Model Compression

Ivan Skorokhodov

July 2, 2020

Overview

Pruning

Pruning

- *Pruning* is removing weights/neurons in a model while preserving the accuracy
- It can be done at different stages:
  - before training (*foresight pruning*)
  - during training
  - after training
  - iteratively train/prune several times

Pruning weights

- Simplest strategies:
  - Apply L_1-regularization during training
  - Prune based on weights magnitudes after training
  - Iterative Magnitude Pruning (IMP): "train $\rightarrow$ prune by magnitude $\rightarrow$ restore original init" several times
- Variational dropout:
  - Perform a variational inference for model weights
  - Obtain μ_i, σ_i^2 associated with each weight
  - If σ_i^2 is large, then the weight is not important $\Rightarrow$ prune it
- Single-Shot Network Pruning (SNIP) prunes at initialization [7]:
  - Compute how much a weight influences the loss:

$$S\left(\theta_i\right) = \lim_{\epsilon \rightarrow 0} \left| \frac{\mathcal{L}\left(\theta_i\right) - \mathcal{L}\left(\theta_i + \epsilon\right)}{\epsilon} \right| = \left| \frac{\partial \mathcal{L}}{\partial \theta_i} \right| \quad (1)$$

  - Prune weights with low scores

Lottery Ticket Hypothesis (LTH)

LTH [4]: *A neural network contains a subnetwork (winning ticket) which, when being trained in isolation, achieves the same accuracy as the base model and does so in the same number of iterations or even faster.*

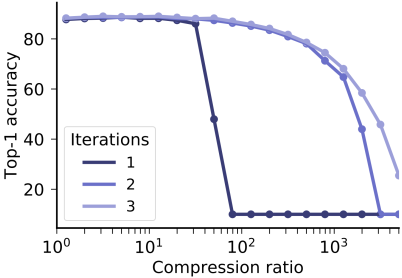
One finds a winning ticket with Iterative Magnitude Pruning (IMP) in n iterations:

1. Randomly initialize a neural network f_{θ_0}, initialize a mask $m^{(0)}$ of all-ones.
2. Train for T iterations, obtain θ_T.
3. Remove $p^{1/n}$ percent of the lowest-magnitude parameters from θ_T by updating the mask $m^{(\ell)}$.
4. Reset the remaining parameters to $\theta_0 \odot m^{(\ell)}$.[1]
5. Repeat steps 2-4 n times, obtain the winning ticket $\theta_0 \odot m^{(n)}$.

[1] [5] found out that it is more beneficial to *rewind* the weights to some θ_k with $k \ll T$ instead of resetting them to θ_0

LTH: caveats

- [6, 9] showed that LTH idea works poorly for large datasets
- IMP requires several full training procedures to work well



- This motivates *foresight pruning*: removing weights once and without training [7, 13, 12]

Synaptic saliency

- All pruning algorithms remove weights based on score values $S(\theta_i)$ associated with each weight θ_i
- These scores can often be represented as

$$S(\theta) = \frac{\partial R}{\partial \theta} \odot \theta \tag{2}$$

  for a function $R(\theta)$ which *operates on top of model outputs* $f_\theta(x)$
- If a score function can be represented as (2) then it is called *synaptic saliency* [12]
- Intuitively, it represents weight importances (but in general can be negative)

Synaptic saliency in real life

Many pruning algorithms have a form similar to (2):

- Skeletonization [10]:

$$\mathcal{R}(\theta) = -L(\theta) \Longrightarrow S(\theta) = -\frac{\partial \mathcal{L}}{\partial \theta} \odot \theta \tag{3}$$

- SNIP [7] has a similar form:

$$S(\theta) = \left| \frac{\partial \mathcal{L}}{\partial \theta} \odot \theta \right| \tag{4}$$

- GraSP [13] has a similar form:

$$S(\theta) = \left(-\mathcal{H} \frac{\partial \mathcal{L}}{\partial \theta} \right) \odot \theta \tag{5}$$

- Magnitude Pruning is not a synaptic saliency:

$$S(\theta_i) = \theta_i^2 = \frac{\partial \mathcal{R}(\theta)}{\partial \theta} \odot \theta \quad \text{for} \quad \mathcal{R}(\theta) = \frac{1}{2} \|\theta\|_2^2, \tag{6}$$

  but $\mathcal{R}(\theta)$ is not a function of outputs!

