HANDS-ON AI I

Convolutional Neural Networks

Andreas Schörgenhumer
Institute for Machine Learning

JOHANNES KEPLER UNIVERSITY LINZ

Institute for Machine Learning

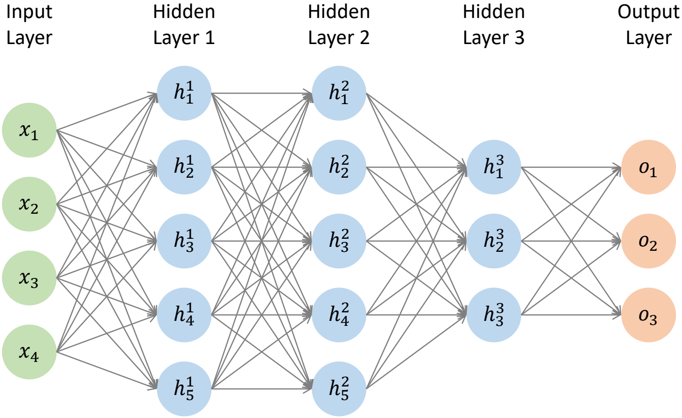
Copyright Statement

Content of Unit 6

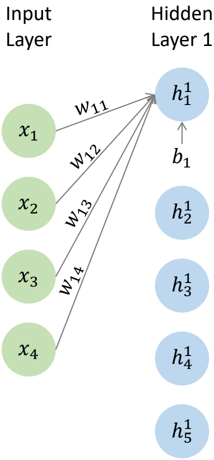
- Short recap on neural networks.
- Introduction to Convolutional Neural Networks (CNNs):
  - Image data properties, receptive field
  - Convolution, kernels
  - Building blocks and structure of CNNs

Recap: Neural Network Components

- Input layer, hidden layers, output layer



Recap: Weight Matrices



- Output h_1^1 of first node/neuron (layer 1):

$$z = b_1 + \sum_{i=1}^{4} w_{1i} x_i \qquad h_1^1 = f(z)$$

  with some activation function f

- Output of entire layer 1:

$$\underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}}_{\boldsymbol{b}^1} + \underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{bmatrix}}_{\boldsymbol{W}^1} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\boldsymbol{x}} \overset{f}{\Rightarrow} \underbrace{\begin{bmatrix} h_1^1 \\ h_2^1 \\ h_3^1 \\ h_4^1 \\ h_5^1 \end{bmatrix}}_{\boldsymbol{h}^1}$$

CONVOLUTIONAL NEURAL NETWORKS (CNNS)

Neural Nets for Image Recognition

- ImageNet Large Scale Visual Recognition Competition (ILSVRC):
  - 1.2M images, 1,000 different classes.
  - Yearly challenge to find the “State of the Art” in image recognition.

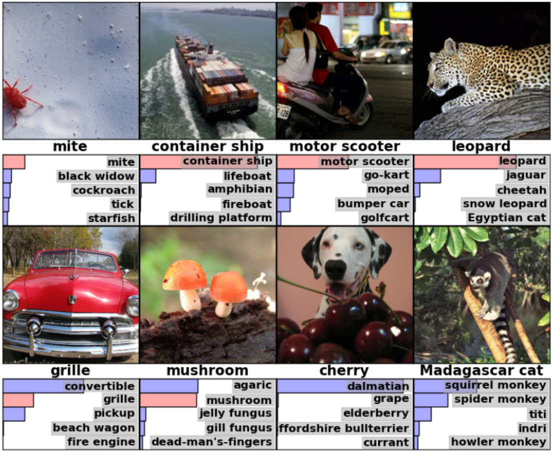
Neural Nets for Image Recognition

- ImageNet Large Scale Visual Recognition Competition (ILSVRC):
  - 1.2M images, 1,000 different classes.
  - Yearly challenge to find the “State of the Art” in image recognition.
  - ILSVRC 2012: Won by the only CNN-based solution.
  - ILSVRC 2013: Best 5 participants were all CNNs (9 of the top 10 were CNNs).
  - ILSVRC 2014: Everyone uses CNNs.

Neural Nets for Image Recognition

- ImageNet Large Scale Visual Recognition Competition (ILSVRC):
  - 1.2M images, 1,000 different classes.
  - Yearly challenge to find the “State of the Art” in image recognition.
  - ILSVRC 2012: Won by the only CNN-based solution.
  - ILSVRC 2013: Best 5 participants were all CNNs (9 of the top 10 were CNNs).
  - ILSVRC 2014: Everyone uses CNNs.
- CNNs are useful whenever there is “local structure” in the data:
  - Pixel data
  - Audio data
  - Voxel data
  - …

ILSVRC 2012: CNNs Classify Images Far Better Than Any Other Methods



[Source: Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems (NIPS). 2012.]

AlexNet

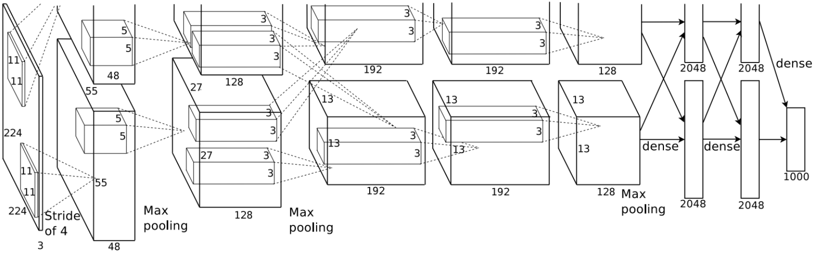


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

- Won ILSVRC 2012 by a landslide
- After Krizhevsky et al. won ILSVRC 2012, “everyone” started using CNNs for image tasks.

[Source: Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems (NIPS). 2012.]

