Introductory Seminar of PyTorch for Deep Learning

Daniele Angioni, Cagliari Digital Lab 2024 - Day 3

Representing Images

Representing Images

As we saw in previous chapters, input data can be represented as set of feature values

Hystorically, datasets are represented as matrices where each row is a sample, and the features are represented in ordered columns

For example, a matrix [100 x 3] represents a dataset of 100 samples with 3 features each.

Representing Images

With general N-dimensional tensors, the representation can be changed, but we will keep the first index as the sample index to keep it consistent.

Hence, **the first dimension will always be the sample index**, and the rest of the dimensions collect generally the rest of the features, but allow more structure than simple row vectors.

This is useful, for example, to deal with images and avoid loss of information regarding the original shape of the images (plus, it is ready for application of operations specific to images, e.g., convolutions).

Representing Images with Tensors

To use PyTorch to create image classifiers (or in general to work with images), we need to be able to represent images in a way that PyTorch can understand

Images are represented as collections of scalars arranged in a specific grid with height x width grid points (pixels)

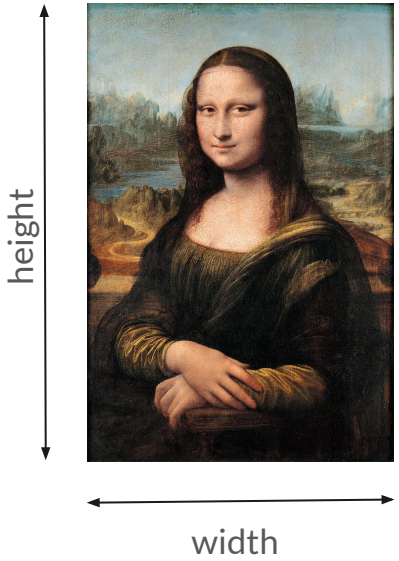
- B/W images are represented with one single scalar per pixel
- RGB images are represented with three values for each pixel (Red, Blue, Green)
- In general, images can have multiple values for each pixel, representing different features (e.g., depth, alpha, temperature, ...)

Representing Images with Tensors

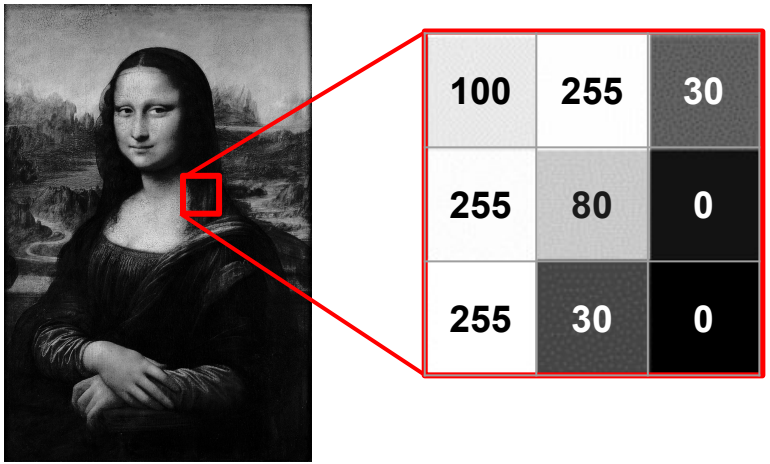
To use PyTorch to create image classifiers (or in general to work with images), we need to be able to represent images in a way that PyTorch can understand

Images are represented as collections of scalar values arranged in a specific grid with height x width grid points (pixels)

The values composing a pixel are often 8-bit unsigned integers, i.e., $2^8 = 256$ possible values in [0, 255]

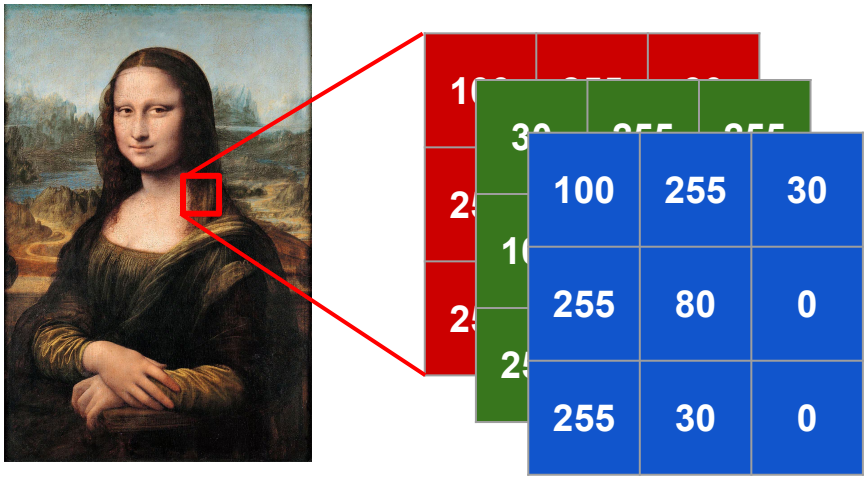


B/W image representation



- For black and white (B/W) images each pixel is represented by a single value
- 255 is pure white, while 0 is pure black
- The shape of a B/W image is [1, height, width]

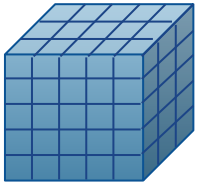
RGB image representation



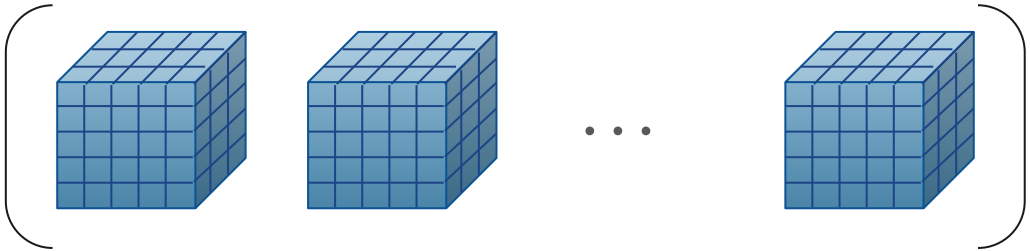
- For coloured images instead, we have 3 channels (**Red**, **Green** **and** **Blue**) for each pixel
- The combination of each 3 values determines the color of the pixel
- In this case, the shape of a coloured image is [**3**, height, width]

In general, pixels can have multiple channels (alpha, temperature,...)

Considering height, width and the channels, each image is a 3D tensor



This means that, a collection of RGB images represented as tensor becomes a 4D tensor (or >4 if additional channels are added)



[#samples, #channels, height, width]

Keep in mind that [#samples, #channels, height, width] is the standardised order of dimensions only for the deep learning community, while in general when we load an image we find the channels in the last dimension

Loading an image

```python
from PIL import Image
import requests
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt

url = 'https://static.wikia.nocookie.net/fansekai/images/1/1d/HA_HA_HA_HA.jpg/'\
      'revision/latest/scale-to-width-down/1000?cb=20220807225343'
response = requests.get(url)
img = Image.open(BytesIO(response.content))
np_img = np.array(img)
print(np_img.shape)

fig, axs = plt.subplots(nrows=1, ncols=3)
axs[0].imshow(np_img)
axs[1].imshow(np_img[200:400, 400:600, :])

np_img[:, :, 0] = 0
np_img[:, :, 2] = 0
axs[2].imshow(np_img[200:400, 400:600, :])
fig.tight_layout()
fig.show()
```

Load the image from URL using the PIL library

Loading an image

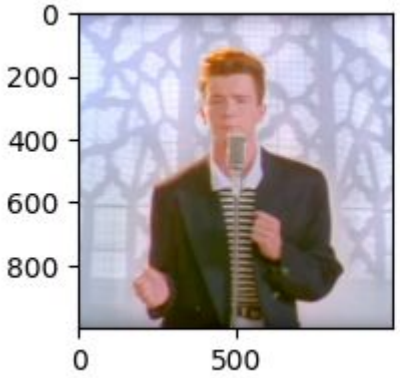
```python
from PIL import Image
import requests
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt

url = 'https://static.wikia.nocookie.net/fansekai/images/1/1d/HA_HA_HA_HA.jpg/'\
      'revision/latest/scale-to-width-down/1000?cb=20220807225343'
response = requests.get(url)
img = Image.open(BytesIO(response.content))
np_img = np.array(img)
print(np_img.shape)

fig, axs = plt.subplots(nrows=1, ncols=3)
axs[0].imshow(np_img)
axs[1].imshow(np_img[200:400, 400:600, :])

np_img[:, :, 0] = 0
np_img[:, :, 2] = 0
axs[2].imshow(np_img[200:400, 400:600, :])
fig.tight_layout()
fig.show()
```

