Deep Learning in Python

Lecture 2: Convolutional Neural Networks

Dr Andreas C. S. Jørgensen

Imperial College London

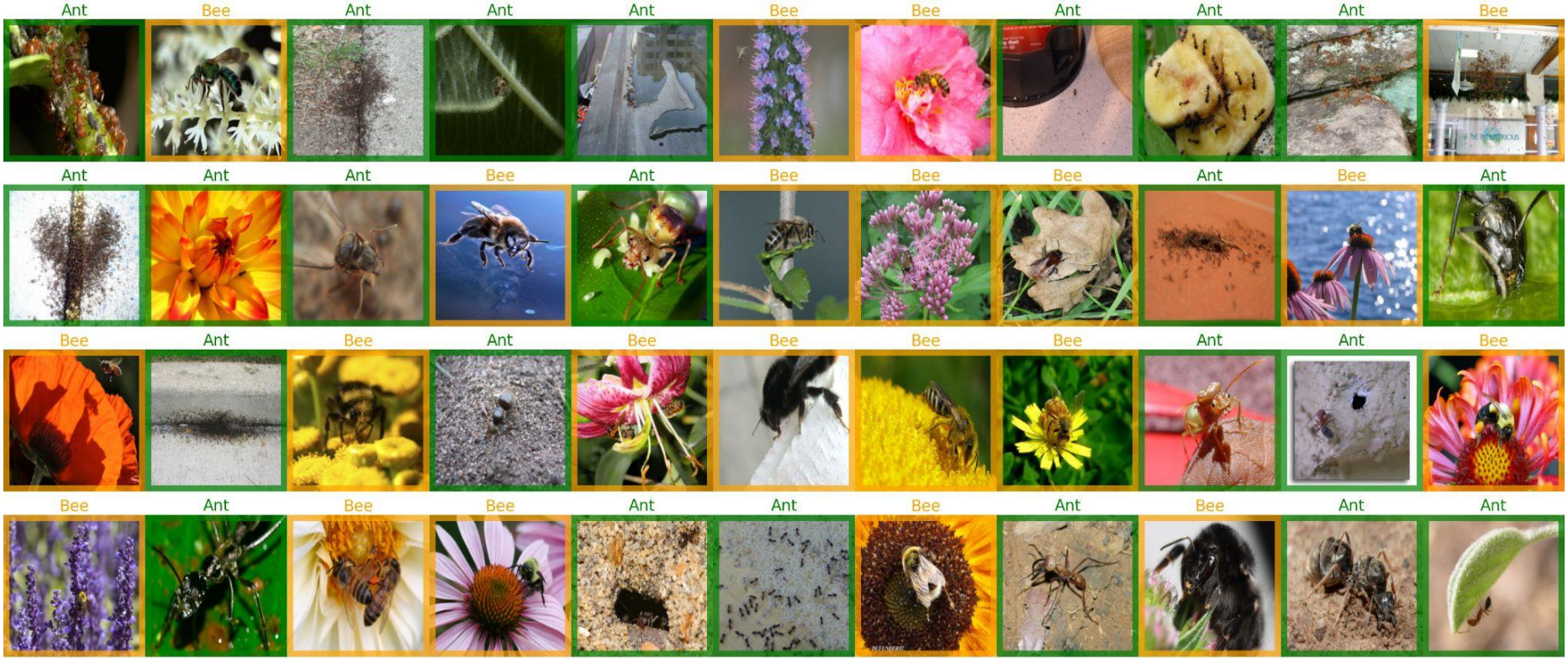
Imperial-X

Outline

Hands-on course in Deep Learning using Python (PyTorch) with focus on computer vision.

Lecture 1	Deep Learning concepts and terminology. PyTorch commands, syntax and libraries (torch, torchvision). Building deep neural networks.
Lecture 2	Convolutional Neural Networks (CNN). Building CNN in PyTorch. Using GPUs with PyTorch.
Lecture 3	Optimisation: How to train and test neural networks robustly. Regularisation and hyper-parameter tuning.

Classify images



Outline

1) Concepts

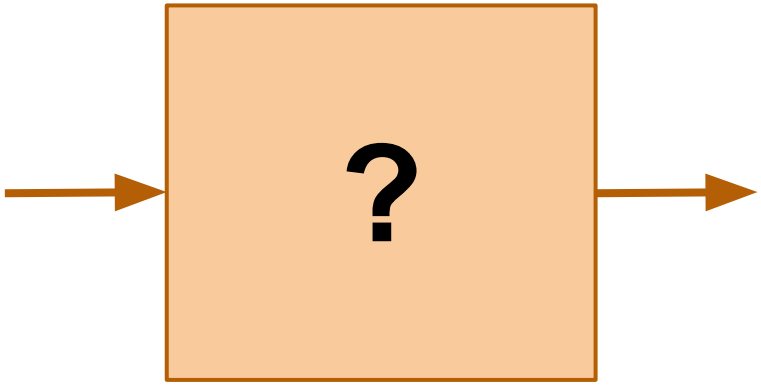
2) Toolkit (PyTorch)