Properties of Image(-like) Data

- Images are extremely **high-dimensional**.
  - Example: 250 x 250 pixels · 3 color channels = 187.5k input dimensions

Properties of Image(-like) Data

- Images are extremely **high-dimensional**.
  - Example: 250 x 250 pixels · 3 color channels = 187.5k input dimensions
- Pixels that are near each other are **highly correlated**.

Properties of Image(-like) Data

- Images are extremely **high-dimensional**.
  - Example: 250 x 250 pixels · 3 color channels = 187.5k input dimensions
- Pixels that are near each other are **highly correlated**.
- Same basic patches (e.g., edges, corners) appear on all positions of the image.

Properties of Image(-like) Data

- Images are extremely **high-dimensional**.
  - Example: 250 x 250 pixels · 3 color channels = 187.5k input dimensions
- Pixels that are near each other are **highly correlated**.
- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

MAIN CONCEPTS OF CNNS

Receptive Field

- Pixels that are near each other are **highly correlated**.
- Same basic patches (e.g., edges, corners) appear on all positions of the image.

Receptive Field

- Pixels that are near each other are **highly correlated**.
- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- We can detect those basic patches by only viewing a small part of the image.

$\rightarrow$ We can use a network with a small receptive field!

Receptive Field

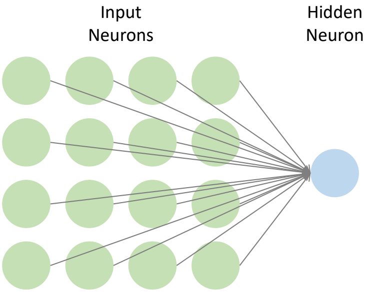
- Pixels that are near each other are **highly correlated**.
- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- We can detect those basic patches by only viewing a small part of the image.

$\rightarrow$ We can use a network with a small receptive field!

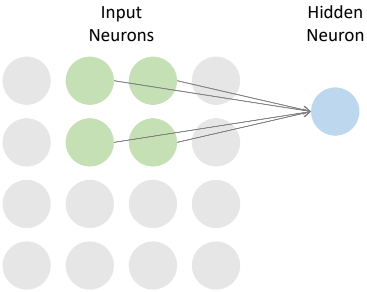
- Receptive field: Connect network to patch of image using weight matrix (=**kernel** or **filter**)

Receptive Field

- **FNN**: In a feed-forward neural network, each hidden neuron is connected to all neurons of the previous layer.
- **CNN**: In a convolutional neural network, a hidden neuron is only connected to a few neurons in the previous layer.



FNN



CNN

What Is a Convolution?

- Mathematical operations on two functions:

$$(h * k)(a, b) = \sum_i \sum_j h(a + i, b + j) k(i, j)$$

- Technically, it is a cross-correlation or sliding dot product (the visual example when talking about weight sharing later on should convey this more clearly).[1]
- By convention, we refer to it as convolution.

[1] Also, the formula only shows the 2D case. While this is the typical scenario (and in this course, we keep it that way), we are not limited to 2D.

What Is a Convolution?

- In a neural network, the convolution is performed on the input (image) matrix.

What Is a Convolution?

- In a neural network, the convolution is performed on the input (image) matrix.
- Applying the kernel $\boldsymbol{W}$ to all image positions (**weight sharing**) can be viewed as convolution.

What Is a Convolution?

- In a neural network, the convolution is performed on the input (image) matrix.
- Applying the kernel $\boldsymbol{W}$ to all image positions (**weight sharing**) can be viewed as convolution.
- We get an output value for each time we apply the kernel.

What Is a Convolution?

- In a neural network, the convolution is performed on the input (image) matrix.
- Applying the kernel $\boldsymbol{W}$ to all image positions (**weight sharing**) can be viewed as convolution.
- We get an output value for each time we apply the kernel.
- We apply an activation function to those outputs afterwards (usually ReLU).

What Is a Convolution?

- In a neural network, the convolution is performed on the input (image) matrix.
- Applying the kernel $\boldsymbol{W}$ to all image positions (**weight sharing**) can be viewed as convolution.
- We get an output value for each time we apply the kernel.
- We apply an activation function to those outputs afterwards (usually ReLU).
- Applying the kernel and the activation function is one layer in a **Convolutional Neural Network (CNN)**.

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

$\rightarrow$ We can reuse the receptive field at all positions of the image to produce the new output (=feature/activation map).

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

$\rightarrow$ We can reuse the receptive field at all positions of the image to produce the new output (=feature/activation map).

- Reusing the kernel weight matrix is called **weight sharing**.

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

$\rightarrow$ We can reuse the receptive field at all positions of the image to produce the new output (=feature/activation map).

- Reusing the kernel weight matrix is called **weight sharing**.
  - We apply our kernel to all image positions while keeping the weights the same.

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

→ We can reuse the receptive field at all positions of the image to produce the new output (=feature/activation map).

- Reusing the kernel weight matrix is called **weight sharing**.
  - We apply our kernel to all image positions while keeping the weights the same.
  - This **significantly reduces** the number of model **parameters**.

Weight Sharing

- Same basic patches (e.g., edges, corners) appear on all positions of the image.
- Often, **invariances to certain variations** are desired (e.g., translation invariance).

→ We can reuse the receptive field at all positions of the image to produce the new output (=feature/activation map).

