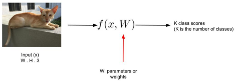
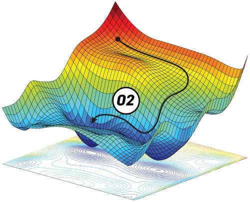
Deep Learning for Computer Vision

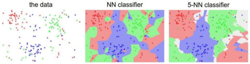
Dr. Konda Reddy Mopuri
Mehta Family School of Data Science and Artificial Intelligence
IIT Guwahati
Aug-Dec 2022

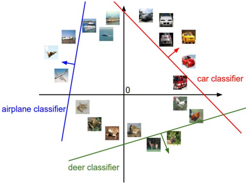
So far in the course

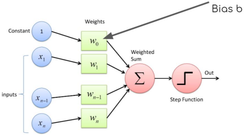
- Scoring function, loss function, gradient descent

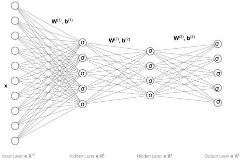






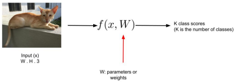


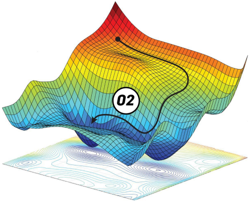


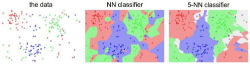


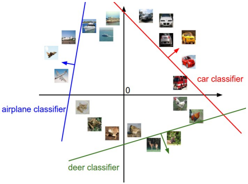
So far in the course

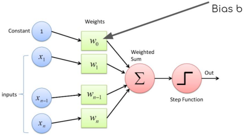
- Scoring function, loss function, gradient descent
- Artificial Neurons and Multi-Layered Perceptron

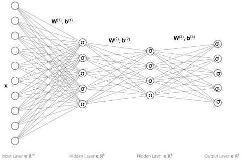






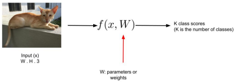


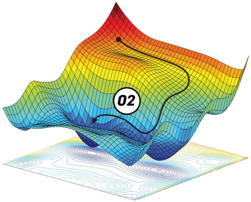


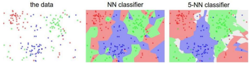


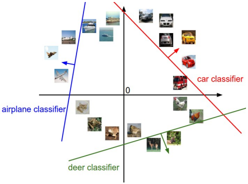
So far in the course

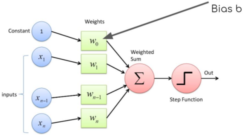
- Scoring function, loss function, gradient descent
- Artificial Neurons and Multi-Layered Perceptron
- Backpropagation

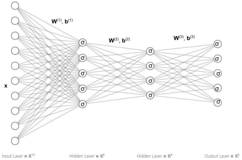












CNNs

- Neurons are similar to that of MLP

CNNs

- Neurons are similar to that of MLP
  - Perform a linear (dot product) operation and have a nonlinearity

CNNs

- Neurons are similar to that of MLP
  - Perform a linear (dot product) operation and have a nonlinearity
- Architecture will have a differentiable loss function, backpropagation is used

CNNs

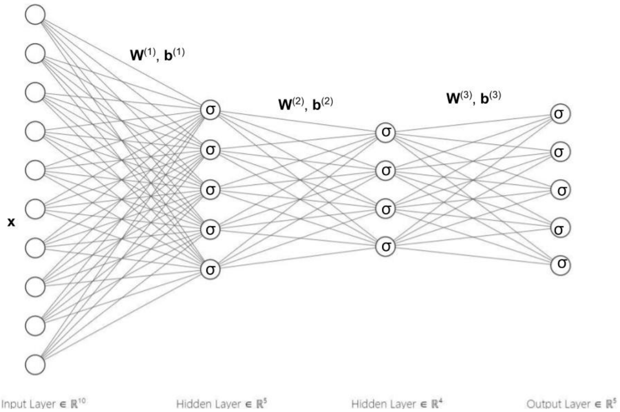
- Neurons are similar to that of MLP
  - Perform a linear (dot product) operation and have a nonlinearity
- Architecture will have a differentiable loss function, backpropagation is used
- Same tips and tricks apply

CNNs

- Neurons are similar to that of MLP
  - Perform a linear (dot product) operation and have a nonlinearity
- Architecture will have a differentiable loss function, backpropagation is used
- Same tips and tricks apply
- So, what changes?

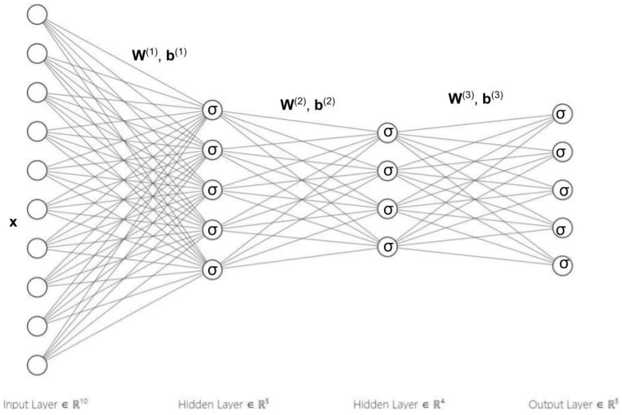
An MLP

- Input is a vector



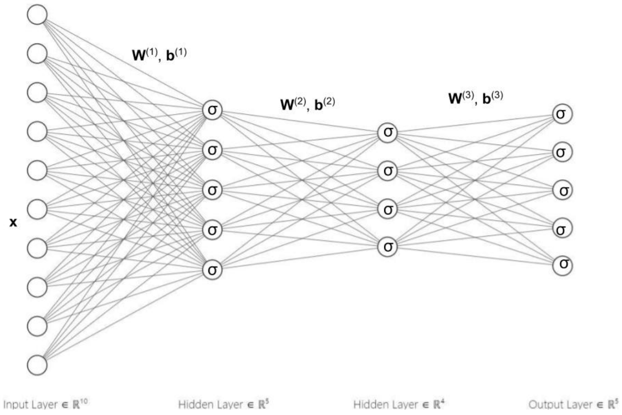
An MLP

- Input is a vector
- Series of densely connected hidden layers



An MLP

- Input is a vector
- Series of densely connected hidden layers
- Neurons in each layer are independent



