AMLD – Deep Learning in PyTorch

4. Convolution Neural Networks

François Fleuret
http://fleuret.org/amld/
February 10, 2018
Convolutional layers
If they were handled as normal “unstructured” vectors, large-dimension signals such as sound samples or images would require models of intractable size.

For instance a linear layer taking a $256 \times 256$ RGB image as input, and producing an image of same size would require

$$(256 \times 256 \times 3)^2 \approx 3.87e+10$$

parameters, with the corresponding memory footprint ($\approx 150$Gb !), and excess of capacity.
Moreover, this requirement is inconsistent with the intuition that such large signals have some “invariance in translation”. A representation meaningful at a certain location can / should be used everywhere.
Moreover, this requirement is inconsistent with the intuition that such large signals have some “invariance in translation”. A representation meaningful at a certain location can / should be used everywhere.

A convolutional layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.
\[ W = \begin{bmatrix}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{bmatrix} \]
$W - w + 1$

Input

$\begin{bmatrix} 1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1 \end{bmatrix}$

Kernel

$\begin{bmatrix} 1 & 2 & 0 & -1 \end{bmatrix}$

Output

$W - w + 1$
The diagram illustrates a convolution operation in a 1D convolutional neural network. The input and output are represented as follows:

- **Input**:
  - A sequence of values: 1, 4, -1, 0, 2, -2, 1, 3, 3, 1

- **Kernel** (`w`):
  - A smaller sequence representing the kernel: 1, 2, 0, -1

- **Output**:
  - A single value resulting from the convolution: 9

The convolution operation is defined as:

\[ \text{Output} = (W - w + 1) \]

Where `W` is the input size and `w` is the kernel size.
Input

\[
\begin{array}{cccccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1 \\
\end{array}
\]

\[W\]

\[w\]

Output

\[
\begin{array}{cc}
9 & 0 \\
\end{array}
\]

\[W - w + 1\]
\[ \text{Input} \]

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 0 & -1 \\
\end{array}
\]

\[ \text{Output} \]

\[
\begin{array}{cccc}
9 & 0 & 1 \\
\end{array}
\]

\[ W - w + 1 \]
Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{array}
\]

\[W\]

\[
\begin{array}{cccc}
1 & 2 & 0 & -1
\end{array}
\]

\[w\]

Output

\[
\begin{array}{cccc}
9 & 0 & 1 & 3
\end{array}
\]

\[W - w + 1\]
Let's consider a convolution operation with a kernel `W` of size `w` and an input matrix `W - w + 1`.
Input

\[ \begin{array}{cccccc}
1 & 4 & -1 & 0 & 2 & -2 \\
\end{array} \]

\[ W \]

\[ \begin{array}{cccccc}
1 & 3 & 3 & 1 \\
\end{array} \]

Output

\[ \begin{array}{cccccc}
9 & 0 & 1 & 3 & -5 & -3 & 6 \\
\end{array} \]

\[ W - w + 1 \]
\[ W - w + 1 \]
Formally, in 1d, given

\[ x = (x_1, \ldots, x_W) \]

and a “convolutional kernel” (or “filter”) of width \( w \)

\[ u = (u_1, \ldots, u_w) \]
Formally, in 1d, given
\[ x = (x_1, \ldots, x_W) \]
and a “convolutional kernel” (or “filter”) of width \( w \)
\[ u = (u_1, \ldots, u_w) \]
the convolution \( x \ast u \) is a vector of size \( W - w + 1 \), with
\[
(x \ast u)_i = (x_i, \ldots, x_{i+w-1}) \cdot u = \sum_{j=1}^{w} x_{i-1+j} u_j
\]
for instance
\[
(1, 2, 3, 4) \ast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).
\]
Formally, in 1d, given

\[ x = (x_1, \ldots, x_W) \]

and a “convolutional kernel” (or “filter”) of width \( w \)

\[ u = (u_1, \ldots, u_w) \]

the convolution \( x \odot u \) is a vector of size \( W - w + 1 \), with

\[
(x \odot u)_i = (x_i, \ldots, x_{i+w-1}) \cdot u = \sum_{j=1}^{w} x_{i-1+j} u_j
\]

for instance

\[
(1, 2, 3, 4) \odot (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).
\]

⚠️ This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.
Convolution can implement a differential operator

\[(0, 0, 0, 0, 1, 2, 3, 4, 4, 4) \ast (-1, 1) = (0, 0, 0, 1, 1, 1, 0, 0, 0)\].
Convolution can implement a differential operator

\[(0, 0, 0, 0, 1, 2, 3, 4, 4, 4) \ast (-1, 1) = (0, 0, 0, 1, 1, 1, 0, 0, 0).\]
Convolution can implement a differential operator

\((0, 0, 0, 0, 1, 2, 3, 4, 4, 4) \ast (-1, 1) = (0, 0, 0, 1, 1, 1, 0, 0, 0)\).

or a crude “template matcher”
Convolution can implement a differential operator

\[(0, 0, 0, 0, 1, 2, 3, 4, 4, 4) \ast (-1, 1) = (0, 0, 0, 1, 1, 1, 0, 0, 0).\]

or a crude “template matcher”

Both of these computation examples are indeed “invariant by translation”.

It generalizes naturally to a multi-dimensional input, although specification can become complicated.
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

Its most usual form for “convolutional networks” processes a 3d tensor as input (i.e. a multi-channel 2d signal) to output a 2d tensor.

In this case, if the input tensor is of size $C \times H \times W$, the kernel is a tensor of size $C \times h \times w$ and the output will be of size $(H - h + 1) \times (W - w + 1)$. 
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

Its most usual form for “convolutional networks” processes a 3d tensor as input (i.e. a multi-channel 2d signal) to output a 2d tensor.

