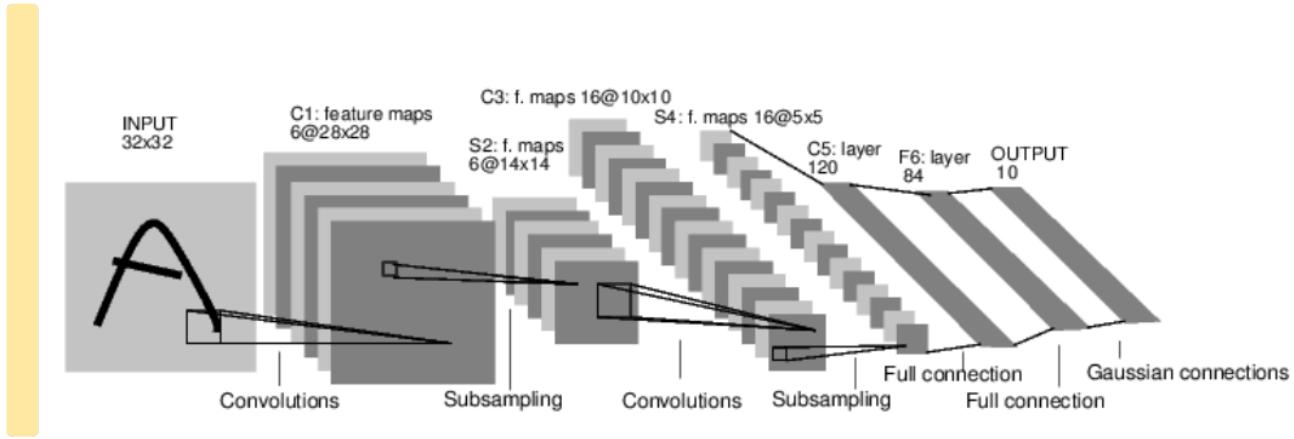
Una possibilità potrebbe essere per esempio GoogleNet perchè è piccola come modello ed è molto buona in quanto accuratezza. Se invece possiamo avere un modello più grande possiamo usare Resnet152 (più operazioni, più memoria).

Una rappresentazione più recente è questa:



LeNet (LeNet-5)

Questo è stato la prima rete convoluzionale. Prende in input immagini 32×32 , è stata creata per riconoscere le lettere alle poste.

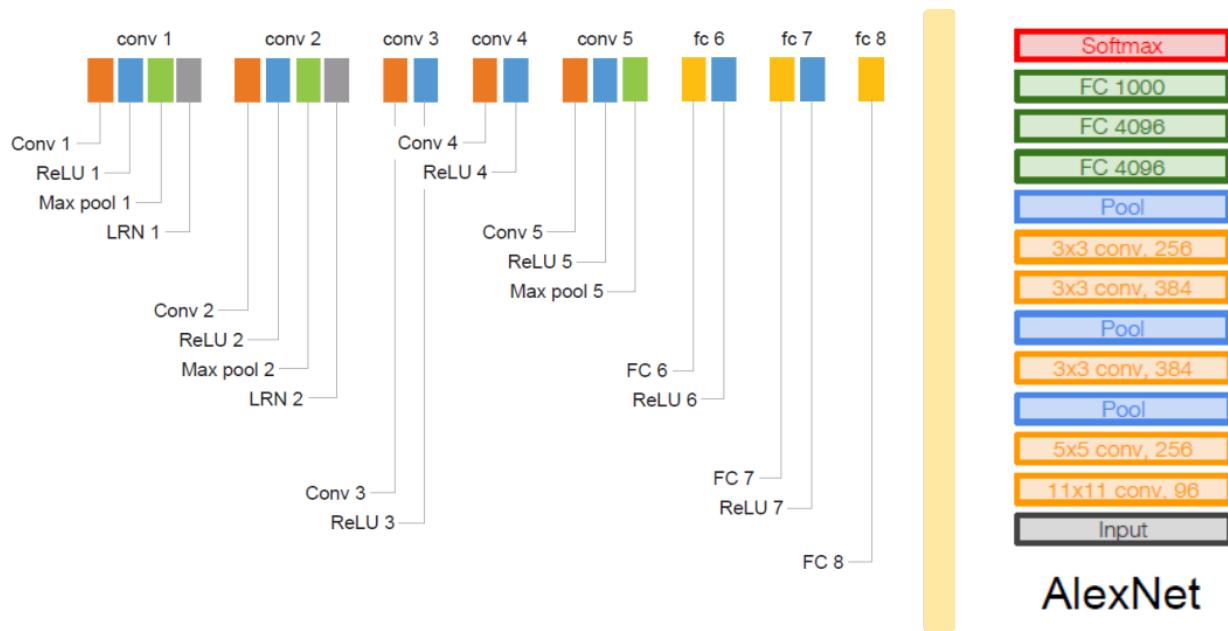


Abbiamo l'input, iniziamo con un layer convoluzionale con 6 filtri, poi abbiamo lo spatial pooling che diminuisce la grandezza ma non il numero di layer, poi abbiamo un altro layer convoluzionale con 16 filtri, un altro spatial pooling, e quindi facciamo reshape in un vettore monodimensionale, seguito da due layer di NN normale fully connected, l'ultimo va in 10 diversi output (corrisponde al numero di classi, questo modello era fatto per riconoscere i 10 numeri dei postal codes).

AlexNet

AlexNet è creata in questo modo. A destra abbiamo informazioni su ciascun layer, con l'input in fondo e l'output in cima.

(le due nell'immagine sono diverse, quella a sinistra è quella revisionata, guardiamo quella a destra, che non ha le normalizzazioni ed è fatta in modo leggermente diverso)



Abbiamo l'input, iniziamo con un livello convoluzionale che ha 96 filtri di grandezza 11×11 (ben di più di LeNet). Poi abbiamo un altro layer con 256 filtri con dimensione 5×5 . Poi abbiamo un livello di pooling, etc etc

Dopo l'ultimo blocco convoluzionale distruggiamo l'informazione spaziale, abbiamo un vettore monodimensionale, e abbiamo un livello fully connected con 4096 neuroni, poi uno con 4096 e poi uno con 1000 (1000 classi del dataset delle immagini), ed infine l'attivazione softmax per avere un output tra 0 e 1 (e le probabilità si sommano a 1) per ciascuna classe.