Transform it to a numpy array and visualize the shape

```
(1000, 1000, 3)
```

Loading an image

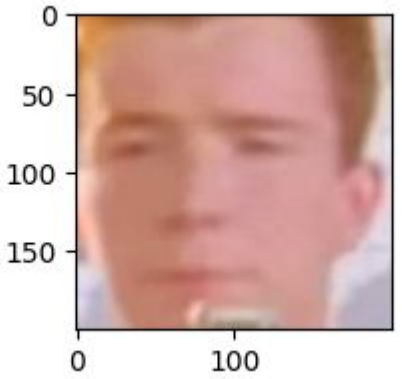
```python
from PIL import Image
import requests
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt

url = 'https://static.wikia.nocookie.net/fansekai/images/1/1d/HA_HA_HA_HA.jpg/'\
      'revision/latest/scale-to-width-down/1000?cb=20220807225343'
response = requests.get(url)
img = Image.open(BytesIO(response.content))
np_img = np.array(img)
print(np_img.shape)

fig, axs = plt.subplots(nrows=1, ncols=3)
axs[0].imshow(np_img)
axs[1].imshow(np_img[200:400, 400:600, :])

np_img[:, :, 0] = 0
np_img[:, :, 2] = 0
axs[2].imshow(np_img[200:400, 400:600, :])
fig.tight_layout()
fig.show()
```



Show the entire image

Loading an image

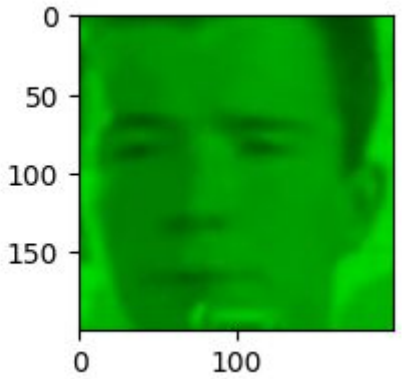
```
1  from PIL import Image
2  import requests
3  from io import BytesIO
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  url = 'https://static.wikia.nocookie.net/fansekai/images/1/1d/HA_HA_HA_HA.jpg/'\
8        'revision/latest/scale-to-width-down/1000?cb=20220807225343'
9  response = requests.get(url)
10 img = Image.open(BytesIO(response.content))
11 np_img = np.array(img)
12 print(np_img.shape)
13
14 fig, axs = plt.subplots(nrows=1, ncols=3)
15 axs[0].imshow(np_img)
16 axs[1].imshow(np_img[200:400, 400:600, :])
17
18 np_img[:, :, 0] = 0
19 np_img[:, :, 2] = 0
20 axs[2].imshow(np_img[200:400, 400:600, :])
21 fig.tight_layout()
22 fig.show()
```



Show a only the rows from 200 to 400 and columns from 400 to 600

Loading an image

```python
from PIL import Image
import requests
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt

url = 'https://static.wikia.nocookie.net/fansekai/images/1/1d/HA_HA_HA_HA.jpg/'\
      'revision/latest/scale-to-width-down/1000?cb=20220807225343'
response = requests.get(url)
img = Image.open(BytesIO(response.content))
np_img = np.array(img)
print(np_img.shape)

fig, axs = plt.subplots(nrows=1, ncols=3)
axs[0].imshow(np_img)
axs[1].imshow(np_img[200:400, 400:600, :])

np_img[:, :, 0] = 0
np_img[:, :, 2] = 0
axs[2].imshow(np_img[200:400, 400:600, :])
fig.tight_layout()
fig.show()
```



Show the same as before after zeroing all channels but the second (the green one)

Tensor Images

```
1  import torchvision
2  transform = torchvision.transforms.ToTensor()
3  torch_img = transform(np_img)
4  print(np_img.max())
5  print(torch_img.max())
6  print(np_img.shape)
7  print(torch_img.shape)
```

The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.

Tensor Images

```
1  import torchvision
2  transform = torchvision.transforms.ToTensor()
3  torch_img = transform(np_img)
4  print(np_img.max())
5  print(torch_img.max())
6  print(np_img.shape)
7  print(torch_img.shape)
```

To obtain a tensor representation of an image we can use instantiate a transforms.ToTensor() object and pass the numpy image to it

Tensor Images

```
1  import torchvision
2  transform = torchvision.transforms.ToTensor()
3  torch_img = transform(np_img)
4  print(np_img.max())
5  print(torch_img.max())
6  print(np_img.shape)
7  print(torch_img.shape)
```

This transformation automatically scale the values from [0, 255] to floating points in [0, 1] …

```
254
tensor(0.9961)
```

Tensor Images

```
1  import torchvision
2  transform = torchvision.transforms.ToTensor()
3  torch_img = transform(np_img)
4  print(np_img.max())
5  print(torch_img.max())
6  print(np_img.shape)
7  print(torch_img.shape)
```

… and reshape it to have the channel dimension first.

```
(1000, 1000, 3)
```

```
torch.Size([3, 1000, 1000])
```

Datasets

Samples are often loaded in groups, and groups of samples are called **batches**. In general, when loading a dataset (or a batch), we have an additional dimension to consider.

In traditional machine learning libraries (e.g., scikit-learn), the data reporesentation follows the standard [num_samples, num_features], where each sample is a (atten) row vector, and we stack multiple samlpes in the first dimension.

For example, a dataset of 300 samples represented each with 3 features will have dimensions 300 x 3

Datasets

With modern libraries for deep learning (including PyTorch), the samples are not one-dimensional anymore, but they can have arbitrary shape. In the RGB image example, a batch of images will have 4 dimensions:

Normalizing the data

In Deep Learning it is common practice to normalize the data so that the pixel values lie in a specific distribution.

This means that if we have images from different sources, by applying normalization we make sure they have similar characteristics

The operation applies to each channel the following operation: $x = \frac{x - mean}{std}$

```
normalizer = torchvision.transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                              std=[0.5, 0.5, 0.5])
norm_img = normalizer(torch_img)
print(f"Before normalization: [{torch_img.min()}, {torch_img.max()}]")
print(f"After normalization: [{norm_img.min()}, {norm_img.max()}]")
```

```
Before normalization: [0.0, 0.9960784316062927]
After normalization: [-1.0, 0.9921568632125854]
```

Data augmentation

We can also define other transformations, for example random rotation, random zoom in and out, etc.

These are useful to obtain data augmentations, i.e., artificial variation of the images

Check the documentation of the torchvision.transforms package

A dataset of tiny images: MNIST

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples.

Each of these B/W image has a shape of [28 × 28] and represents a digit from 0 to 9

The labels are, as well, numbers from 0 to 9

Downloading MNIST

```python
from torchvision import datasets
import matplotlib.pyplot as plt

data_path = 'data'
train_set = datasets.MNIST(root=data_path,
                           train=True,
                           download=True)
test_set = datasets.MNIST(root=data_path,
                          train=False,
                          download=True)
print(f"Samples in training set: {len(train_set)}")
print(f"Samples in testing set: {len(test_set)}")

image, label = train_set[0]
plt.imshow(image, cmap='gray')
plt.title(f"Sample label: {label}")
plt.show()
```

Common datasets can be easily downloaded using the torchvision.datasets package

Downloading MNIST

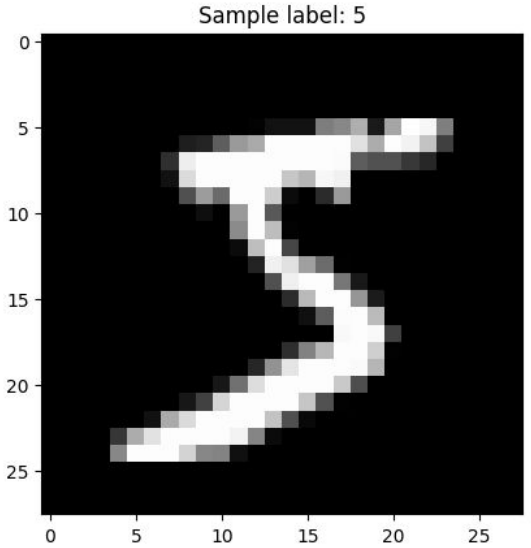
```python
from torchvision import datasets
import matplotlib.pyplot as plt

data_path = 'data'
train_set = datasets.MNIST(root=data_path,
                           train=True,
                           download=True)
test_set = datasets.MNIST(root=data_path,
                          train=False,
                          download=True)
print(f"Samples in training set: {len(train_set)}")
print(f"Samples in testing set: {len(test_set)}")

image, label = train_set[0]
plt.imshow(image, cmap='gray')
plt.title(f"Sample label: {label}")
plt.show()
```

We can access the number of samples with the python built-in len() function

```
Samples in training set: 60000
Samples in testing set: 10000
```

