Lezione 9 - Model compression - 26/11/2024

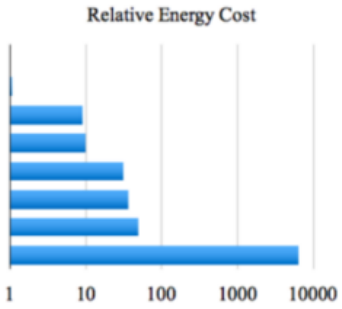
Neural Network model compression

Potremmo partire direttamente con un modello più piccolo, ma in realtà il modello migliore di fare learning è di partire con un "**overparameterized model**", ovvero uno più grande di quello che ti serve realmente.

Il problema del **model compression** arriva dal bisogno di eseguire NN su **device mobili**, date le limitazioni del dispositivo e dell'app store (per esempio l'apple store limita download sopra a 100MB se non sei connesso al wifi).

Ciascuna operazione ha un impatto dal punto di vista della **batteria**. L'idea è che se abbiamo un'operazione di integer 32bit addition, quella su float costa 10 volte di più rispetto all'int.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**



Nelle NN la principale operazione che abbiamo è la **moltiplicazione** (insieme all'addizione). Possiamo vedere che la moltiplicazione può costare 30/40 volte di più rispetto all'addizione.

Questo dipende anche dal tipo di memoria che usiamo per l'operazione, e dove il modello è salvato. Se dobbiamo andare sulla DRAM (perchè il modello è troppo grande per la SRAM cahce) al posto di rimanere sulla SRAM Cache, il costo è molto più alto.

L'idea è quella di avere un modello talmente piccolo che possiamo rimanere su quella memoria a costo basso a livello di batteria.

Quindi il focus è molto diverso, il problema ora è la batteria.

Una soluzione potrebbe essere quella di eseguire la rete neurale del **cloud**, e ritornare la risposta. Però abbiamo il **delay per la trasmissione** nell'internet. Se per esempio abbiamo una self-driving car che deve controllare il suo intorno, non possiamo avere network delay, dobbiamo essere in real time. c'è anche un problema di costo perchè il sistema cloud runna sempre. Infine c'è il problema della privacy dell'utente, perchè dobbiamo mandare i dati dell'utente al cloud. In alcuni paesi questo non si può fare, oppure dipende dalla posizione geografica del server.

Gli **obiettivi** del model compression sono 3:

- vogliamo un modello di **dimensione più piccola** (potremmo partire da un modello più grande e ridurlo man mano, ma non vogliamo solo questo)
- non vogliamo perdere nulla in **accuratezza**, vogliamo mantenere lo stesso livello di performance del modello più grande
- vogliamo che possibilmente il modello sia **più veloce** nella fase di inferenza

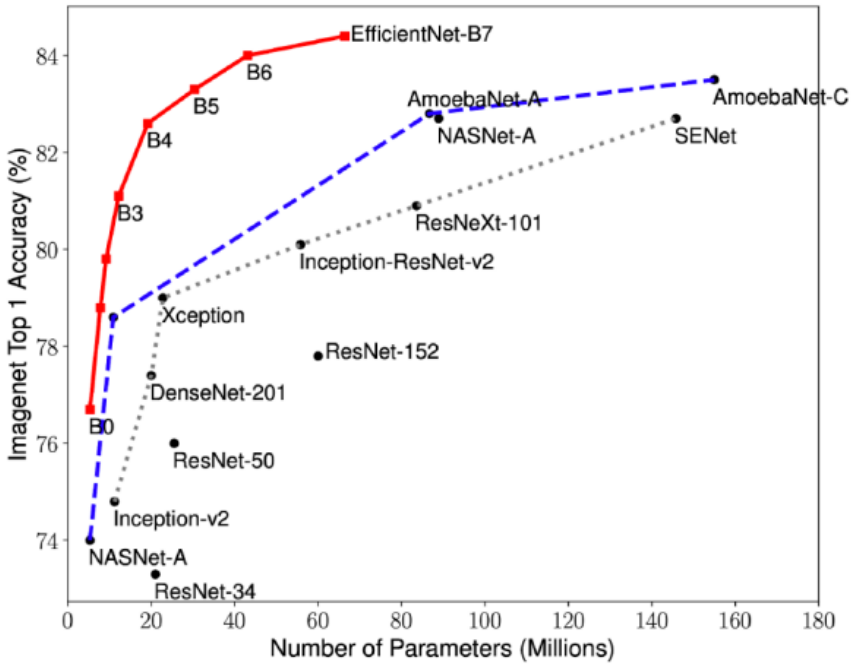
Per esempio Alexnet è possibile ridurlo da 240MB a 6.9MB senza loss in accuracy (35x meno grande).

VGG16 si può ridurre da 552MB a 11.3MB (49x)