- Reusing the kernel weight matrix is called **weight sharing**.
  - We apply our kernel to all image positions while keeping the weights the same.
  - This **significantly reduces** the number of model **parameters**.
  - Interactive kernel demo: https://setosa.io/ev/image-kernels/

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

$$
\begin{aligned}
&\, 0 \cdot 0 + 0 \cdot 0 + 0.5 \cdot 0 \\
&+ 0 \cdot 0 + 0.5 \cdot 1.0 + 0 \cdot 0 \\
&+ 0 \cdot 0 + 0 \cdot 0 + 0.5 \cdot 0 \\
&= 0.5
\end{aligned}
$$

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

$$
\begin{aligned}
& 0 \cdot 0 \; + \; 0 \cdot 0 \; + \; 0.5 \cdot 0 \\
+ & \, 0 \cdot 1.0 + 0.5 \cdot 0 + \; 0 \cdot 0 \\
+ & \; 0 \cdot 0 \; + \; 0 \cdot 0 \; + 0.5 \cdot 0.4 \\
= & \, 0.2
\end{aligned}
$$

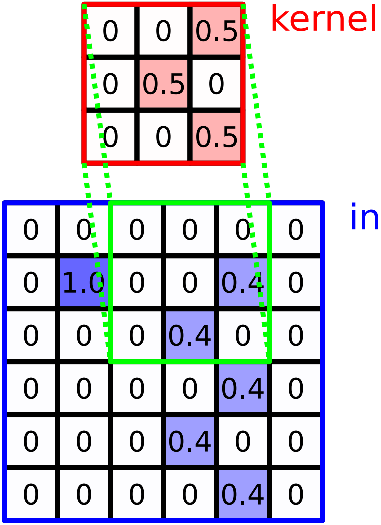
input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

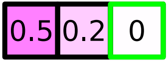
0.5	0.2

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)



$$
\begin{aligned}
&0 \cdot 0 + \ \ 0 \cdot 0 \ \ + 0.5 \cdot 0 \\
&+ 0 \cdot 0 + 0.5 \cdot 0 + 0 \cdot 0.4 \\
&+ 0 \cdot 0 + 0 \cdot 0.4 + 0.5 \cdot 0 \\
&= 0
\end{aligned}
$$

output



Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

$$
\begin{aligned}
&\quad\; 0 \cdot 0 + 0 \cdot 0 + 0.5 \cdot 0 \\
&+ 0 \cdot 0 + 0.5 \cdot 0.4 + 0 \cdot 0 \\
&+ 0 \cdot 0.4 + 0 \cdot 0 + 0.5 \cdot 0 \\
&= 0.2
\end{aligned}
$$

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5	0.2	0	0.2

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

$$
\begin{aligned}
& 0 \cdot 0 + 0 \cdot 1.0 + 0.5 \cdot 0 \\
+ & 0 \cdot 0 + 0.5 \cdot 0 + 0 \cdot 0 \\
+ & 0 \cdot 0 + 0 \cdot 0 + 0.5 \cdot 0 \\
= & 0
\end{aligned}
$$

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5	0.2	0	0.2
0			

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

$$\begin{aligned} & 0 \cdot 1.0 + \ 0 \cdot 0 \ + 0.5 \cdot 0 \\ + & \ 0 \cdot 0 \ + 0.5 \cdot 0 + 0 \cdot 0.4 \\ + & \ 0 \cdot 0 \ + \ 0 \cdot 0 \ + 0.5 \cdot 0 \\ = & \ 0 \end{aligned}$$

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5	0.2	0	0.2
0	0		

Weight Sharing (k: 3×3, i: 6×6, o: 4×4)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5	0.2	0	0.2
0	0	0.6	0
0	0.4	0	0.2
0	0	0.6	0

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?
- We can pick **existing kernels**. Examples:
  - **Sobel filter**/operator for detecting edges
  - **Gaussian blur filter** for blurring

Sobel Filter: Vertical Edge Detection

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

Sobel Filter: Horizontal Edge Detection

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

Gaussian Blur Filter

$$\rightarrow \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \rightarrow$$

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?
- We can pick **existing kernels**. Examples:
  - **Sobel filter**/operator for detecting edges
  - **Gaussian blur filter** for blurring

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?
- We can pick **existing kernels**. Examples:
  - **Sobel filter**/operator for detecting edges
  - **Gaussian blur filter** for blurring
- In convolutional neural networks, however, we actually want to **learn** the kernels ourselves!

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?
- We can pick **existing kernels**. Examples:
  - **Sobel filter**/operator for detecting edges
  - **Gaussian blur filter** for blurring
- In convolutional neural networks, however, we actually want to **learn** the kernels ourselves!
- The kernels are just small weight matrices $\boldsymbol{W}$ which we can learn the same way as in regular neural networks.

Kernels

- We saw how the convolution with kernels can be applied. But how do we know which kernels/kernel values to choose?
- We can pick **existing kernels**. Examples:
  - **Sobel filter**/operator for detecting edges
  - **Gaussian blur filter** for blurring
- In convolutional neural networks, however, we actually want to **learn** the kernels ourselves!
- The kernels are just small weight matrices $\boldsymbol{W}$ which we can learn the same way as in regular neural networks.
- Hopefully, we learn useful kernels, e.g., to detect edges, corners, color patches, body parts, ...

PADDING

Padding

- When applying a convolution with a kernel of size > 1, the output will be smaller than the input (see examples before).

Padding

- When applying a convolution with a kernel of size > 1, the output will be smaller than the input (see examples before).
- **Padding** is used to keep the input and output size the same:
  - **Zero-Padding**: Add zeros at borders of input
  - **Repeat-Padding**: Duplicate the border values
  - Other padding methods: Mean, weighted sum, ...