In this case, if the input tensor is of size $C \times H \times W$, the kernel is a tensor of size $C \times h \times w$ and the output will be of size $(H - h + 1) \times (W - w + 1)$.

In a standard convolutional layer, $D$ such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

Its most usual form for “convolutional networks” processes a 3d tensor as input (i.e. a multi-channel 2d signal) to output a 2d tensor.

In this case, if the input tensor is of size $C \times H \times W$, the kernel is a tensor of size $C \times h \times w$ and the output will be of size $(H - h + 1) \times (W - w + 1)$.

In a standard convolutional layer, $D$ such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.

⚠️ We say “2d signal” even though it has $C$ channels, since it is a feature vector indexed by a 2d location without structure on the feature indexes.
Input

\[ H - h + 1 \]

\[ W - w + 1 \]
Kernel $h \times w + 1 \times 1$
The equation for the Convolution operation is:

\[ (D-h+1) \times (H-w+1) \times 1 \]
Input

Kernel

Output

$$\text{Kernels} = D - h + 1, \quad W - w + 1$$
Kernel

\[
(D - h + 1) \times (W - w + 1) \times 1
\]
The diagram illustrates a convolutional neural network layer where an input tensor is convolved with a kernel to produce an output tensor. The input tensor is represented as a 3D array with dimensions $H \times W \times C$, where $H$ is the height, $W$ is the width, and $C$ is the number of channels. The kernel is a smaller 3D array with dimensions $h \times w \times C$, where $h$ is the height and $w$ is the width of the kernel. The output tensor has dimensions $D \times H - h + 1 \times W - w + 1 \times C$, where $D$ is the number of output channels.
Kernels

\[ D - h + 1 \]

\[ H - w + 1 \]

Input

Kernel

Output

\[ C \]

\[ w \]

\[ h \]
Kernel

\[ H - h + 1 \]

\[ W - w + 1 \]

\[ 1 \]
The convolution operation involves a kernel (also known as a filter) that slides over the input image, performing element-wise multiplications and summing the results. The dimensions of the input and kernel are typically denoted as follows:

- **Input**:
  - Width: \( W \)
  - Height: \( H \)
  - Channels: \( C \)

- **Kernel**:
  - Width: \( w \)
  - Height: \( h \)
  - Channels: \( C \)

The output of the convolution operation is calculated as:

\[
(D - h + 1)(W - w + 1)
\]

where:
- \( D \) is the depth of the output feature map.
- \( W \) is the width of the input.
- \( H \) is the height of the input.
- \( h \) is the height of the kernel.
- \( w \) is the width of the kernel.

The output feature map has a size of \((D - h + 1)(W - w + 1)\) for each channel..
The equation for the dimensions of the output $O$ after applying a convolution with a kernel $K$ of size $h \times w$ to an input $I$ of size $H \times W \times C$ is given by:

$$D - h + 1$$

$$H - w + 1$$

$$W - w + 1$$

where $C$ represents the number of channels.
\[ D \cdot H - h + 1 \]
\[ W \cdot W - w + 1 \]
\[ C \cdot H - h + 1 \]
\[ C \cdot W - w + 1 \]
Kernel

\[ \mathbf{K} = \mathbf{D}_{h \times w} \cdot \mathbf{X}_{H \times W} \]

Input

\[ \mathbf{X}_{H \times W} \]

Kernels

\[ \mathbf{K}_{h \times w \times D \times C} \]

Output

\[ \mathbf{Y}_{(H-h+1) \times (W-w+1) \times D \times C} \]
Note that convolution preserves the signal support structure.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.
Note that convolution preserves the signal support structure.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

In particular the convolution of a \( C \times H \times W \) tensor with a \( C \times 1 \times 1 \) kernel can be interpreted as applying the same linear classifier at every point separately.
Note that convolution preserves the signal support structure.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

In particular the convolution of a $C \times H \times W$ tensor with a $C \times 1 \times 1$ kernel can be interpreted as applying the same linear classifier at every point separately.

We usually refer to one of the channels generated by a convolutional layer as an activation map.

The sub-area of an input map that influences a component of the output as the receptive field of the latter.
Note that convolution preserves the signal support structure.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

In particular the convolution of a $C \times H \times W$ tensor with a $C \times 1 \times 1$ kernel can be interpreted as applying the same linear classifier at every point separately.

We usually refer to one of the channels generated by a convolutional layer as an activation map.

The sub-area of an input map that influences a component of the output as the receptive field of the latter.

In the context of convolutional networks, a standard linear layer is called a fully connected layer since every input influences every output.
Stride, padding, and dilation
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2,1)$...
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 

![Diagram of convolution operation with input and output showing padding and stride effects.](image-url)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2,1)$, a stride of $(2,2)$, and a kernel of size $C \times 3 \times 3$. 

![Diagram showing input and output with padding and stride examples](image-url)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 

[Diagram of convolution operation]
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2,1)$, a stride of $(2,2)$, and a kernel of size $C \times 3 \times 3$. 

![Input and Output Diagram]
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2,1)$, a stride of $(2,2)$, and a kernel of size $C \times 3 \times 3$. 

![Diagram showing input and output with padding and stride applied to convolution operations.](image-url)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 

![Input and Output Diagram](image-url)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 

![Diagram showing input and output of a convolution operation with padding and stride.](image)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 

![Diagram of convolution operation with padding and stride](image)
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$. 
Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$.

Padding can be useful to generate an output of same size as the input.
Recently, the notion of **dilated convolutions** was introduced to increase the receptive fields without increasing the number of parameters (Yu and Koltun, 2015).

The **dilation** parametrizes the expansion of the filter. It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.
Recently, the notion of **dilated convolutions** was introduced to increase the receptive fields without increasing the number of parameters (Yu and Koltun, 2015).