Downloading MNIST

```
1  from torchvision import datasets
2  import matplotlib.pyplot as plt
3
4  data_path = 'data'
5  train_set = datasets.MNIST(root=data_path,
6                             train=True,
7                             download=True)
8  test_set = datasets.MNIST(root=data_path,
9                            train=False,
10                           download=True)
11 print(f"Samples in training set: {len(train_set)}")
12 print(f"Samples in testing set: {len(test_set)}")
13
14 image, label = train_set[0]
15 plt.imshow(image, cmap='gray')
16 plt.title(f"Sample label: {label}")
17 plt.show()
```



and we can access each sample one index at a time on the dataset object

Dataset transformations

```
1  from torchvision import datasets, transforms
2
3  data_path = 'data'
4  transformed_train_set = datasets.MNIST(root=data_path,
5                                         train=True,
6                                         download=True,
7                                         transform=transforms.ToTensor())
8  transformed_test_set = datasets.MNIST(root=data_path,
9                                        train=False,
10                                       download=True,
11                                       transform=transforms.ToTensor())
12 sample, label = transformed_train_set[0]
13 print(type(sample), sample.dtype, sample.shape)
```

```
<class 'torch.Tensor'> torch.float32 torch.Size([1, 28, 28])
```

We can also apply the transform to the whole dataset at loading time

Loading the data in batches

Even with a dataset of small images it can be hard to load in memory all the dataset at once.

In PyTorch we can use the DataLoader class to dynamically load a given subset of samples at a time.

We can specify the batch size and optionally shuffle the samples before loading them.

```
from torch.utils.data import DataLoader

train_loader = DataLoader(transformed_train_set,
                          batch_size=64,
                          shuffle=True)
test_loader = DataLoader(transformed_test_set,
                         batch_size=64,
                         shuffle=True)
```

Data Loaders

The dataloader can become an iterator by calling the iter function:

```
samples, labels = next(iter(train_loader))
print(samples.shape, labels.shape)
```

```
torch.Size([64, 1, 28, 28]) torch.Size([64])
```

Or also by putting it in a for loop:

```
for samples, labels in train_loader:
    print(samples.shape, labels.shape)
    break #avoid doing the for loop
```

```
torch.Size([64, 1, 28, 28]) torch.Size([64])
```

Training a Neural Network to recognize digits

MLP for digit recognition

Similar to what already seen in the last chapter, we can use an MLP by considering each B/W pixel value as a feature.

Since each image has dimension 28x28=784 we need 784 neurons in the first layer, while we need 10 neurons for the output, one for each class (digits from 0 to 9)

For the hidden dimension we choose for example to use 10 neurons.

MLP for digit recognition

```python
from torch import nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(784, 512)
        self.linear2 = nn.Linear(512, 10)
        self.activation = nn.ReLU()

    def forward(self, x):
        # we have to flatten the samples that are 28x28
        x = x.view(-1, 784)
        x = self.activation(self.linear1(x))
        x = self.linear2(x)
        return x
```

The view operator allow to efficiently change the shape while maintaining the same underlying data structure

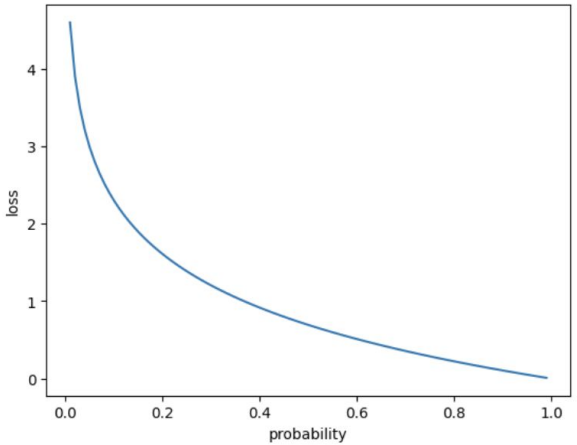
Choosing the loss function

In case of multi-class classification, the common choice is to use the Cross-Entropy loss.

$$H(\hat{y}, y) = \sum_{k \in \mathcal{Y}} -y_k \cdot log(\hat{y}_k)$$

By automatically adopting a one-hot encoding for the labels, the Cross-Entropy Loss:

- force high score on the output neuron corresponding to the correct class
- force low score on the remaining ones



The Training Pipeline

Now we are ready to train our model! For the sake of modularity we can define the function training_pipeline. For now, we use SGD and CrossEntropyLoss by default, but we can always change these two components whenever we need.

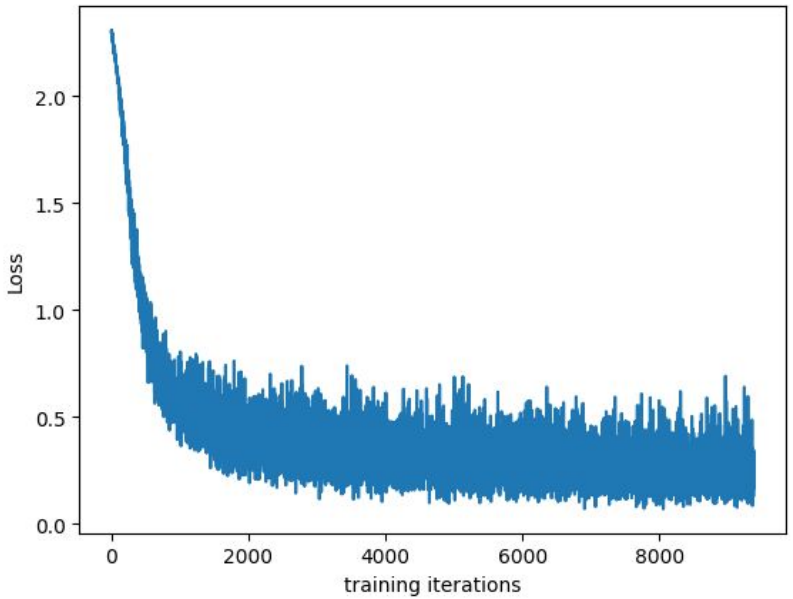
```
from torch.optim import SGD
from torch.nn import CrossEntropyLoss

def training_pipeline(model,
                      train_loader,
                      epochs=10,
                      learning_rate=1e-2
                      ):
    optimizer = SGD(model.parameters(), lr=learning_rate)
    loss_fn = CrossEntropyLoss()
```

```
    model.cuda()       # send model parameters to the GPU
    model.train()       # set the training mode
    loss_path = []      # list to store the running loss values
    # iterations over the whole train_set
    for epoch in range(epochs):
        # iterations over each batch
        for batch_idx, (samples, labels) in enumerate(train_loader):
            # send the data to the GPU
            samples = samples.cuda()
            labels = labels.cuda()

            out = model(samples)        # forward pass
            loss = loss_fn(out, labels) # compute the loss
            loss.backward()             # backpropagate the gradients
            optimizer.step()            # update the parameters
            optimizer.zero_grad()       # reset the gradients

            loss_path.append(loss.item())   # store last loss value
            # Debugging prints every N iterations
            if batch_idx % 100 == 0:
                print(f"Epoch [{epoch + 1}/{epochs}], "\
                      f"Batch [{batch_idx + 1}/{len(train_loader)}], "\
                      f"Loss: {loss.item()}")
    return loss_path
```

```
model = MLP()
loss_path = training_pipeline(model, train_loader,
                              epochs=10, learning_rate=1e-2)
```

```
Epoch [1/10], Batch [1/938], Loss: 2.2991743087768555
Epoch [1/10], Batch [101/938], Loss: 2.057236671447754
Epoch [1/10], Batch [201/938], Loss: 1.7656588554382324
Epoch [1/10], Batch [301/938], Loss: 1.3940001726150513
Epoch [1/10], Batch [401/938], Loss: 0.9924795627593994
Epoch [1/10], Batch [501/938], Loss: 0.9198691844940186
Epoch [1/10], Batch [601/938], Loss: 0.7775295972824097
Epoch [1/10], Batch [701/938], Loss: 0.8085736036300659
Epoch [1/10], Batch [801/938], Loss: 0.5896978378295898
Epoch [1/10], Batch [901/938], Loss: 0.6162244081497192
Epoch [2/10], Batch [1/938], Loss: 0.506310760974884
Epoch [2/10], Batch [101/938], Loss: 0.6563737988471985
Epoch [2/10], Batch [201/938], Loss: 0.5900071263313293
Epoch [2/10], Batch [301/938], Loss: 0.4526040554046631
Epoch [2/10], Batch [401/938], Loss: 0.5129085183143616
Epoch [2/10], Batch [501/938], Loss: 0.4644626975059509
Epoch [2/10], Batch [601/938], Loss: 0.5175845623016357
Epoch [2/10], Batch [701/938], Loss: 0.23565511405467987
Epoch [2/10], Batch [801/938], Loss: 0.30298447608947754
Epoch [2/10], Batch [901/938], Loss: 0.5376992225646973
```

```
1  import matplotlib.pyplot as plt
2  plt.plot(loss_path)
3  plt.xlabel('training iterations')
4  plt.ylabel('Loss')
```



Evaluating the model

The most common metric to measure performance in a multi-class problem is the **accuracy metric**, *i.e.*, the fraction of correct samples among the dataset used to test.