Padding

- When applying a convolution with a kernel of size > 1, the output will be smaller than the input (see examples before).
- **Padding** is used to keep the input and output size the same:
  - **Zero-Padding**: Add zeros at borders of input
  - **Repeat-Padding**: Duplicate the border values
  - Other padding methods: Mean, weighted sum, ...
- Can be applied to input image or in between layers to keep the original input size

Zero-Padding (k: 3×3, i: 4×4, o: 2×2)

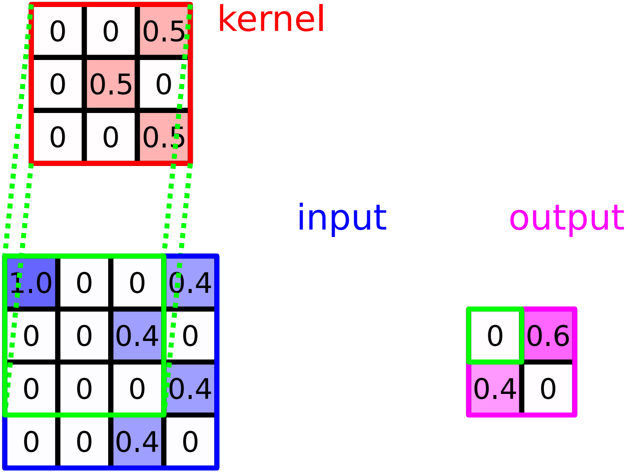
kernel

0	0	0.5
0	0.5	0
0	0	0.5

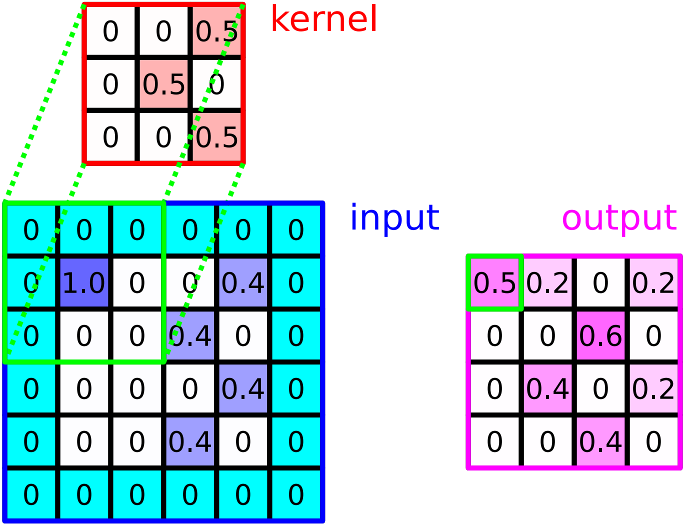
input

1.0	0	0	0.4
0	0	0.4	0
0	0	0	0.4
0	0	0.4	0

Zero-Padding (k: 3×3, i: 4×4, o: 2×2)



Zero-Padding (k: 3×3, i: $4+1 \times 4+1$, o: 4×4)



STRIDING

Striding

- **Striding** controls how much the kernels/filters are moved.

Striding

- **Striding** controls how much the kernels/filters are moved.
- The smaller the stride, the more the receptive filters overlap.

Striding

- **Striding** controls how much the kernels/filters are moved.
- The smaller the stride, the more the receptive filters overlap.
- Striding is one way of **downsampling** images.

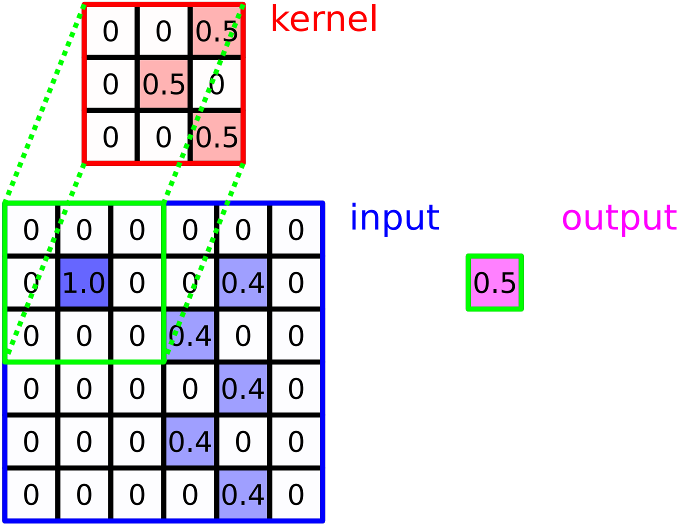
Striding

- **Striding** controls how much the kernels/filters are moved.
- The smaller the stride, the more the receptive filters overlap.
- Striding is one way of **downsampling** images.
- A stride > 1 will lead to loss of information (no problem if we keep the essential information) but will also reduce computational load and memory requirements.

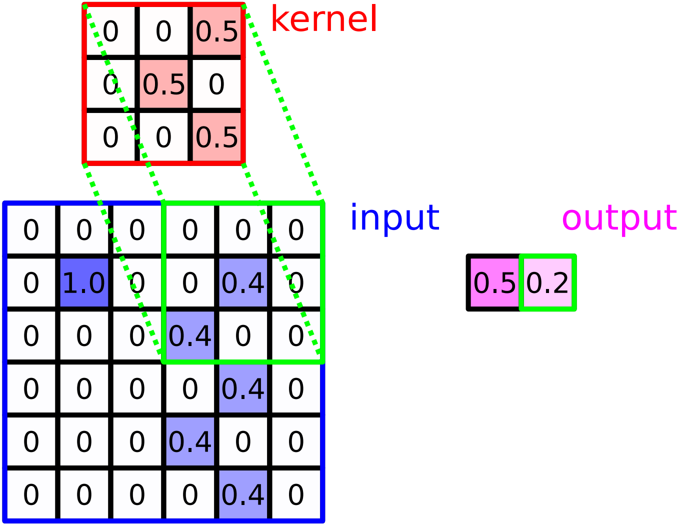
Striding

- **Striding** controls how much the kernels/filters are moved.
- The smaller the stride, the more the receptive filters overlap.
- Striding is one way of **downsampling** images.
- A stride > 1 will lead to loss of information (no problem if we keep the essential information) but will also reduce computational load and memory requirements.
- A stride > 1 will increase the receptive field through depth of network.