An MLP for processing an image

- Say, we want to process a 200×200 RGB image

An MLP for processing an image

- Say, we want to process a 200×200 RGB image
- Vectorizing leads to $200 \times 200 \times 3 \rightarrow 120K$ neurons in the input layer

An MLP for processing an image

- Say, we want to process a 200×200 RGB image
- Vectorizing leads to $200 \times 200 \times 3 \rightarrow 120K$ neurons in the input layer
- A hidden layer of same size leads to $\approx 1.44e^{10}$ weights $\rightarrow \approx 58GB$

An MLP for processing an image

- Say, we want to process a 200×200 RGB image
- Vectorizing leads to $200 \times 200 \times 3 \rightarrow 120K$ neurons in the input layer
- A hidden layer of same size leads to $\approx 1.44e^{10}$ weights $\rightarrow \approx 58GB$
- Full connectivity blows the number of weights $\rightarrow$ hardware limits, overfitting, etc.

An MLP for processing an image

- Say, we want to process a 200×200 RGB image
- Vectorizing leads to $200 \times 200 \times 3 \rightarrow 120K$ neurons in the input layer
- A hidden layer of same size leads to $\approx 1.44e^{10}$ weights $\rightarrow \approx 58GB$
- Full connectivity blows the number of weights $\rightarrow$ hardware limits, overfitting, etc.
- Flattening removes the structure

Large Signals

- Have invariance in translation

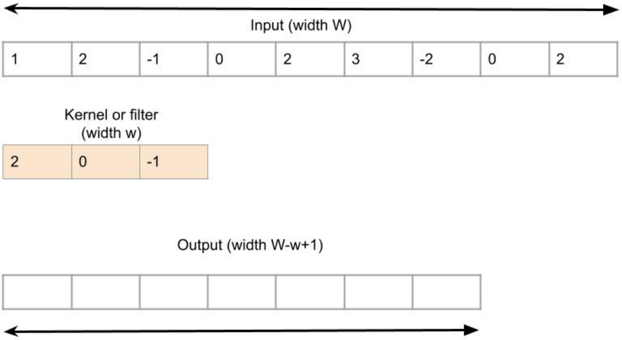
Large Signals

- Have invariance in translation
- Features may occur at different locations in the signal

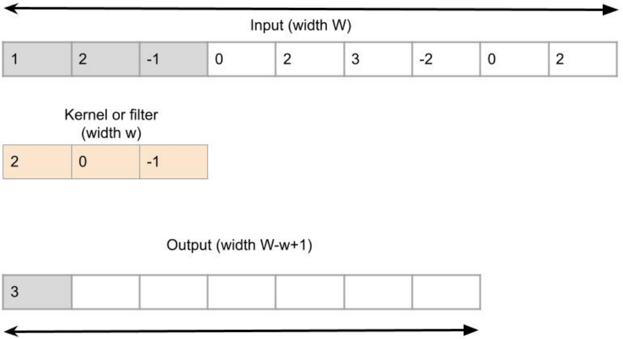
Large Signals

- Have invariance in translation
- Features may occur at different locations in the signal
- Convolution incorporates this idea: Applies same linear operation at all the locations and preserves the structure