The **dilation** parametrizes the expansion of the filter. It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

This notion comes from signal processing, where it is referred to as *algorithme à trous*, hence the term sometime used of “convolution à trous”.
Dilation = 1
Dilation = 1

Input

Output
Dilation = 1
Dilation = 1
Dilation = 1
Dilation = 1

Input

Output
Dilation = 1

Input

Output
Dilation = 2

Input

Output
Dilation = 2
Dilation = 2
Dilation = 2

Input

Output
Dilation = 2

Francois Fleuret

AMLD – Deep Learning in PyTorch / 4. Convolution Neural Networks
Dilation = 2
Dilation = 2

Input

Output
A convolution with a 1d kernel of size $k$ and dilation $d$ can be interpreted as a convolution with a filter of size $1 + (k - 1)d$ with only $k$ non-zero coefficients.

For with $k = 3$ and $d = 4$, the difference between the input map size and the output map size is $1 + (3 - 1)4 - 1 = 8$.

```python
>>> from torch import nn, Tensor
>>> from torch.autograd import Variable
>>> x = Variable(Tensor(1, 1, 20, 30).normal_())
>>> f = nn.Conv2d(1, 1, kernel_size = 3, dilation = 4)
>>> f(x).size()
torch.Size([1, 1, 12, 22])
```
Having a dilation greater than one increases the units’ receptive field size without increasing the number of parameters.

**Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision.**
Having a dilation greater than one increases the units’ receptive field size without increasing the number of parameters.

Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision.

Such networks have the advantage of simplicity:

- non-linear operations are only in the activation function,
- joint operations (combining multiple activations to produce one) are only in the convolutional layers.
Pooling
In many cases, a feed-forward network computes a low-dimension signal (e.g. a few scores) from a very high-dimension signal (e.g. an image).
In many cases, a feed-forward network computes a low-dimension signal (e.g. a few scores) from a very high-dimension signal (e.g. an image).

As for convolution, it makes sense to reduce the signal’s size in a way that preserves its structure, just “down-scaling it”.

The standard operation to do this is **pooling**, and aims at grouping several activations into a single “more meaningful” one.
1d example of **max-pooling** with a kernel of size 2:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>-2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The average pooling would compute the mean per block instead of the max.
1d example of **max-pooling** with a kernel of size 2:

Input:

```
1  4  -1  0  2  -2  1  3  3  1
```

Output:

```
4
```
1d example of max-pooling with a kernel of size 2:

Input:

```
1  4 -1  0  2 -2  1  3  3  1
```

Output:

```
4  0
```

The average pooling would compute the mean per block instead of the max.
1d example of **max-pooling** with a kernel of size 2:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  4  -1  0  2 -2  1  3  3  1</td>
<td>4  0  2</td>
</tr>
</tbody>
</table>

The average pooling would compute the mean per block instead of the max.
1d example of max-pooling with a kernel of size 2:

Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1 \\
\end{array}
\]

Output

\[
\begin{array}{cccc}
4 & 0 & 2 & 3 \\
\end{array}
\]
1d example of **max-pooling** with a kernel of size 2:
1d example of **max-pooling** with a kernel of size 2:

**Input**

```
1  4  -1  0  2  -2  1  3  3  1
```

**Output**

```
4  0  2  3  3
```

The average pooling would compute the mean per block instead of the max.
1d example of **max-pooling** with a kernel of size 2:

![Input and Output Diagram]

The **average pooling** would compute the mean per block instead of the max.
Input

\[ r \times w \]

\[ s \times h \]

\[ C \]
Input

Output

$r, w$

$s, h$

$C$
Input

Output

$r,w$

$s,h$

$C$
Input

Output

\( r w \)

\( s h \)

C
Francois Fleuret
Input

Output

\( r \)

\( s \)

\( h \)

\( w \)

\( C \)
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.
Both the convolutional and pooling layers take as input batches of samples, each one being itself a 3d tensor $C \times H \times W$.

The output has the same structure, and tensors have to be explicitly reshaped before being forwarded to a fully connected layer.
Both the convolutional and pooling layers take as input batches of samples, each one being itself a 3d tensor $C \times H \times W$.

The output has the same structure, and tensors have to be explicitly reshaped before being forwarded to a fully connected layer.

```python
>>> mnist = datasets.MNIST('./data/mnist/', train=True, download=True)
>>> d = mnist.train_data
>>> d.size()
torch.Size([60000, 28, 28])
>>> x = d.view(d.size(0), 1, d.size(1), d.size(2))
>>> x.size()
torch.Size([60000, 1, 28, 28])
>>> x = d.view(d.size(0), -1)
>>> x.size()
torch.Size([60000, 784])
```
torch.nn.Module
PyTorch provides a vast collection of modules, which implement standard operations and can be combined into complicated “deep” architectures.

We will look at components to build our first convolutional neural network:

- `torch.nn.functional.relu`
- `torch.nn.functional.max_pool2d`
- `torch.nn.Conv2d`
- `torch.nn.Linear`
- `torch.nn.MSELoss`
PyTorch provides a vast collection of `Module` s, which implement standard operations and can be combined into complicated “deep” architectures.

We will look at components to build our first convolutional neural network:

- `torch.nn.functional.relu`
- `torch.nn.functional.max_pool2d`
- `torch.nn.Conv2d`
- `torch.nn.Linear`
- `torch.nn.MSELoss`

Elements from `torch.nn.functional` are autograd-compliant functions which compute a result from provided arguments alone. This is usually imported as `F`. 
PyTorch provides a vast collection of modules, which implement standard operations and can be combined into complicated “deep” architectures.

We will look at components to build our first convolutional neural network:

- torch.nn.functional.relu
- torch.nn.functional.max_pool2d
- torch.nn.Conv2d
- torch.nn.Linear
- torch.nn.MSELoss

Elements from torch.nn.functional are autograd-compliant functions which compute a result from provided arguments alone. This is usually imported as \( F \).