Here we can measure the accuracy on the test set, composed of digits that the model have never seen during training.

Let's define the `evaluate_accuracy` function:

Evaluating the model

```python
def evaluate_accuracy(model, test_loader):
    correct = 0
    n_samples = 0
    model.cuda()    # put the model parameters in the GPU
    model.eval()    # set the eval mode
    with torch.no_grad():
        for samples, labels in test_loader:
            # load data in the GPU
            samples = samples.cuda()
            labels = labels.cuda()
            out = model(samples)
            y_preds = out.argmax(dim=1)
            correct += (y_preds == labels).sum()
            n_samples += samples.shape[0]
    return correct / n_samples
```

Evaluating the model

```
1  accuracy = evaluate_accuracy(model, test_loader)
2  print(f"Accuracy on the test set: {accuracy.item():.3f}")
```

```
Accuracy on the test set: 0.931
```

With an MLP we classify correctly the 93.1% of the samples in the test set.

Question: do you think it can still perform that well if we shift the pixels by 3 positions?

```python
from torchvision.datasets import MNIST
from torchvision.transforms import Compose, ToTensor, Lambda
from torch.utils.data import DataLoader

def shift_pixels(t, shift=3):
    # shift the tensor of <shift> pixels
    return torch.roll(t, shift)

transforms = Compose([ToTensor(), Lambda(shift_pixels)])
shifted_test_set = datasets.MNIST(root=data_path,
                                  train=False,
                                  download=True,
                                  transform=transforms)
shifted_test_loader = DataLoader(shifted_test_set,
                                 batch_size=64,
                                 shuffle=False)

accuracy = evaluate_accuracy(model, shifted_test_loader)
print(f"Accuracy on the shifted test set: {accuracy.item():.3f}")
```

```
Accuracy on the shifted test set: 0.467
```

Why the accuracy dropped from 93.1% to 46.7% ???

Limits of Fully Connected models

With fully-connected layers, we are making every pixel count independently, and interact with any other pixel in the combination of the next layer.

In other words, we aren't utilizing the **relative position of neighboring or far-away pixels**, since we are treating the image as one big vector of numbers.

Most importantly, if we shift the same image by one pixel or more in any direction, the relationships between pixels will have to be relearned from scratch.

How can we solve it?

We could augment the training data and the number of parameters, to force the network in learning more meaningful representations…

Or we could search for a model that is invariant to image transformations by construction.

How can we solve it?

Goal: weighted sum of a pixel with its immediate neighbors (rather than with all other pixels in the image, as we do with fully-connected layers)

- This would be equivalent to building weight matrices, in which all weights beyond a certain distance from a center pixel are zero
- This will still be a weighted sum: that is, a linear operation
- we want weights to operate in neighborhoods to respond to local patterns, and local patterns to be identified no matter where they occur in the image

How can we solve it?

Of course, this approach is more than impractical. Fortunately, there is a readily available, local, translation-invariant linear operation on the image: **a (discrete)* convolution**.

*continuous convolutions are beyond the scope of this course, but you might have seen them in other courses (they use integrals)

Convolutions

The important concept: Convolutions deliver locality and translation invariance

A discrete convolution is defined for a 2D image as an operator that performs for each pixel, for all a **scalar product** between a **weight matrix** (also called **filter**, or **kernel**), and a subset of contiguous pixels centered on that pixels.

Let's see how a convolution looks like with a simple standard filter.

Check out this video to understand well how convolutions works!

Convolutions: Example

Let's use a 3 x 3 **mean filter**. This filter:

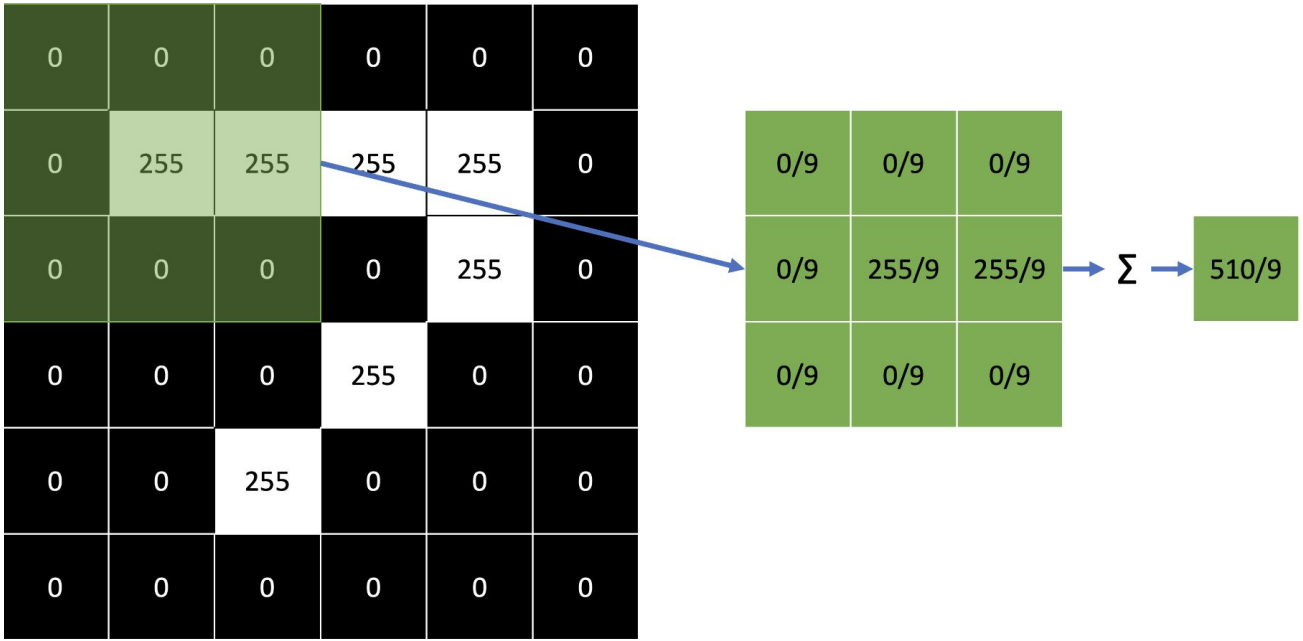
- simply multiplies every pixel by 1/9;
- and sums all results, placing them in the resulting pixel.

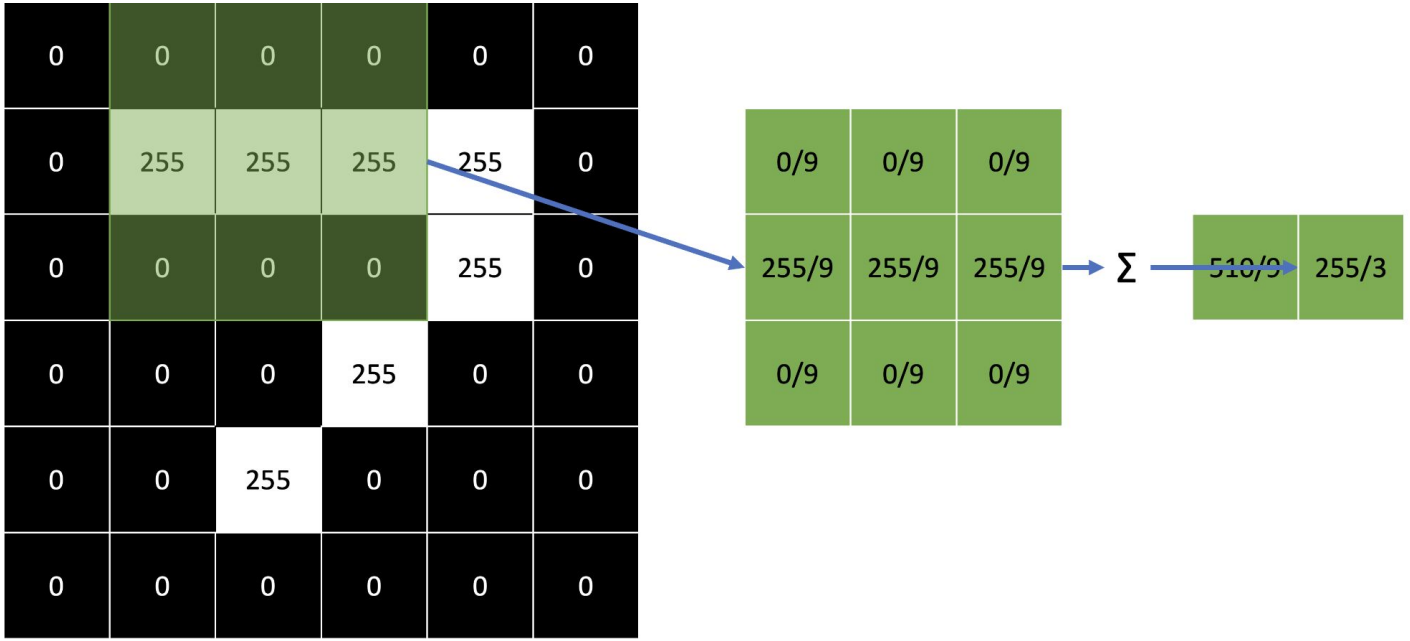
This corresponds to the averaging operation.