La grandezza dei filtri è ridotta man mano. Quindi riduciamo la dimensione spaziale dell'input. Però aumentiamo la dimensione di profondità.

Facciamo un conto, quanti parametri abbiamo? Nel primo layer convoluzionale abbiamo $10 \times 10 \times 100$ (approssimando), quindi 10'000 parametri nel primo layer.

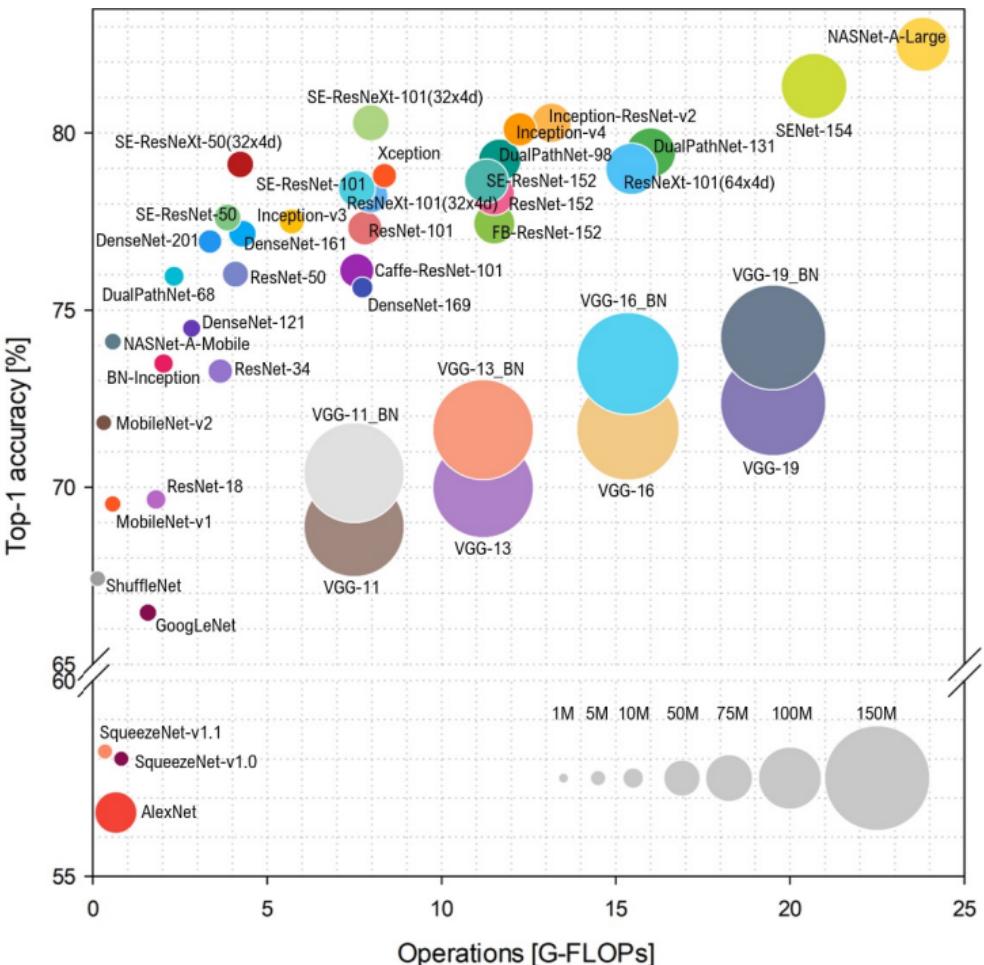
Tra i primi due layer fully connected invece abbiamo più di 16 milioni di parametri. Poi andando al prossimo FC layer abbiamo altri 4 milioni. Quindi solo nel pezzo finale fully connected abbiamo 20 milioni di parametri.

Quindi vediamo che la maggior parte dei parametri è nella parte di fully connected layers. Nelle prossime reti, vedremo che cercheranno di ridurre il numero di parametri in questa parte della rete.

Nella prima parte della rete man mano riduciamo la grandezza delle immagini e aumentiamo il numero di filtri in profondità.

L'idea è quella di fare un **encoding** nei canali in profondità.

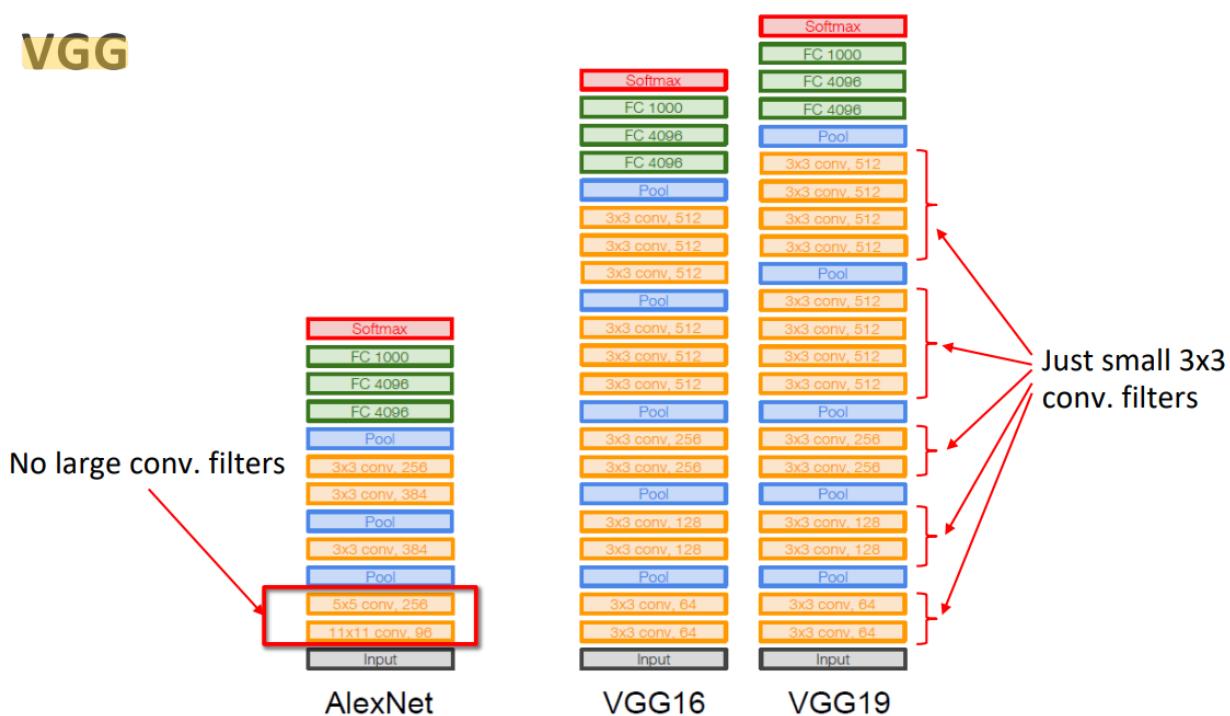
Alexnet è in basso a sx del grafico.