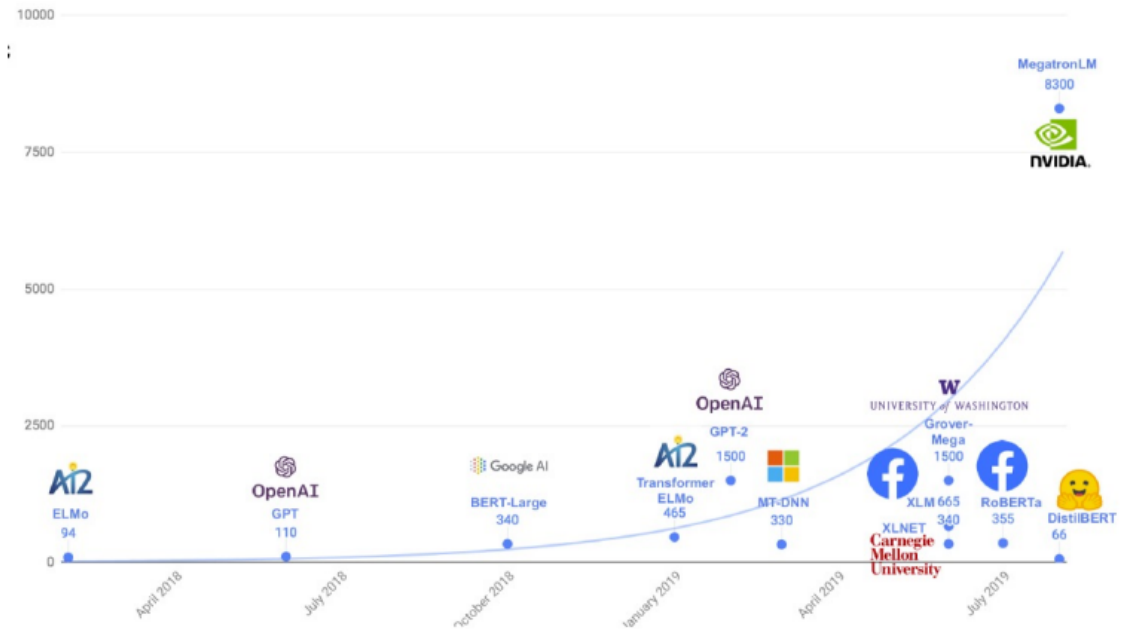
In questo modo possiamo rimanere sulla SRAM, dove viene usata 120 volte meno energia della DRAM.

- AlexNet: 35×, 240MB => 6.9MB
- VGG16: 49× 552MB => 11.3MB
- Both with no loss of accuracy on ImageNet
- Weights fits on-chip SRAM, taking 120x less energy than DRAM

Potremmo avere un'architettura che riduce il numero di operazioni float e che migliora l'efficienza, con meno parametri. Un esempio è la famiglia EfficientNet.



L'importanza del model compression è ancora più importante per le reti transformer. Uno dei modelli migliori di 5 anni fa conteneva 8.3 trilioni parametri, che richiedeva più di 500 GPU per allenare.



GPT-3 contiene 175 milioni di parametri, sono moltissimi anche solo da salvare. Senza compressione, non possiamo salvare questi modelli di LLM su una

macchina locale.

Gli approcci principali sono:

- weight sharing
- network pruning
- low rank matri and tensor decomposition
- knowledge distillation
- quantization

questi 5 partono da un modello grande, e poi comprimiamo

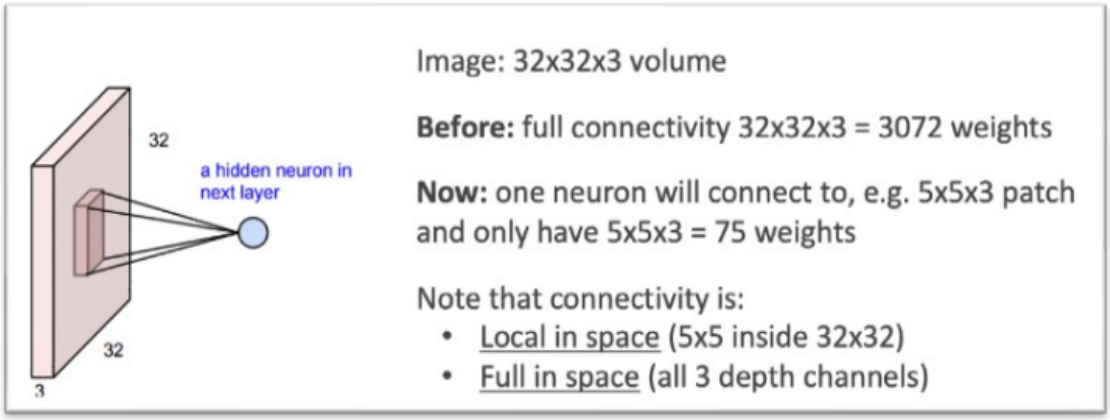
- low resource and efficient architecture

in questo caso si parte subito con il design di un'architettura piccola ed efficiente.

Weight sharing

Il modo più semplice di ridurre una rete è quello di condividere i pesi tra layers.

In questo caso, questa tecnica è applicata prima di trainare il modello originale. Come nella convoluzione, c'è questo weight sharing, prima del training (filters).



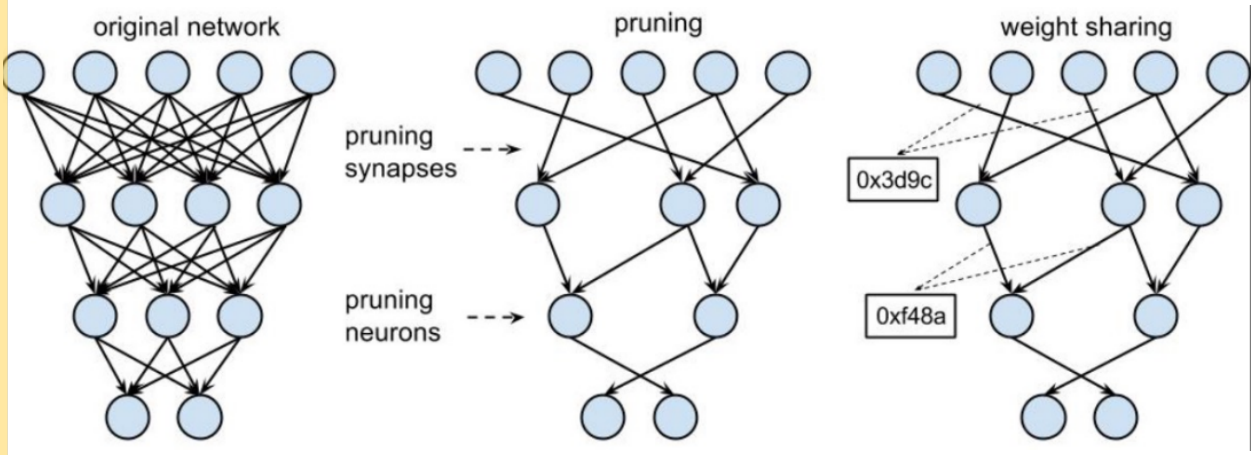
It is not always clear how many and what group of weights should be shared before
there is an unacceptable performance degradation for a given network architecture
and task