Modules from torch.nn are components for networks which embed torch.nn.Parameter\_s to be optimized during training. They can also be criteria (e.g. losses). They usually use torch.nn.functional\_s.
torch.nn.functional.relu(input, inplace=False)

Takes a tensor of any size as input, applies ReLU on each value to produce a result tensor of same size.
torch.nn.functional.relu(input, inplace=False)

Takes a tensor of any size as input, applies ReLU on each value to produce a result tensor of same size.

```plaintext
>>> x = Variable(Tensor(2, 5).normal_())
Variable containing:
-0.2066 -1.7997 -0.0653 0.6481 0.0253
 1.0239  3.0324  1.6431 -1.8925  0.0890
[torch.FloatTensor of size 2x5]

>>> torch.nn.functional.relu(x)
Variable containing:
 0.0000  0.0000  0.0000  0.6481  0.0253
 1.0239  3.0324  1.6431  0.0000  0.0890
[torch.FloatTensor of size 2x5]
```
torch.nn.functional.relu(input, inplace=False)

Takes a tensor of any size as input, applies ReLU on each value to produce a result tensor of same size.

```python
>>> x = Variable(Tensor(2, 5).normal_())
>>> x
Variable containing:
-0.2066  -1.7997  -0.0653   0.6481   0.0253
 1.0239   3.0324  1.6431  -1.8925   0.0890
[torch.FloatTensor of size 2x5]
```

```python
>>> torch.nn.functional.relu(x)
Variable containing:
 0.0000   0.0000   0.0000   0.6481   0.0253
 1.0239   3.0324  1.6431   0.0000   0.0890
[torch.FloatTensor of size 2x5]
```

**inplace** indicates if the operation should modify the argument itself. This may be desirable to reduce the memory footprint of the processing.
torch.nn.functional.max_pool2d(input, kernel_size,
    stride=None, padding=0, dilation=1,
    ceil_mode=False, return_indices=False)

Takes as input either a $C \times H \times W$ or $N \times C \times H \times W$ tensor, and a kernel size which can be a single integer $k$ or a pair $(k, l)$, and applies the max-pooling on each channel of each sample separately.
torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)

Takes as input either a \( C \times H \times W \) or \( N \times C \times H \times W \) tensor, and a kernel size which can be a single integer \( k \) or a pair \((k, l)\), and applies the max-pooling on each channel of each sample separately.

```python
>>> x = Variable(Tensor(2, 2, 6).random_(3))
>>> x
Variable containing:
(0 ,...,)
  0 0 2 0 0 1
  2 1 1 2 0 1
(1 ,...,)
  2 0 0 2 2 0
  2 1 1 0 1 0
[torch.FloatTensor of size 2x2x6]

>>> torch.nn.functional.max_pool2d(x, (1, 2))
Variable containing:
(0 ,...,)
  0 2 1
  2 2 1
(1 ,...,)
  2 2 2
  2 1 1
[torch.FloatTensor of size 2x2x3]
```
class torch.nn.Linear(in_features, out_features, bias=True)

Implements a fully-connected layer with the given input and output dimensions.
class torch.nn.Linear(in_features, out_features, bias=True)

Implements a fully-connected layer with the given input and output dimensions.

```python
>>> f = torch.nn.Linear(in_features = 10, out_features = 4)
>>> f.weight.size()
torch.Size([4, 10])
>>> f.bias.size()
torch.Size([4])
>>> x = Variable(Tensor(523, 10).normal_())
>>> y = f(x)
>>> y.size()
torch.Size([523, 4])
```
class torch.nn.Linear(in_features, out_features, bias=True)

Implements a fully-connected layer with the given input and output dimensions.

```python
>>> f = torch.nn.Linear(in_features = 10, out_features = 4)
>>> f.weight.size()
torch.Size([4, 10])
>>> f.bias.size()
torch.Size([4])
>>> x = Variable(Tensor(523, 10).normal_())
>>> y = f(x)
>>> y.size()
torch.Size([523, 4])
```

⚠️ The weights and biases are automatically randomized at creation. We will come back to that later.
torch.nn.Conv2d(in_channels, out_channels, 
  kernel_size, 
  stride=1, padding=0, dilation=1, groups=1, bias=True)

Implements a standard 2d convolutional layer.

It takes as input either a $C \times H \times W$ or $N \times C \times H \times W$ tensor, and a kernel size which can be a single integer $k$ or a pair $(k, l)$, and applies the convolution on each channel of each sample separately.
torch.nn.Conv2d(in_channels, out_channels, 
    kernel_size, 
    stride=1, padding=0, dilation=1, groups=1, bias=True)

Implements a standard 2d convolutional layer.

It takes as input either a $C \times H \times W$ or $N \times C \times H \times W$ tensor, and a kernel size which can be a single integer $k$ or a pair $(k, l)$, and applies the convolution on each channel of each sample separately.

```python
>>> f = torch.nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> f.weight.size()
torch.Size([5, 4, 2, 3])
>>> f.bias.size()
torch.Size([5])
>>> x = Variable(Tensor(117, 4, 10, 3).normal_())
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

As for the fully connected layer, weights and biases are randomized.
```python
x = mnist_train.train_data.narrow(0, 12, 1).float()

f = torch.nn.Conv2d(1, 5, kernel_size=3)

f.bias.data.zero_()

f.weight.data[0] = Tensor([ [ 0, 0, 0 ],
                           [ 0, 1, 0 ],
                           [ 0, 0, 0 ] ])

f.weight.data[1] = Tensor([ [ 1, 1, 1 ],
                           [ 1, 1, 1 ],
                           [ 1, 1, 1 ] ])

f.weight.data[2] = Tensor([ [ -1, 0, 1 ],
                           [ -1, 0, 1 ],
                           [ -1, 0, 1 ] ])

f.weight.data[3] = Tensor([ [ -1, -1, -1 ],
                           [ 0, 0, 0 ],
                           [ 1, 1, 1 ] ])