Aggiungendo le normalizzazioni e cambiando la rete (facendo quella a sinistra di 2 immagini fa) la rete è stata migliorata.

Da qui siamo passati ad altre architetture che hanno vinto la challenge negli anni successivi:

VGG



VGG è una famiglia di architetture, con diverse grandezze.

Le differenze principali tra le architetture VGG e AlexNet sono innanzitutto il numero di layer convoluzionali. Più layer abbiamo e più non-lineare sarà il classificatore. Quindi con più attivazioni non-lineari abbiamo un classificatore più complesso.

Poi per ridurre il numero di parametri, si riducono i filtri grandi di AlexNet con filtri 3×3 più piccoli.

Hanno poi introdotto una regola, iniziano con 64 canali, 64, poi raddoppiano 128, 128, poi 256, 256 ... e poi rimangono costanti con 512 (in alexnet non c'era una regola, un pattern).

Perchè usiamo i filtri 3×3 al posto di quelli più grandi? Se abbiamo un filtro più grande siamo guardando a una porzione più grande dell'immagine.

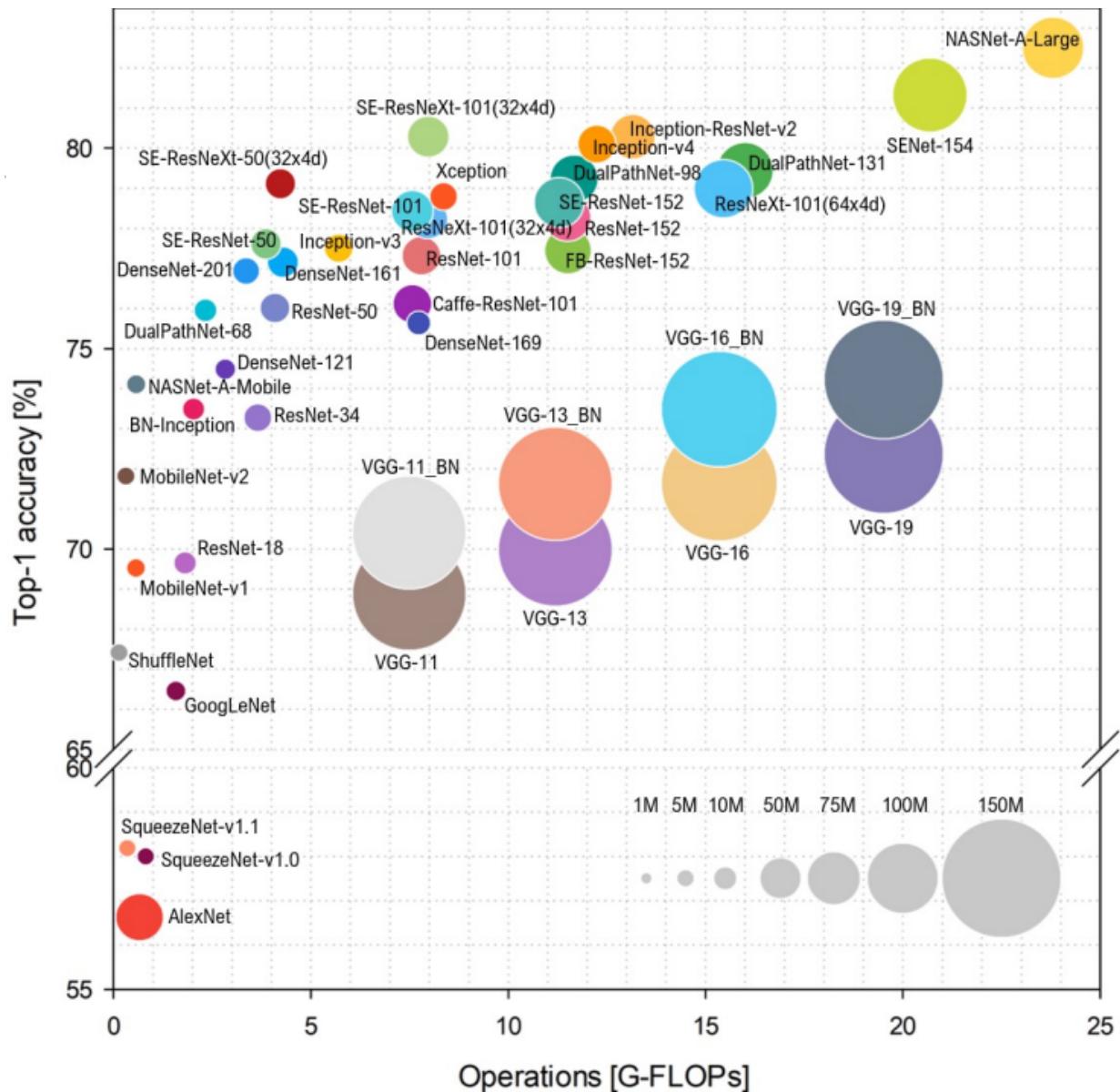
Uno stack di filtri convoluzionali 3×3 ha lo stesso "receptive field" di uno 7×7 , possono vedere la stessa regione. Il receptive field è quanti elementi del volume di input possono avere un effetto sul valore del neurone?

Quindi se abbiamo un filtro 3×3 , lo applichiamo all'immagine, e poi all'output (più piccolo) applichiamo un filtro 3×3 . Questo secondo filtro 3×3 è influenzato da un

“receptive field” più grande nell’immagine originale. Quindi facendo questo processo con più filtri 3×3 porta allo stesso risultato di usare un filtro più grande.