Network pruning

Questo è il metodo più usato per ridurre il numero di parametri di un modello pre-trained.

Partiamo da un modello, e poi facciamo pruning di sinapsi o anche neuroni. Ogni volta che rimuoviamo una connessione, abbiamo un parametro in meno. Se invece rimuoviamo un neurone, rimuoviamo anche tutte le sue connessioni. L'idea è che avendo meno sinapsi, il modello è più piccolo e più veloce.

Se rimuoviamo solamente delle sinapsi, i pesi vengono salvati all'interno della matrice come "0", quindi vengono comunque "salvati". Mentre invece se togliamo un intero neurone, stiamo togliendo un'intera riga o un'intera colonna, quindi stiamo salvando spazio e anche operazioni (perchè non dobbiamo più moltiplicare per 0).

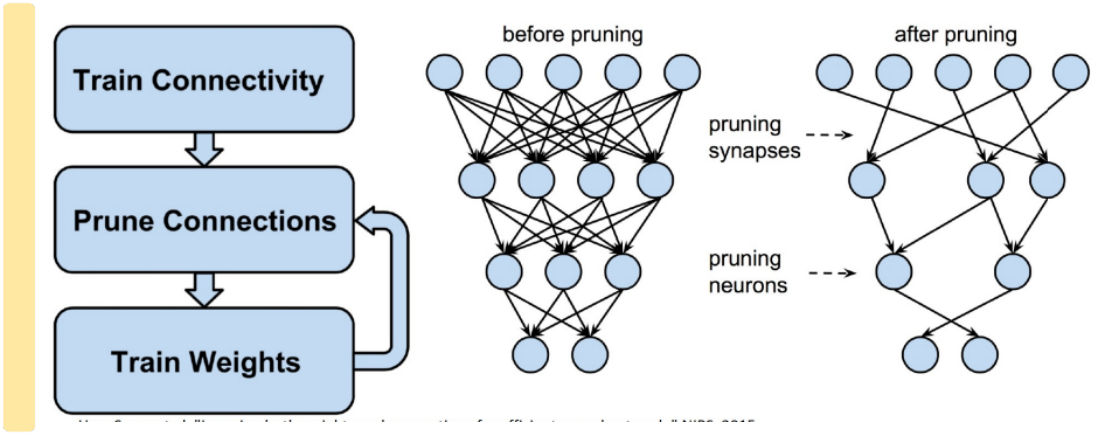


Seconda operazione che può salvare spazio: possiamo fare del weight sharing, facendo una lookup table, unendo i pesi che sono molto simili. Usiamo quindi un pointer, al posto di salvare i valori multiple volte.

Nella realtà questo processo è ripetuto multiple volte, perchè la rete iniziale è già ottimizzata, quindi se rimuoviamo neuroni non possiamo aspettarci che il modello funzioni ancora bene.

Quindi iniziamo con una fully connected architecture, facciamo pruning, e poi facciamo un fine-tuning del modello nella configurazione raggiunta, di modo che il modello possa modificarsi per adattarsi alla nuova configurazione. Poi ripetiamo l'operazione, facendo di nuovo pruning, e poi il retraining.

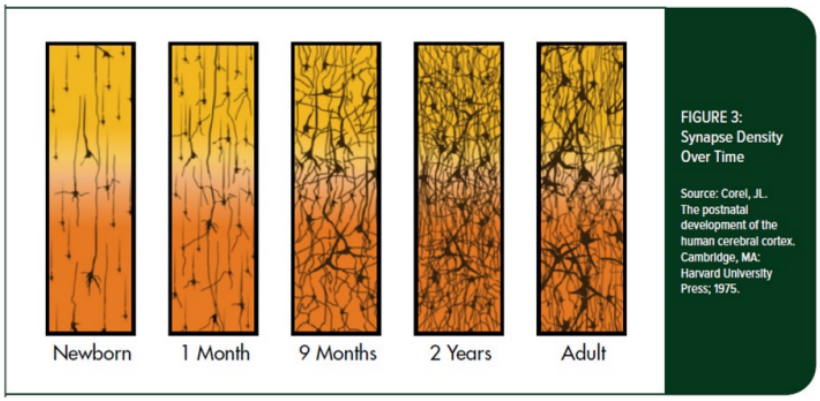
Se vogliamo ridurre il modello del 90%, non possiamo aspettarci che funzioni di botto, rimuovendole tutte, il modello non ha abbastanza tempo per adattarsi. Invece, potremmo rimuovere 10%, lo facciamo adattare, poi un altro 10%... finché raggiungiamo il numero target di parametri.



Potremmo anche scegliere di avere un tipo di compressione diverso rispetto al "lossless" che stiamo cercando di raggiungere ora, potremmo anche avere un approccio "lossy" per ridurre le dimensioni del modello ancora di più.

