From Data to Decisions - M.Sc. Data Science

Compressing deep neural networks

Challenges and theoretical foundations

Presenter: Simone Scardapane

University of Exeter, UK

Table of contents

Schema of a neural network

A (feedforward) neural network (NN) is a *universal approximator* composed of simple units organized in a layered structure:

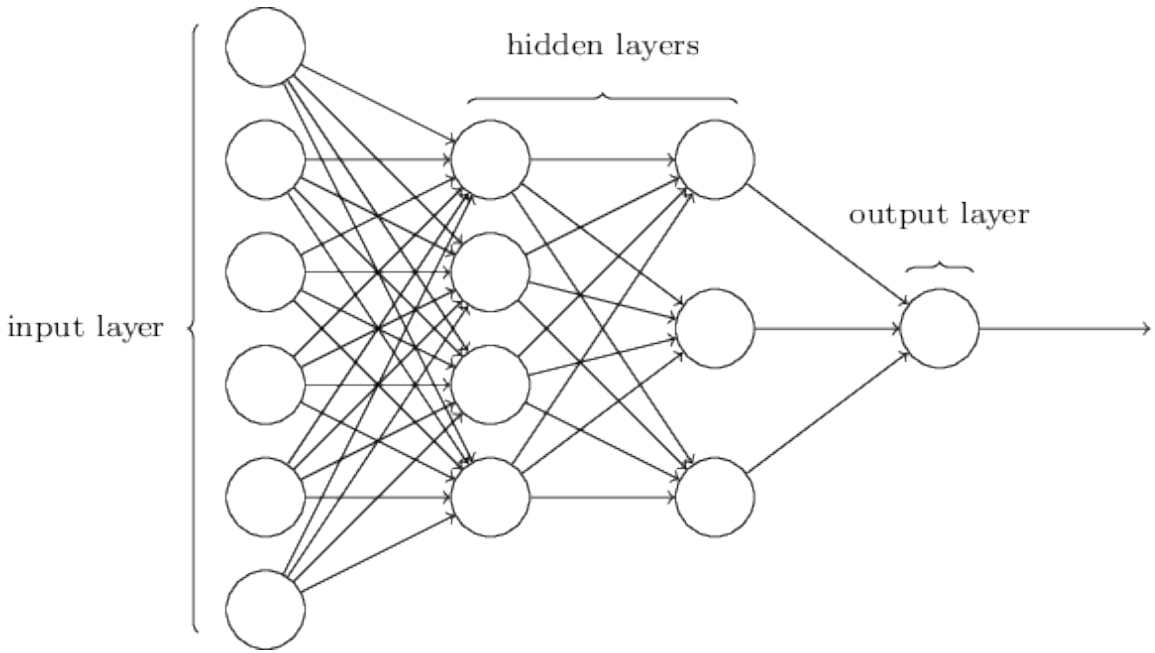


Figure 1 : Schematic depiction of a simple neural network [Image source].

The model of a layer

A layer of a neural network alternates a linear projection with a (generally fixed) nonlinear operation:

$$\mathbf{h}_{l+1} = g\left(\mathbf{W}\mathbf{h}_l + \mathbf{b}\right), \tag{1}$$

where:

- $\mathbf{h}_l / \mathbf{h}_{l+1}$ are the input/output of the layer,
- $\mathbf{W}$ and $\mathbf{b}$ are adapted from data,
- $g(\cdot)$ is a real-valued function, e.g., the rectified linear unit (ReLU):

$$g(s) = \max(0, s). \tag{2}$$

Stacking layers

The neural network is composed by stacking multiple layers to get the final output $\hat{\mathbf{y}}$ from the input $\mathbf{x}$.

Given some training data $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1, \ldots, n$, we train the weights by minimizing some cost function:

$$J(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{y}_i, \hat{\mathbf{y}}_i)\,, \tag{3}$$

where $\mathbf{w} \in \mathbb{R}^Q$ is a vector containing *all* parameters of the network, and $L(\cdot, \cdot)$ is a loss function (e.g., cross-entropy).

Convolutional neural networks

When handling 1D or 2D signals (e.g., images), a common variation is the use of convolutional layers:

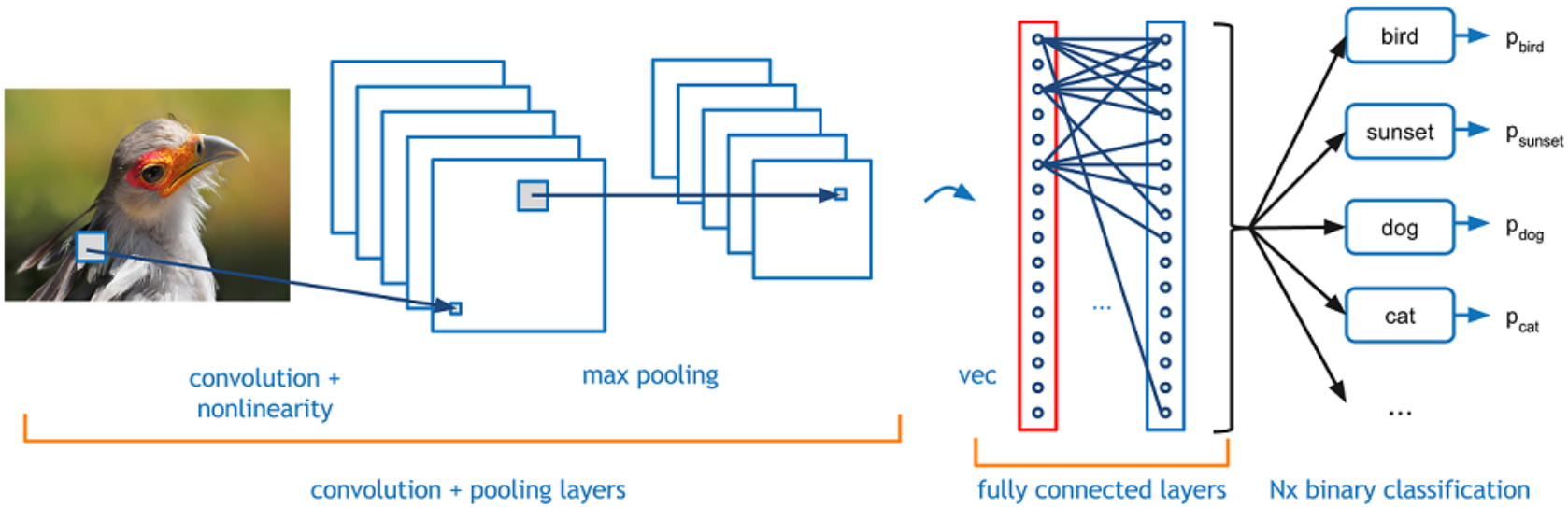


Figure 2 : Depiction of a convolutional neural network (CNN) [Image source].

Optimizing a neural network

Neural networks are generally optimized with stochastic first-order methods, where at each iteration:

1. We sample a mini-batch of data from the entire dataset.
2. We use the mini-batch to compute a (noisy) gradient $\widetilde{\nabla} J(\mathbf{w})$.
3. We use the gradient to move in a descent direction, e.g., by steepest-descent:

$$\mathbf{w} = \mathbf{w} - \eta \widetilde{\nabla} J(\mathbf{w}) \,, \tag{4}$$

where $\eta > 0$ is the learning rate.

---

[1] Bottou, L., Curtis, F.E. and Nocedal, J., 2016. **Optimization methods for large-scale machine learning**. *arXiv preprint arXiv:1606.04838*.

Gradient computation in practice

Most deep learning frameworks allow to efficiently compute the gradient of the loss function (i.e., back-propagation).

As an example, in PyTorch the following code is enough:

```
F.backward()
```

Then the gradient with respect to a variable `w` is given by:

```
print(w.grad)
```

Table of contents

Object recognition with 10k+ classes

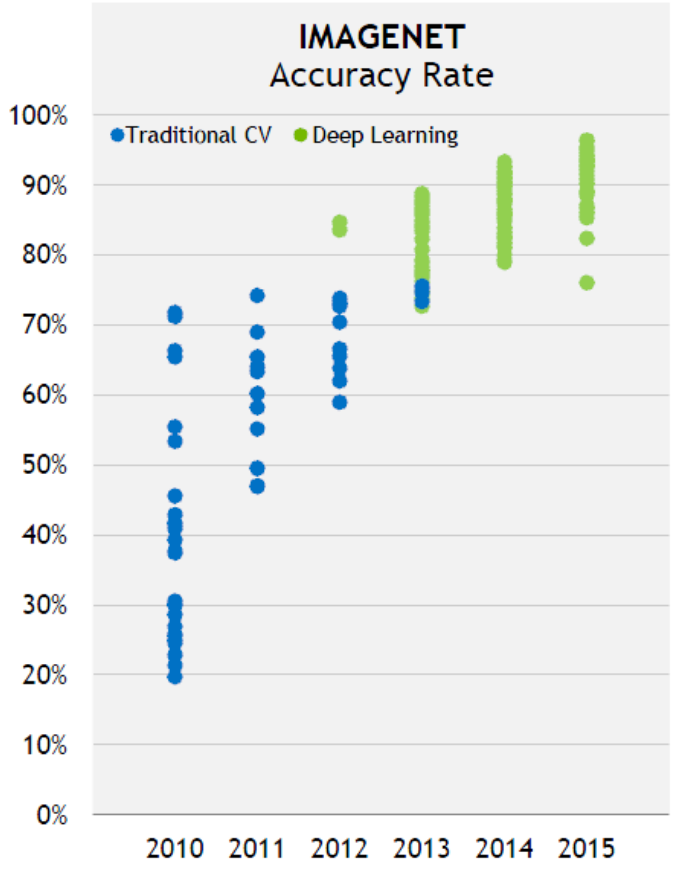


Figure 3 : Accuracy on the ImageNet competition during 2010-2015. 2012 is the first large improvement due to a deep CNN.

