Compressing deep neural networks
Challenges and theoretical foundations

Presenter: Simone Scardapane

University of Exeter, UK
Table of contents

Introduction
   Building and optimizing a neural network
   Some applications of neural networks
   Do we need all these parameters?

Compression by regularization
   Sparse and group-sparse penalties

Compression by quantization
   Limited-precision arithmetic for neural networks
   Binary neural networks
   Multi-stage compression

Other compression techniques
   Distilling the knowledge
   Flexible activation functions

Conclusions
   Conclusions
Schema of a neural network

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

![Schematic depiction of a simple neural network](Image source)

**Figure 1**: Schematic depiction of a simple neural network [Image source].
A layer of a neural network alternates a linear projection with a (generally fixed) nonlinear operation:

\[ h_{l+1} = g(Wh_l + b), \]  

where:

- \( h_l/h_{l+1} \) are the input/output of the layer,
- \( W \) and \( b \) are adapted from data,
- \( g(\cdot) \) is a real-valued function, e.g., the rectified linear unit (ReLU):
  \[ g(s) = \max(0, s). \]
The neural network is composed by stacking multiple layers to get the final output $\hat{y}$ from the input $x$.

Given some training data $(x_i, y_i), i = 1, \ldots, n$, we train the weights by minimizing some cost function:

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

(3)

where $w \in \mathbb{R}^Q$ is a vector containing all parameters of the network, and $L(\cdot, \cdot)$ is a loss function (e.g., cross-entropy).
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 $\tilde{\nabla} J(\mathbf{w})$.
3. We use the gradient to move in a descent direction, e.g., by steepest-descent:

$$
\mathbf{w} = \mathbf{w} - \eta \tilde{\nabla} J(\mathbf{w}),
$$

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

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:

```python
F.backward()
```

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

```python
print(w.grad)
```
Table of contents

Introduction
Building and optimizing a neural network
Some applications of neural networks
Do we need all these parameters?

Compression by regularization
Sparse and group-sparse penalties

Compression by quantization
Limited-precision arithmetic for neural networks
Binary neural networks
Multi-stage compression

Other compression techniques
Distilling the knowledge
Flexible activation functions

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

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

Introduction
  Building and optimizing a neural network
  Some applications of neural networks
  Do we need all these parameters?

Compression by regularization
  Sparse and group-sparse penalties

Compression by quantization
  Limited-precision arithmetic for neural networks
  Binary neural networks
  Multi-stage compression

Other compression techniques
  Distilling the knowledge
  Flexible activation functions

Conclusions
  Conclusions
Parameters vs. Ops.

Figure 7: Even for extremely efficient architectures, we are still talking about *tens of millions* of parameters! [Image source]
Over-parameterization leads to potentially paradoxical results!

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.

“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.”

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

Introduction
   Building and optimizing a neural network
   Some applications of neural networks
   Do we need all these parameters?

Compression by regularization
   Sparse and group-sparse penalties

Compression by quantization
   Limited-precision arithmetic for neural networks
   Binary neural networks
   Multi-stage compression

Other compression techniques
   Distilling the knowledge
   Flexible activation functions

Conclusions
   Conclusions
A common way to regularize neural networks is to impose a $\ell_2$ norm constraint on the weights:

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

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:

$$\nabla \lambda \| w \|_2^2 = 2\lambda w.$$
Sparsity-inducing penalties

The $\ell_2$ norm forces weights to be small. We can force some of them to be \textit{exactly} zero by replacing it with an $\ell_1$ norm:

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

(7)

For linear models, this is called the \textbf{LASSO algorithm}.

The $\ell_1$ norm is not differentiable in 0, but everywhere else we have:

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

(8)
Understanding sparsity

Figure 9: We can understand the action of the $\ell_2$ and $\ell_1$ 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.
Can we modify the $\ell_1$ 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** $G_{in}$: a single element $g_i \in G_{in}, i = 1, \ldots, d$ is the vector of all outgoing connections from the $i$th input neuron to the network.

2. **Hidden groups** $G_h$: an element $g \in G_h$ corresponds to the vector of all outgoing connections from one of the neurons in the hidden layers of the network.

3. **Bias groups** $G_b$: these are one-dimensional groups (scalars) corresponding to the biases on the network.

We define for simplicity the total set of groups as

$$G = G_{in} \cup G_h \cup G_b.$$
Group-sparse regularization can be written as:

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

(9)

where $|g|$ denotes the dimensionality of the vector $g$. Note that for one-dimensional groups, this is equivalent to the $\ell_1$ norm.


Mixed norms

We can also mix several regularization techniques:

- $\ell_2 + \ell_1$: elastic net penalty.
- $\ell_1 + \text{group } \ell_1$: 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

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

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

Introduction
Building and optimizing a neural network
Some applications of neural networks
Do we need all these parameters?

Compression by regularization
Sparse and group-sparse penalties

Compression by quantization
Limited-precision arithmetic for neural networks
Binary neural networks
Multi-stage compression

Other compression techniques
Distilling the knowledge
Flexible activation functions

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

![Diagram of floating point and fixed point formats](image)

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

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

Figure 12: How to Quantize Neural Networks with TensorFlow
# Does it work?

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

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.

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

---

Table of contents

Introduction
   Building and optimizing a neural network
   Some applications of neural networks
   Do we need all these parameters?

