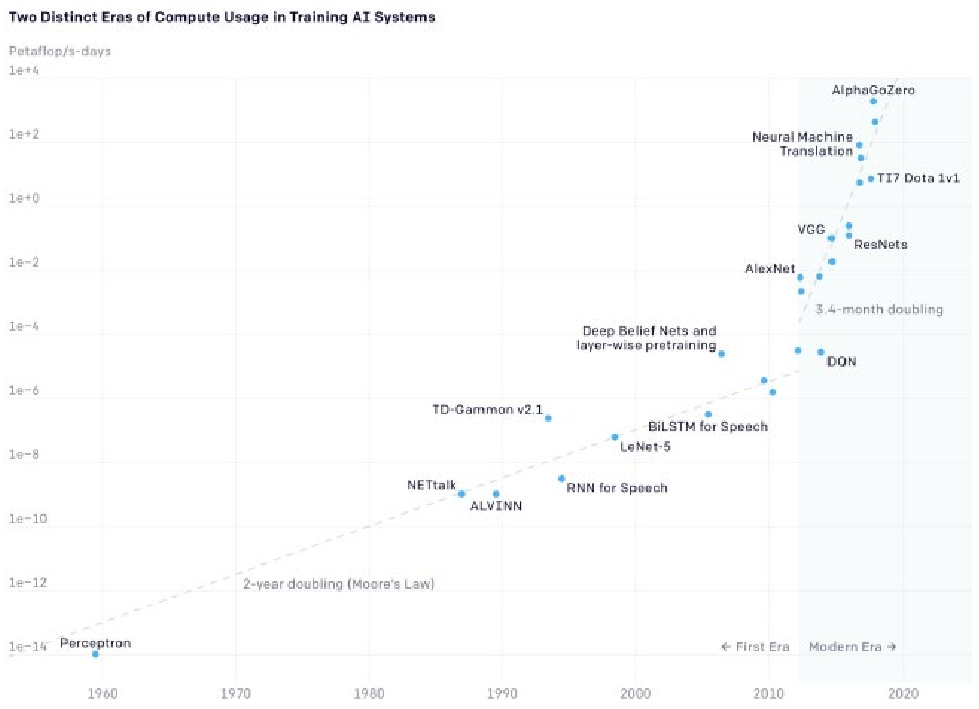
Chapter 12 – Model Compression

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Introduction

Essentially, we know that training models is very expensive in terms of electric means but also in terms of CO_2 footprint. Also costs are a lot high. This is due to the usage of high-performance hardware and also due to the size of the models.



Model	Hardware	Power (W)	Hours	kWh·PUE	CO_2e	Cloud compute cost
Transformer$_{base}$	P100x8	1415.78	12	27	26	\$41–\$140
Transformer$_{big}$	P100x8	1515.43	84	201	192	\$289–\$981
ELMo	P100x3	517.66	336	275	262	\$433–\$1472
BERT$_{base}$	V100x64	12,041.51	79	1507	1438	\$3751–\$12,571
BERT$_{base}$	TPUv2x16	—	96	—	—	\$2074–\$6912
NAS	P100x8	1515.43	274,120	656,347	626,155	\$942,973–\$3,201,722
NAS	TPUv2x1	—	32,623	—	—	\$44,055–\$146,848
GPT-2	TPUv3x32	—	168	—	—	\$12,902–\$43,008

AlphaGo: 1920 CPUs and 280 GPUs,
\$3000 electric bill per game

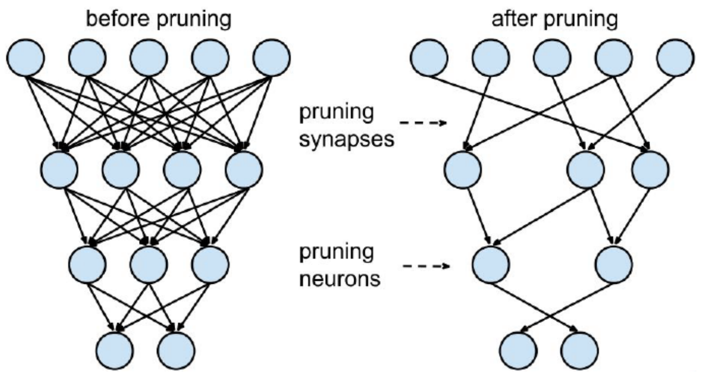
on mobile: drains battery
on data-center: increases TCO

2. Possible Solutions

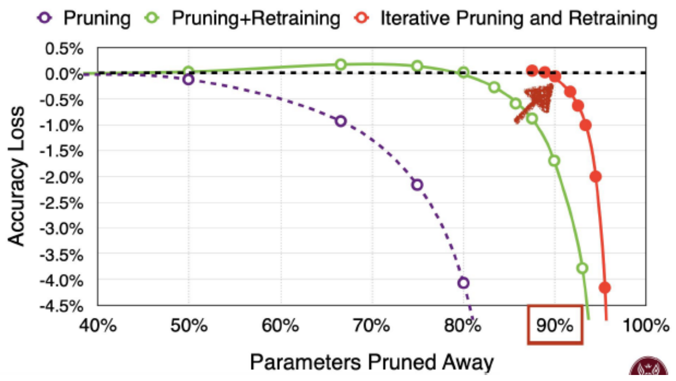
2.1 Pruning

One thing that we could do with Neural Networks is **pruning**, where not all of the neurons are needed, so we either remove the connections between them or we remove directly remove neurons.