Neuron saliency

Synaptic saliency gives rise to *input/output neuron saliency*. Let:

- $\Omega^{\text{in}}(\nu)$ be the set of incoming weights for neuron ν
- $\Omega^{\text{out}}(\nu)$ be the set of outcoming weights for neuron ν

Then:

$$S_\nu^{\text{in}} = \sum_{\theta_i \in \Omega^{\text{in}}(\nu)} S(\theta_i) \quad \text{and} \quad S_\nu^{\text{out}} = \sum_{\theta_i \in \Omega^{\text{out}}(\nu)} S(\theta_i) \tag{7}$$

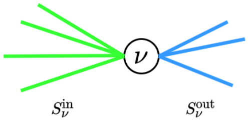


Figure: Neuron saliency

Layer saliency

Neuron saliency gives rise to a *layer saliency*. Let x^ℓ be a vector of neurons in layer ℓ, then layer saliency S^ℓ is:

$$S^\ell = \sum_i S^{\text{in}}_{x_i^\ell} \tag{8}$$

One can show that for a homogenous activation function ϕ we have[2]:

$$S^{\text{in}}_\nu = \frac{\partial R}{\partial x_\nu} x_\nu \quad \text{and} \quad S^{\text{out}}_\nu = \frac{\partial R}{\partial \phi(x_\nu)} \phi(x_\nu), \tag{9}$$

where x_ν is a neuron's value before the non-linearity. This gives us:

$$S^\ell = \left\langle \frac{\partial R}{\partial x^\ell}, x^\ell \right\rangle \tag{10}$$

[2] $\phi(x) = x \cdot \phi'(x)$ — holds for ReLU, LeakyReLU, linear

Conservation laws

Authors proved two conservation laws. If a model's activations are homogenous, then:

1. (Neuron-wise conservation) $S_\nu^{\text{in}} = S_\nu^{\text{out}}$ for all neurons ν
2. (Network-wise conservation) $S^\ell = S^{\ell'}$ for all layers ℓ, ℓ'

Conservation laws in practice

Authors measured how the conservation holds in practice:

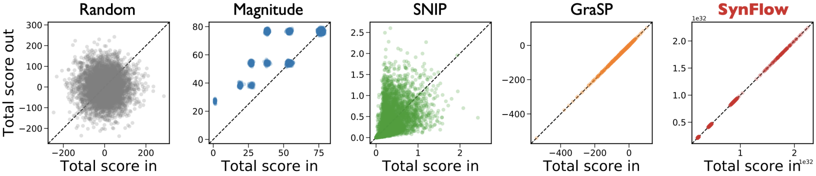


Figure: Input/output score values for each neuron at initialization for VGG-19 model on ImageNet

Layer collapse

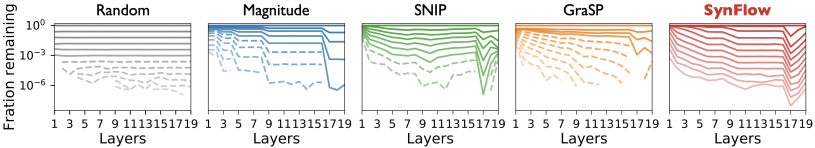


Figure: Fraction of weights remaining in each layer for VGG-19 at initialization on ImageNet. Dashed lines indicates that there is at least one layer without any parameters at all.

- Many pruning algorithms suffer from layer collapse: they prune all the weights in a single layer
- In practice, large layers become over-pruned
- Network-wise conservation law explains this:
  - since $S^{\ell} = S^{\ell'}$ for all layers, then parameters in a large layer on average have much smaller scores

Alleviating layer collapse

- To formalize layer-collapse, authors state the "Maximal Critical Compression Axiom": *a pruning algorithm is good, if it does not prune all the parameters in a single layer if there is something left to prune in other layers*
- One way to avoid layer collapse is local masking: prune each layer individually, but this performs far worse [12]
- Instead authors provide the following solution

Theorem. *If a pruning algorithm, with global-masking, assigns positive scores that respect layer-wise conservation and if the algorithm re-evaluates the scores every time a parameter is pruned, then the algorithm satisfies the Maximal Critical Compression axiom.*

Synaptic Flow

Following the theorem's idea they propose *Synaptic Flow (SynFlow)* pruning algorithm:

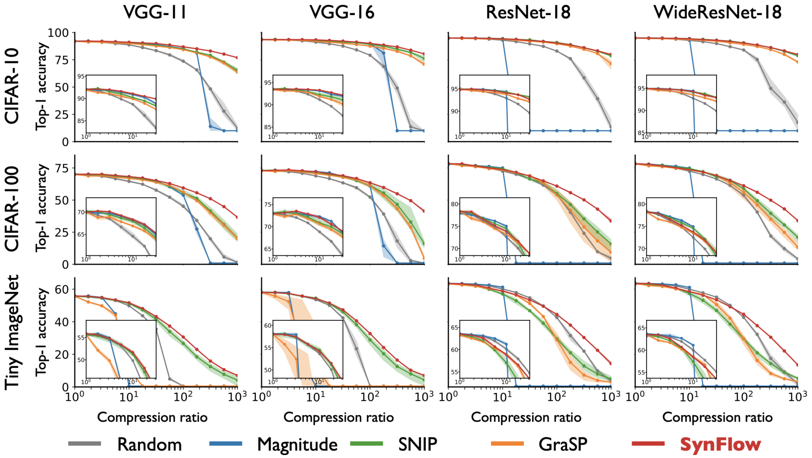
$$\mathcal{R}_{\mathrm{SF}} = \mathbb{1}^T \left(\prod_{\ell=1}^{L} \left| \theta^{[\ell]} \right| \right) \mathbb{1} \tag{11}$$

One can show that:

$$\mathcal{S}_{\mathrm{SF}}\left(w_{ij}^{[\ell]}\right) = \underbrace{\left[\mathbb{1}^\top \prod_{k=\ell+1}^{N} \left| W^{[k]} \right| \right]_i}_{\text{further connections importance}} \cdot \left| w_{ij}^{[\ell]} \right| \cdot \underbrace{\left[\prod_{k=1}^{\ell-1} \left| W^{[k]} \right| \mathbb{1} \right]_j}_{\text{previous connections importance}} \tag{12}$$

- It is an iterative algorithm, applied $n = 100$ times in practice (at initialization)
- The magic is that it does not use any data!

SynFlow results



A problem with SynFlow

- $\mathcal{R}_{\mathsf{SF}}(\theta)$ is not a function of the outputs (but "close": it is a function of the outputs of a model with all weights replaced with their absolute values).
- If we'll now generalize synaptic saliency definition to encompass such functions, then magnitude pruning would also be a positive synaptic saliency
- Then it would have to satisfy a theorem that it does not lead to layer collapse when applied iteratively (but it does)

Pruning neurons (*structured pruning*)

Pruning neurons is usually based on weights pruning:

- Magnitude pruning: prune neurons which weights have the lowest magnitude
- Vardrop pruning: prune neurons which weights have the highest variance

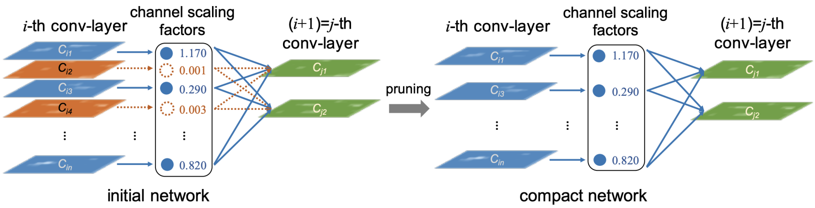
It is usually required to fine-tune the model afterwards.

Network slimming

- Network Slimming [8] associates a scaling parameter $\gamma_{\ell,k}$ for channel k in layer ℓ and regularizes them during training:

$$\mathcal{L}(\theta, \gamma) = \mathcal{L}_{\text{data}}(\theta, \gamma) + \lambda \sum_{\ell,k} \|\gamma_{\ell,k}\| \tag{13}$$

- After training it is necessary to fine-tune the pruned model
- These scalings can be merged with BatchNorm



Network Slimming results

(a) Test Errors on CIFAR-10

Model	Test error (%)	Parameters	Pruned	FLOPs	Pruned
VGGNet (Baseline)	6.34	20.04M	-	7.97×10^8	-
VGGNet (70% Pruned)	**6.20**	2.30M	88.5%	3.91×10^8	51.0%
DenseNet-40 (Baseline)	6.11	1.02M	-	5.33×10^8	-
DenseNet-40 (40% Pruned)	**5.19**	0.66M	35.7%	3.81×10^8	28.4%
DenseNet-40 (70% Pruned)	5.65	0.35M	65.2%	2.40×10^8	55.0%
ResNet-164 (Baseline)	5.42	1.70M	-	4.99×10^8	-
ResNet-164 (40% Pruned)	**5.08**	1.44M	14.9%	3.81×10^8	23.7%
ResNet-164 (60% Pruned)	5.27	1.10M	35.2%	2.75×10^8	44.9%