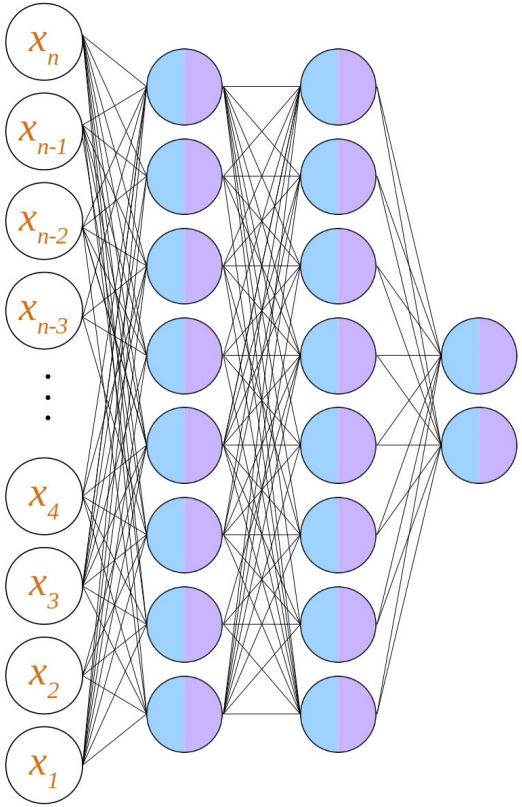
3) Combine 1 and 2 to build CNN

4) Beyond classification

Convolutional Neural Networks

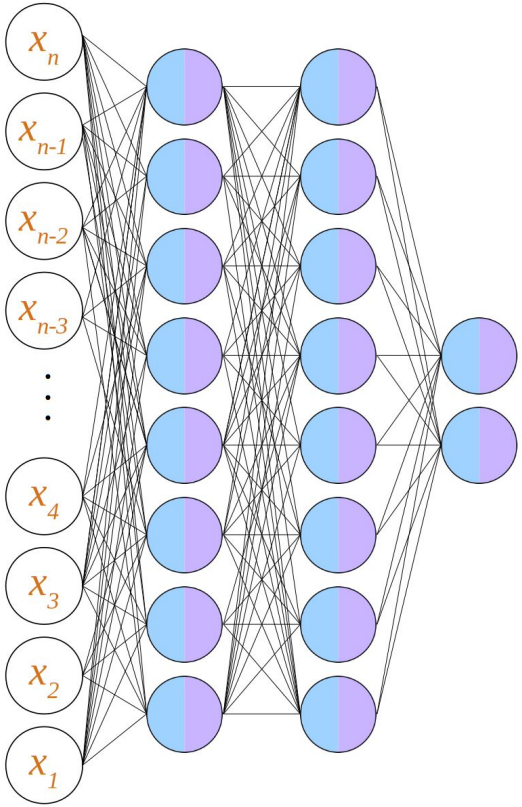
Architecture





Architecture

Create feature vector and extract information through **convolutional** and **pooling** layers.



Convolutional layers

2	5	8	7	9	8
1	3	4	2	8	3
0	4	1	2	1	4
3	3	8	7	5	1
4	1	2	2	9	2
5	1	3	7	4	8

Image

*

-1	-2	-1
0	0	0
1	2	1

Kernel/Filter
(Horizontal edge detection)

=

-11			

Resulting image

Convolution: Sum result of element-wise multiplication with kernel

Convolutional layers

2	5	8	7	9	8
1	3	4	2	8	3
0	4	1	2	1	4
3	3	8	7	5	1
4	1	2	2	9	2
5	1	3	7	4	8

Image

*

-1	-2	-1
0	0	0
1	2	1

Kernel/Filter
(Horizontal edge detection)

=

-11	-20		

Resulting image

Stride: The kernel moves (strides) a number of pixels for every convolution operation

Convolutional layers

2	5	8	7	9	8
1	3	4	2	8	3
0	4	1	2	1	4
3	3	8	7	5	1
4	1	2	2	9	2
5	1	3	7	4	8

Image

*

-1	-2	-1
0	0	0
1	2	1

* Kernel/Filter
(Horizontal edge detection)