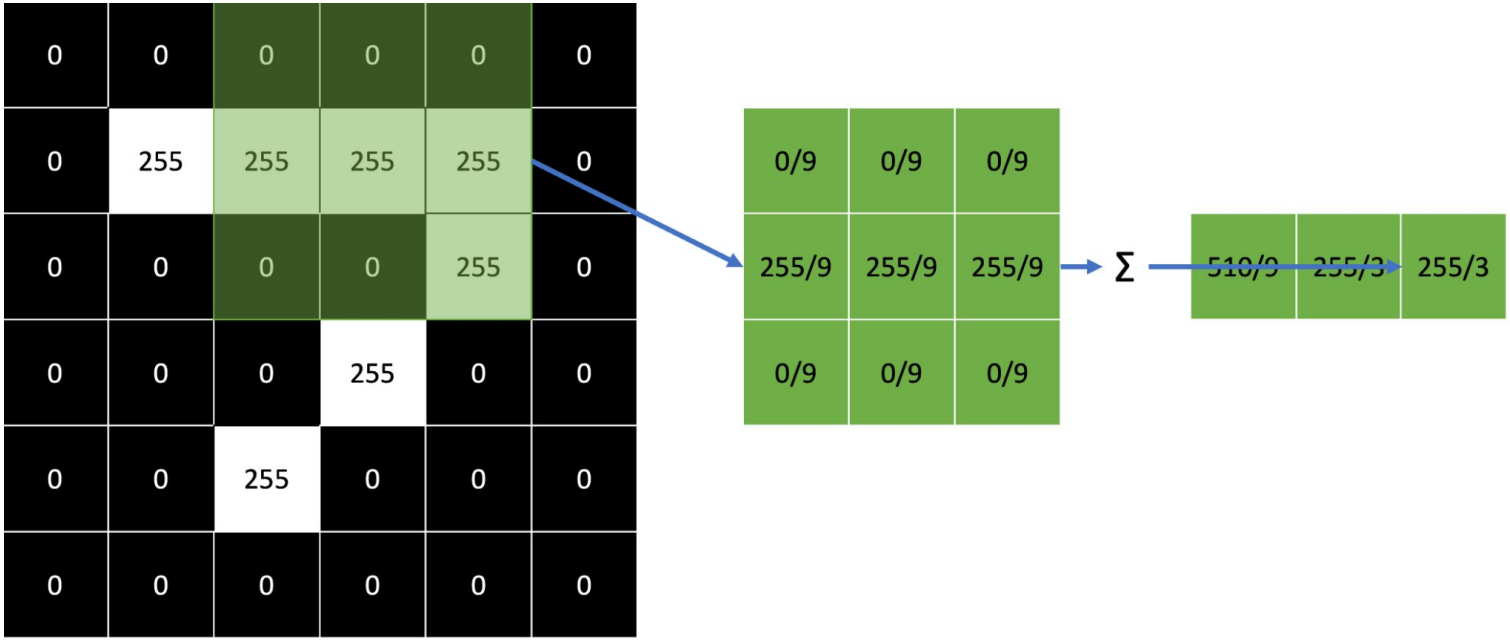
Then, the filter slides of one position and computes again the average.

0	0	0	0	0	0
0	255	255	255	255	0
0	0	0	0	255	0
0	0	0	255	0	0
0	0	255	0	0	0
0	0	0	0	0	0

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9


0 0 0 0 0 0
0 255 255 255 255 0
0 0 0 0 255 0
0 0 0 255 0 0
0 0 255 0 0 0
0 0 0 0 0 0
0/9 0/9 0/9
0/9 255/9 255/9
0/9 0/9 0/9
Σ
510/9


0 0 0 0 0 0
0 255 255 255 255 0
0 0 0 0 255 0
0 0 0 255 0 0
0 0 255 0 0 0
0 0 0 0 0 0
0/9 0/9 0/9
255/9 255/9 255/9
0/9 0/9 0/9
Σ
510/9
255/3

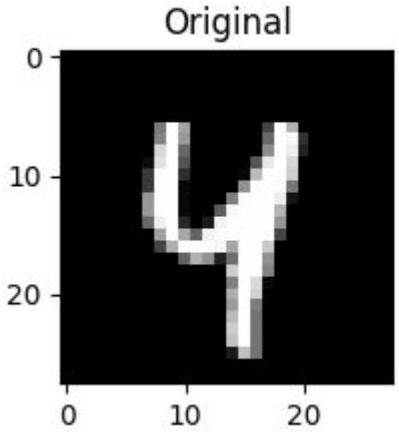

0 0 0 0 0 0
0 255 255 255 255 0
0 0 0 0 255 0
0 0 0 255 0 0
0 0 255 0 0 0
0 0 0 0 0 0
0/9 0/9 0/9
255/9 255/9 255/9
0/9 0/9 0/9
Σ
510/9 255/3 255/3

```python
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np
import cv2

test_set = datasets.MNIST(root='data', train=False, download=True)
image, label = test_set[42]

fig, axs = plt.subplots(nrows=1, ncols=3)

axs[0].imshow(image, cmap='gray')
axs[0].set_title('Original')

# Mean filter
weight = np.array([[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[1].imshow(filtered_image, cmap='gray')
axs[1].set_title('Mean Filter')

# X-derivative filter
weight = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[2].imshow(filtered_image, cmap='gray')
axs[2].set_title('Gradient Filter')

fig.tight_layout()
fig.show()
```

Load an image from the dataset

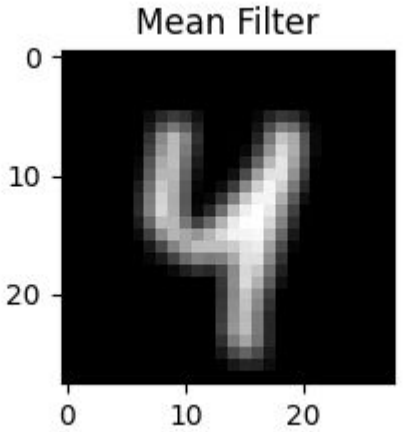
```python
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np
import cv2

test_set = datasets.MNIST(root='data', train=False, download=True)
image, label = test_set[42]

fig, axs = plt.subplots(nrows=1, ncols=3)

axs[0].imshow(image, cmap='gray')
axs[0].set_title('Original')

# Mean filter
weight = np.array([[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[1].imshow(filtered_image, cmap='gray')
axs[1].set_title('Mean Filter')

# X-derivative filter
weight = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[2].imshow(filtered_image, cmap='gray')
axs[2].set_title('Gradient Filter')

fig.tight_layout()
fig.show()
```

Show the original image



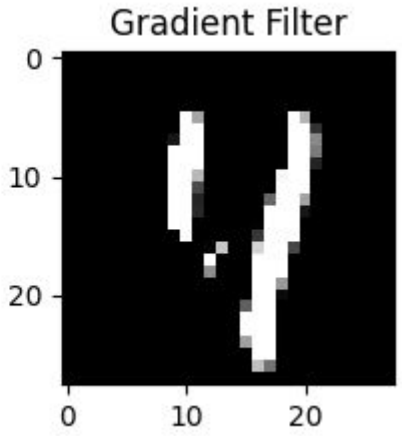
```python
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np
import cv2

test_set = datasets.MNIST(root='data', train=False, download=True)
image, label = test_set[42]

fig, axs = plt.subplots(nrows=1, ncols=3)

axs[0].imshow(image, cmap='gray')
axs[0].set_title('Original')

# Mean filter
weight = np.array([[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[1].imshow(filtered_image, cmap='gray')
axs[1].set_title('Mean Filter')

# X-derivative filter
weight = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[2].imshow(filtered_image, cmap='gray')
axs[2].set_title('Gradient Filter')

fig.tight_layout()
fig.show()
```

With the mean filter we obtain a blurred version of the image



```python
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np
import cv2

test_set = datasets.MNIST(root='data', train=False, download=True)
image, label = test_set[42]

fig, axs = plt.subplots(nrows=1, ncols=3)

axs[0].imshow(image, cmap='gray')
axs[0].set_title('Original')

# Mean filter
weight = np.array([[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[1].imshow(filtered_image, cmap='gray')
axs[1].set_title('Mean Filter')

# X-derivative filter
weight = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]])
filtered_image = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)
axs[2].imshow(filtered_image, cmap='gray')
axs[2].set_title('Gradient Filter')

fig.tight_layout()
fig.show()
```

While with the gradient filter we can highlight the edges



Note that this is a convolution with only one channel. To have convolution for RGB images, the lter has to have one kernel for each channel.