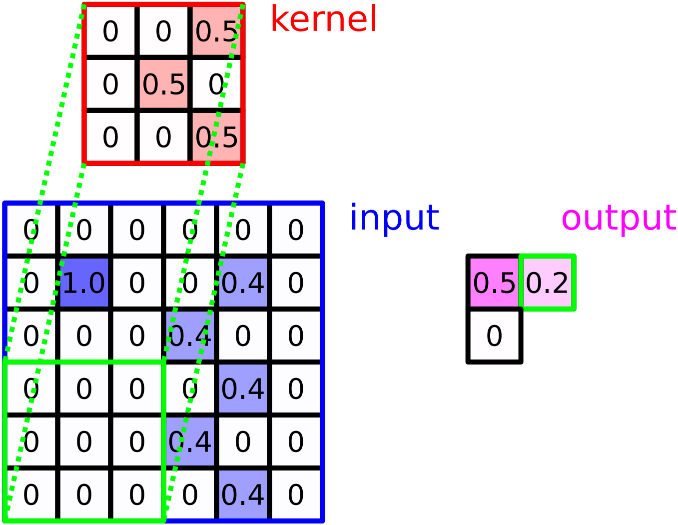
Striding: stride=3 (k: 3×3, i: 6×6, o: 2×2)



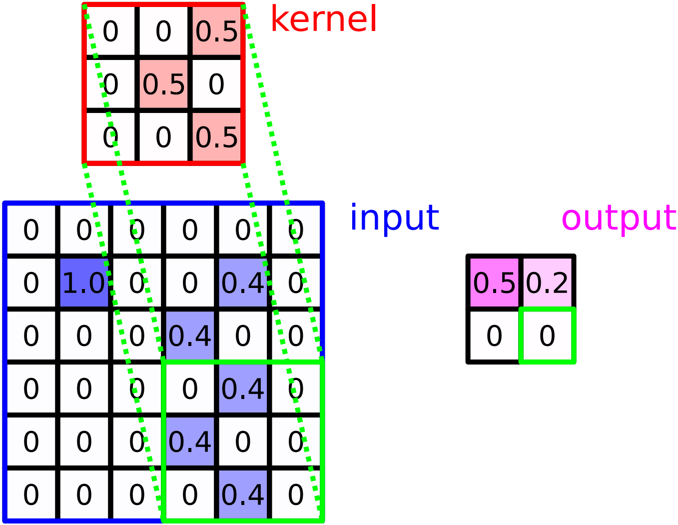
Striding: stride=3 (k: 3×3, i: 6×6, o: 2×2)



Striding: stride=3 (k: 3×3, i: 6×6, o: 2×2)



Striding: stride=3 (k: 3×3, i: 6×6, o: 2×2)



Striding: stride=3 (k: 3×3, i: 6×6, o: 2×2)

kernel

0	0	0.5
0	0.5	0
0	0	0.5

input

0	0	0	0	0	0
0	1.0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0
0	0	0	0.4	0	0
0	0	0	0	0.4	0

output

0.5	0.2
0	0

POOLING

Pooling

- Another way of **downsampling** images is **pooling**.

Pooling

- Another way of **downsampling** images is **pooling**.
- There are different ways to perform pooling. Most popular:
  - **Average Pooling**: take the average value in a $k \times k$ field
  - **Max Pooling**: take the maximum value in a $k \times k$ field
  - **N-Max Pooling**: take the mean over the n maximum values in a $k \times k$ field

Pooling

- Another way of **downsampling** images is **pooling**.
- There are different ways to perform pooling. Most popular:
  - **Average Pooling**: take the average value in a $k \times k$ field
  - **Max Pooling**: take the maximum value in a $k \times k$ field
  - **N-Max Pooling**: take the mean over the n maximum values in a $k \times k$ field
- Pooling will lead to loss of information (no problem if we keep the essential information) but will also reduce computational load and memory requirements.

Pooling

- Another way of **downsampling** images is **pooling**.
- There are different ways to perform pooling. Most popular:
  - **Average Pooling**: take the average value in a $k \times k$ field
  - **Max Pooling**: take the maximum value in a $k \times k$ field
  - **N-Max Pooling**: take the mean over the n maximum values in a $k \times k$ field
- Pooling will lead to loss of information (no problem if we keep the essential information) but will also reduce computational load and memory requirements.
- Pooling will increase the receptive field through depth of network.

Pooling

- Another way of **downsampling** images is **pooling**.
- There are different ways to perform pooling. Most popular:
  - **Average Pooling**: take the average value in a $k \times k$ field
  - **Max Pooling**: take the maximum value in a $k \times k$ field
  - **N-Max Pooling**: take the mean over the n maximum values in a $k \times k$ field
- Pooling will lead to loss of information (no problem if we keep the essential information) but will also reduce computational load and memory requirements.
- Pooling will increase the receptive field through depth of network.
- Pooling is a fixed operation compared to “strided” convolutions, i.e., there are no parameters to learn.

Max Pooling: size=2 (i: 4×4, o: 2×2)

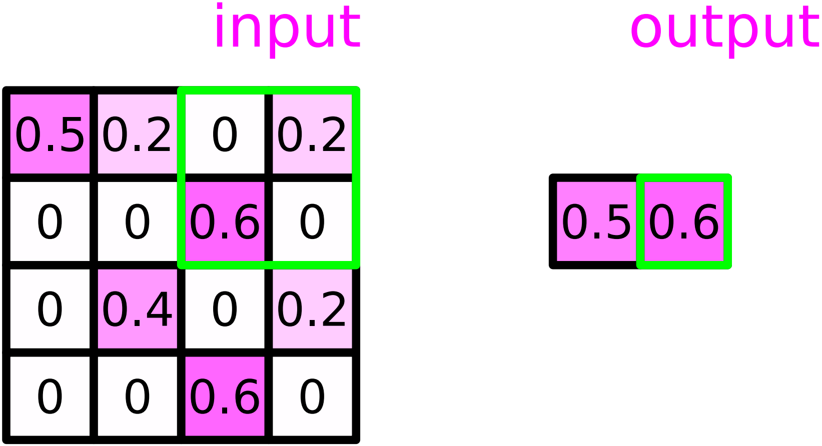
input

0.5	0.2	0	0.2
0	0	0.6	0
0	0.4	0	0.2
0	0	0.6	0

output

0.5

Max Pooling: size=2 (i: 4×4, o: 2×2)