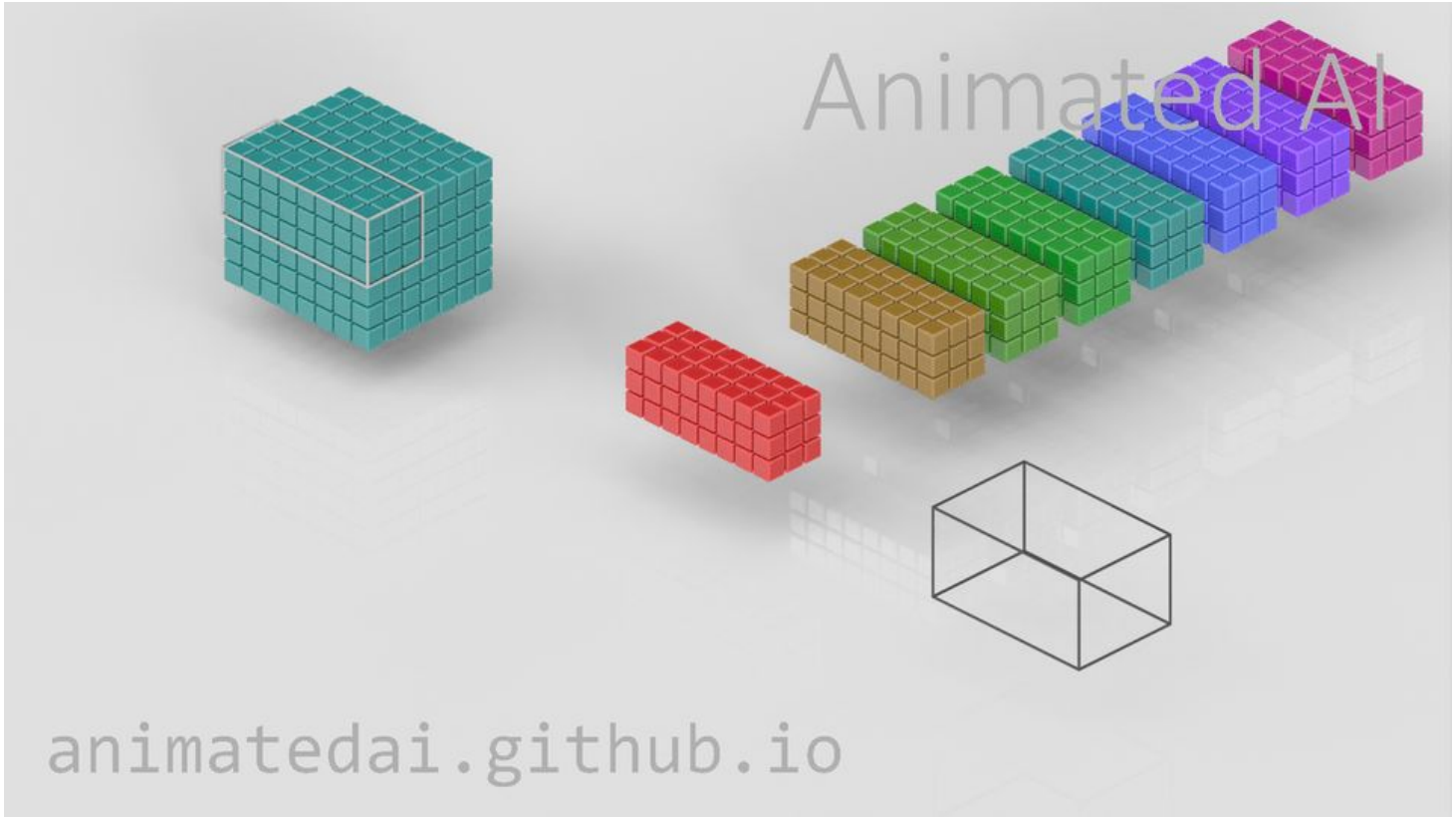
=

-11	-20	-45	-70
6	13	11	-3
-1	-1	9	14
-7	-12	-6	5

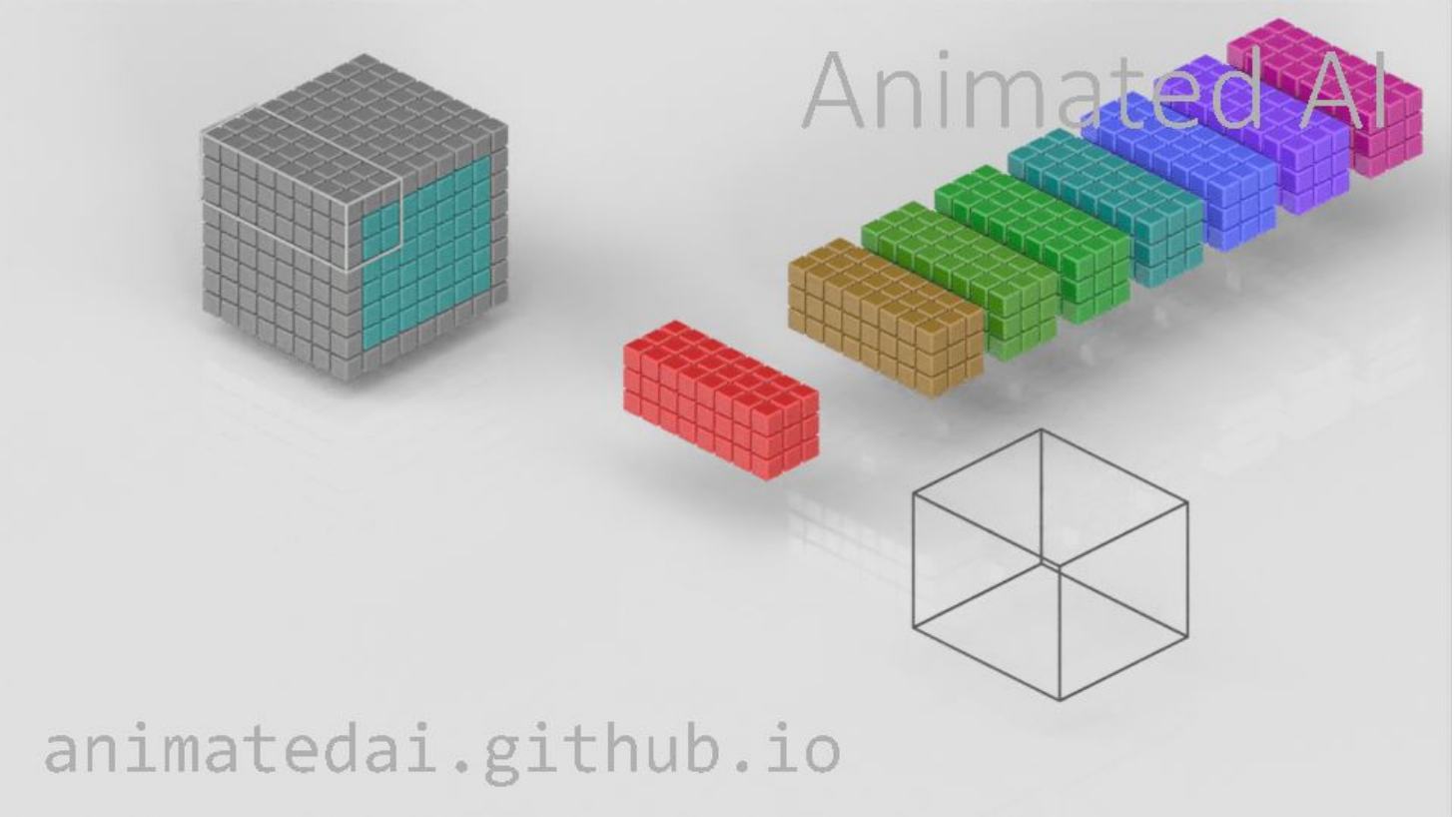
Resulting image

Stride: The kernel moves (strides) a number of pixels for every convolution operation

Convolutional layers



Padding



Other convolutions

- Other dimensions than 3×3.
- Depthwise convolutions: One filter per channel. The output is stacked.
- 1×1 (pointwise) convolutions: Add up channels.
- Dilated convolutions: Filters with gaps (gives a wider field of view).
- Transposed convolutions: Increase spatial dimensions adding details.