f.weight.data[4] = Tensor([ [ 0, -1, 0 ],
                           [ -1, 4, -1 ],
                           [ 0, -1, 0 ] ])

y = f(Variable(x.view(1, x.size(0), x.size(1), x.size(2))).data)

save_2d_tensor_as_image(f.weight.data.squeeze(), 'conv-filters-{:d}.png',
                        signed = True)

save_2d_tensor_as_image(x, 'conv-mnist-orig.png', signed = True)

save_2d_tensor_as_image(y.squeeze(), 'conv-mnist-results-{:d}.png',
                        signed = True)
```
3

3
François Fleuret

AMLD – Deep Learning in PyTorch / 4. Convolution Neural Networks

31 / 53
torch.nn.MSELoss()

Implements the “Mean square error” loss: It takes two arbitrary shaped tensors of same size and computes the sum of the squared errors, **divided by the total number of components in the tensors.**
torch.nn.MSELoss()

Implements the “Mean square error” loss: It takes two arbitrary shaped tensors of same size and computes the sum of the squared errors, divided by the total number of components in the tensors.

```python
>>> f = torch.nn.MSELoss()
>>> x = Variable(Tensor([ 3 ]))
>>> y = Variable(Tensor([ 0 ]))
>>> f(x, y)
Variable containing:
 9
[torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
 3
[torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
 0.9000
[torch.FloatTensor of size 1]
```
torch.nn.MSELoss()

Implements the “Mean square error” loss: It takes two arbitrary shaped tensors of same size and computes the sum of the squared errors, divided by the total number of components in the tensors.

```python
>>> f = torch.nn.MSELoss()
>>> x = Variable(Tensor([ 3 ]))
>>> y = Variable(Tensor([ 0 ]))
>>> f(x, y)
Variable containing:
  9
  [torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
  3
  [torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
  0.9000
  [torch.FloatTensor of size 1]
```

The first parameter is traditionally called the “input” and the second the “target”. For some losses, these two quantities may be of different dimensions or even type (e.g. for classification).
Module inputs and outputs are Variable s. Data Tensor s should be converted before forwarding them into a model.

```python
>>> mnist = torchvision.datasets.MNIST('./data/mnist/')
>>> x = mnist.train_data.float()
>>> x = x.view(x.size(0), -1)
>>> f = nn.Linear(x.size(1), 10)
>>> y = f(x)
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch.nn/modules/module.py", line 325, in __call__
    result = self.forward(*input, **kwargs)
    return F.linear(input, self.weight, self.bias)
    return torch.addmm(bias, input, weight.t())
RuntimeError: addmm(): argument 'mat1' (position 1) must be Variable, not torch.FloatTensor
>>> x = torch.autograd.Variable(x)
>>> y = f(x)
```
Criteria do not compute the gradient with respect to the target, and will not accept a `Variable` with `requires_grad` to `True` as the target.