Max Pooling: size=2 (i: 4×4, o: 2×2)

input

0.5	0.2	0	0.2
0	0	0.6	0
0	0.4	0	0.2
0	0	0.6	0

output

0.5	0.6
0.4	

Max Pooling: size=2 (i: 4×4, o: 2×2)

input

0.5	0.2	0	0.2
0	0	0.6	0
0	0.4	0	0.2
0	0	0.6	0

output

0.5	0.6
0.4	0.6

Max Pooling: size=2 (i: 4×4, o: 2×2)

input

0.5	0.2	0	0.2
0	0	0.6	0
0	0.4	0	0.2
0	0	0.6	0

output

0.5	0.6
0.4	0.6

INPUTS AND OUTPUTS

Inputs in CNNs

- Until now, we assumed grayscale images with 1 channel.

Inputs in CNNs

- Until now, we assumed grayscale images with 1 channel.
- RGB images have 3 **channels** for red, green, blue, typically stored in a shape of (width, height, 3).

Inputs in CNNs

- Until now, we assumed grayscale images with 1 channel.
- RGB images have 3 **channels** for red, green, blue, typically stored in a shape of (width, height, 3).
- After the convolutional operations, channels are also called **feature maps** or **activation maps**.

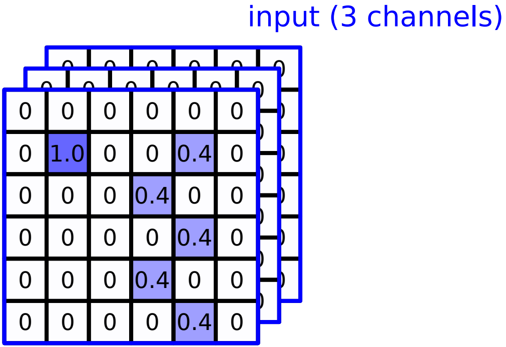
Inputs in CNNs

- Until now, we assumed grayscale images with 1 channel.
- RGB images have 3 **channels** for red, green, blue, typically stored in a shape of (width, height, 3).
- After the convolutional operations, channels are also called **feature maps** or **activation maps**.
- We need to make sure our kernel matches the number of channels (e.g., if the input is 3D, the kernel must be 3D as well).

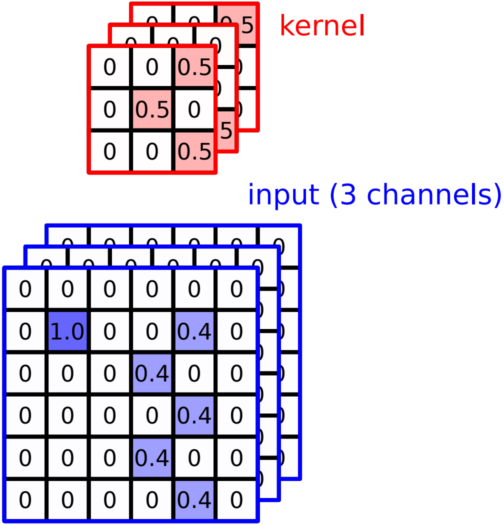
Inputs in CNNs

- Until now, we assumed grayscale images with 1 channel.
- RGB images have 3 **channels** for red, green, blue, typically stored in a shape of (width, height, 3).
- After the convolutional operations, channels are also called **feature maps** or **activation maps**.
- We need to make sure our kernel matches the number of channels (e.g., if the input is 3D, the kernel must be 3D as well).
- Regardless of the number of channels, a single feature map/channel will be produced (it just computes the sum of the channel-wise convolutions).

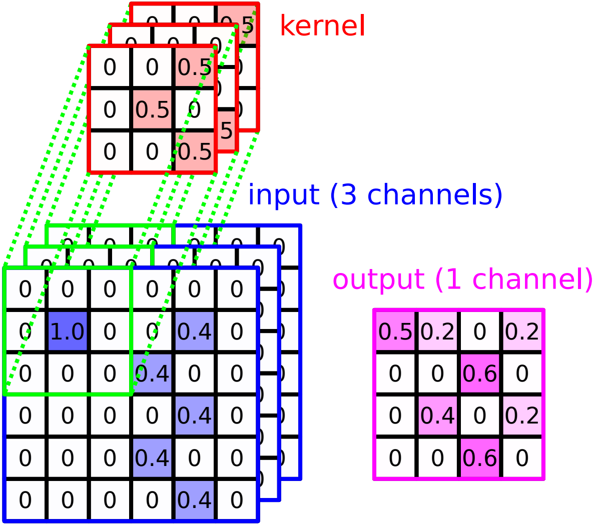
Inputs in CNNs



Inputs in CNNs



Inputs in CNNs



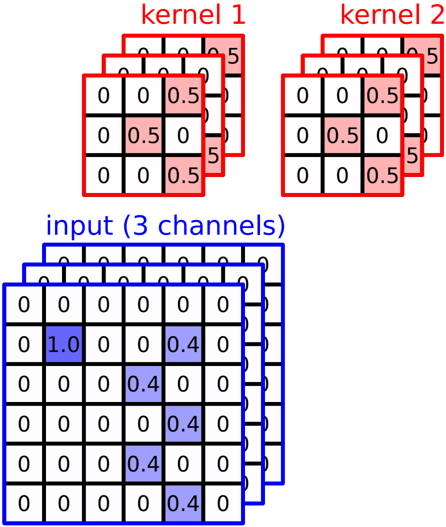
Outputs in CNNs

- We usually want to apply **multiple kernels** to the image.