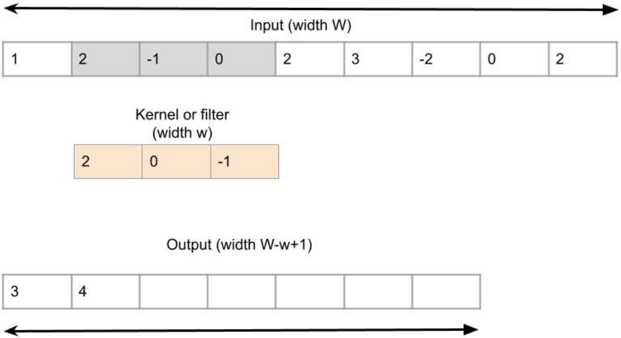
Convolution



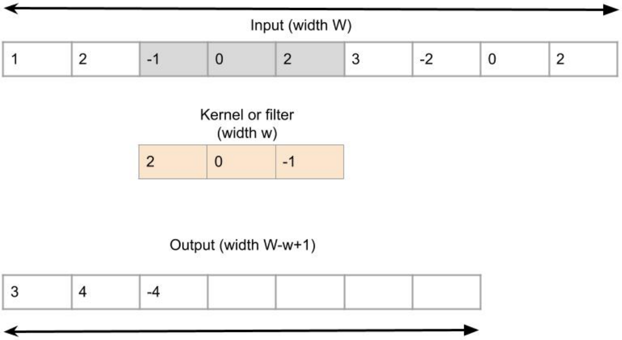
Convolution



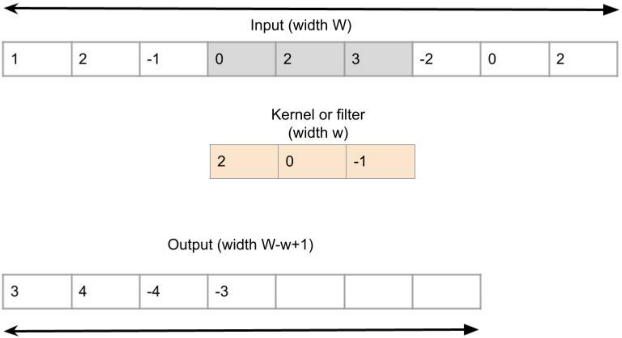
Convolution



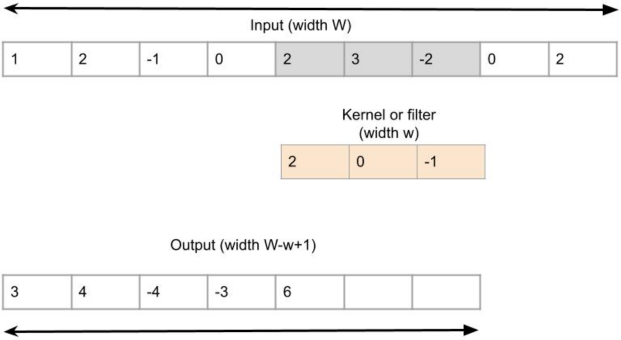
Convolution



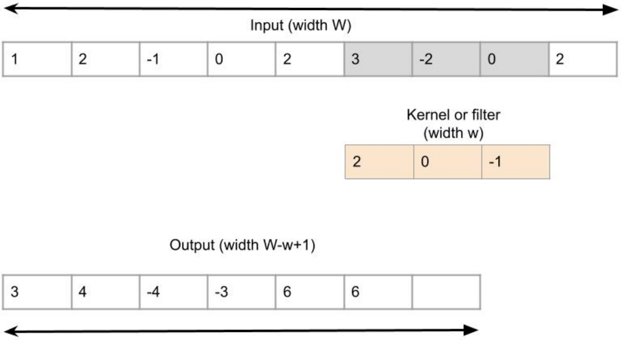
Convolution



Convolution



Convolution



Convolution

Input (width W)

1	2	-1	0	2	3	-2	0	2

Kernel or filter
(width w)

2	0	-1

Output (width W-w+1)

3	4	-4	-3	6	6	-6

Convolution

- Preserves the structure

Convolution

- Preserves the structure
  - if the i/p is a 2D tensor $\rightarrow$ o/p is also a 2D tensor

Convolution

- Preserves the structure
  - if the i/p is a 2D tensor $\rightarrow$ o/p is also a 2D tensor
  - There exist a relation between the locations of i/p and o/p values

Convolution

- Let $\mathbf{x} = (x_1, x_2, \dots x_W)$ is the input, $\mathbf{k} = (k_1, k_2, \dots k_w)$ is the kernel

Convolution

- Let $\mathbf{x} = (x_1, x_2, \ldots x_W)$ is the input, $\mathbf{k} = (k_1, k_2, \ldots k_w)$ is the kernel
- The result $(x \circledast k)$ of convolving $\mathbf{x}$ with $\mathbf{k}$ will be a 1D tensor of size $W - w + 1$

$$
\begin{aligned}
(x \circledast k)_i =& \sum_{j=1}^{w} x_{i-1+j} k_j \\
=& (x_i, \ldots x_{i+w-1}) \cdot \mathbf{k}
\end{aligned}
$$

Convolution

- Powerful feature extractor

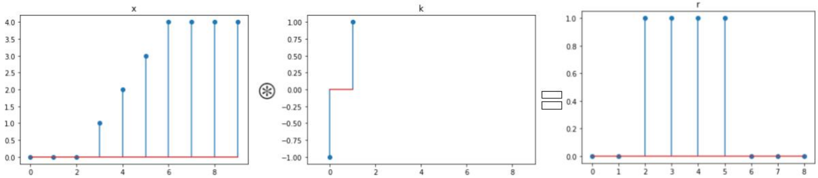
Convolution

- Powerful feature extractor
- For instance, it can perform differential operation and look for interesting patterns in the input

Convolution

- Powerful feature extractor
- For instance, it can perform differential operation and look for interesting patterns in the input
-

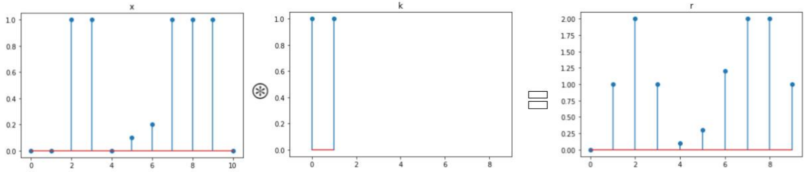
$$(0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,1,1,1,1,0,0,0)$$



Convolution

- Powerful feature extractor
- For instance, it can perform differential operation and look for interesting patterns in the input
-

$$(0, 0, 1, 1, 0, 0.1, 0.2, 1, 1, 1, 0) \circledast (1, 1) = (0, 1, 2, 1, 0.1, 0.3, 1.2, 2, 2, 1)$$



Convolution

- Naturally generalizes to multiple dimensions

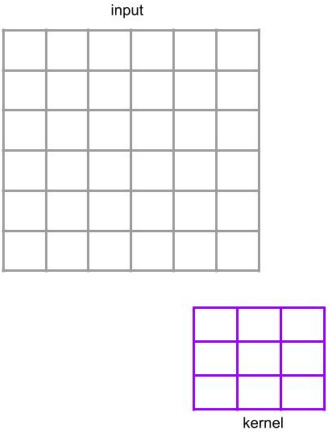
Convolution

- Naturally generalizes to multiple dimensions
- In their most usual form, CNNs process 3D tensors of size $C \times H \times W$ with kernels of size $C \times h \times w$ and result in 2D tensors of size $H - h + 1 \times W - w + 1$

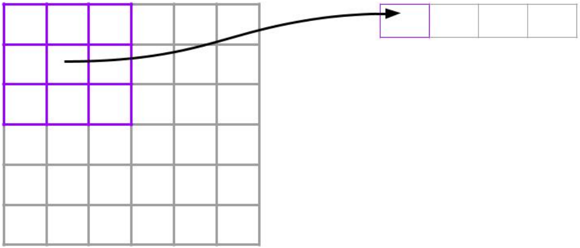
Convolution

- Naturally generalizes to multiple dimensions
- In their most usual form, CNNs process 3D tensors of size $C \times H \times W$ with kernels of size $C \times h \times w$ and result in 2D tensors of size $H - h + 1 \times W - w + 1$
- Note that we generally refer to these inputs as 2D signal (despite having C channels), because, they are referenced as vectors indexed by 2d locations without structure in the channel dimension