Image captioning

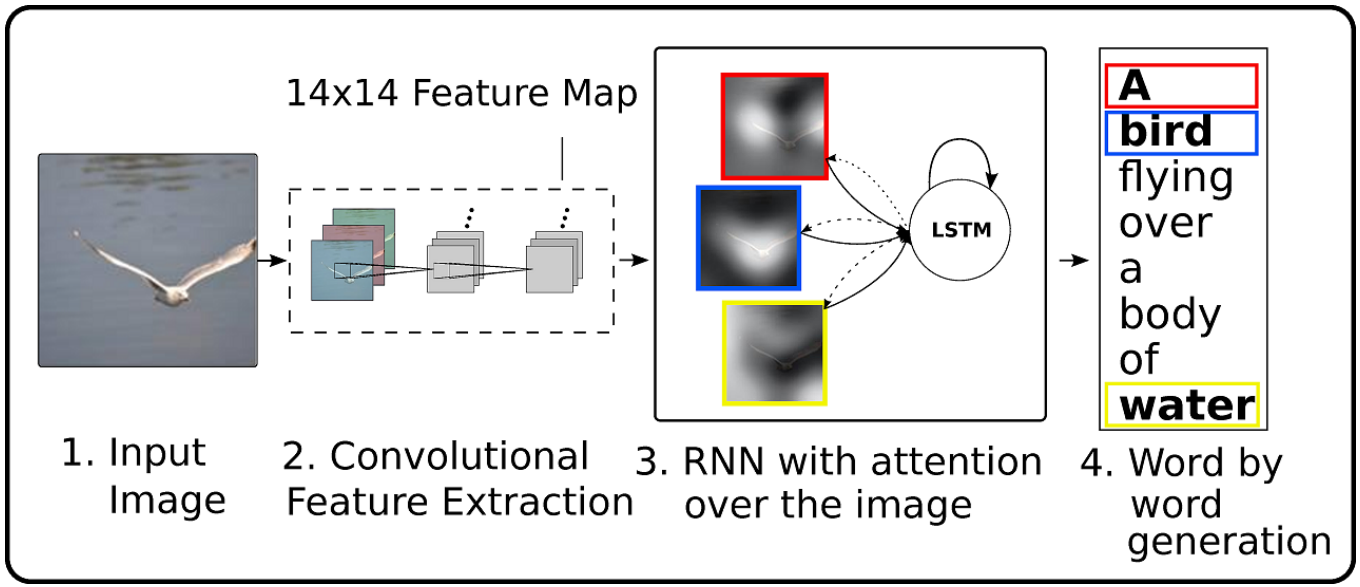


Figure 4 : Recurrent neural networks are NNs endowed with a short-term memory updated at every step [Image source].

Image-to-image translation

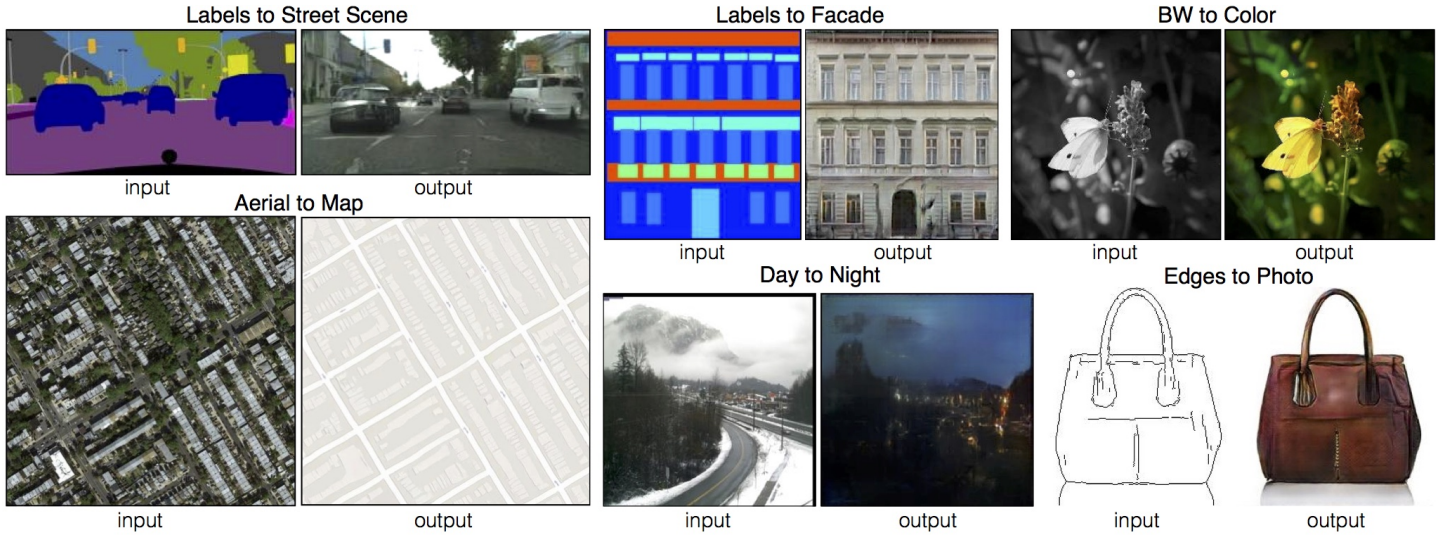


Figure 5 : Isola, P., Zhu, J.Y., Zhou, T. and Efros, A.A., 2016. **Image-to-image translation with conditional adversarial networks**. *arXiv preprint arXiv:1611.07004*.

Reinforcement learning

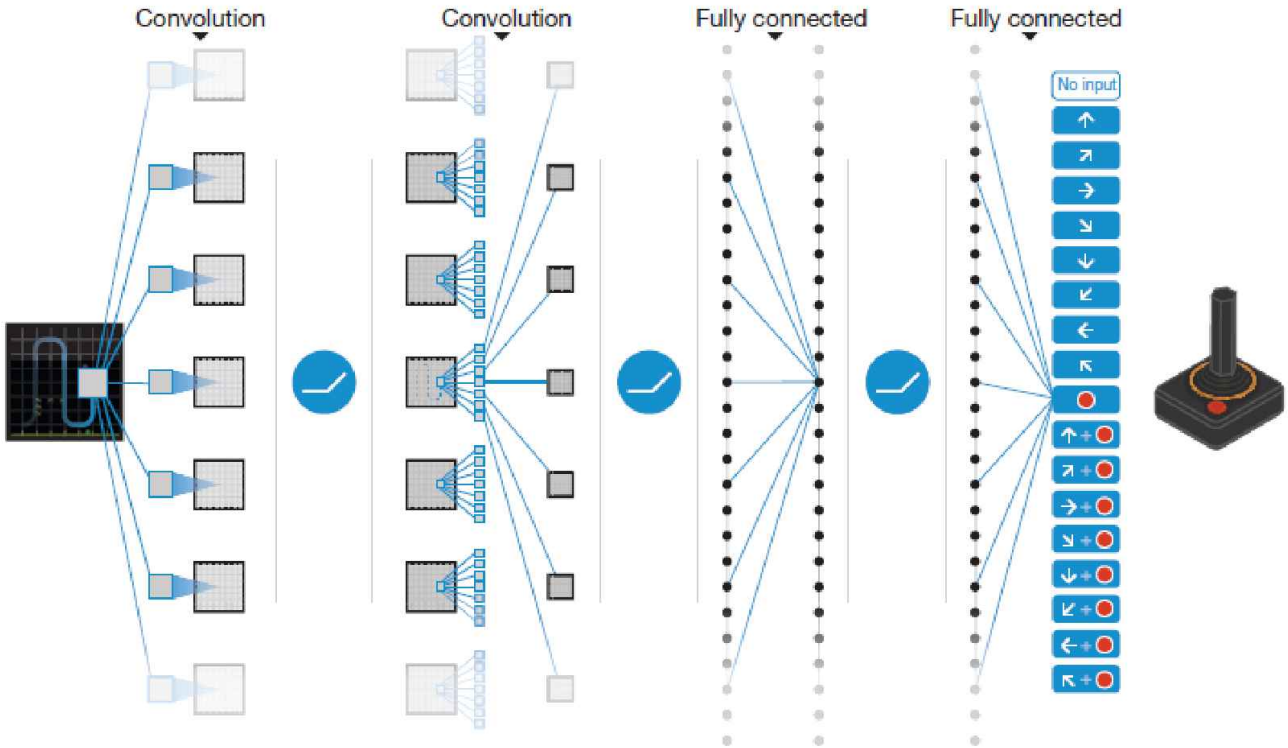


Figure 6 : Mnih, V., et al., 2015. **Human-level control through deep reinforcement learning**. *Nature*, 518(7540), pp.529-533.

Reasons for the sudden success

Apart from the software tools, several other reasons contributed to this explosion:

- Large data availability (allowing to train larger networks with many more parameters).
- Increase in computing power (especially GPUs).
- Development of techniques to train deeper networks (e.g., ReLU, dropout, attention mechanisms, etc.). We went from 1/2 hidden layers to hundred/thousands of them.

As a consequence, NNs today have a **huge number of free parameters** (i.e., very large Q), exploiting all these advancements in computational power and optimization.

Table of contents

Parameters vs. Ops.

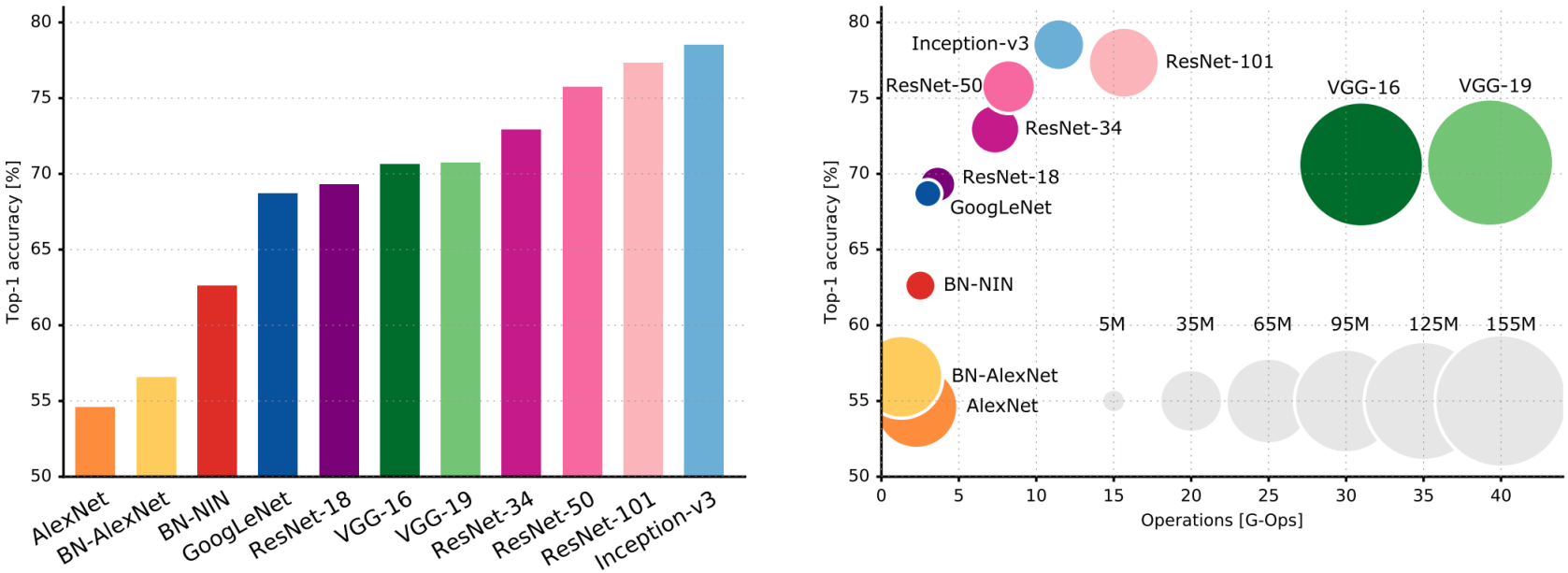
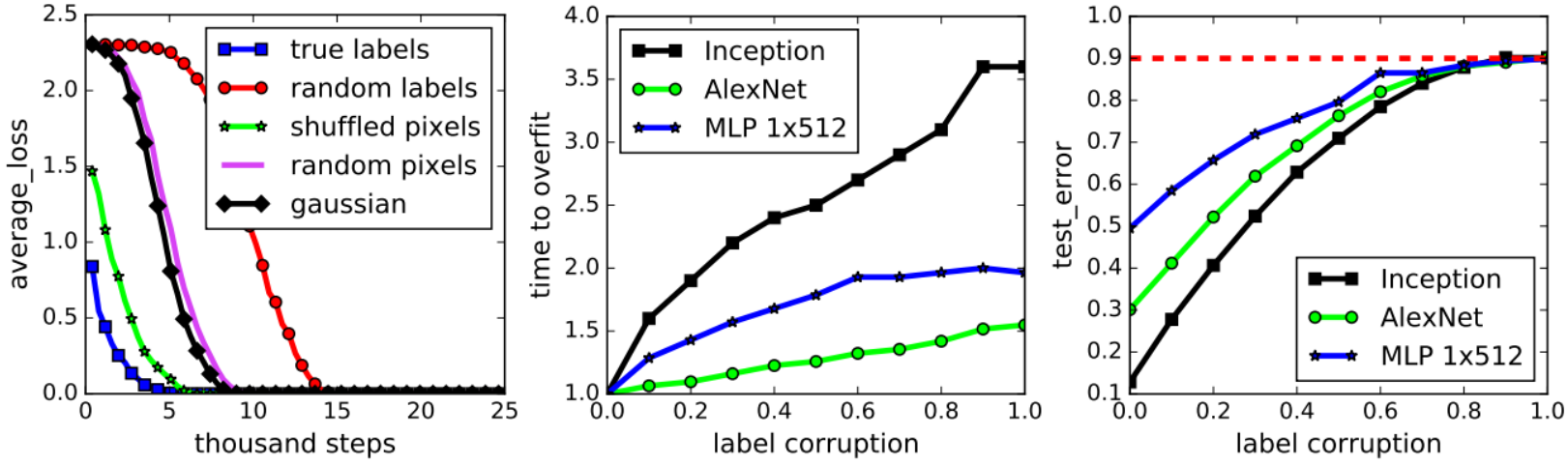