Compression by regularization
   Sparse and group-sparse penalties

Compression by quantization
   Limited-precision arithmetic for neural networks
   Binary neural networks
   Multi-stage compression

Other compression techniques
   Distilling the knowledge
   Flexible activation functions

Conclusions
   Conclusions
Deterministic and stochastic rounding

It is 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:

\[
w_b = \text{sign}(w),
\]

(10)

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

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

(11)

where:

\[
\sigma(w) = \text{clip} \left( \frac{w + 1}{2}, 0, 1 \right).
\]

(12)
Results on benchmark datasets

<table>
<thead>
<tr>
<th>Method</th>
<th>MNIST</th>
<th>CIFAR-10</th>
<th>SVHN</th>
</tr>
</thead>
<tbody>
<tr>
<td>No regularizer</td>
<td>1.30 ± 0.04%</td>
<td>10.64%</td>
<td>2.44%</td>
</tr>
<tr>
<td>BinaryConnect (det.)</td>
<td>1.29 ± 0.08%</td>
<td>9.90%</td>
<td>2.30%</td>
</tr>
<tr>
<td>BinaryConnect (stoch.)</td>
<td>1.18 ± 0.04%</td>
<td><strong>8.27%</strong></td>
<td>2.15%</td>
</tr>
<tr>
<td>50% Dropout</td>
<td>1.01 ± 0.04%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maxout Networks [29]</td>
<td>0.94%</td>
<td>11.68%</td>
<td>2.47%</td>
</tr>
<tr>
<td>Deep L2-SVM [30]</td>
<td><strong>0.87%</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network in Network [31]</td>
<td></td>
<td>10.41%</td>
<td>2.35%</td>
</tr>
<tr>
<td>DropConnect [21]</td>
<td></td>
<td></td>
<td>1.94%</td>
</tr>
<tr>
<td>Deeply-Supervised Nets [32]</td>
<td>9.78%</td>
<td><strong>1.92%</strong></td>
<td></td>
</tr>
</tbody>
</table>

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.

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


*A variant with three possible values per weights including 0.*


*First paper to train both weights and activations with binary values.*


*Theoretical analysis of most quantization techniques.*
Table of contents

Introduction
  Building and optimizing a neural network
  Some applications of neural networks
  Do we need all these parameters?

Compression by regularization
  Sparse and group-sparse penalties

Compression by quantization
  Limited-precision arithmetic for neural networks
  Binary neural networks
  Multi-stage compression

Other compression techniques
  Distilling the knowledge
  Flexible activation functions

Conclusions
  Conclusions
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×, while quantization further improves the compression rate: between 27× and 31×. Huffman coding gives more compression: between 35× and 49×. The compression rate already included the meta-data for sparse representation. The compression scheme doesn’t incur any accuracy loss.

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

![Figure 14](image)

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.
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.

![Histograms showing weight and index distributions.](image)

**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.

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

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

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

Introduction
  Building and optimizing a neural network
  Some applications of neural networks
  Do we need all these parameters?

Compression by regularization
  Sparse and group-sparse penalties

Compression by quantization
  Limited-precision arithmetic for neural networks
  Binary neural networks
  Multi-stage compression

Other compression techniques
  Distilling the knowledge
  Flexible activation functions

Conclusions
  Conclusions
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.

"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."

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

\[ g(s) = \frac{a(1 - \exp\{-bs\})}{1 + \exp\{-bs\}}. \quad (13) \]

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

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

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:

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

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.

Maxout neurons

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

$$g(h) = \max_{i=1,...,K} \{w_i^T h + b_i\}.$$  (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 $K$.

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:

$$g(s) = \sum_{i=1}^{D} \alpha_i \kappa(s, d_i),$$  \hspace{1cm} (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.

Kernel selection

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

\[ \kappa(s, d_i) = \exp \left\{ -\gamma (s - d_i)^2 \right\}, \quad (18) \]

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:

\[ \gamma = \frac{1}{6\Delta^2}, \quad (19) \]

where \( \Delta \) 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 $\gamma$. 
Other than initializing the mixing coefficients randomly, we can also approximate any initial function using kernel ridge regression (KRR):

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

(20)

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

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.
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

<table>
<thead>
<tr>
<th>Activation function</th>
<th>Testing AUC</th>
<th>Trainable parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU (five hidden layers)</td>
<td>0.8739(0.001)</td>
<td></td>
</tr>
<tr>
<td>ELU (five hidden layers)</td>
<td>0.8739(0.001)</td>
<td></td>
</tr>
<tr>
<td>SELU (five hidden layers)</td>
<td>0.8745(0.002)</td>
<td></td>
</tr>
<tr>
<td>PReLU (five hidden layers)</td>
<td>0.8748(0.001)</td>
<td></td>
</tr>
<tr>
<td>Maxout (one layer)</td>
<td>0.8744(0.001)</td>
<td></td>
</tr>
<tr>
<td>Maxout (two layers)</td>
<td>0.8744(0.002)</td>
<td></td>
</tr>
<tr>
<td>APL (one layer)</td>
<td>0.8744(0.002)</td>
<td></td>
</tr>
<tr>
<td>APL (two layers)</td>
<td>0.8757(0.002)</td>
<td></td>
</tr>
<tr>
<td>KAF (one layer)</td>
<td>0.8756(0.001)</td>
<td></td>
</tr>
<tr>
<td>KAF (two layers)</td>
<td>0.8758(0.001)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Results on the SUSY benchmark.
Table of contents

Introduction
  Building and optimizing a neural network
  Some applications of neural networks
  Do we need all these parameters?

Compression by regularization
  Sparse and group-sparse penalties

Compression by quantization
  Limited-precision arithmetic for neural networks
  Binary neural networks
  Multi-stage compression

Other compression techniques
  Distilling the knowledge
  Flexible activation functions

Conclusions
  Conclusions
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.