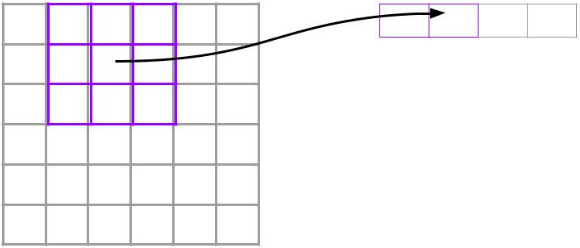
2D Convolution



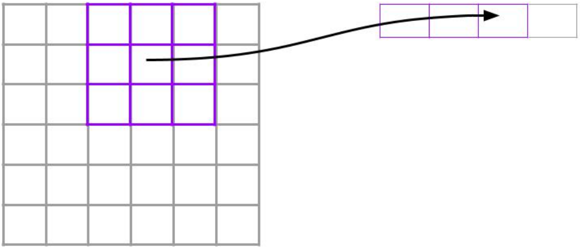
2D Convolution



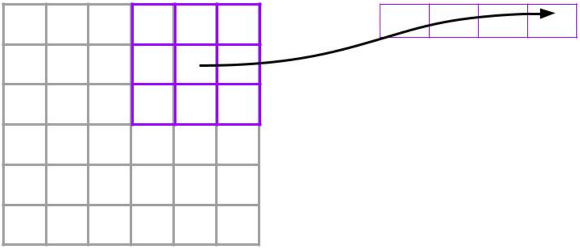
2D Convolution



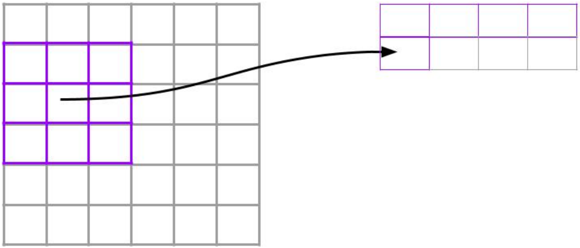
2D Convolution



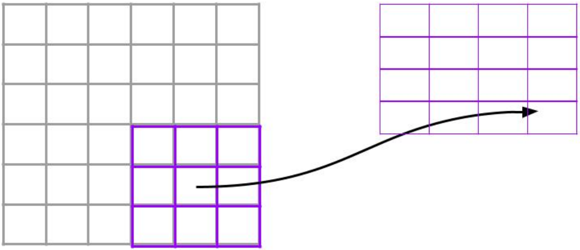
2D Convolution



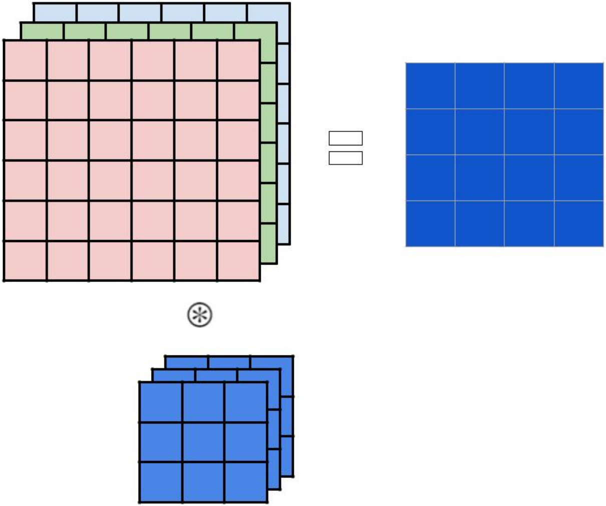
2D Convolution



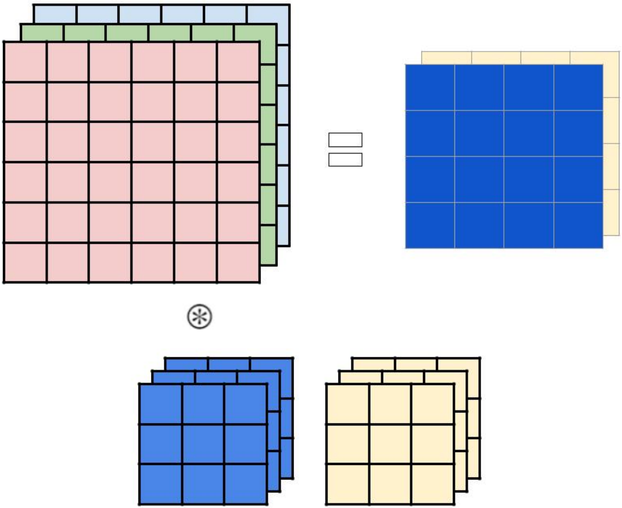
2D Convolution



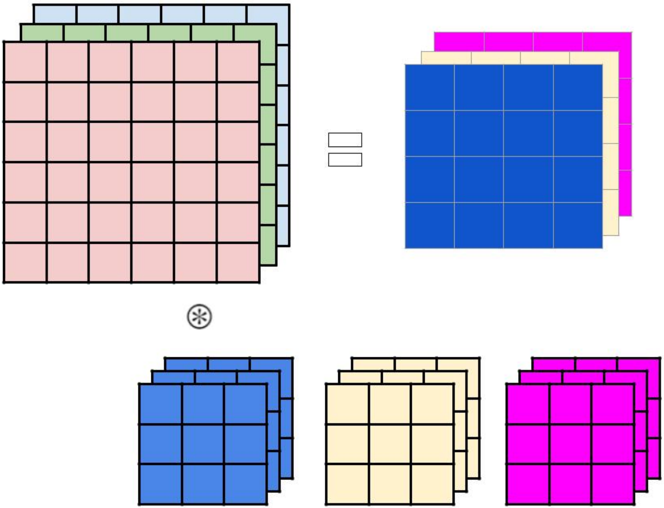
2D Convolution



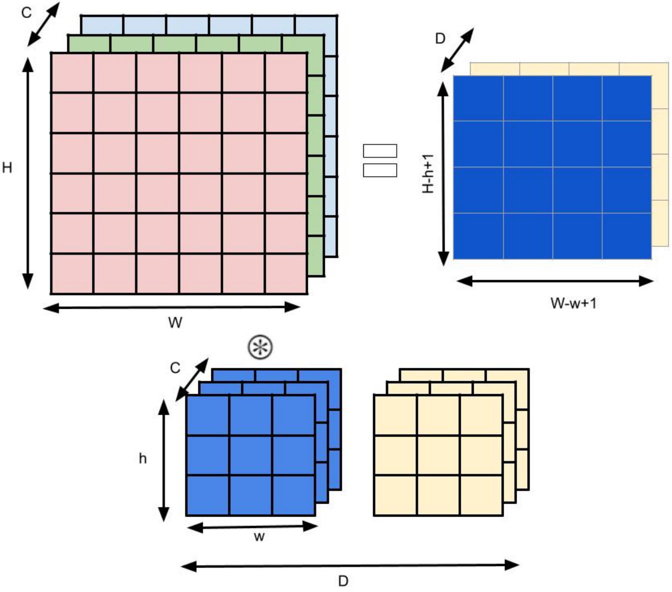
2D Convolution



2D Convolution



2D Convolution

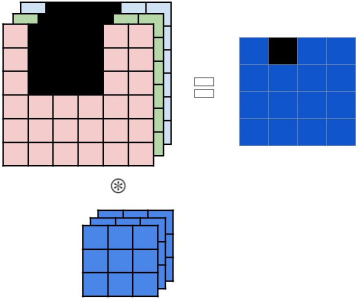


2D Convolution

- Kernel is not convolved in the channel dimension

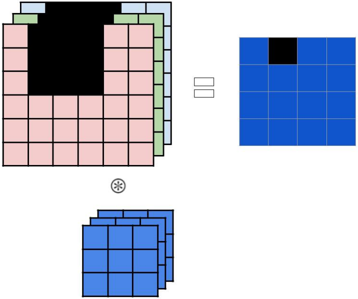
2D Convolution

- Kernel is not convolved in the channel dimension
- Another way to interpret convolution is that an affine function is applied on an input block of size $C \times h \times w$



2D Convolution

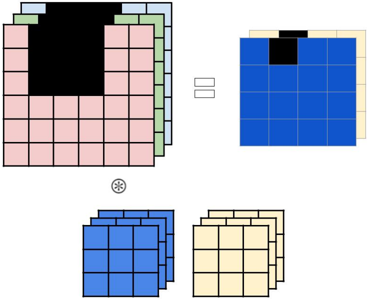
- Kernel is not convolved in the channel dimension
- Another way to interpret convolution is that an affine function is applied on an input block of size $C \times h \times w$



- Same affine function is applied on all such blocks in the input

2D Convolution

- Kernel is not convolved in the channel dimension
- Another way to interpret convolution is that an affine function is applied on an input block of size $C \times h \times w$ and results in output of size $D \times 1 \times 1$



- Same affine function is applied on all such blocks in the input

Convolution

- Preserves the input structure

Convolution

- Preserves the input structure
  - 1D signal outputs 1D signal, 2D signal outputs 2D signal

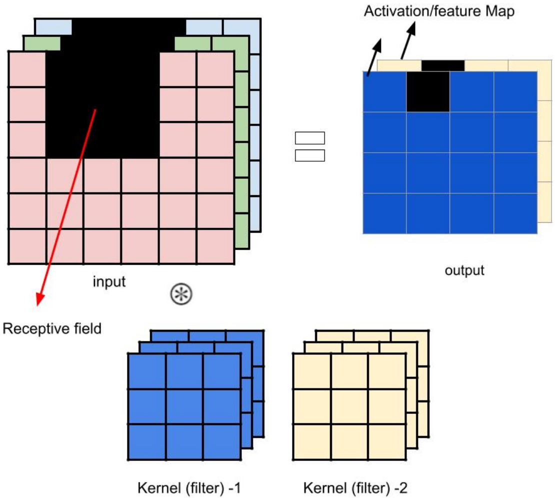
Convolution

- Preserves the input structure
  - 1D signal outputs 1D signal, 2D signal outputs 2D signal
  - Adjacent components in o/p are influenced by adjacent parts in the i/p

Convolution

- Preserves the input structure
  - 1D signal outputs 1D signal, 2D signal outputs 2D signal
  - Adjacent components in o/p are influenced by adjacent parts in the i/p