```python
>>> f = torch.nn.MSELoss()
>>> x = Variable(Tensor([3, 2]), requires_grad = True)
>>> y = Variable(Tensor([0, -2]), requires_grad = True)
>>> f(x, y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    result = self.forward(*input, **kwargs)
    _assert_no_grad(target)
    "nn criterions don’t compute the gradient w.r.t. targets - please " \nAssertionError: nn criterions don’t compute the gradient w.r.t. targets - please mark these variables as volatile or not requiring gradients
```
Such a network can be trained with the Mean-Squared Error loss, even though this is classification.
Such a network can be trained with the Mean-Squared Error loss, even though this is classification. To do so, given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{1, \ldots, C\}, \ n = 1, \ldots, N,$$

we will consider an output with as many units as there are classes, and the target will be a tensor $z \in \mathbb{R}^{N \times C}$, with $−1$ everywhere but for the correct labels:

$$\forall n, \ z_{n,m} = \begin{cases} 
1 & \text{if } m = y_n \\
-1 & \text{otherwise.}
\end{cases}$$

Although MSE is a regression loss, using it like this gives excellent results.
Such a network can be trained with the Mean-Squared Error loss, even though this is classification. To do so, given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{1, \ldots, C\}, \ n = 1, \ldots, N,$$

we will consider an output with as many units as there are classes, and the target will be a tensor $z \in \mathbb{R}^{N \times C}$, with $-1$ everywhere but for the correct labels:

$$\forall n, z_{n,m} = \begin{cases} 1 & \text{if } m = y_n \\ -1 & \text{otherwise.} \end{cases}$$

For instance, with $N = 5$ and $C = 3$, we would have

$$\begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} -1 & 1 & -1 \\ 1 & -1 & -1 \\ 1 & -1 & -1 \\ -1 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix}.$$
Such a network can be trained with the Mean-Squared Error loss, even though this is classification. To do so, given a training set

\[(x_n, y_n) \in \mathbb{R}^D \times \{1, \ldots, C\}, \; n = 1, \ldots, N,\]

we will consider an output with as many units as there are classes, and the target will be a tensor \(z \in \mathbb{R}^{N \times C}\), with \(-1\) everywhere but for the correct labels:

\[
\forall n, \; z_{n,m} = \begin{cases} 
1 & \text{if } m = y_n \\
-1 & \text{otherwise.}
\end{cases}
\]

For instance, with \(N = 5\) and \(C = 3\), we would have

\[
\begin{pmatrix}2 \\
1 \\
1 \\
3 \\
2 \end{pmatrix} \Rightarrow \begin{pmatrix}-1 & 1 & -1 \\
1 & -1 & -1 \\
1 & -1 & -1 \\
-1 & -1 & 1 \\
-1 & 1 & -1 \end{pmatrix}.
\]

Although MSE is a regression loss, using it like this gives excellent results.
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td>32</td>
<td>832</td>
</tr>
<tr>
<td>$32 \times (5^2 + 1) = 832$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td>460</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F_{\text{max}} \text{pool2d}(x, \text{kernel size}=3)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$64 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F_{\text{relu}}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x.\text{view}(-1, 256)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$256 \times (256 + 1) = 51$</td>
<td></td>
<td>400</td>
</tr>
<tr>
<td>$200 \times 256 = 51$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F_{\text{relu}}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$200 \times 200 = 200$</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>$200 \times 200 = 200$</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>$10 \times (200 + 1) = 2$</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$10 \times 200 = 200$</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>$F_{\text{relu}}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total 105,506 parameters and 1,333,200 products for the forward pass.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 24^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td><code>nn.Conv2d(1, 32, kernel_size=5)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nn.Conv2d(1, 32, kernel_size=5)</code></td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 24^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.max_pool2d(x, kernel_size=3)</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$256$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nn.Linear(256, 200)</code></td>
<td>$200 \times (256 + 1) = 51,400$</td>
<td>$200 \times 256 = 51,200$</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10$</td>
<td>$10 \times (200 + 1) = 2,001$</td>
<td>$10 \times 200 = 2,000$</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total 105,506 parameters and 1,333,200 products for the forward pass.
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nn.Conv2d(1, 32, kernel_size=5)</code></td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 24^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.max_pool2d(x, kernel_size=3)</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.relu</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total $105,506$ parameters and $1,333,200$ products for the forward pass.
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel_size=5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel_size=3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>F.relu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel_size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 4^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nn.Conv2d(1, 32, kernel_size=5)</code></td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.max_pool2d(x, kernel_size=3)</code></td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.relu</code></td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nn.Conv2d(32, 64, kernel_size=5)</code></td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 4^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.relu</code></td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>F.max_pool2d(x, kernel_size=2)</code></td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel.size=5)</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 24^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel.size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel.size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 4^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel.size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $\times$ 28 $\times$ 28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel.size=5)</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>32 $\times$ 24 $\times$ 24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel.size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32 $\times$ 8 $\times$ 8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32 $\times$ 8 $\times$ 8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel.size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 4^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>64 $\times$ 4 $\times$ 4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel.size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64 $\times$ 2 $\times$ 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64 $\times$ 2 $\times$ 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x.view(-1, 256)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel_size=5)</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel_size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 4^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.view(-1, 256)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(256, 200)</td>
<td>$200 \times (256 + 1) = 51,400$</td>
<td>$200 \times 256 = 51,200$</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel_size=5)</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel_size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel_size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 2^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max_pool2d(x, kernel_size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.view(-1, 256)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$256$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(256, 200)</td>
<td>$200 \times (256 + 1) = 51,400$</td>
<td>$200 \times 256 = 51,200$</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 28 \times 28$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel_size=5)</td>
<td>$32 \times (5^2 + 1) = 832$</td>
<td>$32 \times 2^2 \times 5^2 = 460,800$</td>
</tr>
<tr>
<td>$32 \times 24 \times 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 8 \times 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel_size=5)</td>
<td>$64 \times (32 \times 5^2 + 1) = 51,264$</td>
<td>$32 \times 64 \times 2^2 \times 5^2 = 819,200$</td>
</tr>
<tr>
<td>$64 \times 4 \times 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$64 \times 2 \times 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.view(-1, 256)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$256$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(256, 200)</td>
<td>$200 \times (256 + 1) = 51,400$</td>
<td>$200 \times 256 = 51,200$</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(200, 10)</td>
<td>$10 \times (200 + 1) = 2,010$</td>
<td>$10 \times 200 = 2,000$</td>
</tr>
<tr>
<td>$10$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

<table>
<thead>
<tr>
<th>Tensor sizes / operations</th>
<th>Nb. parameters</th>
<th>Nb. products</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \times 28 \times 28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(1, 32, kernel_size=5)</td>
<td>32 \times (5^2 + 1) = 832</td>
<td>32 \times 2^2 \times 5^2 = 460,800</td>
</tr>
<tr>
<td>32 \times 24 \times 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32 \times 8 \times 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32 \times 8 \times 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Conv2d(32, 64, kernel_size=5)</td>
<td>64 \times (32 \times 5^2 + 1) = 51,264</td>
<td>32 \times 64 \times 2^2 \times 5^2 = 819,200</td>
</tr>
<tr>
<td>64 \times 4 \times 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.max.pool2d(x, kernel_size=2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64 \times 2 \times 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64 \times 2 \times 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.view(-1, 256)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(256, 200)</td>
<td>200 \times (256 + 1) = 51,400</td>
<td>200 \times 256 = 51,200</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.relu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nn.Linear(200, 10)</td>
<td>10 \times (200 + 1) = 2,010</td>
<td>10 \times 200 = 2,000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total **105,506** parameters and **1,333,200** products for the forward pass.
The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

⚠️ Although they do not have a “_” in their names, these `Module` operations make changes in-place.
The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

Although they do not have a “_” in their names, these `Module` operations make changes in-place.

A typical snippet of code to use the GPU would be

```python
if torch.cuda.is_available():
    model.cuda()
    criterion.cuda()
    train_input, train_target = train_input.cuda(), train_target.cuda()
    test_input, test_target = test_input.cuda(), test_target.cuda()
```
A very simple way to leverage multiple GPUs is to use