Il motivo di spezzettare il filtro 7×7 in 2 3×3 è che facciamo una rete più deep, con 2 attivazioni al posto di 1, quindi rendiamo la rete più non-lineare. In più abbiamo anche meno parametri, perchè abbiamo filtri più piccoli. Con 7×7 abbiamo 49 parametri, con 2 3×3 abbiamo 9 parametri. Quindi è anche più flessibile.

(in queste non c’è la normalizzazione, ci sono versione modificate che ce l’hanno)



Quelli con normalizzazione sono quelli subito sopra che finiscono con N.

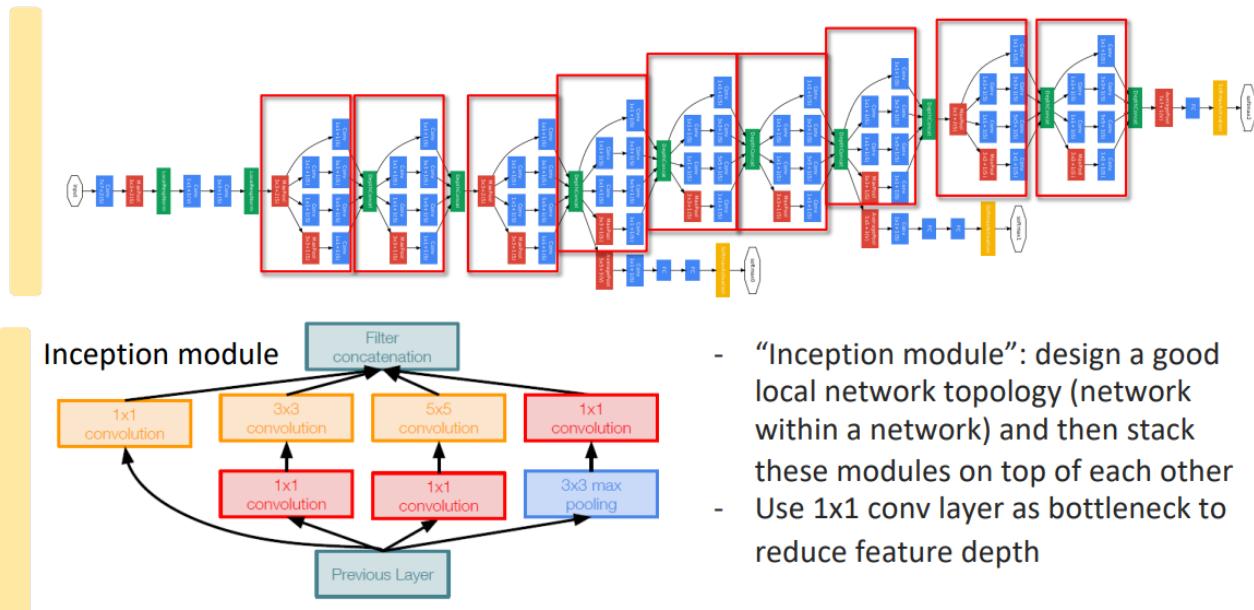
Ci sono però molti milioni di parametri nella parte fully connected, e insieme alle ripetizioni di layer convoluzionali, tutto insieme rende questo modello uno di quelli più grandi per le CNN.

I diversi modelli della famiglia VGG hanno più o meno operazioni. Le performance sono molto meglio di AlexNet.

GoogleLeNet (Inception)

è simile a LeNet. Questa si chiama anche Inception v1. Il nome arriva dal film.

Abbiamo dei moduli di "inception" che sono ripetuti multiple volte nella rete.



Questo modulo processa l'input in diversi percorsi, con diverse grandezze di kernel. Ciascuna parte ha anche filtri con dimensioni diverse, per estrarre informazioni diverse.

Ottieniamo 4 output diversi che vengono stackati nel canale di profondità.

La profondità dell'architettura inizia a diventare molto grande, quindi la vediamo ruotata sopra.

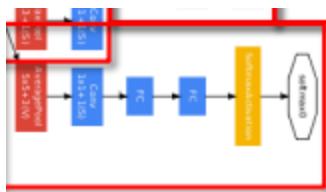
Abbiamo un totale di 9 moduli inception stackati uno dopo l'altro.

L'output ha un'attivazione softmax (giallo). Possiamo vedere che abbiamo altri 2 blocchi gialli però. Questi 2 sono classificatori ausiliari. L'idea è quella che

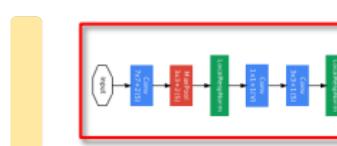
iniziamo a computare l'errore con il gradiente dall'output. Man mano che back-propaghiamo, il gradiente diventa molto piccolo, quasi 0, quindi inizia a diventare molto piccolo e quindi non fa update dei layers. Questi classificatori ausiliari aggiungono informazioni man mano per riuscire a propagare l'errore a tutti i layers.

Quindi durante il training abbiamo una loss che è una combinazione lineare delle 3 losses pesate, la prima a destra è quella più importante, le altre sono progressivamente meno importanti, andando a sinistra (seconda meno importante, terza a destra ancora meno).