For pruning, one evaluates the importance of neurons, then remove the least important ones and fine-tune the network again. Then we should ask whether to continue pruning or not based on some criteria (for instance: accuracy drop more than 10% / network size smaller than X% etc.).

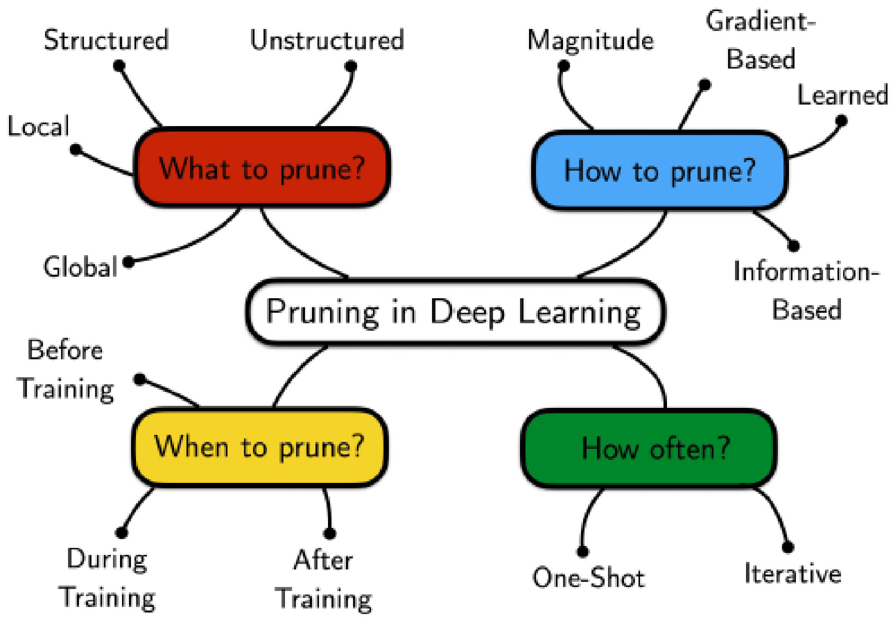


There is also a Train – Prune – Recovery process, which is pretty self-explanatory. We substantially prune and retrain, having always 0% accuracy loss but with 90/95% of parameters pruned.



There are some cons though, since pruning cannot be made efficient on parallel hardware that exploit efficient matrix multiplication hardware components (GPUs/TPUs).

2.2 The General Overview of Pruning



2.3 Lottery Ticket Hypothesis

The question here involves training an *efficient* pruned network. In order to explore this section, we introduce the **lottery ticket hypotesis:**

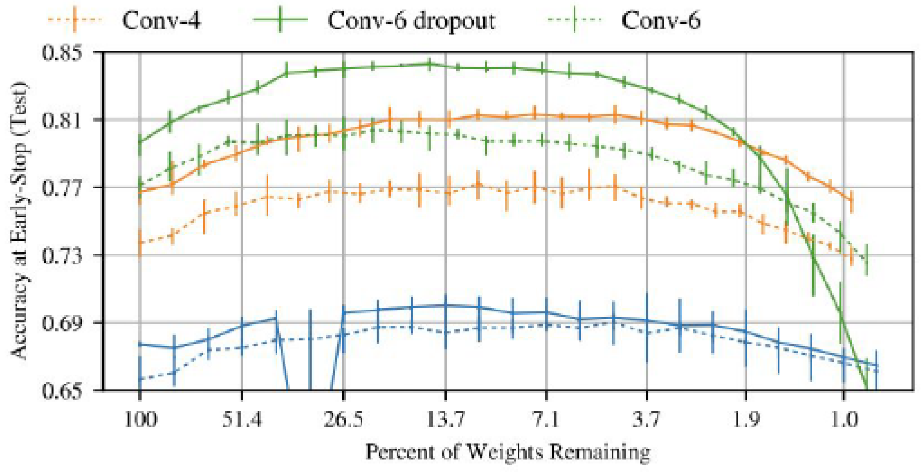
“A randomly-initialized dense neural network (where weights are not set to 0) contains a sub-network that is initialized such that when trained in isolation, it can match the test accuracy of the original network after training for at most the same number of iterations”.

More formally, for defining the pruned network, we have a pruning mask $m \in \{0, 1\}$ which is element-wise multiplied by θ (from original network), effectively having a new network initialized as $m \odot \theta_0$.

The lottery ticket hypotesis **states** that there exist an m such that its training time is almost the same (with accuracy equal or even higher) than the original network with significantly fewer parameters. ($\left||m|\right|_0 \ll |\theta|$).