Figure 7 : Even for extremely efficient architectures, we are still talking about *tens of millions* of parameters! [Image source]

Fitting random noise

Over-parameterization leads to potentially *paradoxical results*!



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Zhang, C., Bengio, S., Hardt, M., Recht, B. and Vinyals, O., 2016. **Understanding deep learning requires rethinking generalization**. *arXiv preprint arXiv:1611.03530*.

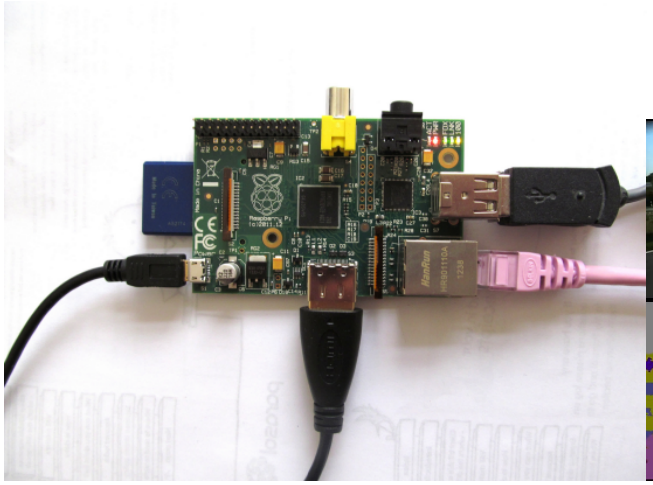
Predicting the parameters

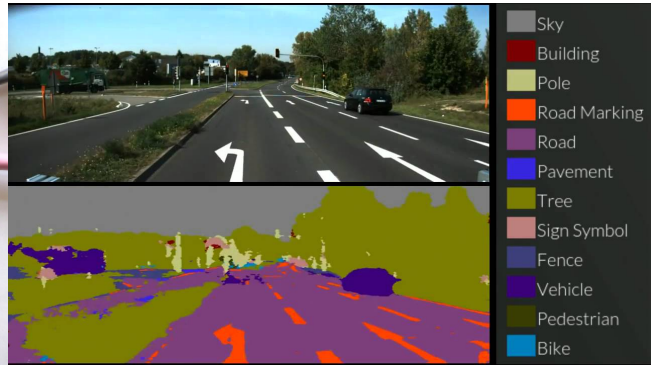
"*We demonstrate that there is significant redundancy in the parameterization of several deep learning models. Given only a few weight values for each feature it is possible to accurately predict the remaining values. [...]* **In the best case we are able to predict more than 95% of the weights of a network without any drop in accuracy.**"

Denil, M., Shakibi, B., Dinh, L. and de Freitas, N., 2013. **Predicting parameters in deep learning**. In *Advances in Neural Information Processing Systems* (pp. 2148-2156).

DL is needed in embedded environments!

DL is increasingly needed in embedded environments where space and memory are still concerns.





DL is needed in distributed environments!

In addition, there is a growing interest in distributed environments where agents need to exchange lots of information, including the weights of the network.

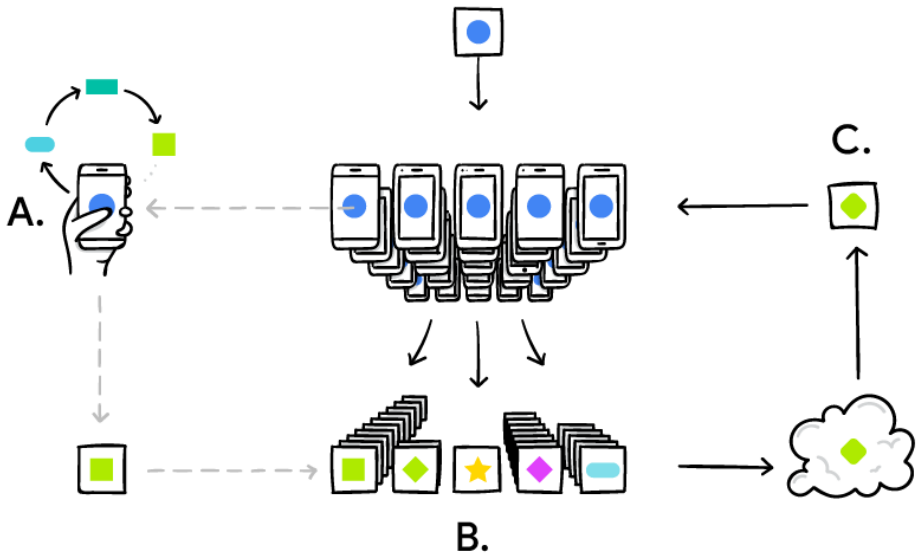


Figure 8 : McMahan, B. and Ramage, D., 2017. **Federated learning: Collaborative machine learning without centralized training data**. *Technical report, Google*.

Table of contents

Regularization

A common way to regularize neural networks is to impose a ℓ_2 norm constraint on the weights:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{y}_i, \hat{\mathbf{y}}_i) + \lambda \sum_{i=1}^{Q} w_i^2 \,, \tag{5}$$

where $\lambda > 0$ can be chosen by the user to balance the amount of regularization.

Sometimes this is called **weight decay** because of the way its gradient acts on the weights:

$$\nabla \lambda \left\|\mathbf{w}\right\|_2^2 = 2\lambda \mathbf{w} \,. \tag{6}$$

Sparsity-inducing penalties

The ℓ_2 norm forces weights to be small. We can force some of them to be *exactly* zero by replacing it with an ℓ_1 norm:

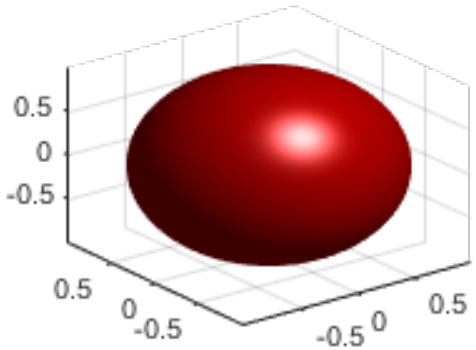
$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{y}_i, \hat{\mathbf{y}}_i) + \lambda \sum_{i=1}^{Q} |w_i| \,. \tag{7}$$

For linear models, this is called the **LASSO algorithm**.

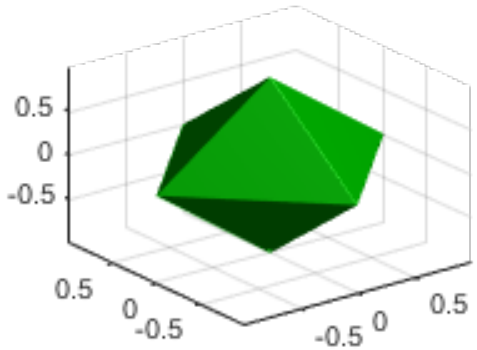
The ℓ_1 norm is not differentiable in 0, but everywhere else we have:

$$\frac{\partial |w_i|}{\partial w_i} = \text{sign}(w_i) \,. \tag{8}$$

Understanding sparsity



(a) ℓ_2 norm

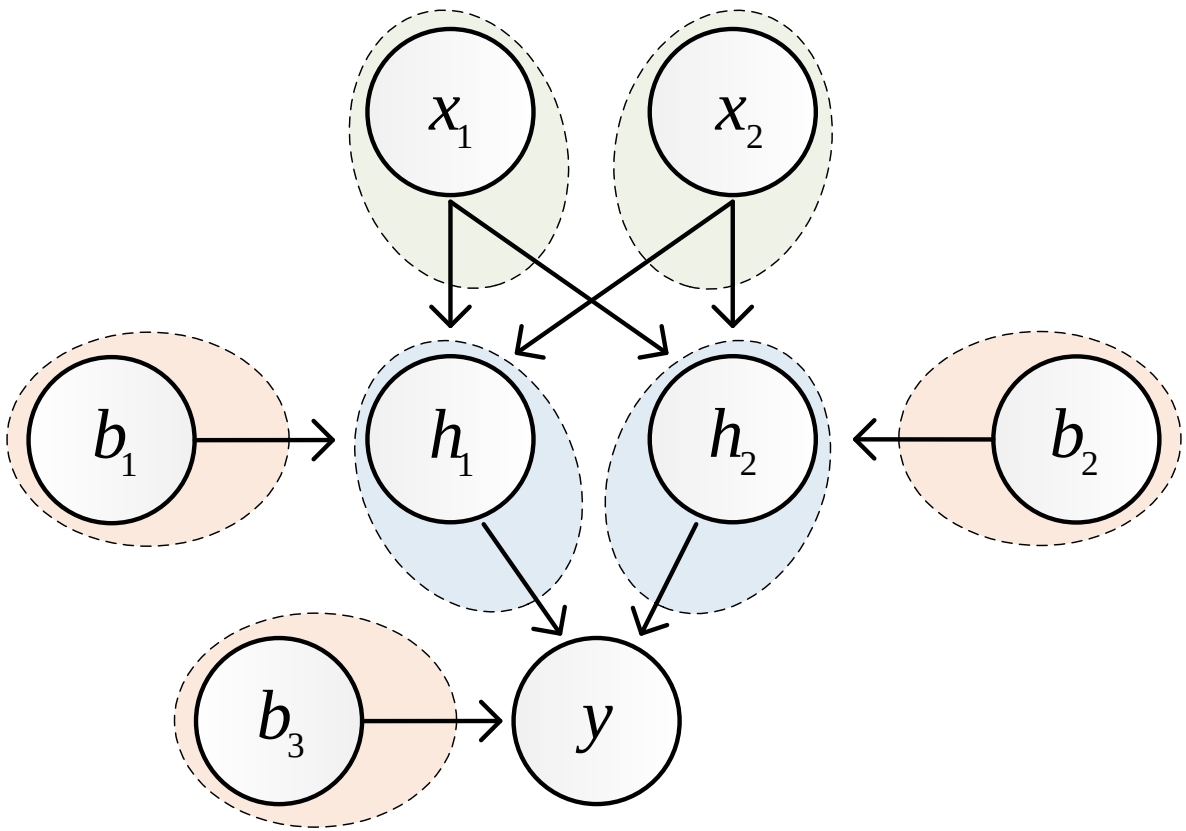


(b) ℓ_1 norm

Figure 9 : We can understand the action of the ℓ_2 and ℓ_1 norms by looking at their shapes in the Euclidean space. Norms can be thought of as constraining the weight vectors to lie inside (or on the boundary) of such shapes.