- If the channel dimension has a metric meaning (e.g. time) 3D convolution can be employed (e.g. frames in a video)

Terminology in Convolution



Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`

Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
- `weight` is $D \times C \times h \times w$ dimensional kernels

Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
- `weight` is $D \times C \times h \times w$ dimensional kernels
- `bias` D dimensional

Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
- `weight` is $D \times C \times h \times w$ dimensional kernels
- `bias` D dimensional
- `input` is $N \times C \times H \times W$ dimensional signal

Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
- `weight` is $D \times C \times h \times w$ dimensional kernels
- `bias` D dimensional
- `input` is $N \times C \times H \times W$ dimensional signal
- Output is $N \times D \times (H - h + 1) \times (W - w + 1)$ tensor

Convolution function in PyTorch

- `F.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
- `weight` is $D \times C \times h \times w$ dimensional kernels
- `bias` D dimensional
- `input` is $N \times C \times H \times W$ dimensional signal
- Output is $N \times D \times (H - h + 1) \times (W - w + 1)$ tensor
- Autograd compliant

Convolution function in PyTorch

```
input = torch.empty(128, 3, 20, 20).normal_()
weight = torch.empty(5, 3, 5, 5).normal_()
bias = torch.empty(5).normal_()
output = F.conv2d(input, weight, bias)
output.size()
torch.Size([128, 5, 16, 16])
```

Look/Access the filters

```
 weight[0,0]
tensor([[-0.6974, 0.1342, -0.2632, -0.4672, 0.1827],
[-0.1184, -0.2164, 0.2772, -0.1099, 0.0103],
[-0.8272, 0.3580, 0.2398, -0.5795,-0.9472],
[-1.1734, -0.1019, 0.7394, 0.3342, 0.1699],
[ 1.9271, 0.1250, 0.4222, 0.2014, 1.1100]])
```

Conv layer in PyTorch

- `Class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)`

Conv layer in PyTorch

- `Class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)`
- `kernel_size` can be either a pair `(h, w)` or a single value k interpreted as `(k, k)`.