```python
torch.nn.DataParallel(module, device_ids)
```

The `forward` of the resulting module will

1. Split the input mini-batch along the first dimension in as many mini-batches as there are GPUs in `device_ids`
2. send them to the `forwards` of clones of `module` located on each GPU,
3. concatenate the results.
class Dummy(nn.Module):
    def __init__(self, m):
        super(Dummy, self).__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), torch.cuda.current_device())
        return self.m(x)
```python
class Dummy(nn.Module):
    def __init__(self, m):
        super(Dummy, self).__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), torch.cuda.current_device())
        return self.m(x)

x = Variable(Tensor(50, 10).normal_())
m = Dummy(nn.Linear(10, 5))
x = x.cuda()
m = m.cuda()

print('Without data_parallel')
y = m(x)

print()

mp = nn.DataParallel(m, range(torch.cuda.device_count()))

print('With data_parallel')
y = mp(x)
```

prints

Without data_parallel
Dummy.forward torch.Size([50, 10]) 0

With data_parallel
Dummy.forward torch.Size([25, 10]) 0
Dummy.forward torch.Size([25, 10]) 1
Creating a module
To create a `Module`, one has to inherit from the base class and implement the constructor `__init__(self, ...)` and the forward pass `forward(self, x)`.
To create a `Module`, one has to inherit from the base class and implement the constructor `__init__(self, ...) and the forward pass `forward(self, x)`.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

As long as you use autograd-compliant operations, the backward pass is implemented automatically.
To create a Module, one has to inherit from the base class and implement the constructor `__init__(self, ...)` and the forward pass `forward(self, x)`.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

As long as you use autograd-compliant operations, the backward pass is implemented automatically.
To create a Module, one has to inherit from the base class and implement the constructor `__init__(self, ...) and the forward pass `forward(self, x).

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

As long as you use autograd-compliant operations, the backward pass is implemented automatically.
Module's added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.
Module's added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    model = Net()

    for k in model.parameters():
        print(k.size())
```
Modules added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    model = Net()
    for k in model.parameters():
        print(k.size())
```

prints

```
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```
Modules added as attributes are seen by `Module.parameters()`, which returns an iterator over the model’s parameters for optimization.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

model = Net()

for k in model.parameters():
    print(k.size())
```

prints

```text
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```
Module's added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

model = Net()

for k in model.parameters():
    print(k.size())

prints

torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
Modules added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

        model = Net()

        for k in model.parameters():
            print(k.size())
```

prints

```python
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```
Modules added as attributes are seen by `Module.parameters()`, which returns an iterator over the model’s parameters for optimization.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

model = Net()

for k in model.parameters():
    print(k.size())

prints

torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])```
Parameters added as attributes are also seen by `Module.parameters()`.
Parameters added as attributes are also seen by `Module.parameters()`.

⚠️ Parameters added in dictionaries or arrays are not seen.
Parameters added as attributes are also seen by `Module.parameters()`.

⚠️ Parameters added in dictionaries or arrays are not seen.

class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.ouch = {}
        self.ouch[0] = nn.Linear(543, 21)

model = Buggy()

for k in model.parameters():
    print(k.size())

prints

torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
Parameters added as attributes are also seen by `Module.parameters()`.

⚠️ Parameters added in dictionaries or arrays are not seen.

```python
class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.ouch = {}
        self.ouch[0] = nn.Linear(543, 21)

model = Buggy()

for k in model.parameters():
    print(k.size())
```

prints

```none```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
```
Parameters added as attributes are also seen by `Module.parameters()`.

⚠️ Parameters added in dictionaries or arrays are not seen.

```python
class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.ouch = {}
        self.ouch[0] = nn.Linear(543, 21)

model = Buggy()

for k in model.parameters():
    print(k.size())
```

prints

```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
```
Parameters added as attributes are also seen by `Module.parameters()`.

⚠ Parameters added in dictionaries or arrays are not seen.

class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.ouch = {}
        self.ouch[0] = nn.Linear(543, 21)

model = Buggy()

for k in model.parameters():
    print(k.size())

prints

torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
The proper policy then is to use \texttt{Module.add.module(name, module)}

```python
class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

for k in model.parameters():
    print(k.size())
```
The proper policy then is to use `Module.add_module(name, module)`

class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

for k in model.parameters():
    print(k.size())

prints

torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
torch.Size([21])
The proper policy then is to use `Module.add_module(name, module)`

class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

for k in model.parameters():
    print(k.size())

prints

torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
torch.Size([21])
The proper policy then is to use `Module.add_module(name, module)`

class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

for k in model.parameters():
    print(k.size())

prints

torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
torch.Size([21])

These modules are added as attributes, and can be accessed with `getattr`.
The proper policy then is to use `Module.add_module(name, module)`

```python
class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

for k in model.parameters():
    print(k.size())
```

prints

```python
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
torch.Size([21])
```

These modules are added as attributes, and can be accessed with `getattr`.

`Module.register_parameter(name, parameter)` allows to similarly register Parameter's explicitly.
Another option is to add modules in a field of type `nn.ModuleList`, which is a list of modules properly dealt with by PyTorch’s machinery.

```python
class AnotherNotBuggy(nn.Module):
    def __init__(self):
        super(AnotherNotBuggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.other_stuff = nn.ModuleList()
        self.other_stuff.append(nn.Linear(50, 75))
        self.other_stuff.append(nn.Linear(125, 999))

model = AnotherNotBuggy()

for k in model.parameters():
    print(k.size())
```

prints

```python
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([75, 50])
torch.Size([75])
torch.Size([999, 125])
torch.Size([999])
```
Image classification, standard convnets
The most standard networks for image classification are the LeNet family (LeCun et al., 1998), and its modern extensions, among which AlexNet (Krizhevsky et al., 2012) and VGGNet (Simonyan and Zisserman, 2014).

They share a common structure of several convolutional layers seen as a feature extractor, followed by fully connected layers seen as a classifier.
The most standard networks for image classification are the LeNet family (LeCun et al., 1998), and its modern extensions, among which AlexNet (Krizhevsky et al., 2012) and VGGNet (Simonyan and Zisserman, 2014).

They share a common structure of several convolutional layers seen as a feature extractor, followed by fully connected layers seen as a classifier.