Sparsity over neurons

Can we modify the ℓ_1 norm to remove *groups* of weights? Using this, we could remove entire neurons, biases, or inputs by collecting the incoming or outgoing weights:



Defining groups

We consider three groups of variables:

1. **Input groups** $\mathcal{G}_{\text{in}}$: a single element $\mathbf{g_i} \in \mathcal{G}_{\text{in}}, i = 1, \ldots, d$ is the vector of all outgoing connections from the ith input neuron to the network.
2. **Hidden groups** $\mathcal{G}_{\text{h}}$: an element $\mathbf{g} \in \mathcal{G}_{\text{h}}$ corresponds to the vector of all outgoing connections from one of the neurons in the hidden layers of the network.
3. **Bias groups** $\mathcal{G}_{\text{b}}$: these are one-dimensional groups (scalars) corresponding to the biases on the network.

We define for simplicity the total set of groups as

$$\mathcal{G} = \mathcal{G}_{\text{in}} \cup \mathcal{G}_{\text{h}} \cup \mathcal{G}_{\text{b}} \,.$$

Group-sparse regularization

Group sparse regularization can be written as:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{y}_i, \hat{\mathbf{y}}_i) + \lambda \sum_{\mathbf{g}\in\mathcal{G}} \sqrt{|\mathbf{g}|}\, \|\mathbf{g}\|_2 \,, \tag{9}$$

where $|\mathbf{g}|$ denotes the dimensionality of the vector $\mathbf{g}$. Note that for one-dimensional groups, this is equivalent to the ℓ_1 norm.

---

[1] Scardapane, S., Comminiello, D., Hussain, A. and Uncini, A., 2017. **Group sparse regularization for deep neural networks**. *Neurocomputing*, 241, pp.81-89.

[2] Yuan, M. and Lin, Y., 2006. **Model selection and estimation in regression with grouped variables**. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1), pp.49-67.

Mixed norms

We can also mix several regularization techniques:

- $\ell_2 + \ell_1$: elastic net penalty.
- $\ell_1 +$ group ℓ_1: sparse group LASSO penalty.

Combining two or more of them, we can apply different effects to different groups of neurons, at the cost of introducing more hyper-parameters.

Visualizing group LASSO

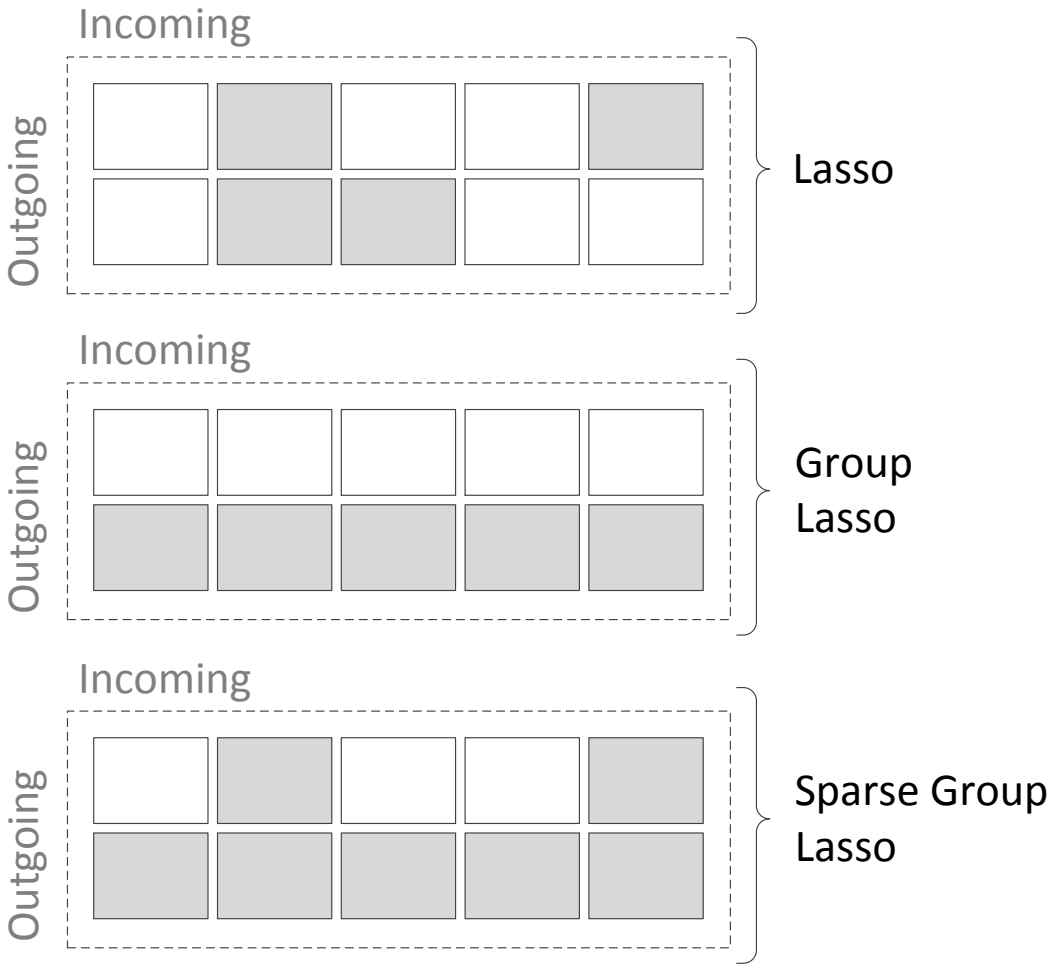
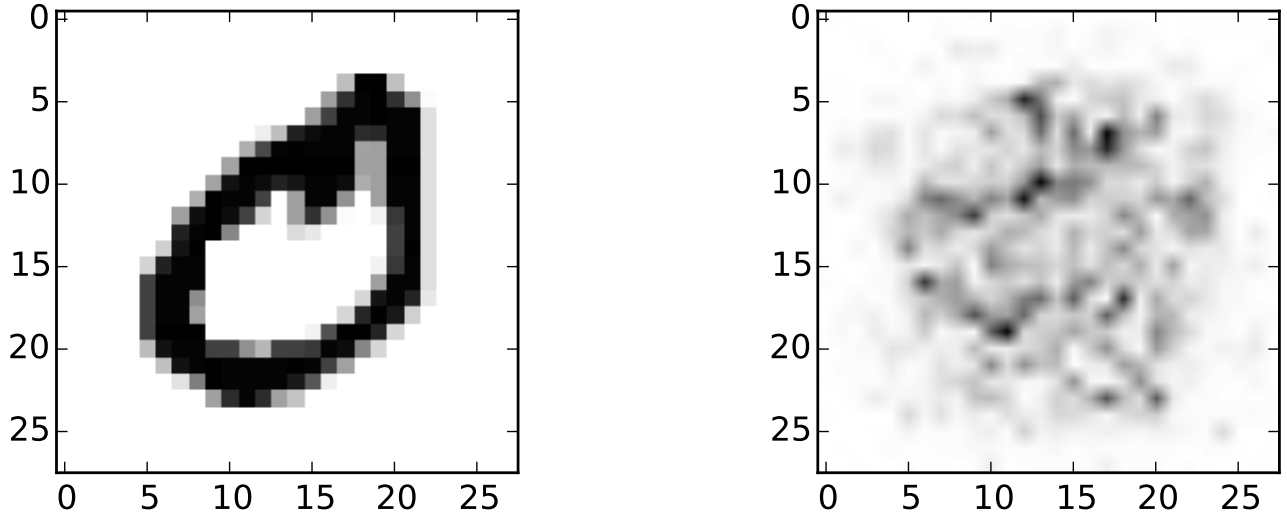


Figure 10 : A visual comparison of the effect of the different regularizers on the weight matrices connecting the layers.

Some results on MNIST



(a) Example of digit (b) Selected features

Figure 11 : Visualization of the selected features for the MNIST dataset. (a) Example of input pattern to the network (number 0). (b) Overall strength of outgoing weights from the respective input pixel (white are lowest, black are highest).

Quantitative results on other datasets

Dataset	Measure	L2-NN	SG-L1-NN
SSD	Acc. [%]	0.98	0.97
	Sparsity [%]	[0.17, 0.36, 0.36, 0.16]	[0.64, 0.81, 0.76, 0.54]
	Neurons	[48.0, 35.5, 24.8, 26.3]	[47.4, 19.0, 14.8, 15.9]
MNIST	Acc. [%]	0.98	0.97
	Sparsity [%]	[0.60, 0.60, 0.34, 0.08]	[0.96, $\approx$ 1.0, 0.98, 0.48]
	Neurons	[676.4, 311, 249.9, 93.7]	[581.8, 44.7, 41.0, 60.6]
COVER	Acc. [%]	0.84	0.83
	Sparsity [%]	[0.04, 0.10, 0.22, 0.14]	[0.45, 0.82, 0.84, 0.49]
	Neurons	[54.0, 49.0, 47.3, 18.7]	[52.7, 30.0, 16.0, 11.3]

Table of contents

Real number representation on hardware

Neural networks generally work with 32/64 bit floating point precision. *Is this precision truly required for most applications?*