Conv layer in PyTorch

- `Class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)`
- `kernel_size` can be either a pair `(h, w)` or a single value k interpreted as `(k, k)`.
- Encloses the convolution as a module

Conv layer in PyTorch

- `Class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)`
- `kernel_size` can be either a pair `(h, w)` or a single value k interpreted as `(k, k)`.
- Encloses the convolution as a module
- Initializes the kernel parameters and biases as random

Conv layer in PyTorch

```
f = nn.Conv2d(in_channels = 3, out_channels = 5,
kernel_size = (2, 3))
for n, p in f.named_parameters():
...print(n, p.size())

>>weight torch.Size([5, 3, 2, 3])
>>bias torch.Size([5])
```

Conv layer in PyTorch

```
f = nn.Conv2d(in_channels = 3, out_channels = 5,
kernel_size = (2, 3))
for n, p in f.named_parameters():
...print(n, p.size())

>>weight torch.Size([5, 3, 2, 3])
>>bias torch.Size([5])

input = torch.empty(128, 3, 28, 28).normal_()
output = f(input)
output.size()
>>torch.Size([128, 5, 27, 26])
```

Padding in Convolution

- Adds zeros around the input

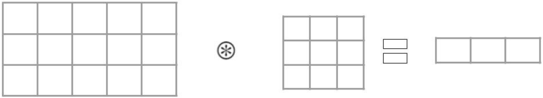
Padding in Convolution

- Adds zeros around the input
- Takes cares of size reduction after convolution

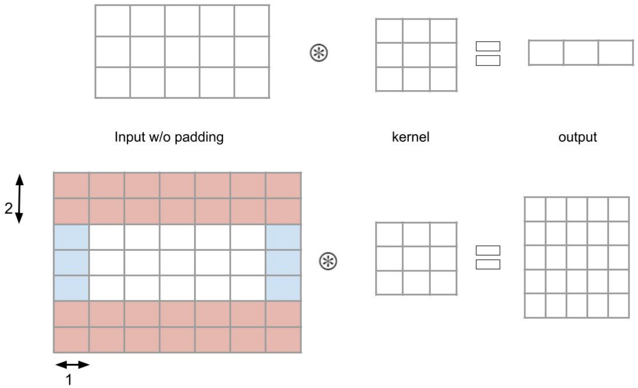
Padding in Convolution

- Adds zeros around the input
- Takes cares of size reduction after convolution
- Instead of zeros, one may pad with signal values at the edges

Padding in Convolution



Padding in Convolution



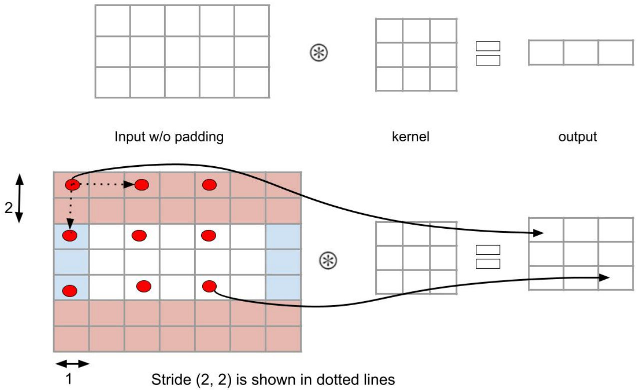
Stride in Convolution

- Specifies the step size taken while performing convolution

Stride in Convolution

- Specifies the step size taken while performing convolution
- Default value is 1, i.e., move the kernel across the signal densely (without skipping)

Padding and Stride in Convolution



Stride (2, 2) is shown in dotted lines

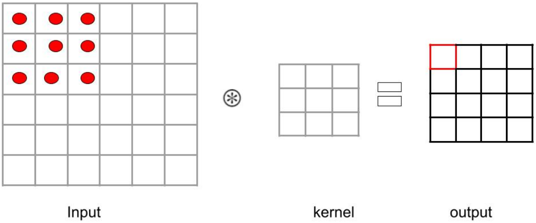
Dilation in Convolution

- Manipulates the size of the kernel via expanding its size without adding weights.

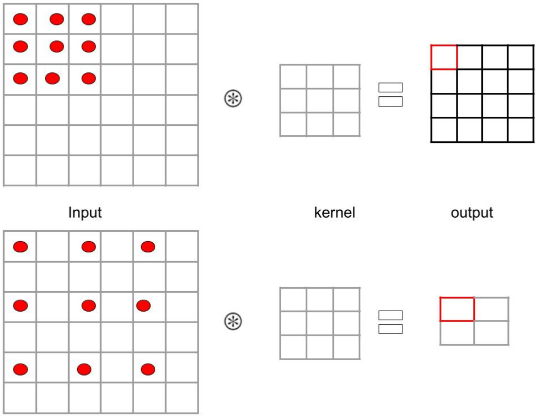
Dilation in Convolution

- Manipulates the size of the kernel via expanding its size without adding weights.
- In other words, it inserts 0s in between the kernel values

Without Dilation



Dilation (2, 2)



Dilation

- Expands the kernel by adding rows and columns of zeros

Dilation

- Expands the kernel by adding rows and columns of zeros
- Default value for dilation is 1, i.e., no zeros placed

Dilation

- Expands the kernel by adding rows and columns of zeros
- Default value for dilation is 1, i.e., no zeros placed
- Any higher value of dilation makes the kernel sparse

Dilation

- Expands the kernel by adding rows and columns of zeros
- Default value for dilation is 1, i.e., no zeros placed
- Any higher value of dilation makes the kernel sparse
- Dilation increases the receptive field

Dilation

- Expands the kernel by adding rows and columns of zeros
- Default value for dilation is 1, i.e., no zeros placed
- Any higher value of dilation makes the kernel sparse
- Dilation increases the receptive field
- It is referred to as 'atrous' convolution

Pooling

Pooling

- Groups multiple activations and replaces by a representative one

Pooling

- Groups multiple activations and replaces by a representative one
- Reduces the dimensionality of the signal progressively $\rightarrow$ considers non-overlapping stride

Pooling

- Groups multiple activations and replaces by a representative one
- Reduces the dimensionality of the signal progressively $\rightarrow$ considers non-overlapping stride
- Also called sub-sampling layer