Questo tipo di approccio è simile a quello che accade nel cervello umano. Queste sono le sinapsi di un bambino:

- **50 trillions** of synapses are generated in the human brain during the first few months of birth.
- **1 year old**, peaked at **1000 trillions**
- Pruning begins to occur.
- **10 years old**, a child has nearly **500 trillions** synapses
- This 'pruning' mechanism removes redundant connections in the brain.



man mano che il bambino cresce, vengono create più connessioni. Ad un certo punto raggiunge un pensiero critico, dopo di che alcune connessioni sono create

ed altre sono droppate.

Abbiamo bisogno di regole per scegliere quali connessioni rimuovere, non possiamo farlo in modo random.

La strategia più semplice è quella di settare una soglia che decide quali sono i pesi che devono essere rimossi, quindi tutti le connessioni che hanno un peso minore di x vengono rimosse (quindi le connessioni con i pesi bassi).

Di solito vengono rimosse una percentuale di pesi.

MBP (magnitude-based pruning) è l'approccio più comunemente usato per la sua semplicità e la sua efficacità su molti modelli

Possiamo avere 2 tipi di MBP:

- possiamo scegliere una soglia che è globale per tutto il modello, scegliendo per esempio che voglio togliere il 50% delle connessioni.
- altrimenti possiamo avere una soglia per ciascun layer.

Il secondo metodo è più flessibile.

Abbiamo diversi modi di **categorizzare le tecniche di pruning**:

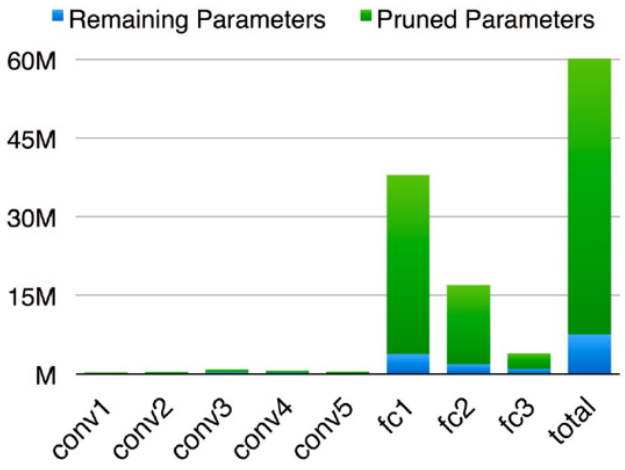
- Magniture-based pruning, dove togliamo i pesi più bassi in base alla soglia o una percentuale
- Metodi che forzano il modo ad avere molti pesi che sono 0, in questo modo possiamo rimuoverli semplicemente successivamente. Quindi abbiamo un run dove forziamo questo comportamento durante il training, di modo da creare candidati ideali.
- Metodi che calcolano la sensibilità della loss function quando i pesi sono rimossi e usano questo criterio per rimuovere i collegamenti che hanno impatto minore.
- Approcci search-based (particle filters, evolutionary algorithms, reinforcement learning) che tendono ad ottimizzare la rimozione dei neuroni nell'architettura. Questo è più indicato per modelli che sono già piccoli e di cui dobbiamo trovare la migliore configurazione.

Magnitude-based-pruning

é quello di cui stavamo parlando prima.

Questa è una rappresentazione di Alexnet dove vediamo i diversi layer e il numero di pesi in ciascun layer.

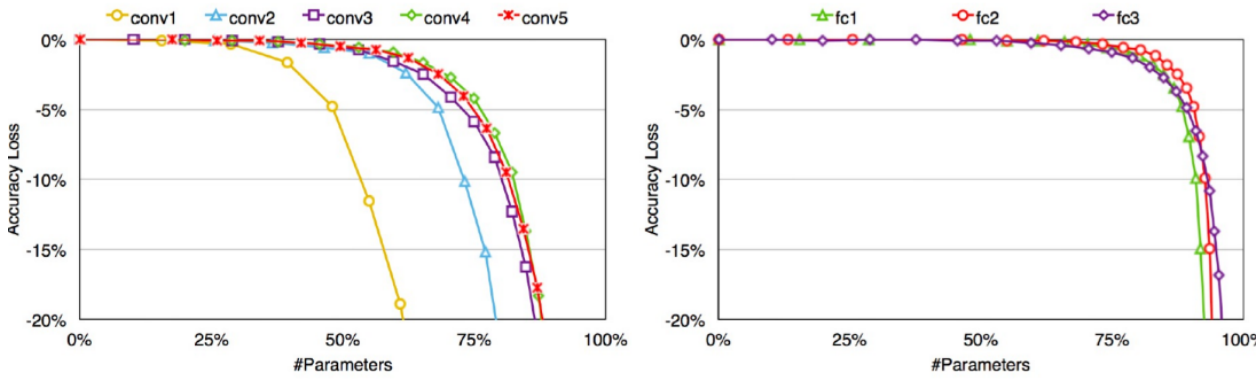
Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10
Total	61M	1.5B	54%	11%	30%



Nell'istogramma possiamo vedere come nei layer convoluzionali hanno pochi parametri rispetto alla parte fully connected. La somma delle barre blu e verdi è quella del modello originale. Quelli verdi sono quelli rimossi con il pruning.

Nella tabella, nella colonna "weights%" vediamo che nei layer convoluzionali manteniamo la maggior parte dei layers.

Qui possiamo vedere 2 diversi comportamenti: a sinistra i layer convoluzionali, se non li rimuoviamo non abbiamo loss nell'accuratezza. Man mano che rimuoviamo i parametri, abbiamo che la performance peggiora. Noi vogliamo raggiungere il numero più grande possibile di parametri rimossi senza perdita di performance. Possiamo vedere il primo layer sia il più sensibile, perde accuratezza molto in fretta. Layer diversi reagiscono differentemente alla rimozione dei parametri.