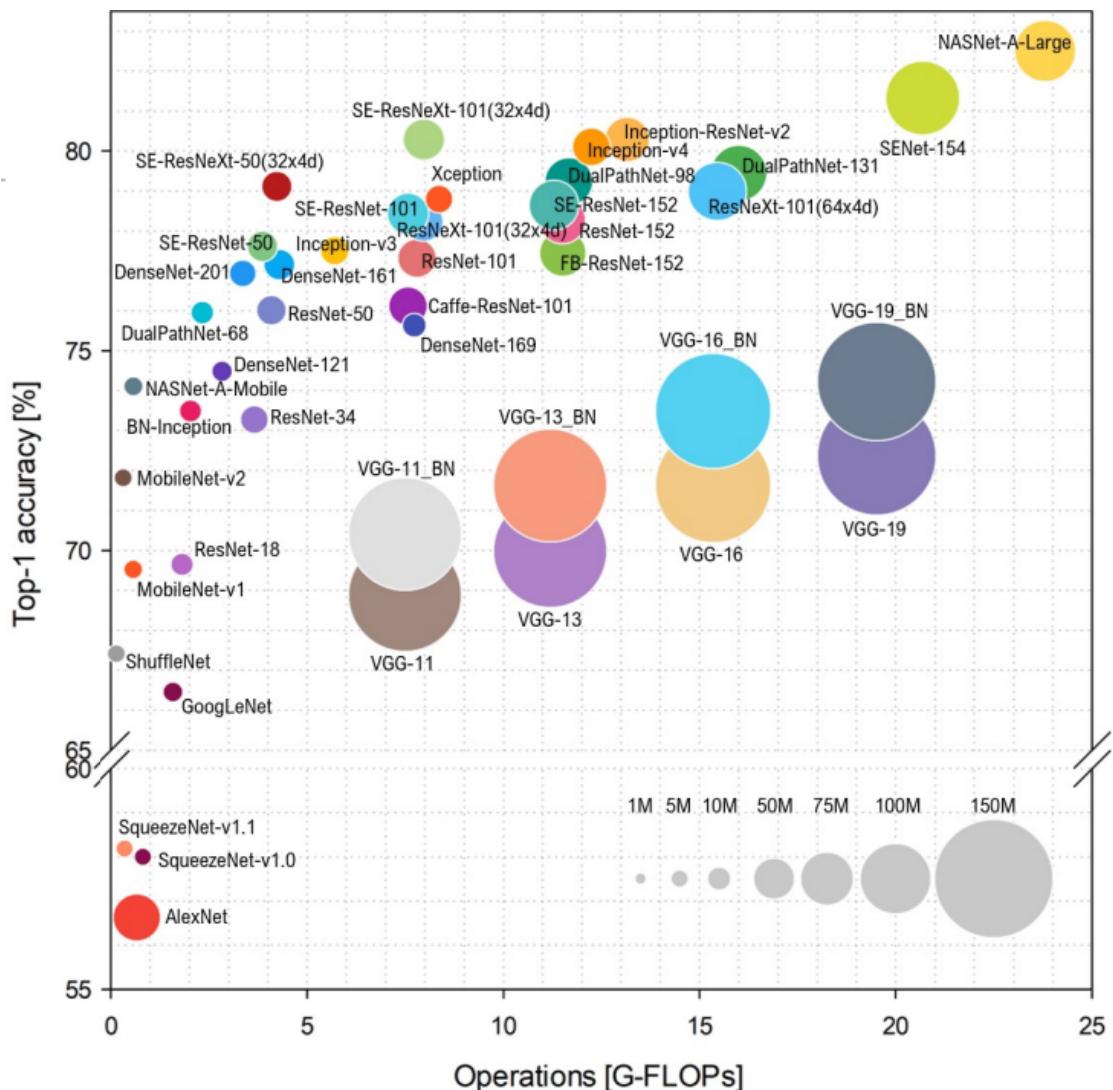
Quando abbiamo finito il training rimuoviamo i due blocchi di output ausiliari e manteniamo solo l'ultimo.



Poi abbiamo lo **stem network**. Questo fa pre processing dell'input.



Possiamo vedere che abbiamo solo 1 fully connected layer in fondo (quello blu in fondo).



Possiamo vedere come GoogLeNet sia molto più piccola.

Vediamo che riducendo il numero di fully connected layers (che è dove ci sono i milioni di parametri), riusciamo ad avere un'architettura molto efficiente.

Abbiamo varianti, come Inception v3 e v4 che sono più grandi.

ResNet

Fin ora abbiamo visto che più deep è la rete e migliori sono i risultati. Però ad un certo punto non possiamo aggiungere più layers per migliorare la rete.

ResNet

- So far the deeper the better
- Can we continue on adding layer and go deeper?

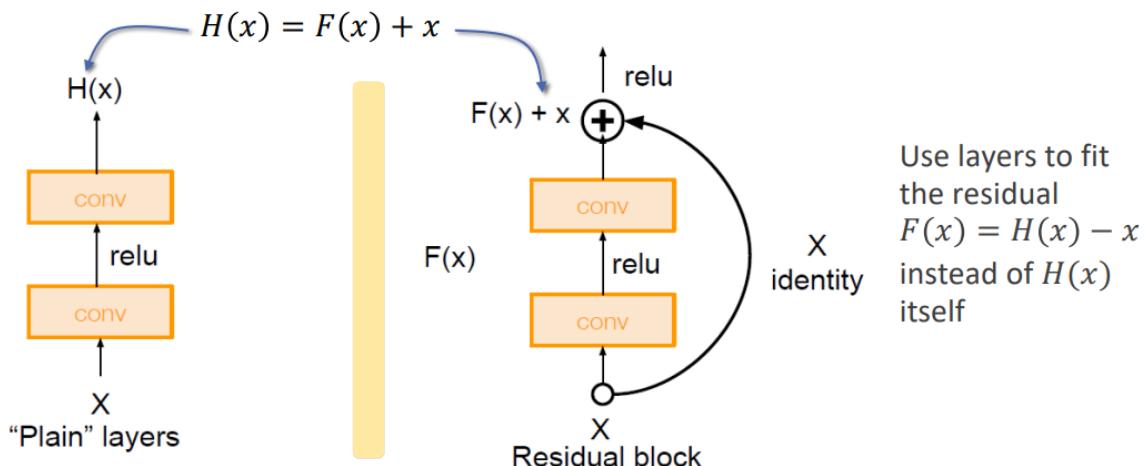


Il grafico a sinistra è googlenetv1, se raddoppi il numero di layers non solo non si riesce a ridurre il training error rispetto al modello più piccolo, ma anzi non raggiunge neanche il livello del modello più piccolo.

Uno dei problemi è che c'è il vanishing gradients, che è ancora più difficile da risolvere in reti molto profonde.

La soluzione sta in ResNet (Residual Network).

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Fin ora siamo nella configurazione a sinistra, abbiamo convolution, non linearità, convolution... L'idea è quella di non usare la rete per imparare il mapping da X a

$H(X)$, ma piuttosto fittiamo $H(x) - x$.