Pooling

- Groups multiple activations and replaces by a representative one
- Reduces the dimensionality of the signal progressively $\rightarrow$ considers non-overlapping stride
- Also called sub-sampling layer
- Generally found between two convolution layers (and parameter free)

Max Pooling

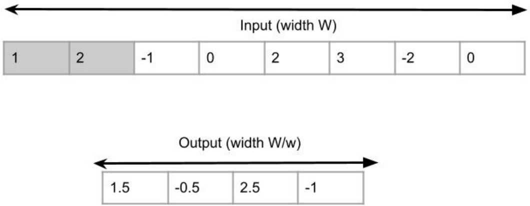
- Standard in CNNs

Max Pooling

- Standard in CNNs
- Computes maximum value over a non-overlapping blocks in the input

Average Pooling

- Computes the average of the receptive field



Pooling in 2D

- Same as 1D, but the receptive field is 2D and non-overlapping

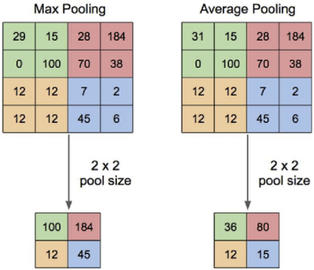


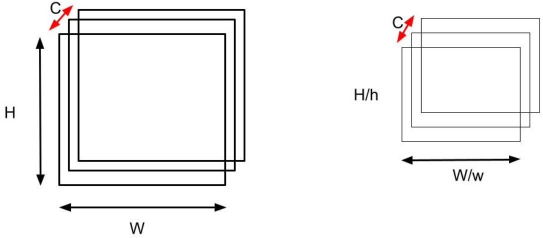
Figure credits: Preston Hoang and Quora

Pooling in 2D

- Contrary to Convolution, Pooling applies channel wise

Pooling in 2D

- Contrary to Convolution, Pooling applies channel wise
- No reduction in number of channels, only spatial size reduction



Pooling provides weak invariance

- Operation is invariant to any permutation within the block

Pooling provides weak invariance

- Operation is invariant to any permutation within the block
- Withstands deformations caused by local translations

Max_Pooling PyTorch

```
F.max_pool2d(input, kernel_size, stride=None, padding=0,
dilation=1, ceil_mode=False, return_indices=False)
```

- Applies max pooling on each of the channels separately

Max_Pooling PyTorch

```
F.max_pool2d(input, kernel_size, stride=None, padding=0,
dilation=1, ceil_mode=False, return_indices=False)
```

- Applies max pooling on each of the channels separately
- `input` is $N \times C \times H \times W$ tensor

Max_Pooling PyTorch

```
F.max_pool2d(input, kernel_size, stride=None, padding=0,
dilation=1, ceil_mode=False, return_indices=False)
```

- Applies max pooling on each of the channels separately
- `input` is $N \times C \times H \times W$ tensor
- `kernel_size` is (h, w) or k

Max_Pooling PyTorch

```
F.max_pool2d(input, kernel_size, stride=None, padding=0,
dilation=1, ceil_mode=False, return_indices=False)
```

- Applies max pooling on each of the channels separately
- `input` is $N \times C \times H \times W$ tensor
- `kernel_size` is (h, w) or k
- Result would be a tensor of size $N \times C \times \lfloor H/h \rfloor \times \lfloor W/w \rfloor$

Pooling in PyTorch

- Default stride is the kernel size (for convolution, it is 1)

Pooling in PyTorch

- Default stride is the kernel size (for convolution, it is 1)
- But, it can be modulated if required

Pooling in PyTorch

- Default stride is the kernel size (for convolution, it is 1)
- But, it can be modulated if required
- Default padding is zero

Pooling Layer in PyTorch

```
class torch.nn.MaxPool2d(kernel_size, stride=None,
padding=0, dilation=1, return_indices=False,
ceil_mode=False)
```

Putting it all together

Architecture of a simple CNN

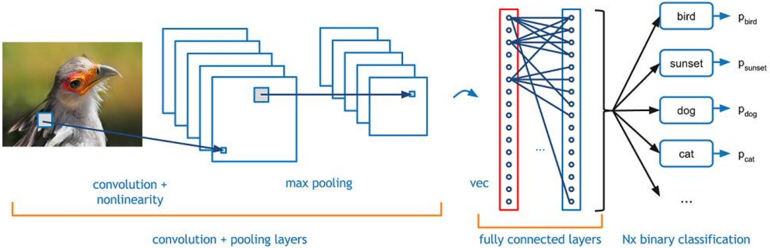
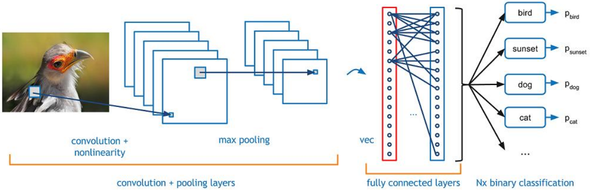


Figure credits: Adit Deshpande

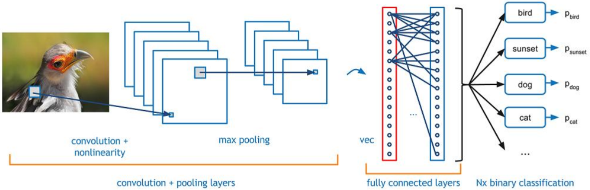
Architecture of a simple CNN



- Initially Conv layer with nonlinearity

Figure credits: Adit Deshpande

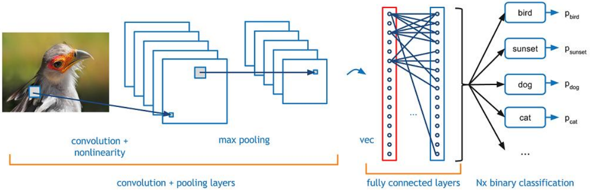
Architecture of a simple CNN



- Initially Conv layer with nonlinearity
- Followed by a few Conv + Nonlinearity layers