Pooling layers

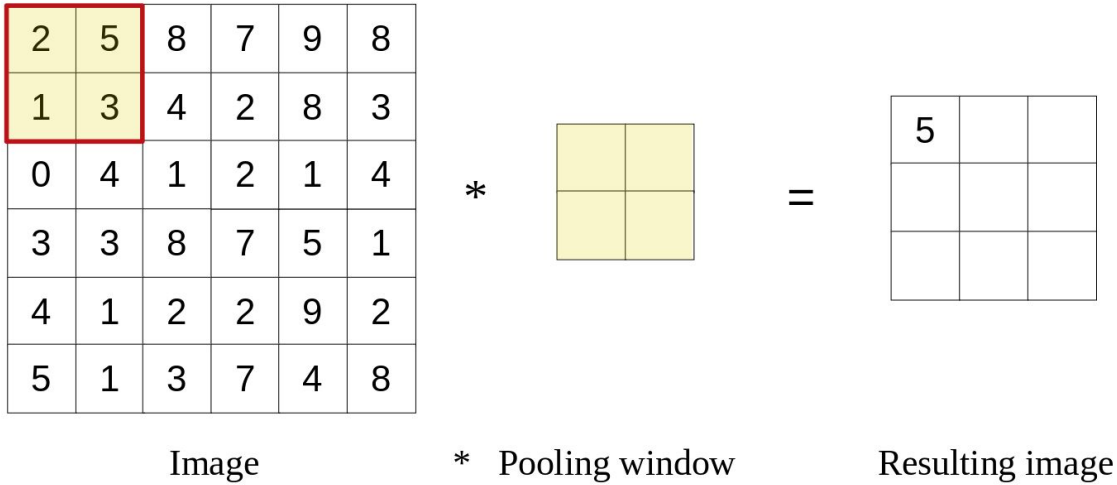
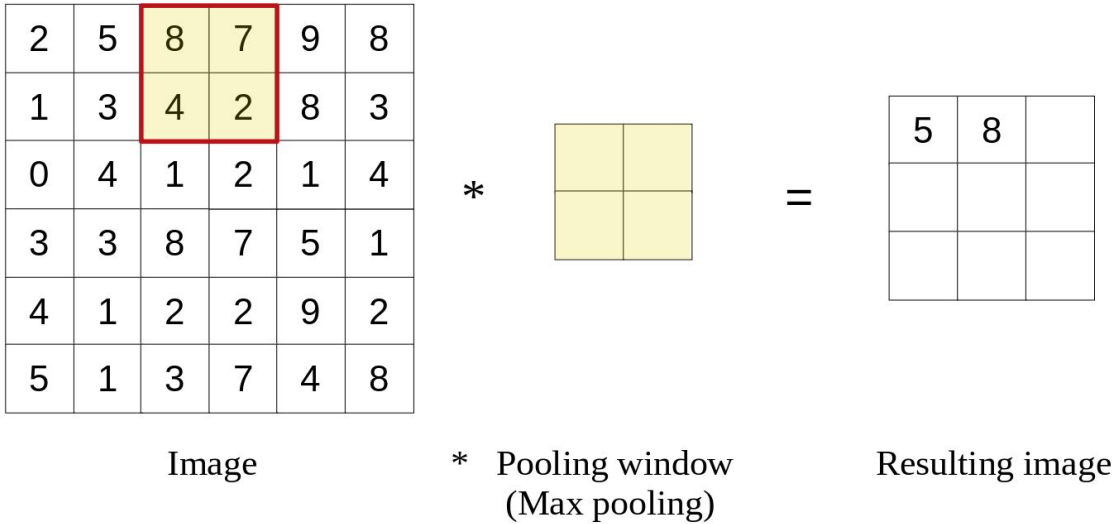


Image * Pooling window Resulting image

Pooling: Reduce feature map size and add spatial invariance.

Pooling layers



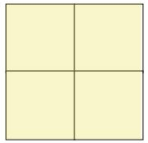
2×2 max pooling: Finds maximum value within window: The most prominent feature. With a stride of 2.

Pooling layers

2	5	8	7	9	8
1	3	4	2	8	3
0	4	1	2	1	4
3	3	8	7	5	1
4	1	2	2	9	2
5	1	3	7	4	8

Image

*



Pooling window
(Max pooling)

=

5	8	9
4	8	5
5	7	9

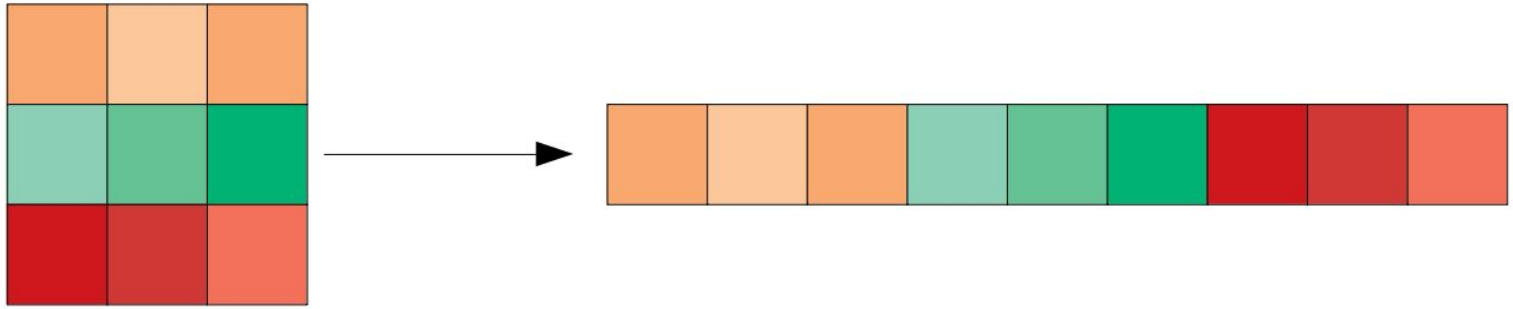
Resulting image

2×2 max pooling: Finds maximum value within window: The most prominent feature. With a stride of 2.