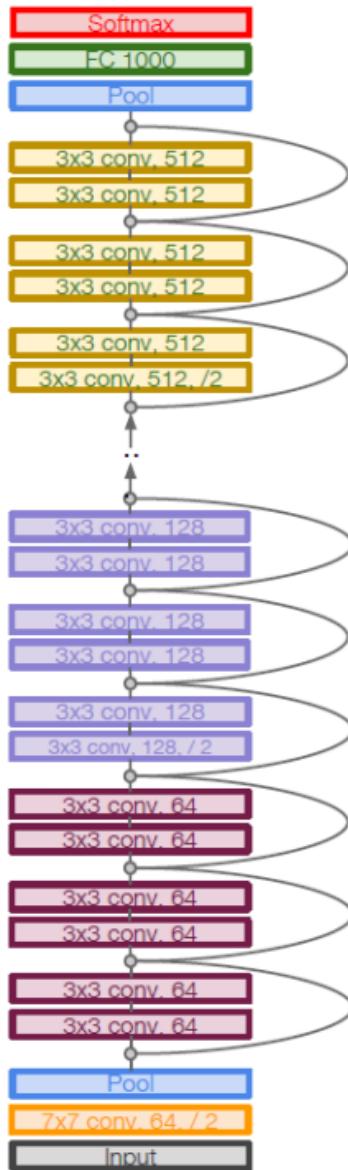
Quindi sostanzialmente, al posto di imparare la funzione, diamo al modello la funzione identità, e dobbiamo imparare solamente la differenza con l'identità della funzione.

Quindi abbiamo l'input, passiamo per l'architettura, otteniamo $F(x)$ e ad un certo punto sommiamo x (dopo un tot di layers). Quindi copiamo l'input x , e lo passiamo avanti di qualche layer, e la rete deve imparare $F(x) + x$, conoscendo x .

Questa è chiamata una **skip connection**. Semplicemente copia l'input, dandolo ad un tot di layer dopo.

Questo è importante perchè se prima dovevamo fare back propagation che andava indietro layer per layer, ora facciamo la stessa cosa ma seguiamo anche la direzione della freccia curvata in direzione opposta, che è x . La derivata di x rispetto a x è 1, quindi abbiamo un path dove possiamo propagare il gradiente senza cambiamenti, senza riduzione. Quindi riusciamo a backpropagare il gradiente anche in reti molto grandi, perchè abbiamo creato un percorso dove il gradiente può passare senza essere ridotto.

Questo blocco di residuo è ripetuto molte volte nell'architettura ResNet.



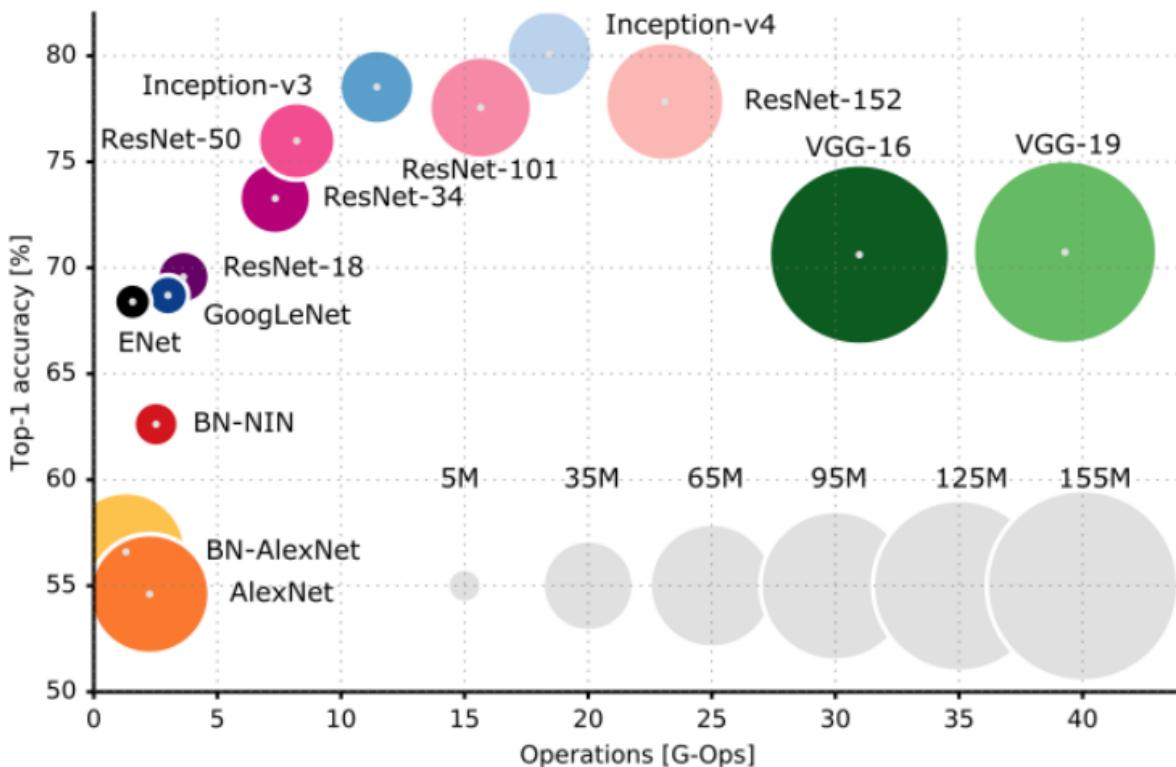
Quindi abbiamo l'accumulazione dei gradients che passano per tutti i possibili paths (diritto come una nn normale, uno che passa per qualche skip in cima e poi va diritto, etc tutte le combinazioni possibili)

Questo risolve il problema del vanishing gradient perchè abbiamo dei percorsi dove il gradiente può passare senza cambiamenti.

Quindi abbiamo l'input, iniziamo con un layer convoluzionale 7×7 con 64 canali, poi negli altri layer convoluzionali abbiamo filtri 3×3 , finchè raggiungiamo il pooling layer che fa il reshape.

Per cosa sta il /2? Questo è il parametro di **stride**. Al posto di usare pooling layers, ogni volta che siamo raddoppiando il numero di canali facciamo stride/2, quindi stiamo salvando computazioni perchè stiamo facendo delle operazioni di pooling già nel layer convoluzionali.

Quando raddoppiamo il numero di canali, dimezziamo la dimensione spaziale, usando stride = 2.



Un altro concetto importante è quello del transfer learning.