A destra invece vediamo come possiamo rimuovere una grande quantità di parametri nella parte fully connected senza perdere accuratezza. Vediamo infatti che abbiamo mantenuto solamente 9-25% dei parametri.

Pruning using weight regularization

Di solito abbiamo un penalty term che viene aggiunto alla loss finale. Spingiamo i pesi verso zero esplicitamente. Di solito si usa la penalizzazione con l2. Questo metodo non è molto usato però.

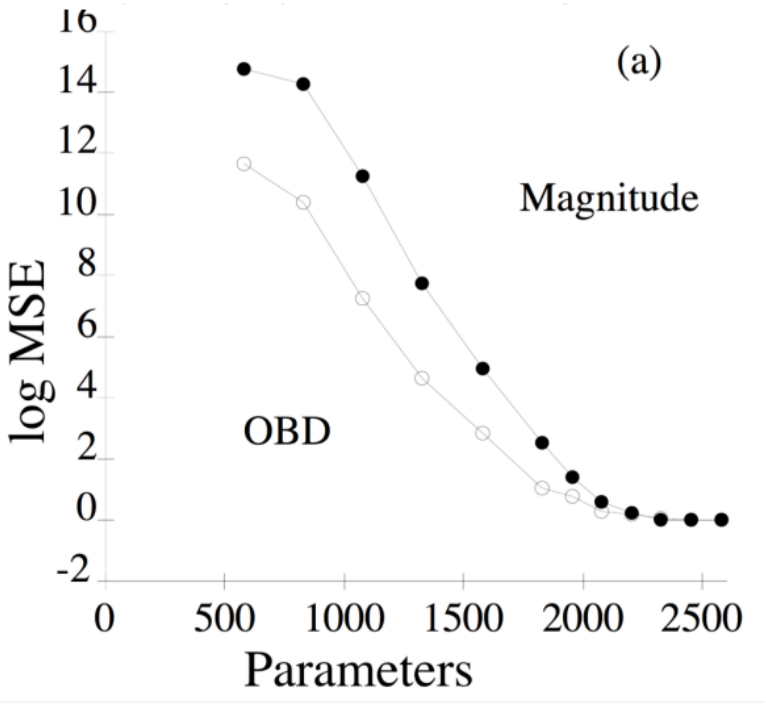
Hessian-based pruning: Optimal Brain Damage

L'idea è di calcolare la "saliency" dei pesi rispetto alla loss function, e poi rimuovere i parametri con bassa saliency. Quindi rimuoviamo i parametri che hanno un impatto piccolo sull'error finale.

1. Choose a neural network architecture.
2. Train the network until a reasonable solution is obtained.
3. Compute the second derivatives for each parameter.
4. Compute the saliencies for each parameter $S_k = \frac{\partial^2 E}{\partial^2 u_k} u_k^2$.
5. Sort the parameters by saliency and delete some low-saliency parameters
6. Iterate to step #2

Sembra semplice, ma perché non è usata sempre? Se abbiamo una funzione con 10 variabili diverse, dobbiamo calcolare 10 derivate di primo ordine, ma abbiamo un aumento quadratico di derivate di secondo ordine. Quindi se già abbiamo 1 milione di parametri, abbiamo un numero enorme di derivate del secondo ordine. Questo è il motivo per il quale questo metodo non può essere usato in realtà.

Il secondo approccio è quello di dire, sulla matrice hessiana, non ci importano le cross-derivatives, usiamo solo quelle diagonali. Anche così però abbiamo bisogno più computazioni rispetto alle derivate del primo d'ordine, che devono anche essere salvate oltre che computate.



OBD (optimal brain damage) funziona meglio, più comprimiamo il modello.

Structured vs unstructured pruning

Un'altra distinzione importante che dobbiamo fare, è la differenza tra i metodi strutturati e quelli non strutturati.

I metodi strutturati cercano di mantenere la densità della rete. Quindi rimuoviamo gruppi di pesi, cercando di mantenere la rete densa. Quindi se abbiamo un layer fully connected, andiamo a togliere un intero neurone, di modo che il resto della rete rimanga fully connected. Questo rimuove flessibilità.

Nel secondo caso invece il risultato sarà una rete sparsa, e questo fa in modo che rimanga una buona performance. La matrice finale sarà più piccola, quindi abbiamo meno operazioni.

- **structured** aims to preserve network density for computational efficiency (faster computation at the expense of less flexibility) by removing groups of weights

- **unstructured** is unconstrained to which weights or activations are removed but the sparsity means that the dimensionality of the layers does not change.

- Hence, sparsity in unstructured pruning techniques provide good performance at the expense of slower computation.
- For example, MBP produces a sparse network that requires sparse matrix multiplication (SMP) libraries to take full advantage of the memory reduction and speed benefits for inference. However, SMP is generally slower than dense matrix multiplication and therefore there has been work towards preserving subnetworks which omit the need for SMP libraries

Uno dei vantaggi del metodo structured è che può essere usato anche nei layer convoluzionali. Quindi potremmo togliere un filtro alla filter bank.