Pooling layers

- 2×2 or 3×3 max pooling: Retains most prominent feature. Even when moved by a pixel, we get the same result, i.e., the pooling is not sensitive to position.
- Average pooling: Smoothed representation.
- Global average pooling: The global average of the entire picture.
- Adaptive pooling: Vary the pooling stride and window size.

Flattening



We now have a feature vector and can feed the data into a neural network.

Transfer learning

Rather than training a new network, we just *tweak* an existing one. First, we replace the output layer with a layer that produces the right number of outputs.

Fine tuning: Adjust the parameters through training on the data set.

Feature extraction: Only calibrate the final layer, but "freeze" the rest of the network.

Only viable option for **small datasets**!

Step by step

Components of a Convolutional Neural Network:

- Use convolutional layers to detect patterns and features (e.g., edges, textures).
- Use pooling layers to reduce feature map size and introduce spatial invariance.
- Pass the extracted features to the type of architecture we have seen: Fully-connected layers and an output layer (using the cross-validation loss for classification).

New tools required?

GPUs

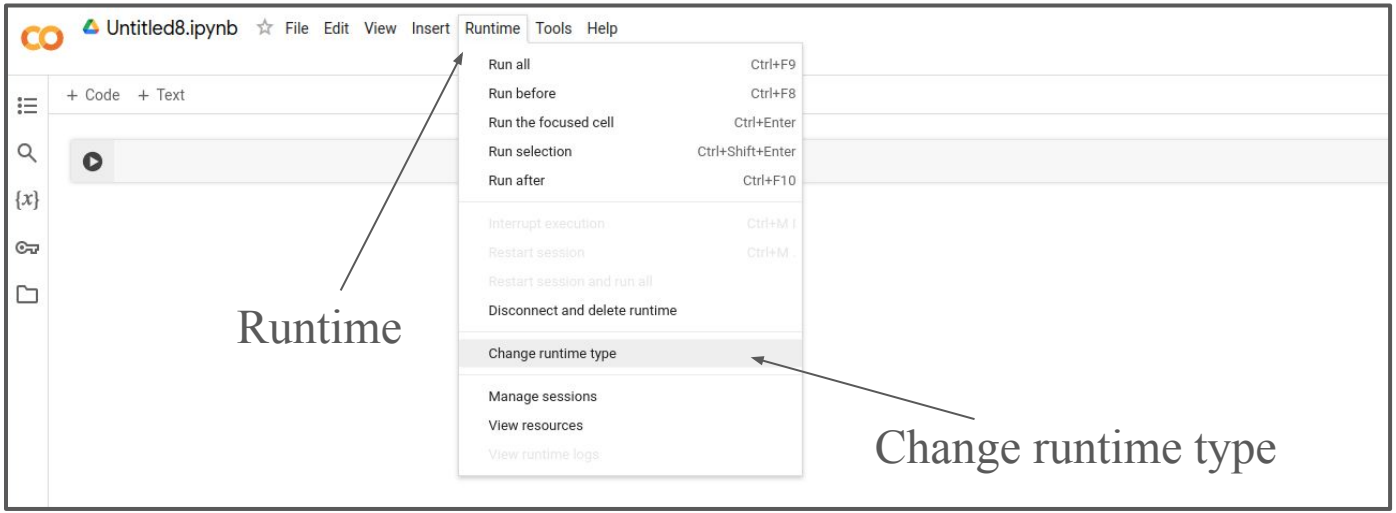
Graphics processing units can perform many computations in parallel and can thus lead to speed up for parallelizable tasks.

```
# Check GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

GPUs

Graphics processing units can perform many computations in parallel and can thus lead to speed up for parallelizable tasks.

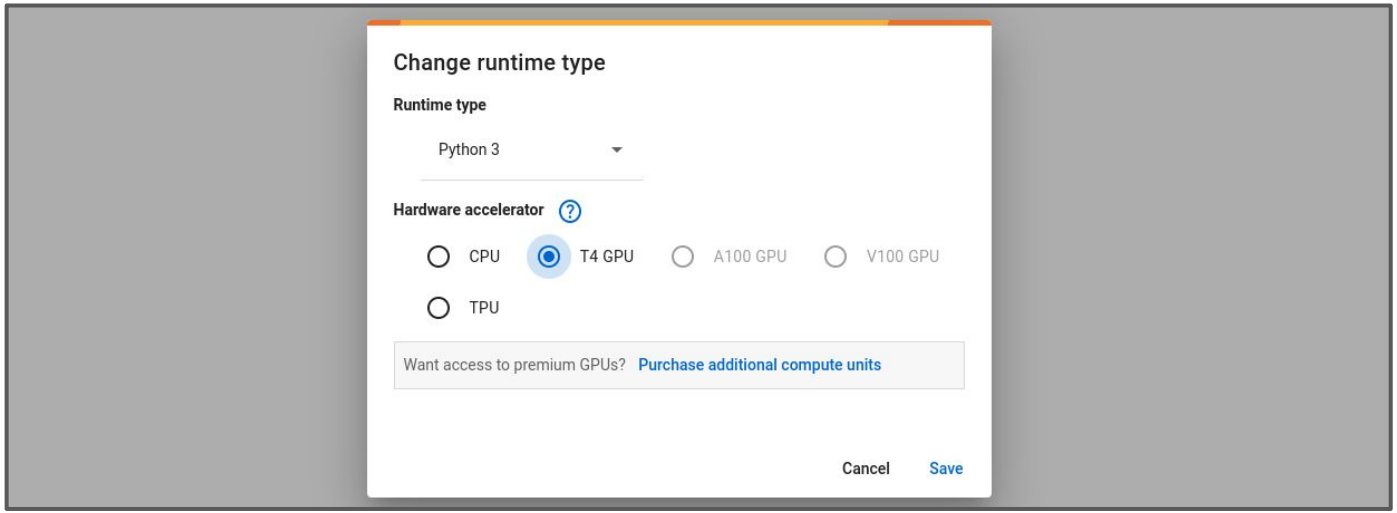
```
# Check GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