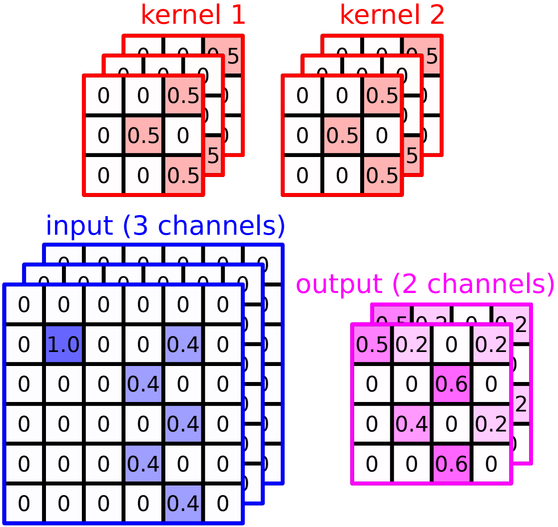
Outputs in CNNs

- We usually want to apply **multiple kernels** to the image.
- For each (multi-dimensional) kernel, we create a **feature map/channel** in the CNN output.

Outputs in CNNs



Outputs in CNNs



CREATING A COMPLETE CNN

Multiple Levels of Convolutions

- A typical CNN architecture has **several layers** of:
  1. Convolution
  2. Non-linearity
  3. Pooling (optional)

Multiple Levels of Convolutions

- A typical CNN architecture has **several layers** of:
  1. Convolution
  2. Non-linearity
  3. Pooling (optional)
- Each kernel produces a new feature map for the next layer.

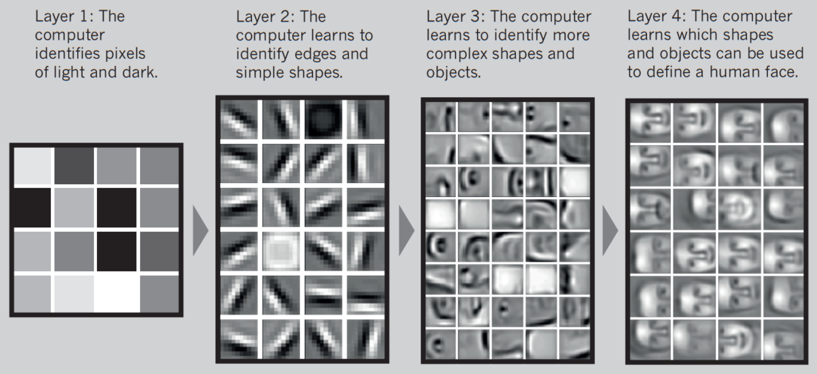
Multiple Levels of Convolutions

- A typical CNN architecture has **several layers** of:
  1. Convolution
  2. Non-linearity
  3. Pooling (optional)
- Each kernel produces a new feature map for the next layer.
- The complexity of detected features tends to increase layer by layer (e.g., first feature map does edge detection, later ones combine it to complex shapes).

Multiple Levels of Convolutions

- A typical CNN architecture has **several layers** of:
  1. Convolution
  2. Non-linearity
  3. Pooling (optional)
- Each kernel produces a new feature map for the next layer.
- The complexity of detected features tends to increase layer by layer (e.g., first feature map does edge detection, later ones combine it to complex shapes).
- Interactive CNN visualization demo:
  `https://poloclub.github.io/cnn-explainer/`

Multiple Levels of Convolutions



[Adapted from: https://www.nature.com/articles/505146a (image credit: Andrew Ng). Also see: Honglak Lee et al. Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks. Communications of the ACM, 54(10). 2011.]

Final Output

- Depending on the task, we might need to perform some additional operations after the convolutions.

Final Output

- Depending on the task, we might need to perform some additional operations after the convolutions.
- Example: **image classification**:
  - We want to employ the same strategy we already know from regular neural networks.

Final Output

- Depending on the task, we might need to perform some additional operations after the convolutions.
- Example: **image classification**:
  - We want to employ the same strategy we already know from regular neural networks.
  - This means that we would like to have a flat vector of size K with all the class probabilities.

Final Output

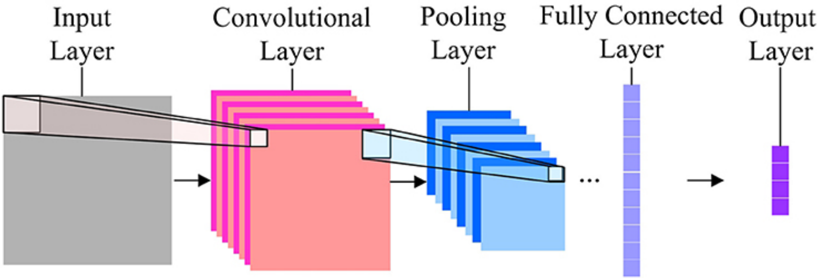
- Depending on the task, we might need to perform some additional operations after the convolutions.
- Example: **image classification**:
  - We want to employ the same strategy we already know from regular neural networks.
  - This means that we would like to have a flat vector of size K with all the class probabilities.
  - We know that the softmax function can produce the probabilities, but the current output of our CNN is a multi-dimensional feature map, which is incompatible.

Final Output

- Depending on the task, we might need to perform some additional operations after the convolutions.
- Example: **image classification**:
  - We want to employ the same strategy we already know from regular neural networks.
  - This means that we would like to have a flat vector of size K with all the class probabilities.
  - We know that the softmax function can produce the probabilities, but the current output of our CNN is a multi-dimensional feature map, which is incompatible.
  - We somehow need to transform this output to a flat vector of size K.