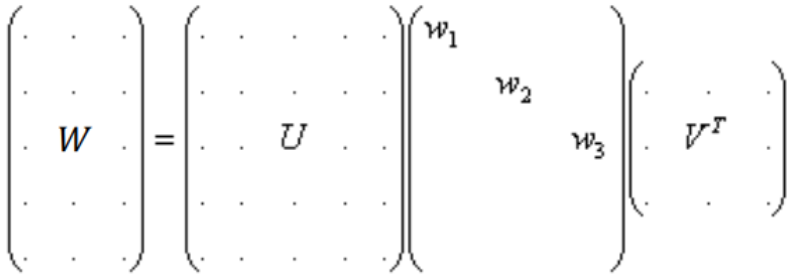
Low rank matrix and tensor decomposition

L'idea di questo approccio arriva dal fatto che la maggior parte dei parametri è nella parte fully connected. Possiamo anche implementare layer fully connected con una moltiplicazione a singola matrice. Quindi se abbiamo N unità nell'input e M nell'output, tutti i possibili pesi possono essere salvati in una matrice NxM. Poi, possiamo fare una decomposizione a singoli valori, di modo che la matrice W possa essere espressa come la moltiplicazione di 3 diverse matrici. Abbiamo che la matrice S è diagonale, con valori che diminuiscono man mano (i w).

$$W = USV^T$$

$$W \in \mathbb{R}^{m\times k}, U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times k}, V^T \in \mathbb{R}^{k\times k}$$

S is diagonal, decreasing magnitudes of SV along the diagonal



L'idea è che se manteniamo solamente i valori più grandi, ora abbiamo nuove matrici dove abbiamo rimosso colonne e righe di modo che alla fine della moltiplicazione, la nuova W ha la stessa dimensione di quella originale. Ma ora il rank della nuova matrice è più piccolo di quella originale.

- By only keeping the t (< k) singular values with largest magnitude:

$$\widetilde{W} = \widetilde{U}\widetilde{S}\widetilde{V^T}$$

$$\widetilde{W} \in \mathbb{R}^{m\times k}, \widetilde{U} \in \mathbb{R}^{m\times t}, \tilde{S} \in \mathbb{R}^{t\times t}, \widetilde{V^T} \in \mathbb{R}^{t\times k}$$

So that now:

$$Rank(\widetilde{W}) = t < k = Rank(W)$$

Quindi riducendo il numero di colonne della prima matrice, delle right nell'ultima, e riducendo S a txt, il risultato è comunque una matrice della stessa dimensione di quella originale, però è un'approssimazione con rank piccolo dell'originale.

Il vantaggio è che originalmente avevamo m*k elementi, mentre ora invece abbiamo m*t + t + t*k

Before:

$$W = USV^T \qquad W \in \mathbb{R}^{m\times k}, U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times k}, V^T \in \mathbb{R}^{k\times k}$$

Now:

$$\widetilde{W} = \widetilde{U}\widetilde{S}\widetilde{V^T} \qquad \widetilde{W} \in \mathbb{R}^{m\times k}, \widetilde{U} \in \mathbb{R}^{m\times t}, \tilde{S} \in \mathbb{R}^{t\times t}, \widetilde{V^T} \in \mathbb{R}^{t\times k}$$

Storage for W : $O(mk)$

Storage for $\widetilde{W}$: $O(mt + t + tk)$

Compression rate: $O(\frac{mk}{mt+t+tk})$

Theoretical error: $\left\|AW - A\widetilde{W}\right\|_F \leq w_{t+1}\|A\|_F$

Quindi abbiamo un compression rate che dipende da quanti valori singoli stiamo mantenendo. La differenza quando moltiplichiamo lo stesso input con la matrice originale e con la matrice approssimata è dominato dai ?? valori singoli che stiamo (o non stiamo?) usando.

L'idea è che non salviamo la matrice W, ma invece salviamo U, V, S così che abbiamo un modello più piccolo. Useremo questa operazione per recuperare la matrice originale, e useremo la matrice approssimata perchè sappiamo che il risultato è molto vicino a quello originale.

Knowledge distillation

L'idea di questo approccio è che vogliamo imparare un modello piccolo da una rete grande, usando la rete grande come supervisione per il modello piccolo, e poi usando un'entropia, o una distanza, tra le loro stime.

Quindi il primo modello sta insegnando al secondo modello.

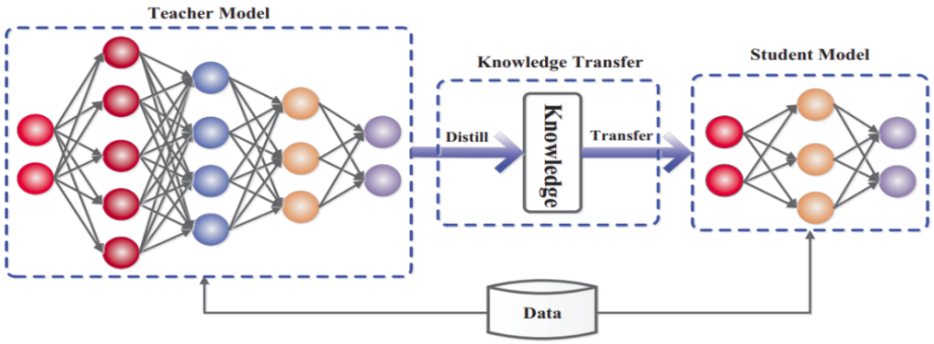
è stato dimostrato che imparare in questo modo permette di raggiungere risultati migliori rispetto a imparare da zero con un modello nuovo più piccolo.

Quindi non stiamo imparando solamente dalla ground truth (classe), ma invece il modello ci da le probabilità di tutte le classi, e vogliamo che il modello piccolo imiti

questo comportamento. Quindi sta imparando come il modello più grande prende decisioni.

I modelli di transformer sono spesso distillati, di modo che possano essere eseguiti in locale.