GPUs

Graphics processing units can perform many computations in parallel and can thus lead to speed up for parallelizable tasks.

```
# Check GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



Building CNNs

New layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class




        # Define layers and activations functions




    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\ldots\phi_2(\phi_1(\boldsymbol{x}))\ldots))$$

New layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class

        self.act1 = nn.ReLU() # Activation function



        self.fc1 = nn.Linear(6*8*8, 120) # Fully connected
        self.fc2 = nn.Linear(120, 84) # Fully connected
        self.fc3 = nn.Linear(84, 10) # Fully connected

    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\ldots\phi_2(\phi_1(\boldsymbol{x}))\ldots))$$

New layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class
        self.conv1 = nn.Conv2d(3, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.act1 = nn.ReLU() # Activation function
        self.pool1 = nn.MaxPool2d(2, 2) # Pooling
        self.conv2 = nn.Conv2d(6, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.flat = nn.Flatten() # Flattening
        self.fc1 = nn.Linear(6*8*8, 120) # Fully connected
        self.fc2 = nn.Linear(120, 84) # Fully connected
        self.fc3 = nn.Linear(84, 10) # Fully connected

    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\ldots\phi_2(\phi_1(\boldsymbol{x}))\ldots))$$

New layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class
        self.conv1 = nn.Conv2d(3, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.act1 = nn.ReLU() # Activation function
        self.pool1 = nn.MaxPool2d(2, 2) # Pooling
        self.conv2 = nn.Conv2d(6, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.flat = nn.Flatten() # Flattening
        self.fc1 = nn.Linear(6*8*8, 120) # Fully connected    6 channels of 8x8 pixels (starting at 32x32 pixels)
        self.fc2 = nn.Linear(120, 84) # Fully connected
        self.fc3 = nn.Linear(84, 10) # Fully connected

    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\dots\phi_2(\phi_1(\boldsymbol{x}))\dots))$$

New layers

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class
        self.conv1 = nn.Conv2d(3, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.act1 = nn.ReLU() # Activation function
        self.pool1 = nn.MaxPool2d(2, 2) # Pooling
        self.conv2 = nn.Conv2d(6, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.flat = nn.Flatten() # Flattening
        self.fc1 = nn.Linear(6*8*8, 120) # Fully connected
        self.fc2 = nn.Linear(120, 84) # Fully connected
        self.fc3 = nn.Linear(84, 10) # Fully connected

    def forward(self, x):
        x = self.pool1(self.act1(self.conv2(self.pool1(self.act1(self.conv1(x))))))
        x = self.flat(x)
        x = self.fc3(self.act1(self.fc2(self.act1(self.fc1(x)))))
        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\dots\phi_2(\phi_1(\boldsymbol{x}))\dots))$$

New layers

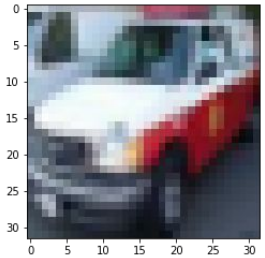
```
class CNN(nn.Module):
    def __init__(self):
        super().__init__() # Inherit from parent class
        self.conv1 = nn.Conv2d(3, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.act1 = nn.ReLU() # Activation function
        self.pool1 = nn.MaxPool2d(2, 2) # Pooling
        self.conv2 = nn.Conv2d(6, 6, kernel_size=(3,3), stride=1, padding=1) # Convolutional
        self.flat = nn.Flatten() # Flattening
        self.fc1 = nn.Linear(6*8*8, 120) # Fully connected
        self.fc2 = nn.Linear(120, 84) # Fully connected
        self.fc3 = nn.Linear(84, 10) # Fully connected

    def forward(self, x):
        x = self.pool1(self.act1(self.conv2(self.pool1(self.act1(self.conv1(x))))))
        x = self.flat(x)
        x = self.fc3(self.act1(self.fc2(self.act1(self.fc1(x)))))
        return x
```

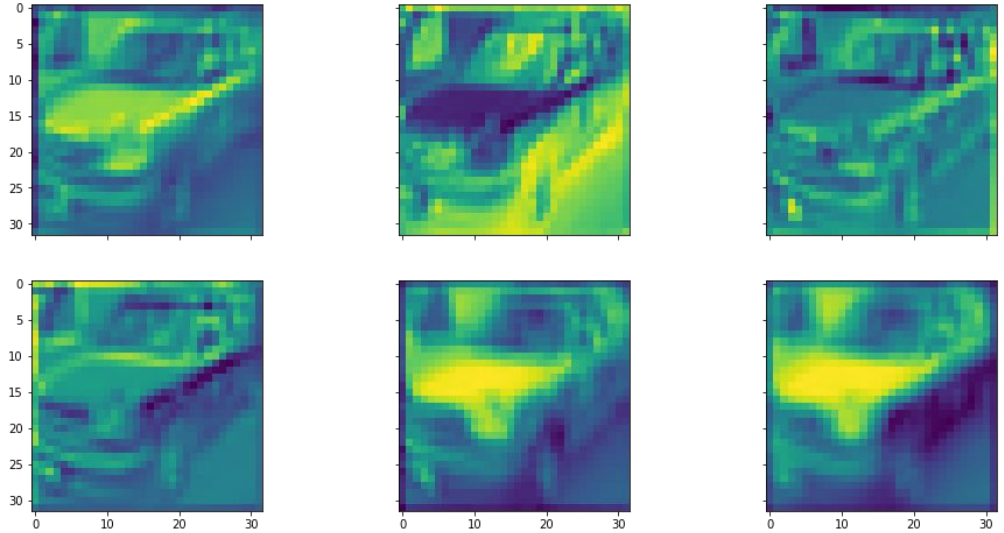
$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\dots\phi_2(\phi_1(\boldsymbol{x}))\dots))$$

New layers

```
X = torch.tensor([trainset.data[60]], dtype=torch.float32).permute(0,3,1,2)
with torch.no_grad():
    X = cnn.conv1(X)
```



conv1(X)



Training

```
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(cnn.parameters(), lr=1e-3, momentum=0.9)
```

Training

```
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(cnn.parameters(), lr=1e-3, momentum=0.9)
```

In each epoch, looping over batches,

1) Make model predictions.

2) Compute the loss.

3) Compute the gradient of the loss.

4) Update parameters.