Final Output

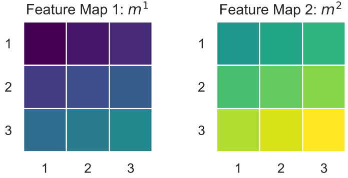
- Solution: Reshape the multi-dimensional output into a vector (a.k.a. **flatten**), and then apply a regular **fully connected layer** that maps the flattened size to K.



[Source: Min Peng et al. Dual Temporal Scale Convolutional Neural Network for Micro-Expression Recognition. Frontiers in Psychology 8. 2017.]

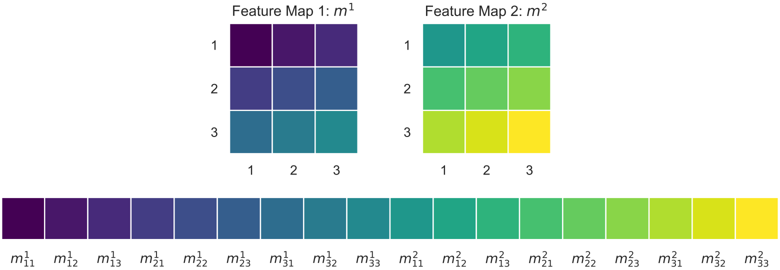
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.



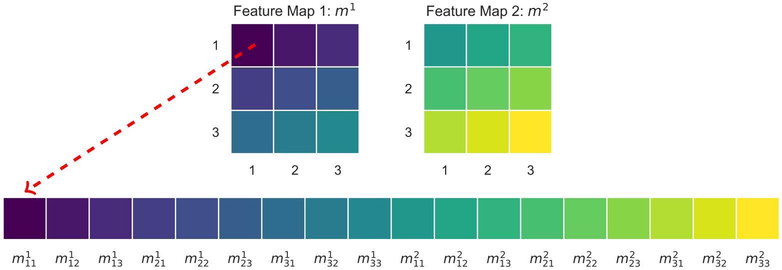
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



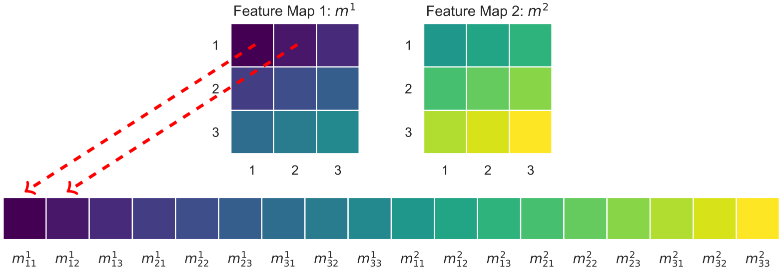
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



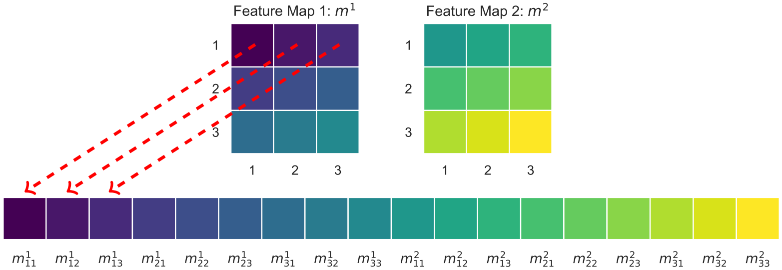
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



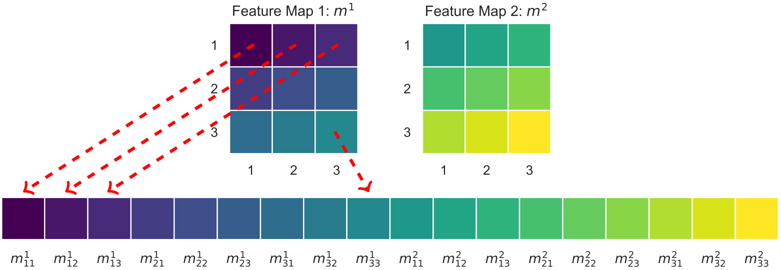
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



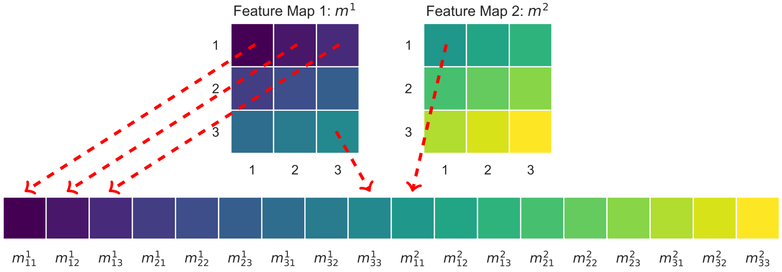
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



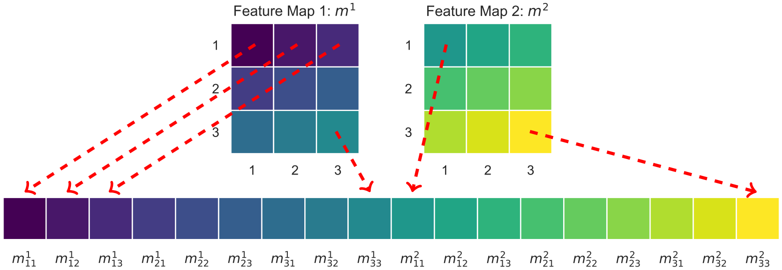
Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size 18



Flatten Example

- Assume our CNN ultimately produces 2 feature maps m^i, each of size $3 \times 3 \Rightarrow 2 \cdot 3 \cdot 3 = 18$ flat elements.
- For each map, run through all elements from left to right and top to bottom and put the corresponding element into our flat vector of size $18 \Rightarrow$ ready for regular NN input!