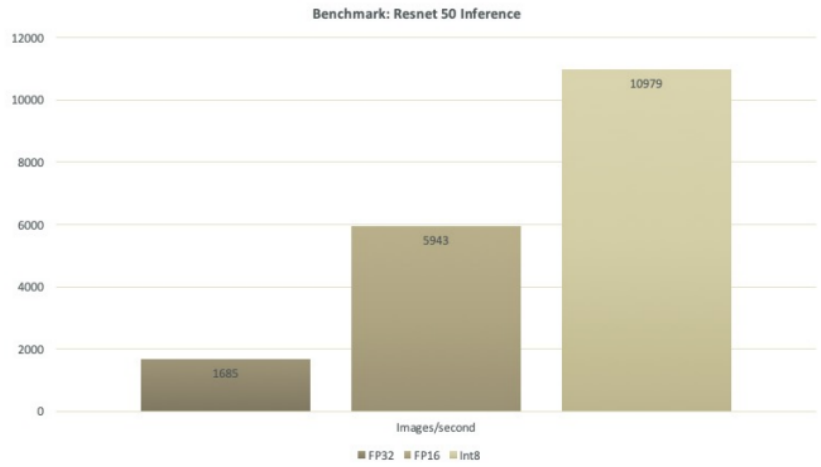
Se supponiamo di avere molti dati senza label, possiamo usare il modello principale per creare i label (non per forza sono giusti, ma noi vogliamo proprio imparare le decisioni del modello). A questo punto trainiamo il secondo modello su questi dati labellati dal primo modello.



Quantization

La quantizzazione è il processo di rappresentare i valori con un numero ridotto di bit. Questo può essere applicato come una seconda operazione (nel primo step per esempio rimuoviamo delle connessioni).

Tipically, when training on the GPU, values are stored in 32-bit floating point (FP) single precision. Half-precision for floating point (FP-16) and integer arithmetic (INT-16) are also commonly considered.



Lo score è il throughput, quante immagini al secondo. Vediamo come solo spostandoci da float32 a float16 la performance aumenti significativamente. Allo stesso modo, ridurre i bits della rappresentazione fino all'int, il throughput aumenta.

Si può anche usare una precisione di integer ancora più bassa, 4bit, 2bit o anche 1bit.

1bit sembra impossibile ma c'è un paper del 2024 che dice che tutti i LLM sono in 1.58bit (quindi 1, -1 e 0). Hanno mostrato che in questo modo possono ottenere una velocità molto superiore nella fase di inferenza, e una dimensione su disco più piccola, e un costo delle operazioni più basso (perchè al posto di fare una moltiplicazione, cambia il segno e fa addizioni).

Dobbiamo menzionare anche che avere diverse rappresentazioni significa che abbiamo anche diversi range di numeri che possiamo rappresentare.

- For the range of signed integers with n bits, we represent a range of [-2n-1, 2n-2] and for full precision (FP-32) the range is $\pm 3.4e38$.

Se avessimo bisogno di più rappresentazioni nell'intervallo 0-1, possiamo avere uno scaling factor alpha che muove il range nella zona di cui abbiamo bisogno (così possiamo usare gli int).

- For integers to be used to represent weight matrices and activations, a FP scale factor is often used → hence many quantization approaches involve a hybrid of mostly integer formats with FP-32 scaling numbers. This approach is often referred to as mixed-precision (MP)

Low resource and efficient architectures

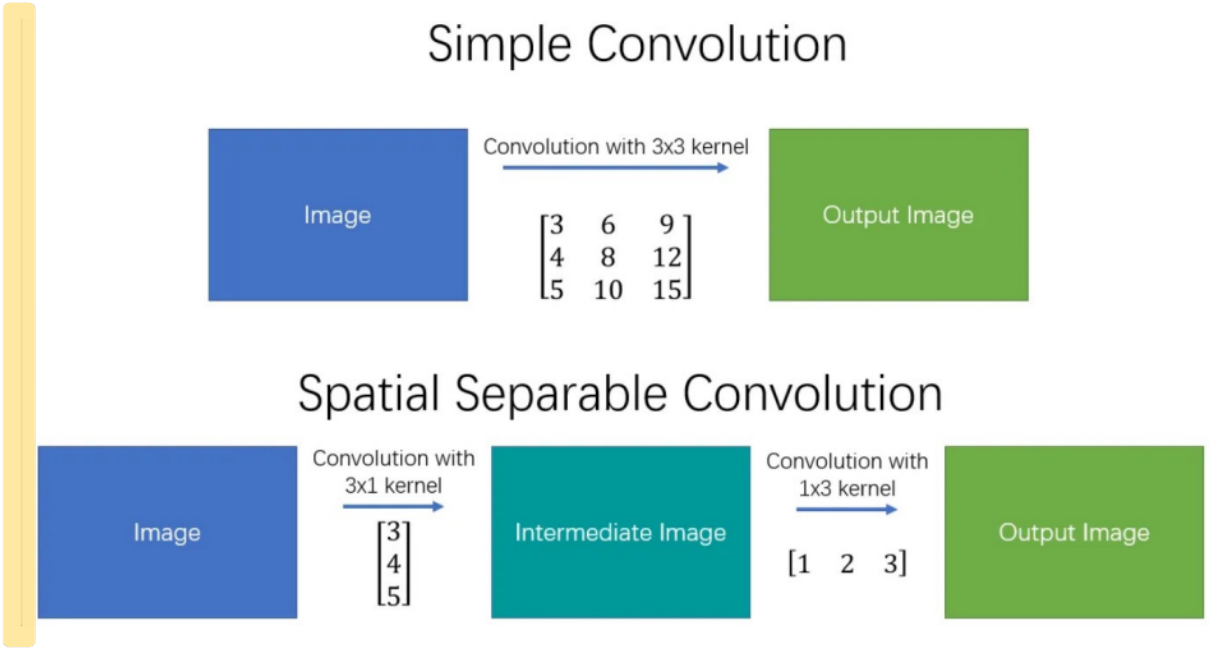
Un'architettura famosa che è molto efficiente è la famiglia MobileNet, che è fatta apposta per essere eseguita su device mobile. In particolare, nell'architettura è stato introdotto in concetto di **depth-wise separable convolutions (DSC)**.