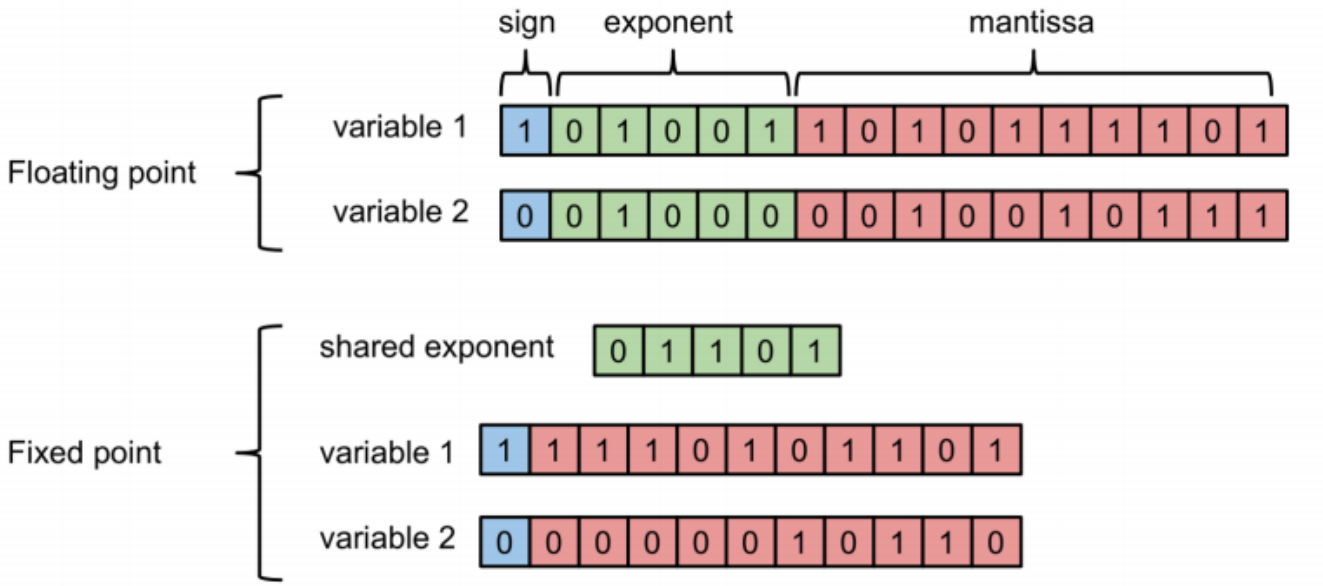


Figure 1: Comparison of the floating point and fixed point formats.

Courbariaux, M., Bengio, Y. and David, J.P., 2015. **Training deep neural networks with low precision multiplications**. *ICLR 2015*.

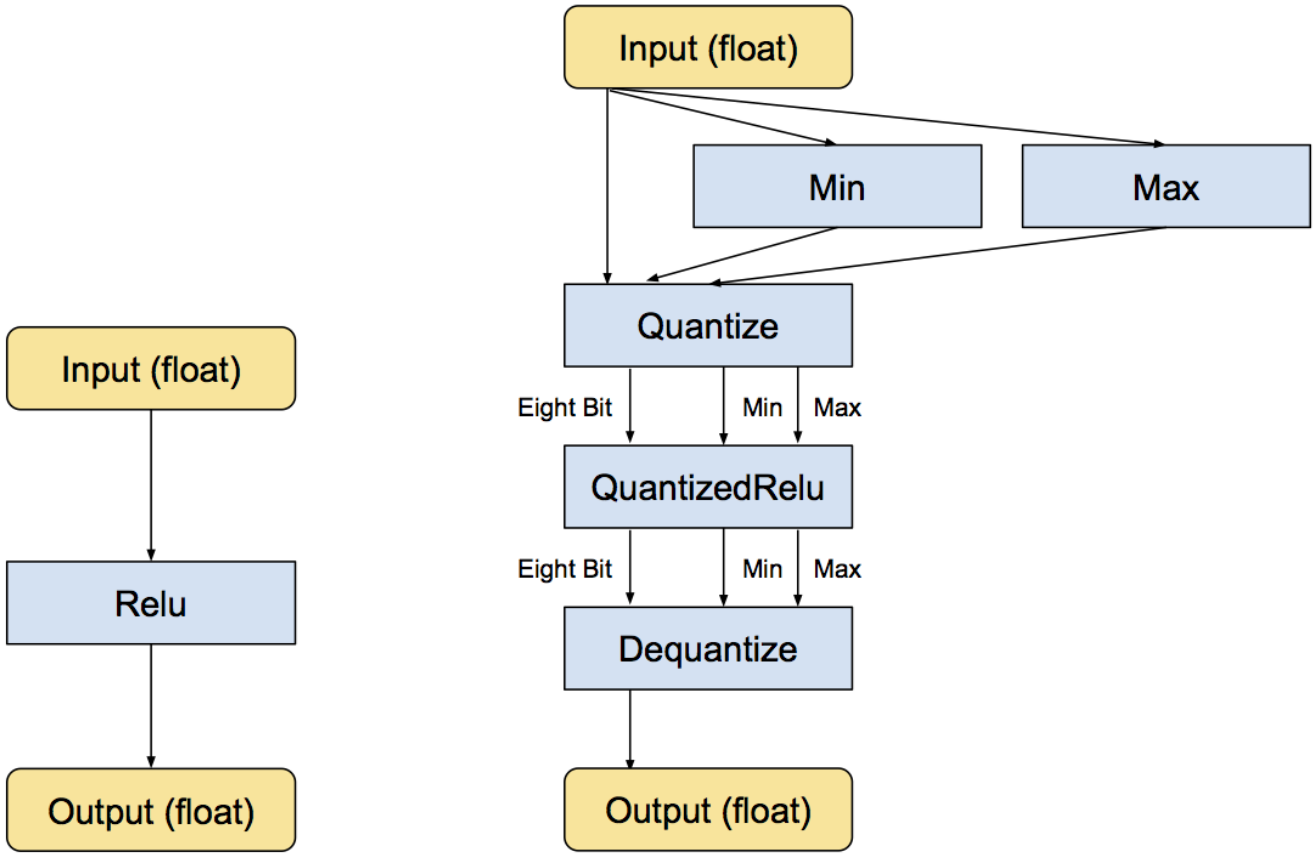
Quantization after training

Simplest possibility: train the neural network using high-precision arithmetic, then **quantize all weights** using, e.g., min-max quantization:

1. For each layer, compute the minimum and maximum values for the weight matrix.
2. Each quantized number in that layer will represent a float number, distributed linearly between minimum and maximum.

Most frameworks allow for an operation of this type, e.g., How to Quantize Neural Networks with TensorFlow.

Quantization in TensorFlow



(a) ReLU (b) Quantized ReLU

Figure 12 : How to Quantize Neural Networks with TensorFlow

Does it work?

Model	32-float	12-bit	10-bit	8-bit	6-bit
MNIST	98.42	98.43	98.44	98.44	98.32
SVHN	96.03	96.03	96.04	96.02	95.46
CIFAR10	93.78	93.79	93.80	93.58	90.86
CIFAR100	74.27	74.21	74.19	73.70	66.32
STL10	77.59	77.65	77.70	77.59	73.40
AlexNet	55.70/78.42	55.66/78.41	55.54/78.39	54.17/77.29	18.19/36.25
VGG16	70.44/89.43	70.45/89.43	70.44/89.33	69.99/89.17	53.33/76.32
VGG19	71.36/89.94	71.35/89.93	71.34/89.88	70.88/89.62	56.00/78.62
ResNet18	68.63/88.31	68.62/88.33	68.49/88.25	66.80/87.20	19.14/36.49
ResNet34	72.50/90.86	72.46/90.82	72.45/90.85	71.47/90.00	32.25/55.71
ResNet50	74.98/92.17	74.94/92.12	74.91/92.09	72.54/90.44	2.43/5.36
InceptionV3	76.41/92.78	76.43/92.71	76.44/92.73	73.67/91.34	1.50/4.82

Note: ImageNet 32-float models are directly from torchvision

Figure 13 : Some results for quantization on pre-trained models in PyTorch (Source).

Reducing precision during training

We can also reduce precision during forward/backward passes, but **keep high-precision arithmetic** for computing updates.

Algorithm 1 Forward propagation with low precision multipliers.

for all layers **do**
  Reduce the precision of the parameters and the inputs
  Apply convolution or dot product (with high precision accumulations)
  Reduce the precision of the weighted sums
  Apply activation functions
end for
Reduce the precision of the outputs

More precision is needed for updating because of the small range of gradients during training.

Courbariaux, M., Bengio, Y. and David, J.P., 2015. **Training deep neural networks with low precision multiplications**. *ICLR 2015*.

Table of contents

Deterministic and stochastic rounding

It it possible to extend the previous ideas to *binary* weights constrained in $\{-1, +1\}$, allowing for very efficient training on specialized hardware.

The rounded value w_b of a single weight w is given by:

$$w_b = \text{sign}(w)\,, \tag{10}$$

Instead of **deterministic rounding**, we can improve using **stochastic rounding** as:

$$w_b = \begin{cases} -1 & \text{with probability } p = \sigma(w)\,, \\ +1 & \text{with probability } 1 - p\,, \end{cases} \tag{11}$$

where:

$$\sigma(w) = \text{clip}\left(\frac{w+1}{2}, 0, 1\right)\,. \tag{12}$$

Results on benchmark datasets

Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	**8.27%**	2.15%
50% Dropout	$1.01 \pm 0.04\%$		
Maxout Networks [29]	0.94%	11.68%	2.47%
Deep L2-SVM [30]	**0.87%**		
Network in Network [31]		10.41%	2.35%
DropConnect [21]			1.94%
Deeply-Supervised Nets [32]		9.78%	**1.92%**

Table 2: Test error rates of DNNs trained on the MNIST (no convolution and no unsupervised pretraining), CIFAR-10 (no data augmentation) and SVHN, depending on the method. We see that in spite of using only a single bit per weight during propagation, performance is not worse than ordinary (no regularizer) DNNs, it is actually better, especially with the stochastic version, suggesting that BinaryConnect acts as a regularizer.

Courbariaux, M., Bengio, Y. and David, J.P., 2015. **Binaryconnect: Training deep neural networks with binary weights during propagations.** In *NIPS* (pp. 3123-3131).

Binarization as noise

Interestingly, binarization can be seen as introducing noise in the optimization process, thus leading to better regularization.

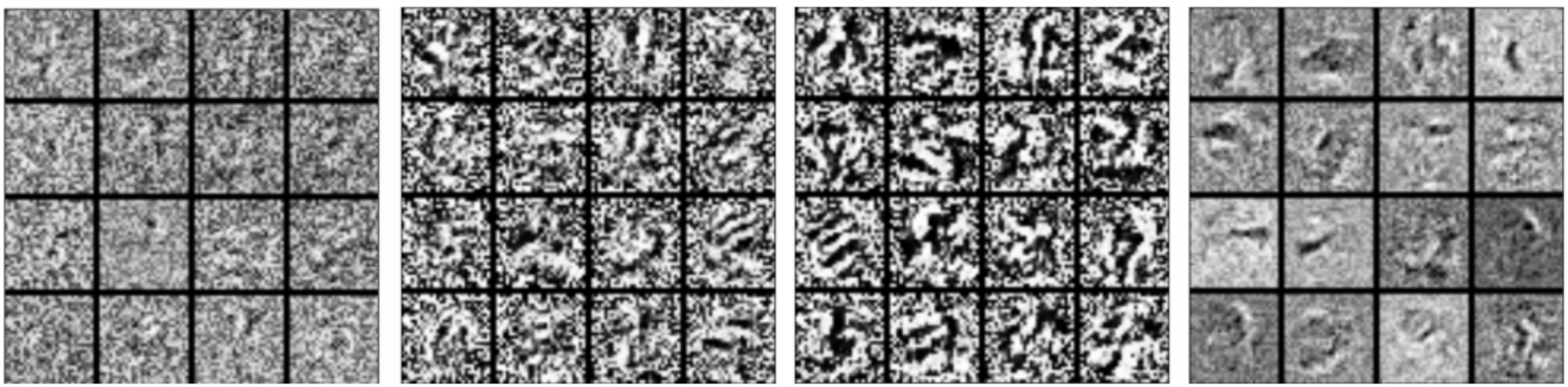


Figure 1: Features of the first layer of an MLP trained on MNIST depending on the regularizer. From left to right: no regularizer, deterministic BinaryConnect, stochastic BinaryConnect and Dropout.