```
for epoch in range(n_epochs):
    for data in trainloader:
        inputs, labels = data
        # 1) Feedforward
        y_pred = cnn(inputs)

        # 2) Compute the loss
        loss = loss_func(y_pred, labels)

        # 3) Backpropagation of errors
        optimizer.zero_grad()
        loss.backward()

        # 4) Update parameters
        optimizer.step()
```

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Fine tuning: Adjust the parameters through training on the data set.

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Fine tuning: Adjust the parameters through training on the data set.

```
from torchvision.models import resnet50

# Instigate the pretrained model
model1 = resnet50(pretrained=True)

# Input features of the last layer of the original model
n_ftrs = model1.fc.in_features

# Replace the last layer with a new output layer that matches the dataset.
n_classes = len(class_names)
model1.fc = nn.Linear(n_ftrs, n_classes)
```

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Fine tuning: Adjust the parameters through training on the data set.

```
from torchvision.models import resnet50

# Instigate the pretrained model
model1 = resnet50(pretrained=True)

# Input features of the last layer of the original model
n_ftrs = model1.fc.in_features

# Replace the last layer with a new output layer that matches the dataset.
n_classes = len(class_names)
model1.fc = nn.Linear(n_ftrs, n_classes)
```

TRAIN AS BEFORE

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Feature extraction: Only calibrate the final layer, but "freeze" the rest of the network.

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Feature extraction: Only calibrate the final layer, but "freeze" the rest of the network.

```
model2 = torchvision.models.resnet18(pretrained=True) # You could also use resnet50 as above

# Freeze the network
for param in model2.parameters():
    param.requires_grad = False

# Input features of the last layer of the original model
n_ftrs = model2.fc.in_features

n_classes = len(class_names)
model2.fc = nn.Linear(n_ftrs, n_classes)
```

Transfer learning

Replace the output layer with a layer that produces the right number of outputs.

Feature extraction: Only calibrate the final layer, but "freeze" the rest of the network.

```
model2 = torchvision.models.resnet18(pretrained=True) # You could also use resnet50 as above

# Freeze the network
for param in model2.parameters():
    param.requires_grad = False

# Input features of the last layer of the original model
n_ftrs = model2.fc.in_features

n_classes = len(class_names)
model2.fc = nn.Linear(n_ftrs, n_classes)
```

TRAIN AS BEFORE

Including GPUs

```
# Check GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Move the model to the GPUs

```
# move to GPU
model = model.to(device)
```

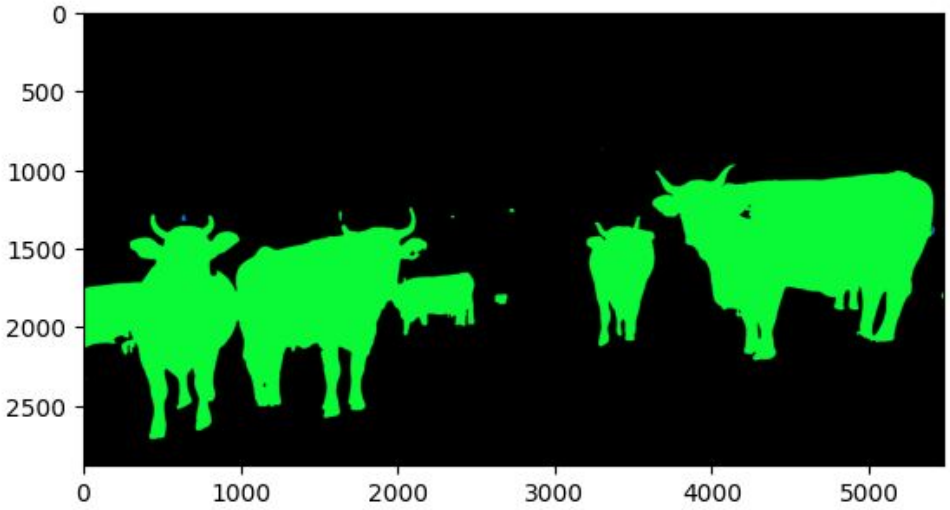
Move mini-batches to the GPUs

```
# Modify training loop
for epoch in range(num_epochs):
    for inputs, labels in trainloader:
        inputs = inputs.to(device)
        labels = labels.to(device)
```