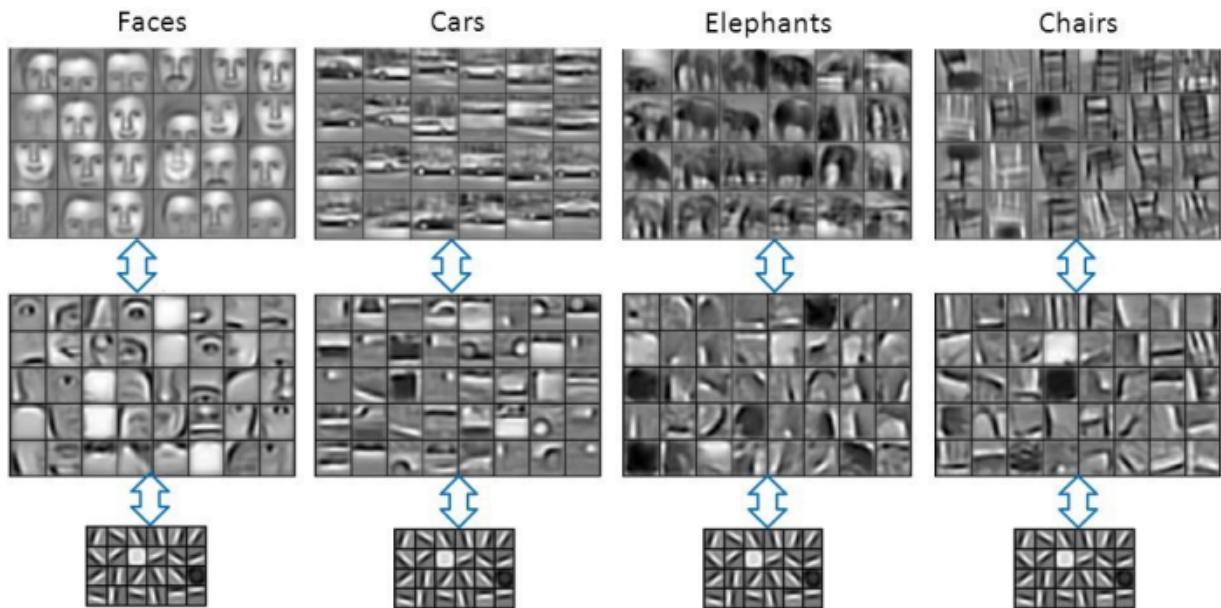
Transfer learning

Tutte le architetture di oggi hanno molti milioni di parametri, quindi avremmo bisogno di dataset molto grandi per fare tuning dei parametri. Cosa succede se non abbiamo questo dataset grande?

In pratica, poco spesso si fa il training di una CNN da zero, è invece molto comune fare pre-training su dataset molto larghi, come IMAGENET, anche se non è

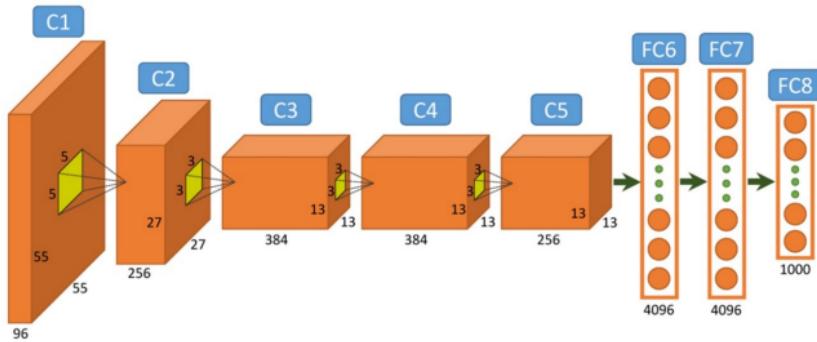
esattamente il problema che vogliamo risolvere, e poi possiamo usare questa CNN per fine-tuning o come feature extractor del nostro problema.

Indipendentemente dalla task, la CNN sta imparando la stessa rappresentazione (la gerarchia) fino ad un certo punto, dove l'astrazione aumenta e diventa task-specific.

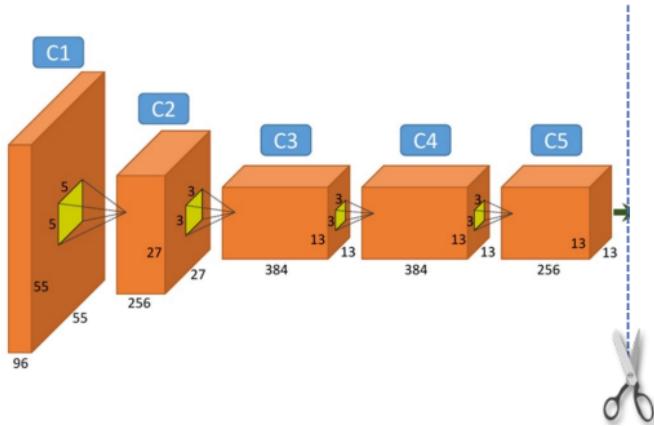


Esempio fine tuning

Task 1 (e.g. ImageNet,
1000 categories)



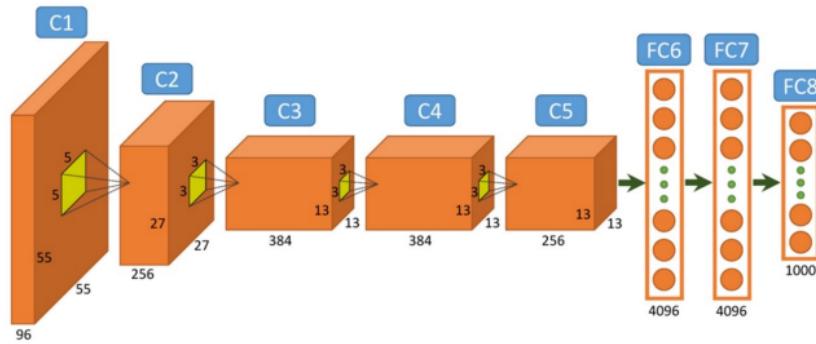
Task 2 (10 categories)



L'idea è quella di tagliare gli ultimi layer che sono task-specific, e li rimpiazziamo.
Freeziamo i layer nella prima parte, non facciamo aggiornamenti.