The performance of AlexNet was a wake-up call for the computer vision community, as it vastly out-performed other methods in spite of its simplicity.
The most standard networks for image classification are the LeNet family (LeCun et al., 1998), and its modern extensions, among which AlexNet (Krizhevsky et al., 2012) and VGGNet (Simonyan and Zisserman, 2014).

They share a common structure of several convolutional layers seen as a feature extractor, followed by fully connected layers seen as a classifier.

The performance of AlexNet was a wake-up call for the computer vision community, as it vastly out-performed other methods in spite of its simplicity.

Recent advances rely on moving from standard convolutional layers to local complex architectures to reduce the model size.
torchvision.models provides a collection of reference networks for computer vision, e.g.:

```python
import torchvision
alexnet = torchvision.models.alexnet()
```
`torchvision.models` provides a collection of reference networks for computer vision, e.g.:

```python
import torchvision
alexnet = torchvision.models.alexnet()
```

The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given their size.
torchvision.models provides a collection of reference networks for computer vision, e.g.:

```python
import torchvision
alexnet = torchvision.models.alexnet()
```

The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given their size.

⚠️ Networks from PyTorch may differ slightly from the reference papers which introduced them historically.
LeNet5 (LeCun et al., 1989). 10 classes, input $1 \times 28 \times 28$.

(features): Sequential (  
(0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
(1): ReLU (inplace)  
(2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
(3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
(4): ReLU (inplace)  
(5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
)

(classifier): Sequential (  
(0): Linear (400 -> 120)  
(1): ReLU (inplace)  
(2): Linear (120 -> 84)  
(3): ReLU (inplace)  
(4): Linear (84 -> 10)  
)
Alexnet (Krizhevsky et al., 2012). 1,000 classes, input $3 \times 224 \times 224$.

(features): Sequential (  
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))  
(1): ReLU (inplace)  
(2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))  
(4): ReLU (inplace)  
(5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(7): ReLU (inplace)  
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(9): ReLU (inplace)  
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(11): ReLU (inplace)  
(12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
)

(classifier): Sequential (  
(0): Dropout (p = 0.5)  
(1): Linear (9216 -> 4096)  
(2): ReLU (inplace)  
(3): Dropout (p = 0.5)  
(4): Linear (4096 -> 4096)  
(5): ReLU (inplace)  
(6): Linear (4096 -> 1000)  
)
We can illustrate the convenience of these pre-trained models on a simple image-classification problem.

To be sure this picture did not appear in the training data, it was not taken from the web.
import PIL, torch, torchvision

# Load and normalize the image
img = torchvision.transforms.ToTensor()(PIL.Image.open('blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()
import PIL, torch, torchvision

# Load and normalize the image
img = torchvision.transforms.ToTensor()(PIL.Image.open('blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

# Load an already trained network and compute its prediction
alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()

output = alexnet(Variable(img))
import PIL, torch, torchvision

# Load and normalize the image
img = torchvision.transforms.ToTensor()(PIL.Image.open('blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

# Load an already trained network and compute its prediction
alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()

output = alexnet(Variable(img))

# Prints the classes
scores, indexes = output.data.view(-1).sort(descending = True)

class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())

for k in range(15):
    print('#{:d} ({:.02f}) {:s}'.format(k, scores[k], class_names[indexes[k]]))
<table>
<thead>
<tr>
<th>#</th>
<th>Rank</th>
<th>Breed</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>12.26</td>
<td>Weimaraner</td>
</tr>
<tr>
<td>#2</td>
<td>10.95</td>
<td>Chesapeake Bay retriever</td>
</tr>
<tr>
<td>#3</td>
<td>10.87</td>
<td>Labrador retriever</td>
</tr>
<tr>
<td>#4</td>
<td>10.10</td>
<td>Staffordshire bullterrier, Staffordshire bull terrier</td>
</tr>
<tr>
<td>#5</td>
<td>9.55</td>
<td>flat-coated retriever</td>
</tr>
<tr>
<td>#6</td>
<td>9.40</td>
<td>Italian greyhound</td>
</tr>
<tr>
<td>#7</td>
<td>9.31</td>
<td>American Staffordshire terrier, Staffordshire terrier, American pit bull terrier, pit bull terrier</td>
</tr>
<tr>
<td>#8</td>
<td>9.12</td>
<td>Great Dane</td>
</tr>
<tr>
<td>#9</td>
<td>8.94</td>
<td>German short-haired pointer</td>
</tr>
<tr>
<td>#10</td>
<td>8.53</td>
<td>Doberman, Doberman pinscher</td>
</tr>
<tr>
<td>#11</td>
<td>8.35</td>
<td>Rottweiler</td>
</tr>
<tr>
<td>#12</td>
<td>8.25</td>
<td>kelpie</td>
</tr>
<tr>
<td>#13</td>
<td>8.24</td>
<td>barrow, garden cart, lawn cart, wheelbarrow</td>
</tr>
<tr>
<td>#14</td>
<td>8.12</td>
<td>bucket, pail</td>
</tr>
<tr>
<td>#15</td>
<td>8.07</td>
<td>soccer ball</td>
</tr>
</tbody>
</table>
#1 (12.26) Weimaraner
#2 (10.95) Chesapeake Bay retriever
#3 (10.87) Labrador retriever
#4 (10.10) Staffordshire bullterrier, Staffordshire bull terrier
#5 (9.55) flat-coated retriever
#6 (9.40) Italian greyhound
#7 (9.31) American Staffordshire terrier, Staffordshire terrier, American pit bull terrier, pit bull terrier
#8 (9.12) Great Dane
#9 (8.94) German short-haired pointer
#10 (8.53) Doberman, Doberman pinscher
#11 (8.35) Rottweiler
#12 (8.25) kelpie
#13 (8.24) barrow, garden cart, lawn cart, wheelbarrow
#14 (8.12) bucket, pail
#15 (8.07) soccer ball
The end
References