Also, there are convolutions also for 1-D and 3-D data. For this course we focus only on the 2-D convolutions.

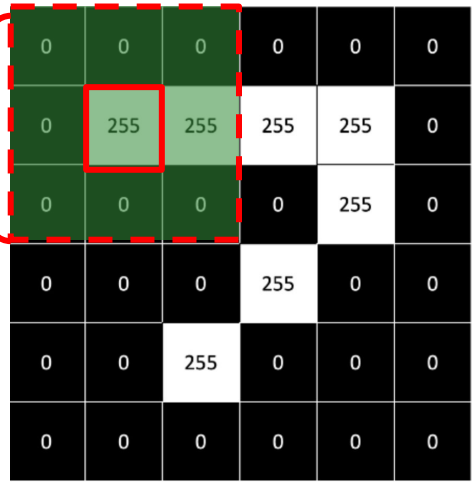
Convolution Parameters

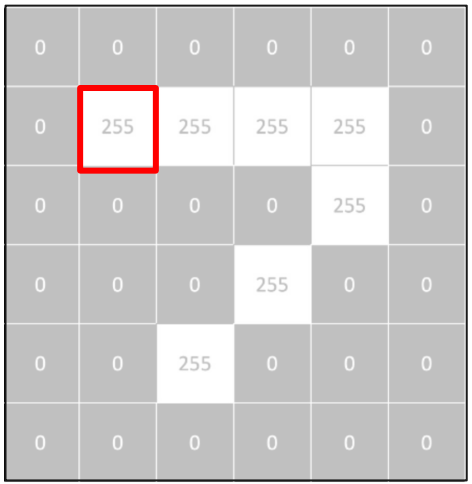
Applying a convolution kernel as a weighted sum of pixels in a 3x3 neighborhood requires that there are neighbors in all directions. To control better the behavior of our filters, we can specify a few parameters:

- **kernel size**: size of the kernel filter $k \times k$
- **stride**: how much the filter moves between one convolution and the other
- **padding**: how much padding to add to the image (to apply convolutions to the border) - adds zeros on the outside

Kernel Size

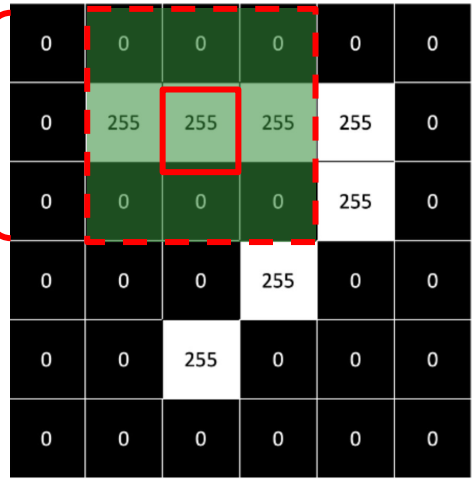
K = 3

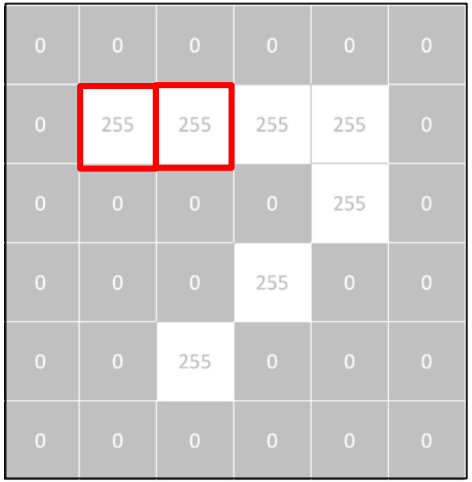




Kernel Size

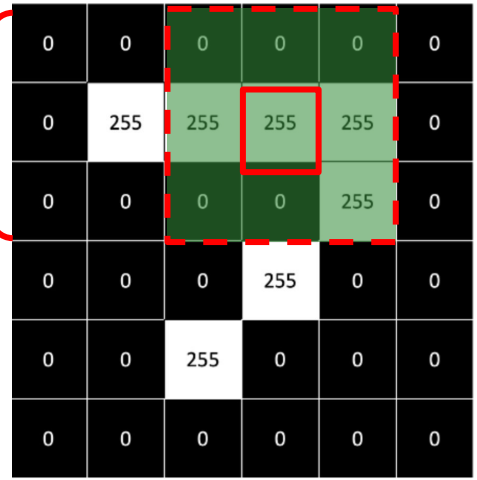
K = 3

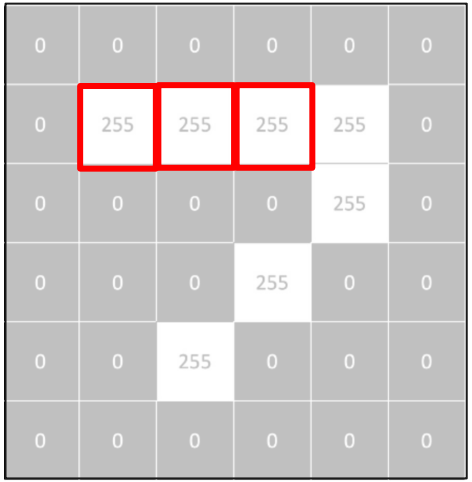




Kernel Size

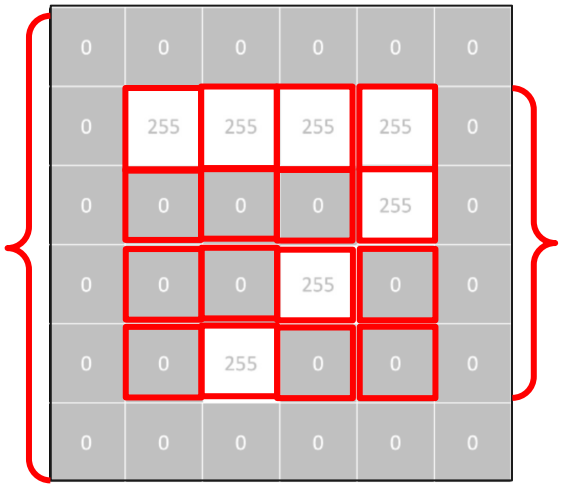
K = 3





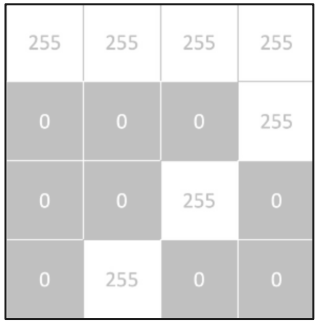
Kernel Size

Input size = 6 x 6

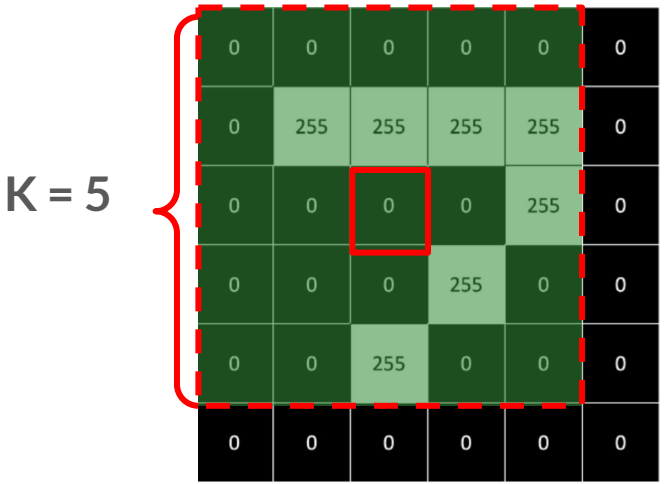


Output size = 4 x 4

Kernel Size

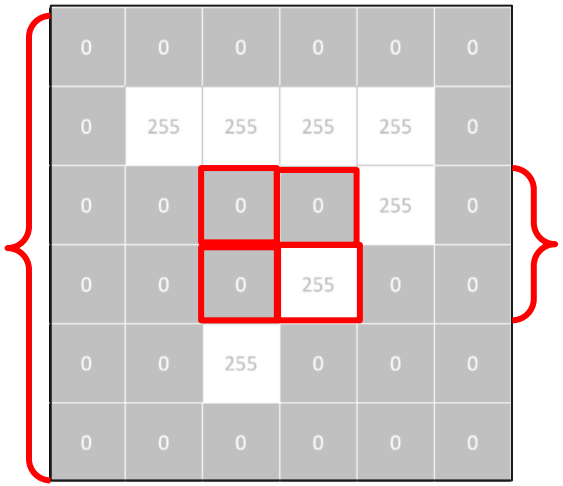


Kernel Size


K = 5
0 0 0 0 0 0
0 255 255 255 255 0
0 0 0 0 255 0
0 0 0 255 0 0
0 0 255 0 0 0
0 0 0 0 0 0

Kernel Size

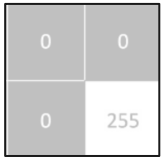
Input size = 6 x 6



Output size = 2 x 2

Kernel Size

In general, if W is the dimension of the input, the output dimension Z scale down to **W - K + 1**