Further readings on binary neural networks

[1] Li, F., Zhang, B. and Liu, B., 2016. **Ternary weight networks**. *arXiv preprint arXiv:1605.04711*.
A variant with three possible values per weights including 0.

[2] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R. and Bengio, Y., 2016. **Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1**. *arXiv preprint arXiv:1602.02830*.
First paper to train both weights and activations with binary values.

[3] Li, H., De, S., Xu, Z., Studer, C., Samet, H. and Goldstein, T., 2017. **Training Quantized Nets: A Deeper Understanding**. *arXiv preprint arXiv:1706.02379*.
Theoretical analysis of most quantization techniques.

Table of contents

A full pipeline with quantization

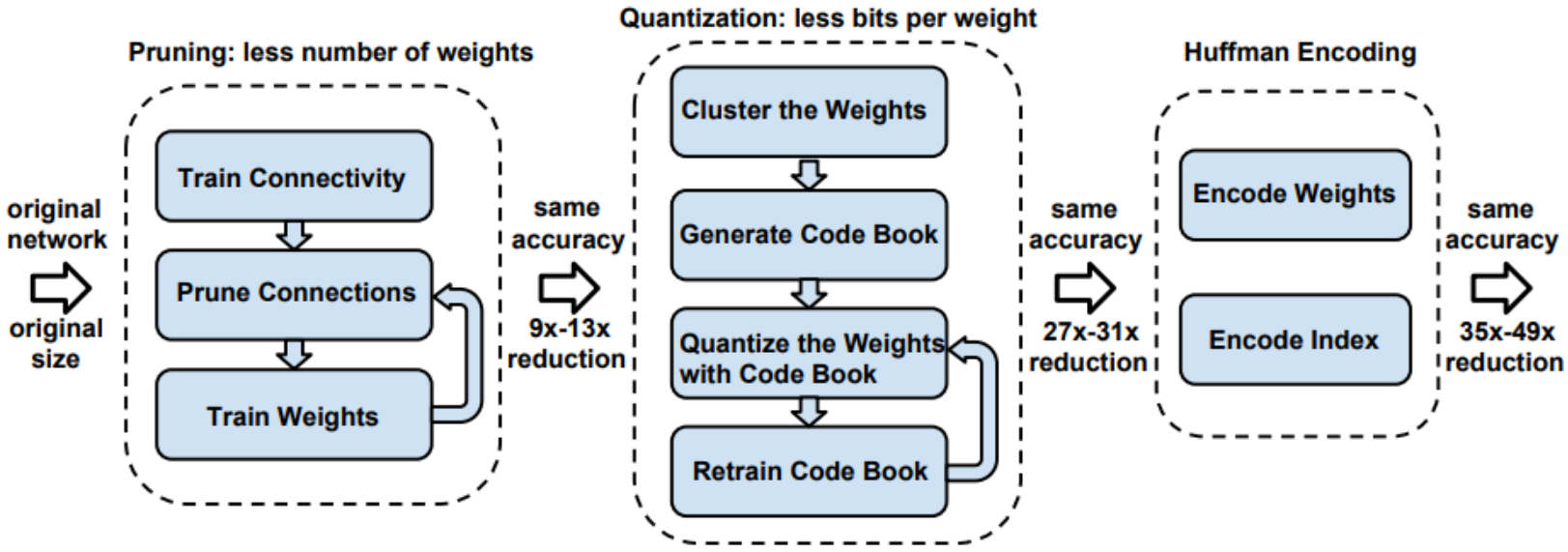


Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by $10\times$, while quantization further improves the compression rate: between $27\times$ and $31\times$. Huffman coding gives more compression: between $35\times$ and $49\times$. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

Han, S., Mao, H. and Dally, W.J., 2015. **Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding**. *arXiv preprint arXiv:1510.00149*.

Pruning step

The pruning part can be seen as a variation of sparse penalties introduced earlier:

1. Train the network normally.
2. Remove connections below a given threshold.
3. Repeat steps 1-2 until convergence.

The resulting sparse matrix can be stored efficiently using a linear number of float values.

Storing the sparse matrix

In addition, the indexes in the sparse matrix are saved using the relative difference, encoded using a few-bit precision arithmetic:

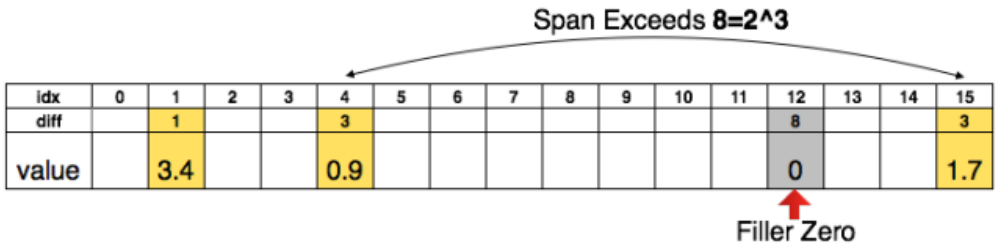


Figure 2: Representing the matrix sparsity with relative index. Padding filler zero to prevent overflow.

Figure 14 : Note how padding is sometimes necessary if the span exceed the bound given by the few bits.

Codebook compression

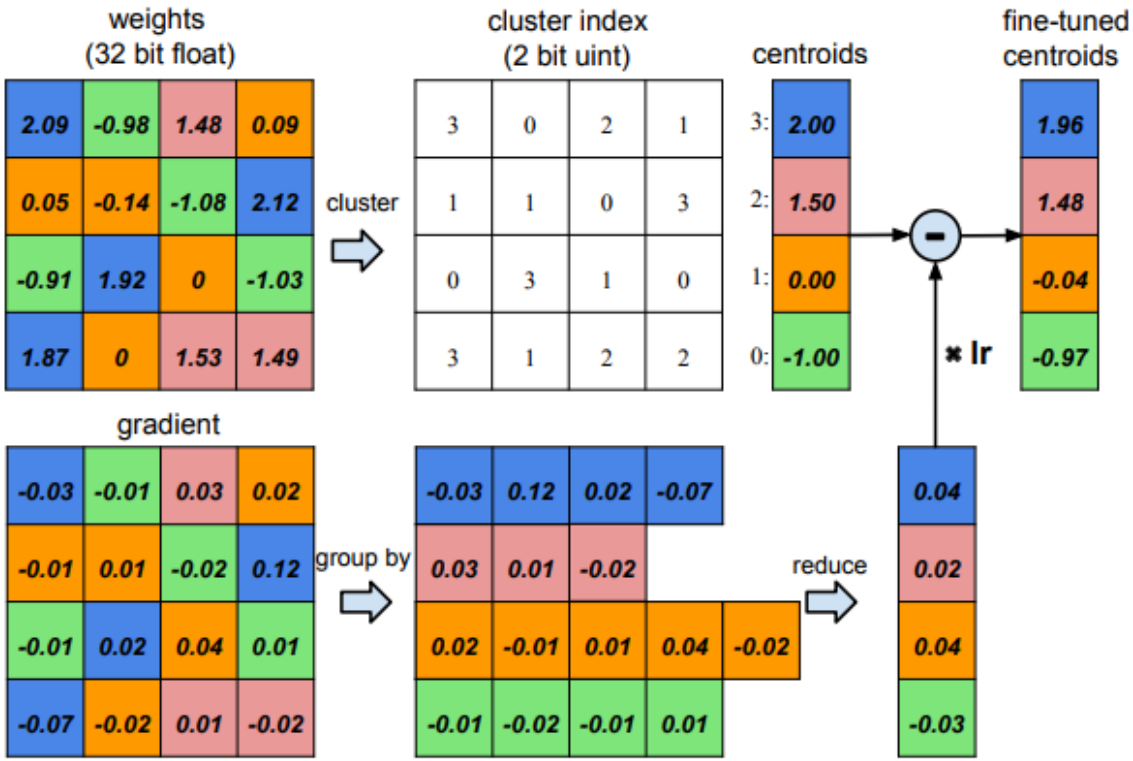


Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

Huffman coding step

Because the weight distribution is biased, Huffman coding is applied as last step. Huffman coding is a lossless compression technique giving shorter codes to symbols appearing more frequently.

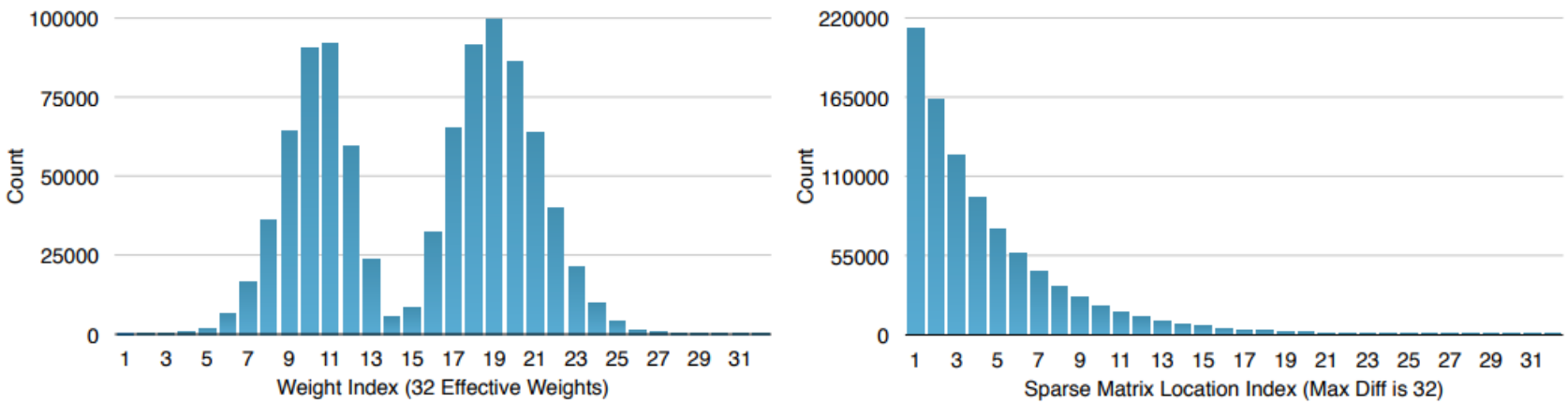


Figure 5: Distribution for weight (Left) and index (Right). The distribution is biased.