Applications of CNNs

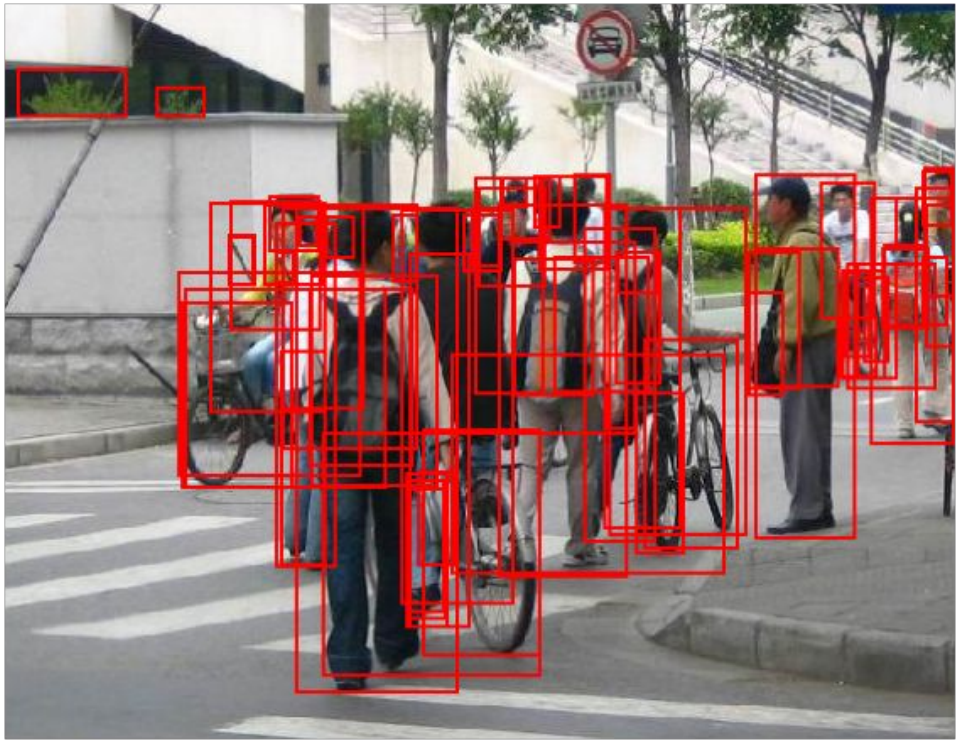
Semantic segmentation

Use CNNs to assign a semantic label to every pixel.




```
model = torch.hub.load('pytorch/vision:v0.10.0', 'deeplabv3_resnet50', pretrained=True) # Pre-trained network (DeepLab3)
```

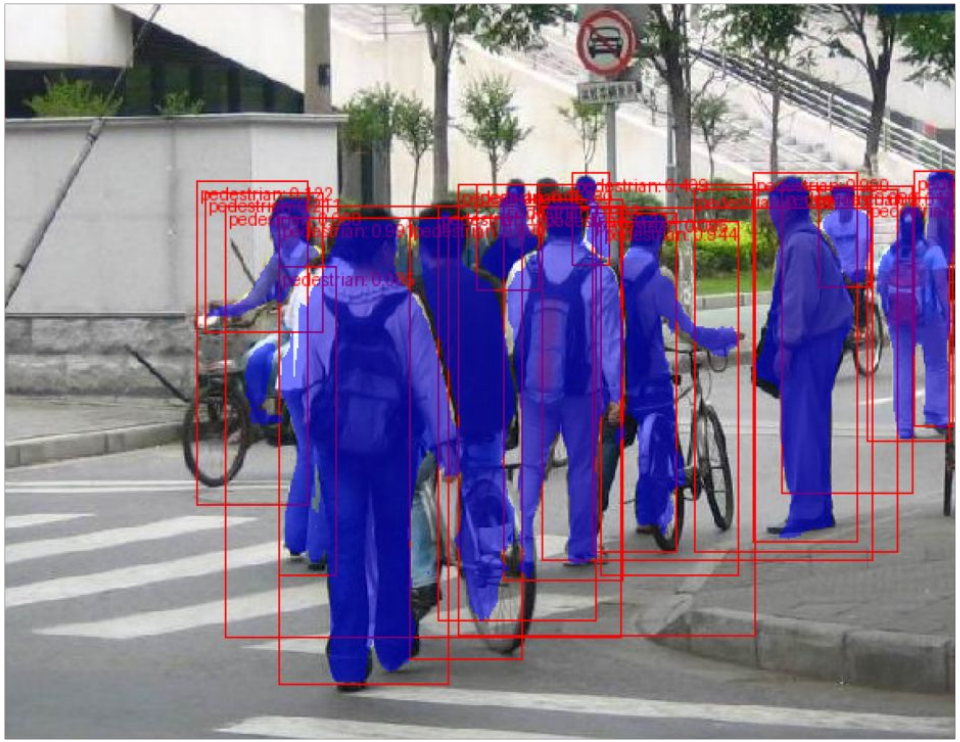
Object detection



Use CNNs to detect and localise objects within an image.

Cf. PyTorch Tutorials on pytorch.org

Instance segmentation



Use CNNs to detect and localise objects within an image.

Cf. PyTorch Tutorials on pytorch.org

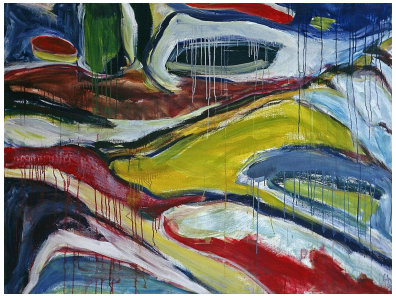
Add instance segmentation to follow individuals.

Image style transfer

Use CNNs to transfer the artistic style of one image to another.

Content

+



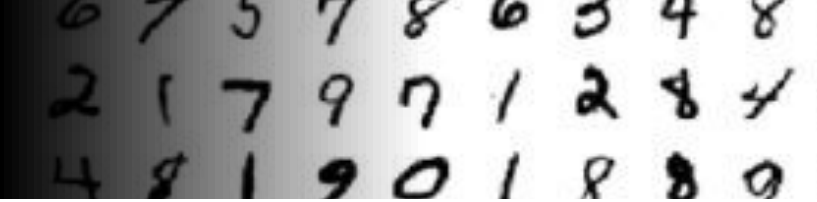
Style

=

New image

```
from torchvision.models import vgg19, VGG19_Weights # Pre-trained network
```

Other examples



- CNNs are widely used for **face recognition** tasks.
- With Recurrent Neural Networks (RNNs), CNNs can **generate image captions**.
- CNNs are used in **generative models** like GANs (Generative Adversarial Networks) for generating realistic images.
- CNNs can be extended to **video data**.
- Super-Resolution: CNNs can be used for **enhancing the resolution** of images.

Summary

Based on this lecture, you should better be able to

1) Explain basic concepts behind Convolutional Neural Networks (CNNs): convolutional and pooling layers). Outline the architecture of CNNs.

2) Translate these concepts into PyTorch.

3) Train and run neural networks on GPUs.

4) Describe the breadth of applications of CNNs.