Figure credits: Adit Deshpande

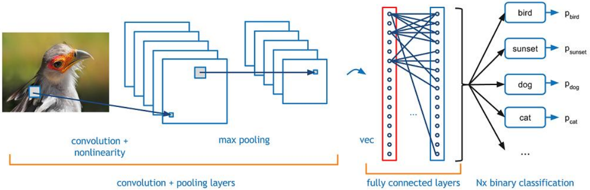
Architecture of a simple CNN



- Initially Conv layer with nonlinearity
- Followed by a few Conv + Nonlinearity layers
- Have Pooling layers in between Conv layers → reduce the feature map size sufficiently

Figure credits: Adit Deshpande

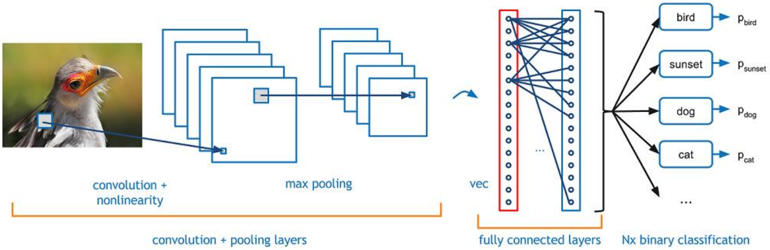
Architecture of a simple CNN



- Initially Conv layer with nonlinearity
- Followed by a few Conv + Nonlinearity layers
- Have Pooling layers in between Conv layers $\rightarrow$ reduce the feature map size sufficiently
- Vectorize and and fully connected layers

Figure credits: Adit Deshpande

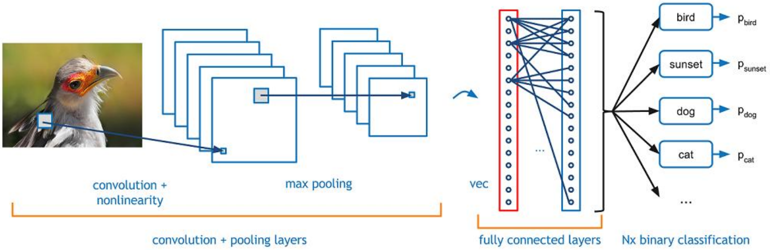
Architecture of a simple CNN



INPUT -> [[CONV -> RELU] *N -> POOL]*M -> [FC->RELU]*K -> FC

Figure credits: Adit Deshpande

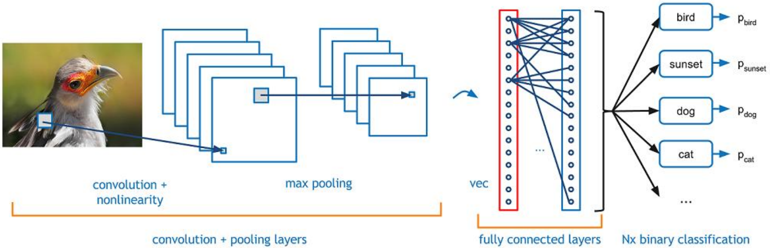
Architecture of a simple CNN



INPUT -> [[CONV -> RELU] *N -> POOL]*M -> [FC->RELU]*K -> FC

Figure credits: Adit Deshpande

Architecture of a simple CNN



INPUT -> [[CONV -> RELU] *N -> POOL]*M -> [FC->RELU]*K -> FC

Figure credits: Adit Deshpande

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`			

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$		

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2+1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`			

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`			

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2+1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$		

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2 + 1)$ $= 51264$	

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2+1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2+1)$ $= 51264$	$64.32.4^2.5^2$ $= 819200$

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1) = 832$	$32.24^2.5^2 = 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2 + 1) = 51264$	$64.32.4^2.5^2 = 819200$
$64 \times 4 \times 4$ `F.max_pool2d(., kernel_size=2)`	$64 \times 2 \times 2$	0	0

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2 + 1)$ $= 51264$	$64.32.4^2.5^2$ $= 819200$
$64 \times 4 \times 4$ `F.max_pool2d(., kernel_size=2)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ / `F.relu(.)`	$64 \times 2 \times 2$	0	0

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2 + 1) = 832$	$32.24^2.5^2 = 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2 + 1) = 51264$	$64.32.4^2.5^2 = 819200$
$64 \times 4 \times 4$ `F.max_pool2d(., kernel_size=2)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ / `F.relu(.)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ `x.view(-1,256)`	256	0	0
256 `nn.Linear(256,200)`	200		

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2+1)$ $= 832$	$32.24^2.5^2$ $= 460800$
$32 \times 24 \times 24$ `F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2+1)$ $= 51264$	$64.32.4^2.5^2$ $= 819200$
$64 \times 4 \times 4$ `F.max_pool2d(., kernel_size=2)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ / `F.relu(.)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ `x.view(-1,256)`	256	0	0
256 `nn.Linear(256,200)`	200	200(256+1)=51400	200.256=51200

Case study: LeNet-like architecture

input size/ layer information	output size	# parameters	# products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)`	$32 \times 24 \times 24$	$32.(5^2+1)$ $= 832$	$32.24^2.5^2$ $= 460800$
`F.max_pool2d(., kernel_size=3)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ / `F.relu(.)`	$32 \times 8 \times 8$	0	0
$32 \times 8 \times 8$ `nn.conv2d(32, 64, kernel_size=5)`	$64 \times 4 \times 4$	$64.(32.5^2+1)$ $= 51264$	$64.32.4^2.5^2$ $= 819200$
$64 \times 4 \times 4$ `F.max_pool2d(., kernel_size=2)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ / `F.relu(.)`	$64 \times 2 \times 2$	0	0
$64 \times 2 \times 2$ `x.view(-1,256)`	256	0	0
256 `nn.Linear(256,200)`	0 200	0 200(256+1)=51400	0 200.256=51200
200 / `F.relu(.)`	200	0	0
200 `nn.Linear(200,10)`	0 10	0 10(200+1)=2010	0 10.200=2000

Recent architectures are more sophisticated

- Note that LeNet is a classical architecture and does not reflect the recent CNNs in complexity

Recent architectures are more sophisticated

- Note that LeNet is a classical architecture and does not reflect the recent CNNs in complexity
- Recent CNN architectures are far more sophisticated [Contents of the next lecture(s)]
  - More depth
  - Regularizers to handle the depth