(b) Test Errors on CIFAR-100

Model	Test error (%)	Parameters	Pruned	FLOPs	Pruned
VGGNet (Baseline)	26.74	20.08M	-	7.97×10^8	-
VGGNet (50% Pruned)	**26.52**	5.00M	75.1%	5.01×10^8	37.1%
DenseNet-40 (Baseline)	25.36	1.06M	-	5.33×10^8	-
DenseNet-40 (40% Pruned)	**25.28**	0.66M	37.5%	3.71×10^8	30.3%
DenseNet-40 (60% Pruned)	25.72	0.46M	54.6%	2.81×10^8	47.1%
ResNet-164 (Baseline)	23.37	1.73M	-	5.00×10^8	-
ResNet-164 (40% Pruned)	**22.87**	1.46M	15.5%	3.33×10^8	33.3%
ResNet-164 (60% Pruned)	23.91	1.21M	29.7%	2.47×10^8	50.6%

Figure: Pruning can also have a regularization effect: pruned models perform better

Pruning caveats

- Pruning weights (theoretically) reduces the number of FLOPs, but:
  - Resulted sparse matrices are not “sparse enough” to provide practical benefits (sparse matrix-vector multiplications are usually based on non-parallel computations)
  - [6, 9] claim that modern SotA weight-pruning algorithms work poorly on large datasets
- Pruning neurons speeds up a model, but:
  - [9] argues that training the pruned model from scratch would give the same performance
  - So the main value is in optimizing the architecture

Break

Hashing

Simple hashing

1. Imagine that we are have a model $f_\theta(x)$ with $\|\theta| = n$
2. Create a *pool of variables* ν s.t. $\|\nu\| \ll n$
3. Hash elements θ_i into ν_j with a hashing function $h : i \mapsto j$ (with collisions).
4. Then we can optimize ν instead of θ

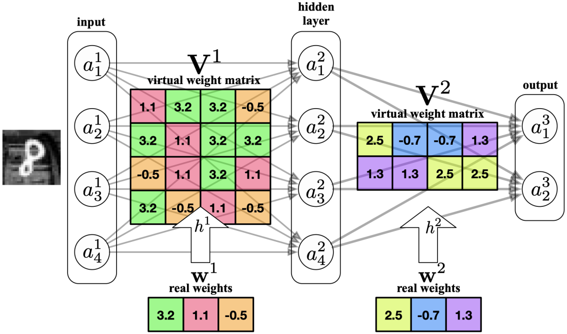


Figure: HashedNet illustration [1]. This can be seen as random weight sharing

Multi-hashing

- One can obtain greater flexibility with *multi-hashing*
- Create m variables buckets $\nu_1, ..., \nu_m$ and m hashing functions $h_1, ..., h_m$
- Each h_k maps parameter's index i into its bucket's index $\nu_k[h_k(i)]$
- To obtain the value for θ_i, compute:

$$\theta_i = \sum_{k=1}^{m} \nu_k[h_k(i)] \tag{14}$$

- Instead of summation one can use other reducing functions, like product, max, min, etc.

Structured Multi-Hashing

- Traditional multi-hashing lacks memory locality which makes slows things down
- [2] solves the problem with a special hashing function
- First, they reshape parameters into matrix M

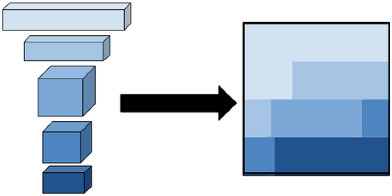


Figure: Structred Multi-Hashing parameters reshaping

- Then they factorize this matrix into low-rank product $M = UV$
- Now θ_i can be seen as a multi-hashing reduction:

$$\theta_i = U_{(s)}^{\top} V^{(t)} = \sum_{k=1}^{m} U_{(s),k} V_k^{(t)} \tag{15}$$

Structured Multi-Hashing results

Compression Method	Target Model Size	Accuracy	Samples Per Second
SMH	5.3M	**0.774**	6060
1X Hash	5.3M	0.762	4000
2X Hash	5.3M	0.765	2800
10X Hash	5.3M	0.770	790
SMH	7.9M	**0.782**	6040
1X Hash	7.9M	0.773	3900
2X Hash	7.9M	0.775	2500
10X Hash	7.9M	0.779	760