Results on standard CNNs

Table 2: Compression statistics for LeNet-300-100. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
ip1	235K	8%	6	4.4	5	3.7	3.1%	2.32%
ip2	30K	9%	6	4.4	5	4.3	3.8%	3.04%
ip3	1K	26%	6	4.3	5	3.2	15.7%	12.70%
Total	266K	8%(12×)	6	5.1	5	3.7	3.1% (**32**×)	2.49% (**40**×)

Table 3: Compression statistics for LeNet-5. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	0.5K	66%	8	7.2	5	1.5	78.5%	67.45%
conv2	25K	12%	8	7.2	5	3.9	6.0%	5.28%
ip1	400K	8%	5	4.5	5	4.5	2.7%	2.45%
ip2	5K	19%	5	5.2	5	3.7	6.9%	6.13%
Total	431K	8%(12×)	5.3	4.1	5	4.4	3.05% (**33**×)	2.55% (**39**×)

All steps are necessary in the pipeline!

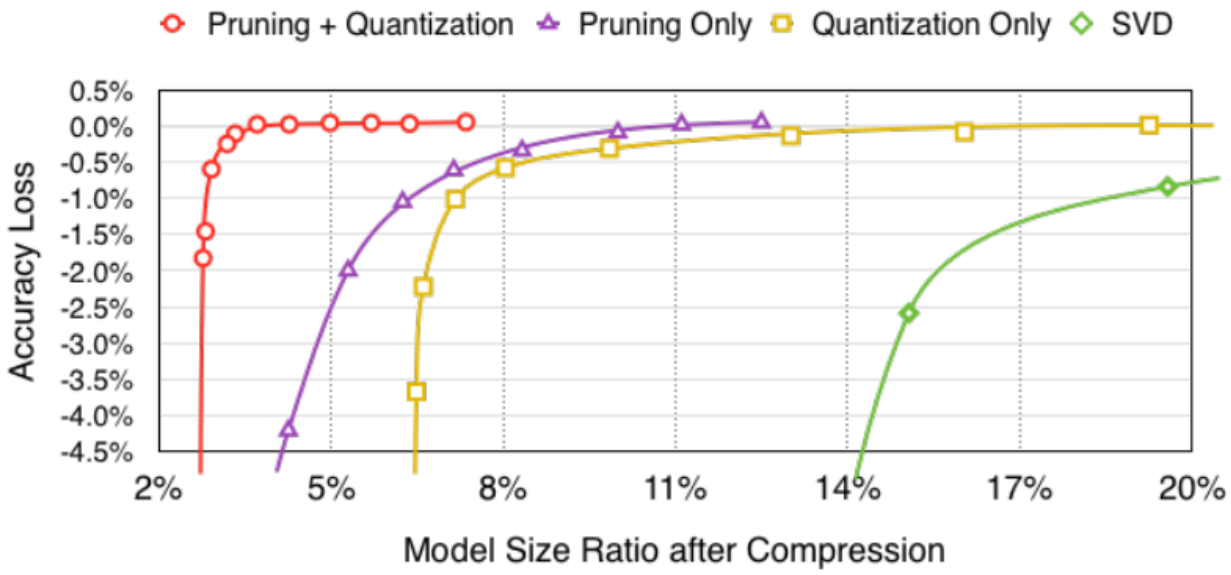


Figure 6: Accuracy v.s. compression rate under different compression methods. Pruning and quantization works best when combined.

Table of contents

Distilling a neural network

Distillation is a technique to compress (trained) neural networks, by training a smaller network to replicate both the original labels and the predicted labels from the first network.

Also known colloquially as "dark knowledge", it allows to build a small model having stronger accuracy than by simply training it from scratch.

Recently, it has been applied to a wealth of applications, including building more interpretable models and defending against adversarial attacks.

Hinton, G., Vinyals, O. and Dean, J., 2015. **Distilling the knowledge in a neural network**. *arXiv preprint arXiv:1503.02531*.

Some results from MNIST

"We trained a single large neural net with two hidden layers of 1200 rectified linear hidden units on all 60.000 training cases. [...] This net **achieved 67 test errors** *whereas a smaller net with two hidden layers of 800 rectified linear hidden units and no regularization* **achieved 146 errors**. *[Using distillation], it achieved* **74 test errors**."

Hinton, G., Vinyals, O. and Dean, J., 2015. **Distilling the knowledge in a neural network**. *arXiv preprint arXiv:1503.02531*.

Table of contents

Do we need all these parameters? (Again)

It might be that we need all these parameters because of the inflexible architecture of the network, including the fixed, simple nonlinearities.

- Can we use *flexible* activation functions?
- Can the use of flexible activation functions allow for a more efficient use of parameters?

Parametric activation functions

Making a single activation function parametric is relatively simple, e.g., we can add a learnable scale and bandwidth to a tanh:

$$g(s) = \frac{a\left(1 - \exp\left\{-bs\right\}\right)}{1 + \exp\left\{-bs\right\}} . \tag{13}$$

Or learn the slope for the negative part of the ReLU (PReLU):

$$g(s) = \begin{cases} s & \text{if } s \geq 0 \\ \alpha s & \text{otherwise} \end{cases} . \tag{14}$$

These *parametric* AFs have a small amount of trainable parameters, but their flexibility is severely limited.

Adaptive piecewise linear units

An APL nonlinearity is the sum of S linear segments:

$$g(s) = \max\{0, s\} + \sum_{i=1}^{S} a_i \max\{0, -s + b_i\} \ . \qquad (15)$$

This is non-parametric because S is a user-defined hyper-parameter controlling the flexibility of the unit.

The APL introduces $S+1$ points of non-differentiability for each neuron, which may damage the optimization algorithm. Also, in practice having $S > 3$ seems to have less effect on the resulting shapes.

---

[1] Agostinelli, F., Hoffman, M., Sadowski, P. and Baldi, P., 2014. **Learning activation functions to improve deep neural networks**. *arXiv preprint arXiv:1412.6830*.

Maxout neurons

A Maxout replaces an entire neuron by taking the maximum over K separate linear projections:

$$g(\mathbf{h}) = \max_{i=1,\ldots,K} \left\{ \mathbf{w}_i^T \mathbf{h} + b_i \right\} . \tag{16}$$

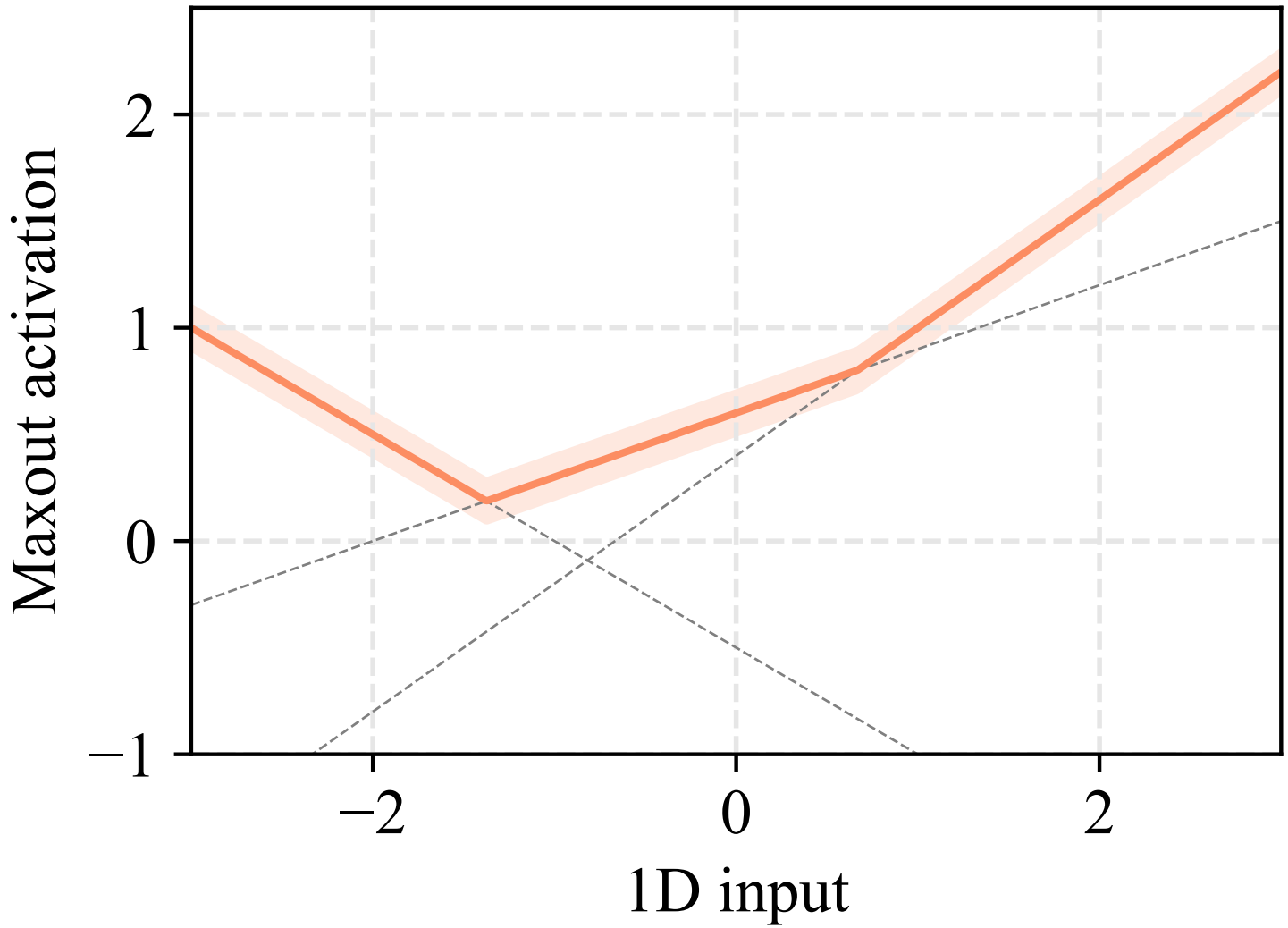
With two maxout neurons, a NN with one hidden layer remains an universal approximator provided K is sufficiently large.

However, it is impossible to plot the functions for $K > 3$, and the number of parameters can in fact *increase* drastically with respect to K.

---

[1] Goodfellow, I.J., Warde-Farley, D., Mirza, M., Courville, A. and Bengio, Y., 2013. **Maxout networks**. *Proc. 30th Int. Conf. on Machine Learning*.

Visualization of a Maxout neuron



Basic structure of the KAF

We model each activation function in terms of a kernel expansion over D terms as:

$$g(s) = \sum_{i=1}^{D} \alpha_i \kappa(s, d_i) \ , \tag{17}$$

where:

1. $\{\alpha_i\}_{i=1}^{D}$ are the **mixing coefficients**;
2. $\{d_i\}_{i=1}^{D}$ are the **dictionary elements**;
3. $\kappa(\cdot, \cdot) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is a **1D kernel function**.

We only adapt the mixing coefficients, and for the dictionary we sample D values over the x-axis, uniformly around zero.

[1] Scardapane, S., Van Vaerenbergh, S. Totaro, S., and Uncini, A., 2017. **Kafnets: kernel-based non-parametric activation functions for neural networks**. *arXiv preprint arXiv:1707.04035*.