Quindi inizializziamo random gli ultimi layers, freeziamo la prima parte e poi
alleniamo l'ultimo pezzo

Task 1 (e.g. ImageNet,
1000 categories)



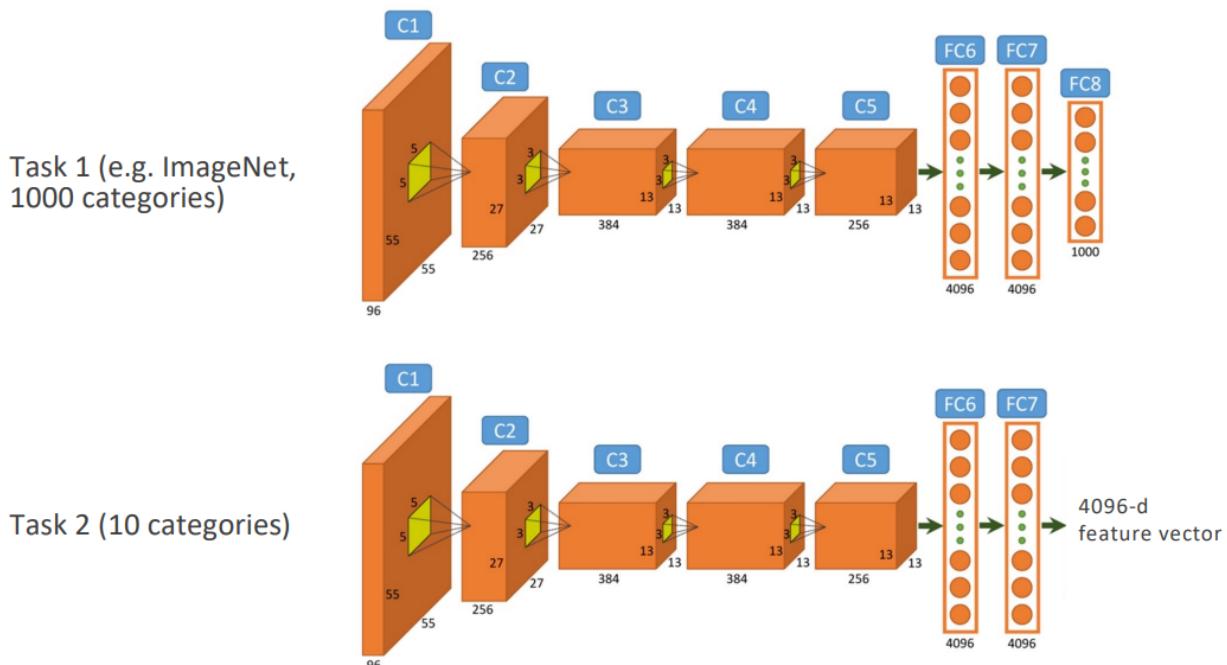
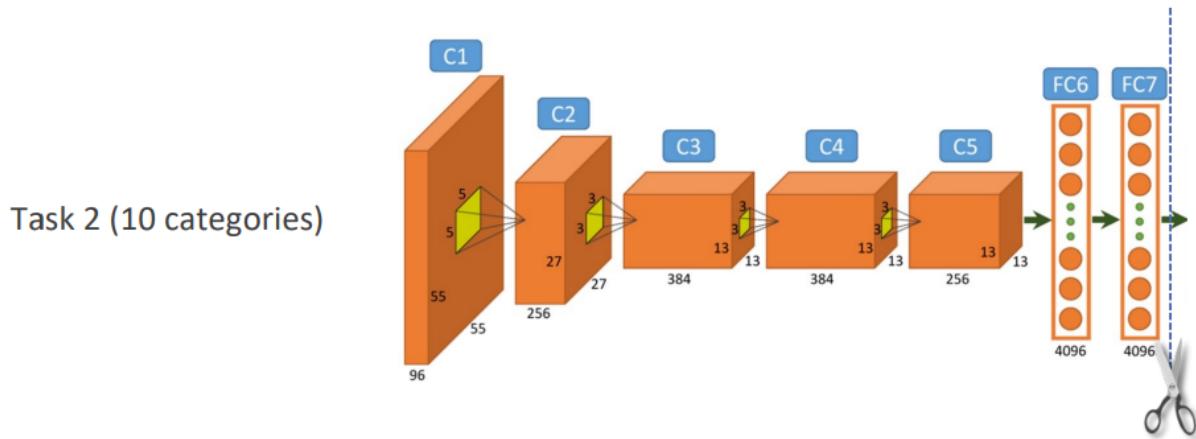
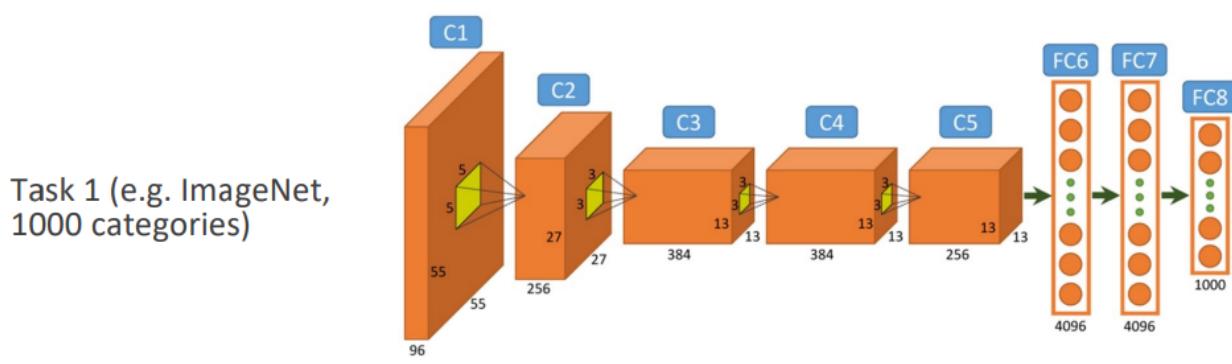
Task 2 (10 categories)



Cambiamo solo l'ultima parte di modo che sia specifica per il nostro problema.

Un'altra opzione è usare la rete come un feature extractor per il nostro problema.

Tagliamo l'architettura ad un certo punto, e poi usiamo l'activation dell'ultimo layer come feature extractor.



Quindi poi aggiungo un layer.

è dimostrato che queste 4096 features sono molto potenti e molto utili.

Come possiamo decidere che tipo di transfer learning dovremmo fare, in base al nostro problema?

- New dataset is **small** and **similar** to original dataset:
→ Train a linear classifier on CNN features from higher layers
- New dataset is **large** and **similar** to original dataset
→ Fine-tune the CNN
- New dataset is **small** but **very different** from original dataset
→ Train a linear classifier on CNN features from lower layers
- New dataset is **large** and **very different** from original dataset
→ Train CNN from scratch or fine-tune it