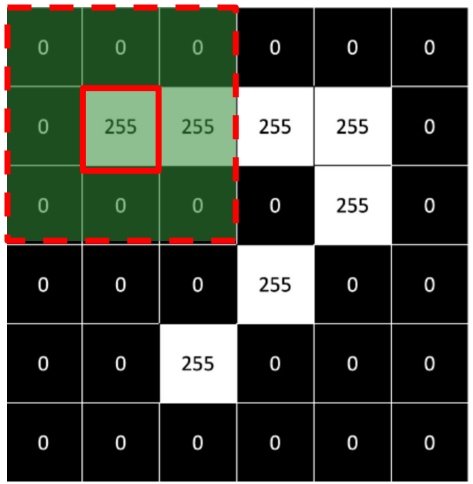
Padding

To keep the same dimension in output we can simply add pixels in the input image.

$Z + K - 1 = W + K - 1 - K + 1$

$Z + K - 1 = W' - K + 1$

$W' = W + K - 1$



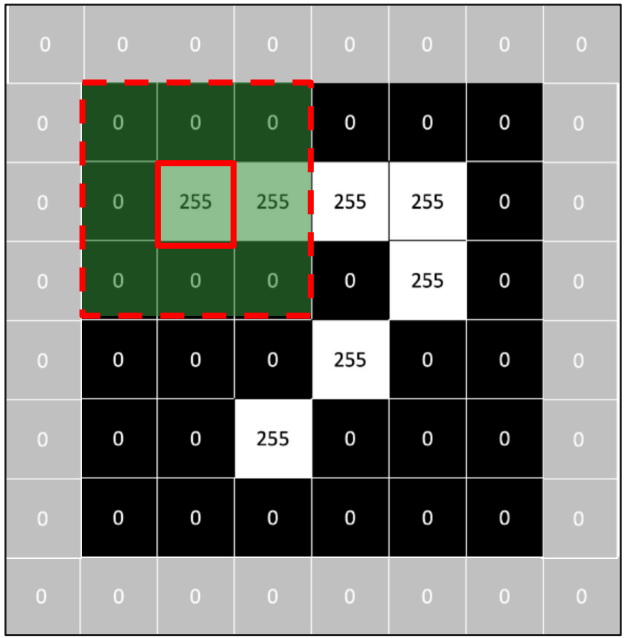
Padding

To keep the same dimension in output we can simply add pixels in the input image.

$Z + K - 1 = W + K - 1 - K + 1$

$Z + K - 1 = W' - K + 1$

$W' = W + K - 1$



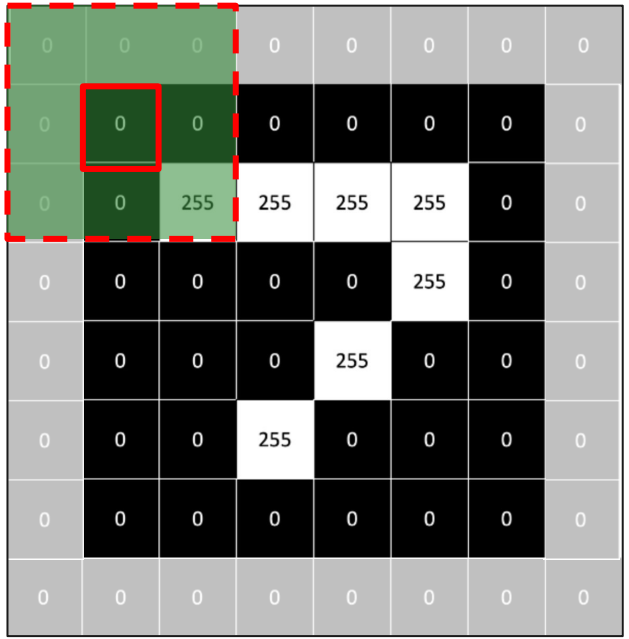
Padding

To keep the same dimension in output we can simply add pixels in the input image.

$Z + K - 1 = W + K - 1 - K + 1$

$Z + K - 1 = W' - K + 1$

$W' = W + K - 1$



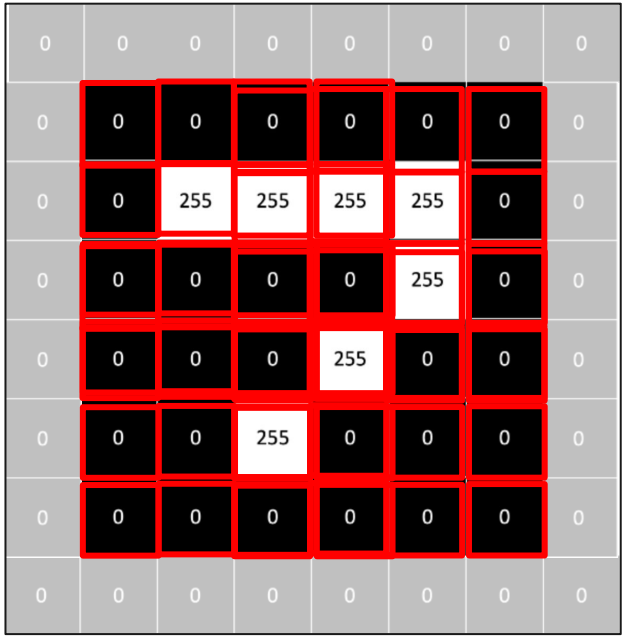
Padding

To keep the same dimension in output we can simply add pixels in the input image.

$Z + K - 1 = W + K - 1 - K + 1$

$Z + K - 1 = W' - K + 1$

$W' = W + K - 1$



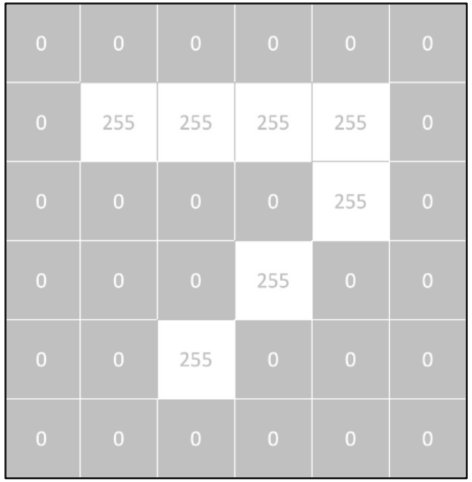
Padding

To keep the same dimension in output we can simply add pixels in the input image.

$Z + K - 1 = W + K - 1 - K + 1$

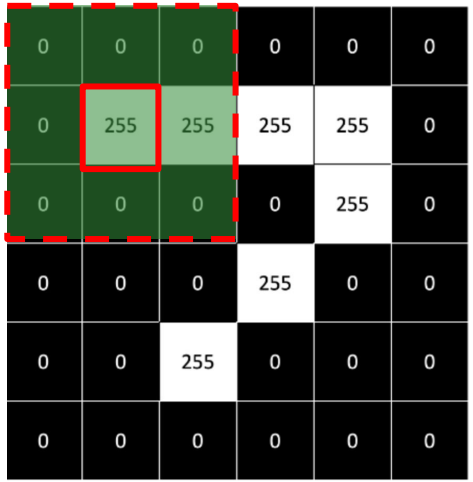
$Z + K - 1 = W' - K + 1$

$W' = W + K - 1$



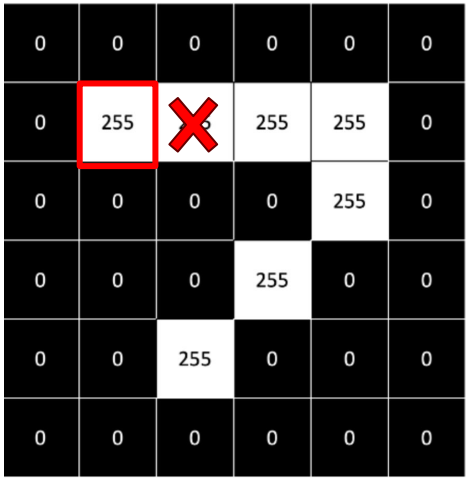
Stride

By applying the convolution we can also **skip pixels** to reduce the size of an image, especially for large ones.