Model	Accuracy	Model Size
B0	76.3%	5M
SMH_{2M} B4	76.6%	2M
SMH_{3M} B5	78.3%	3M
B1	78.8%	7.9M

Figure: SMH works much faster and obtains better results for EfficientNet-B2 on Imagenet. It also performs well under extreme compression (up to $10\times$) for large datasets

Low-rank decomposition

- Low-rank decomposition usually represents weight tensors as a product of low-rank matrices $W = UV$
- This can be done during training or after training
- [15] represents it as $W \approx L + S$, where L is low-rank and S is sparse
- TTD [11] "tensorizes" weights and biases by reshaping them into d-dimensional tensors and applies *tensor train decomposition*:

$$\mathcal{A}(j_1, \ldots, j_d) = \boldsymbol{G}_1[j_1]\,\boldsymbol{G}_2[j_2] \cdots \boldsymbol{G}_d[j_d] \tag{16}$$

  i.e. each element in tensor $\mathcal{A}$ is computed as a product of small matrices

Quantization

Quantization

- Quantization converts model weights (and sometimes activations) to a lower precision
  - One can train a low-precision model from scratch
  - But a more practical approach is to quantize a model after training (*post-time quantization*)
- Practically interesting precisions are fp16, int8
- Theoretically interesting precisions are fp16, int8, int4, ternary, binary

Quantization

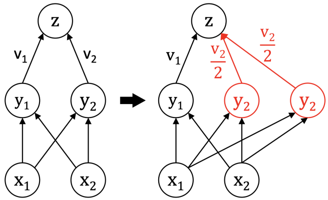
- Any quantizers is defined by scale, shift and precision:

$$Q_p(x, \gamma, \beta) = \text{round}_p\left(\frac{x}{\gamma} + \beta\right) \tag{17}$$

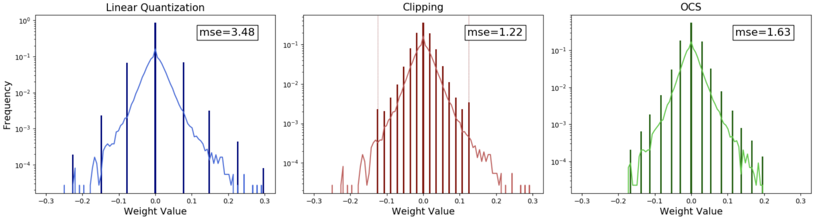
  - precision p defines the range for a value to be rounded to, i.e. for int8 it is $[0, 1, ..., 255]$
  - parameters γ and β normalizes the variable to the required range
- Quantizers use equal distances between quantized values since the common hardware is not supposed to work with non-standard non-linear quantization (i.e. different from floats)
- Usually, precision p is defined by your needs and your hands are tied in changing it
- So the main research question is how to find the best γ and β for your data.

Outlier Channel Splitting (OCS)

- Usually, people select β, γ s.t. KL distance between the quantized values and original values is minimal (for histograms)
- But OCS[16] notices that it makes tails be poorly covered by the quantized distribution
- Previously, people just clipped them away
- Authors remove these outliers (channels with large-magnitude weights) by duplicating them and dividing by 0.5
- This does not change predictions and has negligible overhead



OCS results



(a) (b) (c)

Knowledge distillation

- Knowledge distillation distills a big teacher model M_ϕ into a small student f_θ
- One can use logits instead of one-hot labels since logits carry uncertainty information
- Can be done without data [3]: train the student to produce similar outputs on samples from the generator which tries to fool him.

Architectural tricks

One of the best-performing tricks to compress a model is designing compressed architectures manually:

- Depthwise-separable convolutions
- Groupwise convolutions
- Conv+Bilinear instead of ConvTranspose
- Conv$\rightarrow$ReLU$\rightarrow$MaxPool$\rightarrow$BN instead of Conv$\rightarrow$BN$\rightarrow$ReLU$\rightarrow$MaxPool
- BatchNorm fusion: fuse batchnorm computation into the succeeding layer
- Early-exit: Resnets with dynamic depths that can classify easier examples earlier [14]

- Compression models is important from both practical and theoretical perspective
- Most of the methods are interesting, but not quite practical
- The most practical methods are quantization and “architectural tricks”