Kernel selection

For the experiments, we use the 1D Gaussian kernel defined as:

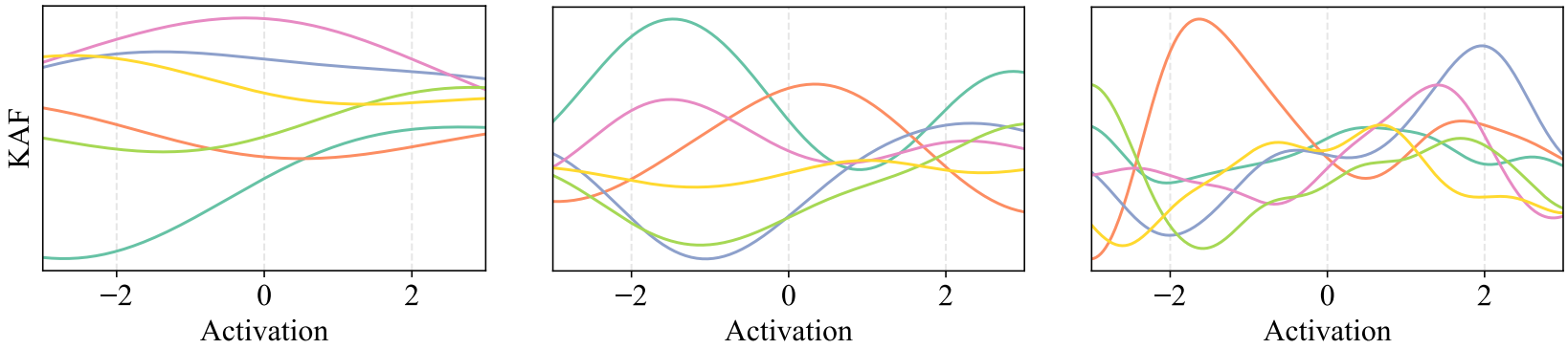
$$\kappa(s, d_i) = \exp\left\{-\gamma\left(s - d_i\right)^2\right\}, \tag{18}$$

where $\gamma \in \mathbb{R}$ is called the kernel bandwidth. Based on some preliminary experiments, we use the following rule-of-thumb for selecting the bandwidth:

$$\gamma = \frac{1}{6\Delta^2}, \tag{19}$$

where Δ is the distance between the grid points.

Choosing the bandwidth



(a) $\gamma = 2.0$ (b) $\gamma = 0.5$ (c) $\gamma = 0.1$

Figure 15 : Examples of KAFs. In all cases we sample uniformly 20 points on the x-axis, while the mixing coefficients are sampled from a normal distribution. The three plots show three different choices for γ.

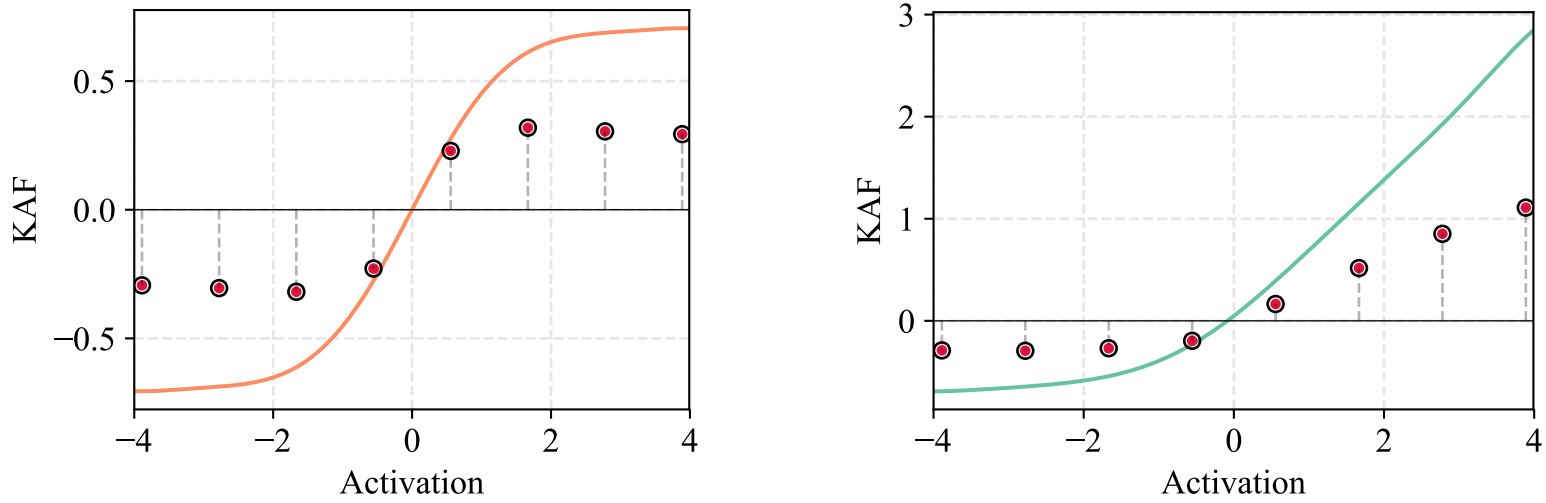
Inizialization of the mixing coefficients

Other than initializing the mixing coefficients randomly, we can also approximate any initial function using kernel ridge regression (KRR):

$$\boldsymbol{\alpha} = (\mathbf{K} + \varepsilon \mathbf{I})^{-1} \mathbf{t}\,, \tag{20}$$

where $\mathbf{K} \in \mathbb{R}^{D \times D}$ is the kernel matrix computed between the desired points $\mathbf{t}$ and the elements of the dictionary $\mathbf{d}$.

Examples of initialization



(a) tanh (b) ELU

Figure 16 : Two examples of initializing a KAF using KRR, with $\varepsilon = 10^{-6}$. (a) A hyperbolic tangent. (b) The ELU function. The red dots indicate the corresponding initialized values for the mixing coefficients.

Advantages of the framework

1. Universal approximation properties.
2. Very simple to vectorize and to accelerate on GPUs.
3. Smooth over the entire domain.
4. Mixing coefficients can be regularized easily, including the use of sparse penalties.

Visualizing the functions

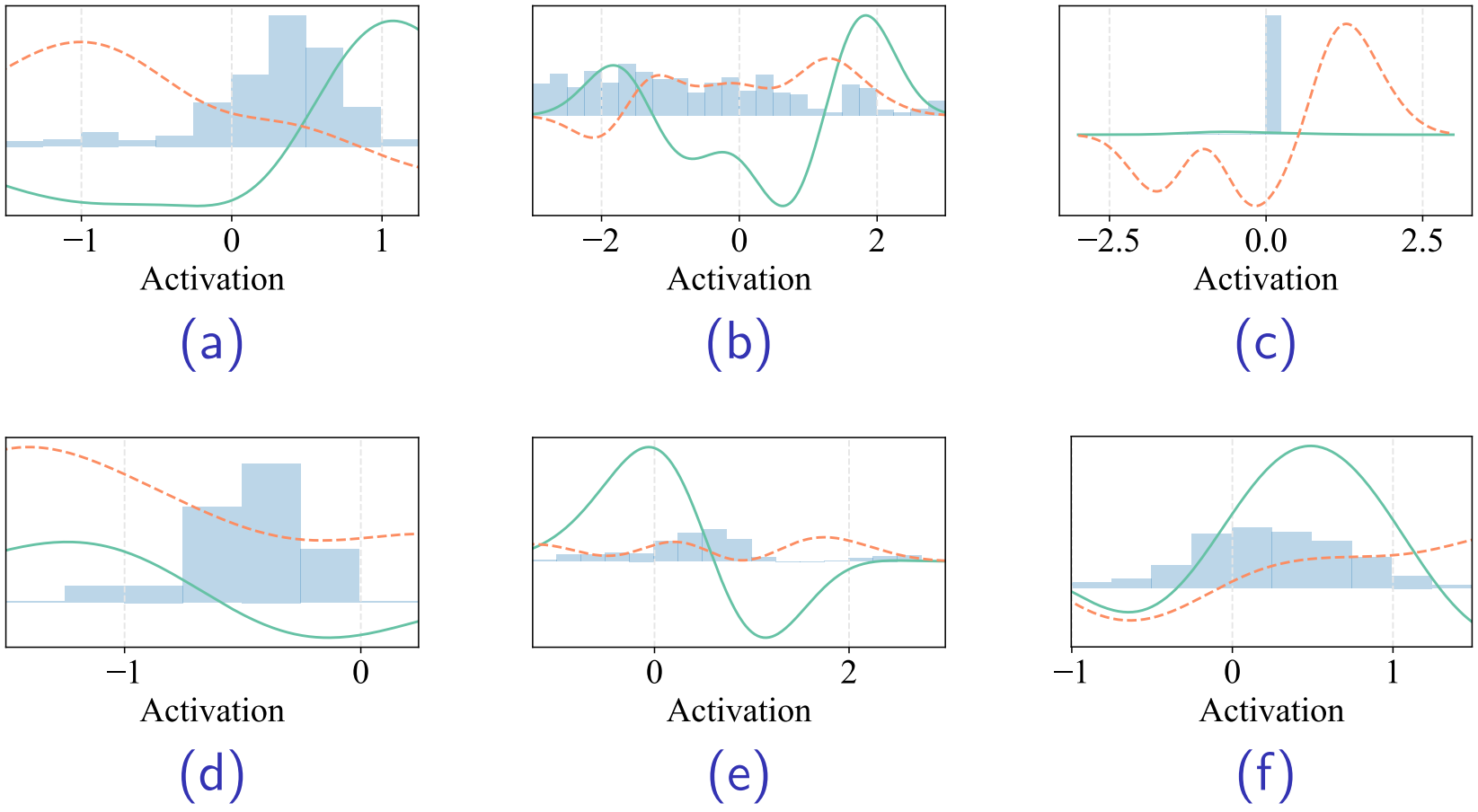


Figure 17 : Examples of 6 trained KAFs (with random initialization) on the Sensorless dataset. On the y-axis we plot the output value of the KAF. The KAF after initialization is shown with a dashed red, while the final KAF is shown with a solid green.

Results on the SUSY benchmark

Activation function	**Testing AUC**	**Trainable parameters**
ReLU (five hidden layers)	0.8739(0.001)	367201
ELU (five hidden layers)	0.8739(0.001)	
SELU (five hidden layers)	0.8745(0.002)	
PReLU (five hidden layers)	0.8748(0.001)	368701
Maxout (one layer)	0.8744(0.001)	17401
Maxout (two layers)	0.8744(0.002)	288301
APL (one layer)	0.8744(0.002)	7801
APL (two layers)	0.8757(0.002)	99901
KAF (one layer)	0.8756(0.001)	12001
KAF (two layers)	0.8758(0.001)	108301

Table 1 : Results on the SUSY benchmark.

Table of contents

Conclusions

- Compressing neural networks is a vital research field with plenty of applicative domains.
- A lot of “old” techniques can be re-purposed to this end.
- More in general, the amount of feasible compression means that a lot more research is needed on efficient allocation of parameters.