L'idea è di fattorizzare la convoluzione in una depth-wise convolution seguita da una 1×1 convolution.

(Spatially) Separable convolution

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Una convoluzione 3×3 può essere sostituita da una 1×3 seguita da una 3×1, in questo modo riduciamo di un terzo il numero di parametri.



Abbiamo ridotto il numero di parametri, e anche di operazioni. Quindi abbiamo un'inferenza più veloce anche.

SqueezeNet è un'altra rete che ha introdotto altri spatial layers chiamati squeeze layers. Non andiamo nel dettaglio.

SqueezeNet (fire module)

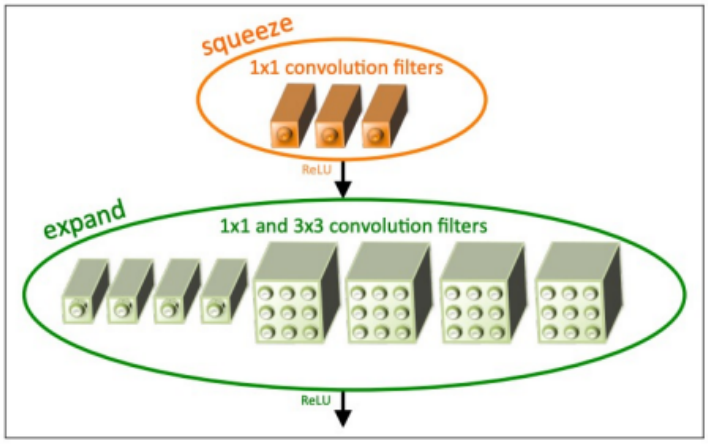


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

ShuffleNet è stata creata per mischiare i canali di output ad un certo punto, per aumentare l'informazione che passa nel resto della rete.

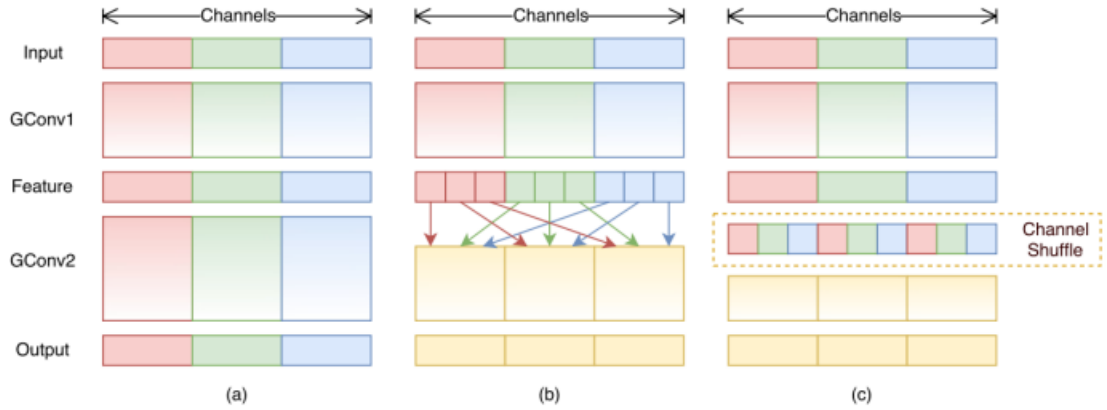


Figure 1. Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

DenseNet usa in modo efficiente i parametri, riutilizzando le mappe di attivazione multiple volte nella rete, in modo simile a ResNet con le connessioni che skippano un layer.

DenseNet

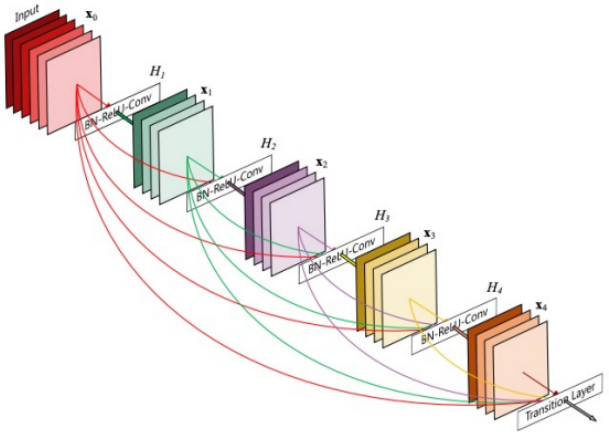
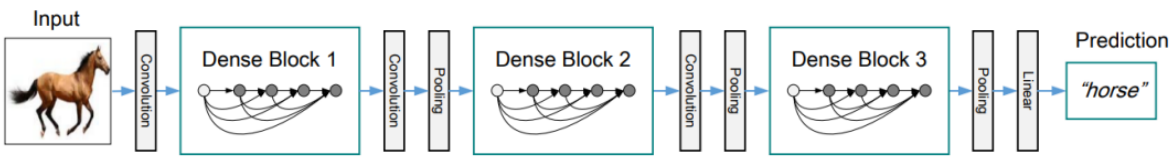


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.



Il problema è che anche se il numero totale di parametri è piccolo, la quantità di memoria usata è alta. Perché bisogna salvare le informazioni per molti layer, perchè dovrà essere riutilizzata.