Schematically:

- Randomly initialize a neural network $f(x;\ \theta_0)$ (where $\theta_0 \sim D_\theta$).
- Train the network for j iterations, arriving at parameters θ_j.
- Prune $p\%$ of the parameters in θ_j, creating a mask m.
- Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x;\ m \odot \theta_0)$.
- Here are some results of CNN pruning with **Dropout** as well. We can see that with simply 3.7% of weights remaining, we still achieve higher performance than normal CNN prune.



2.4 Quantization

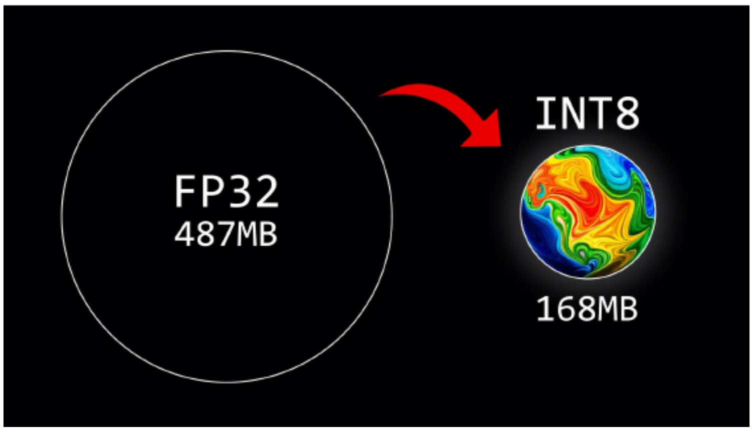
The most effective way to compress a model is applying quantization.

It's a technique where, to save memory, weights can be stored using lower-precision data types ($FP32 -> \ FP16, INT8$ and $INT4$).

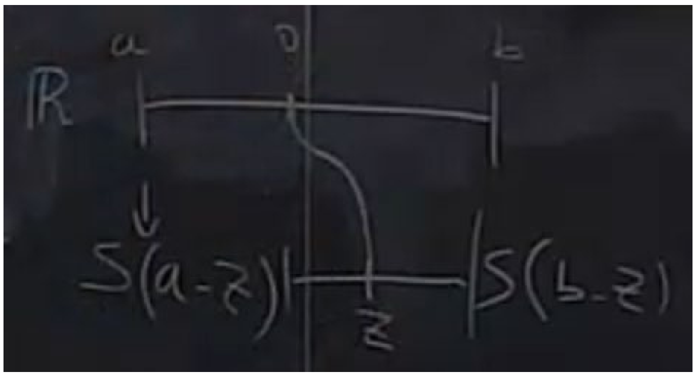
For example, $INT8$ has 3x acceleration ratio along with obvious advantages in power consumption.

In the literature, we have the **Post-Training Quantization** where we train a network (as we usually do) and then take the weights and quantize them, so that we can store a light model. Also, we have **Quantization-Aware Training (QAT)** which aims at training the network as we normally do and then fine-tune the weights through quantization.

Also, it has been proved that from 32-Bit Floating Point to 8-Bit Integer (as in the right image) we can have 75% reduction in memory (and execution time) with very low performance drops.

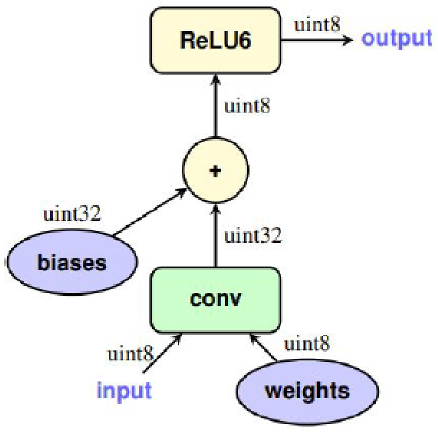


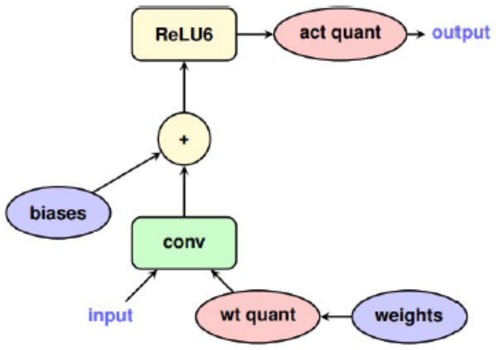
Talking about **QAT**, it follows a quantization scheme which aims at finding 2 parameters: S (Scale) that shrinks down the real number and Z (Zero) which it's the center of the distribution of the quantized values (in other words it's the mapping of the real number 0 in the quantized version). In this way, given q as the quantized representation, our real number r is:



$$r = S(q - Z)$$

Where S is a floating point and Z is an integer.


ReLU6
uint8
output
uint8
+
uint32
biases
uint32
conv
uint8
input
uint8
weights


ReLU6
act quant
output
+
biases
conv
input
wt quant
weights