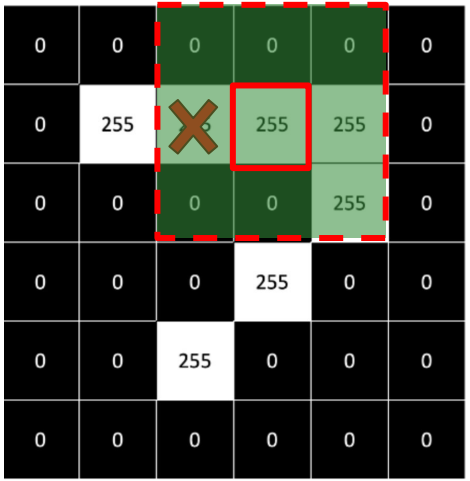
Stride

By applying the convolution we can also **skip some pixels** to reduce the size of an image, especially for large ones.



Stride

By applying the convolution we can also **skip pixels** to reduce the size of an image, especially for large ones.



Output size of a convolution

To compute the output size, we have to take into account all the parameters:

$$Z = \frac{W - K + 2P}{S} + 1$$

- Z = Ouput size
- W = Input size
- K = Filter size
- S = Stride
- P = Padding

Convolutions in PyTorch

```
1  import torch
2  from torch import nn
3
4  conv = nn.Conv2d(in_channels=1, out_channels=1,
5                   kernel_size=3, padding=0, stride=1)
6  with torch.no_grad():
7      conv.weight[:] = torch.tensor([[1/9, 1/9, 1/9],
8                                     [1/9, 1/9, 1/9],
9                                     [1/9, 1/9, 1/9]])
10     conv.bias.zero_()
11
12 from torchvision import transforms
13 image_as_tensor = transforms.ToTensor()(image)
14 convolution = conv(image_as_tensor).detach().numpy()[0]
15
16 plt.figure()
17 plt.imshow(convolution, cmap='gray')
```

Instantiate the nn.Conv2d module with its hyper-parameters (kernel size, padding and stride).

This module acts as a linear layer internally, where the weight are arranged to reproduce exactly the convolution operator.

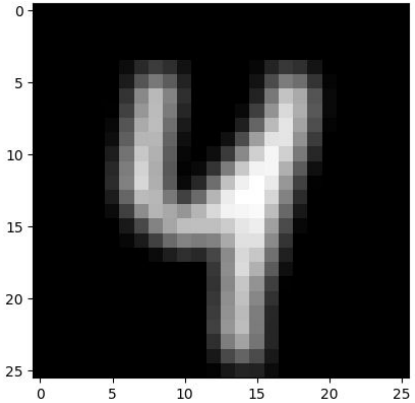
Convolutions in PyTorch

```
1  import torch
2  from torch import nn
3
4  conv = nn.Conv2d(in_channels=1, out_channels=1,
5                   kernel_size=3, padding=0, stride=1)
6  with torch.no_grad():
7      conv.weight[:] = torch.tensor([[1/9, 1/9, 1/9],
8                                     [1/9, 1/9, 1/9],
9                                     [1/9, 1/9, 1/9]])
10     conv.bias.zero_()
11
12 from torchvision import transforms
13 image_as_tensor = transforms.ToTensor()(image)
14 convolution = conv(image_as_tensor).detach().numpy()[0]
15
16 plt.figure()
17 plt.imshow(convolution, cmap='gray')
```

Without extending the computational graph, we manually insert the weights …

Convolutions in PyTorch

```
1  import torch
2  from torch import nn
3
4  conv = nn.Conv2d(in_channels=1, out_channels=1,
5                   kernel_size=3, padding=0, stride=1)
6  with torch.no_grad():
7      conv.weight[:] = torch.tensor([[1/9, 1/9, 1/9],
8                                     [1/9, 1/9, 1/9],
9                                     [1/9, 1/9, 1/9]])
10     conv.bias.zero_()
11
12 from torchvision import transforms
13 image_as_tensor = transforms.ToTensor()(image)
14 convolution = conv(image_as_tensor).detach().numpy()[0]
15
16 plt.figure()
17 plt.imshow(convolution, cmap='gray')
```



… leading to the same results obtained with the mean filter

Extracting features with convolutions

Moving from fully connected layers to convolutions, we achieve locality and translation invariance. However, the network needs to get multiple shapes and patterns into its neurons. We have to refine the information extracted from the **raw pixels**.

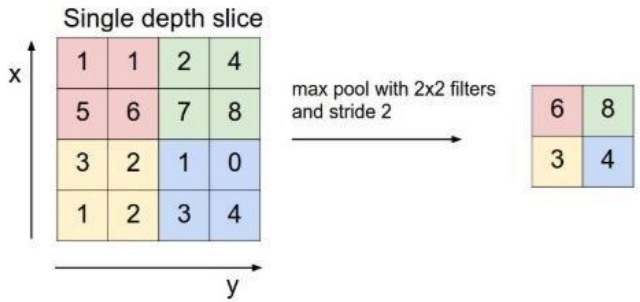
We can stack one convolution after the other and at the same time downsampling the image between successive convolutions to extract the information from the input.

From large to small: downsampling

Scaling an image by half is the equivalent of taking four neighboring pixels as input and producing one pixel as output. We can:

- **Average Pooling** - Average the four pixels
- **Max Pooling** - Take the makimum of the four pixels
- **Strided convolution** - reduces the pixels as output

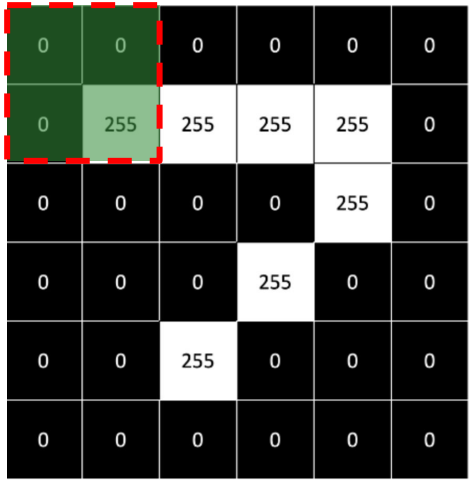
From large to small: downsampling



We will be focusing on the **max pooling**. Intuitively, the output images from a convolution layer, especially since they are followed by an activation just like any other linear layer, tend to have a high magnitude where certain features corresponding to the estimated kernel are detected.

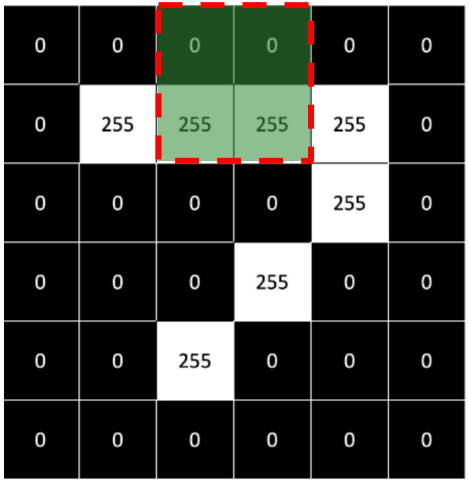
By keeping the highest value in the 2 × 2 neighborhood as the downsampled output, we ensure that the features that are found survive the downsampling, at the expense of the weaker responses.

Max Pooling



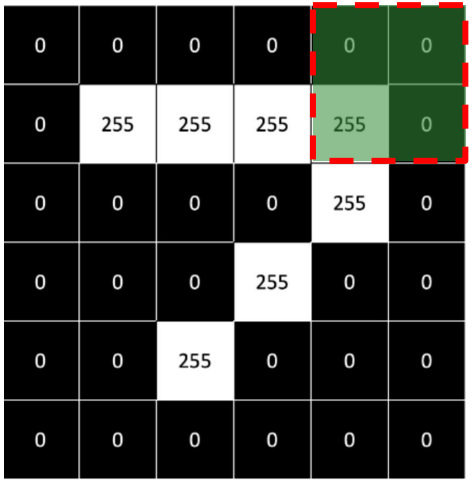
255		

Max Pooling



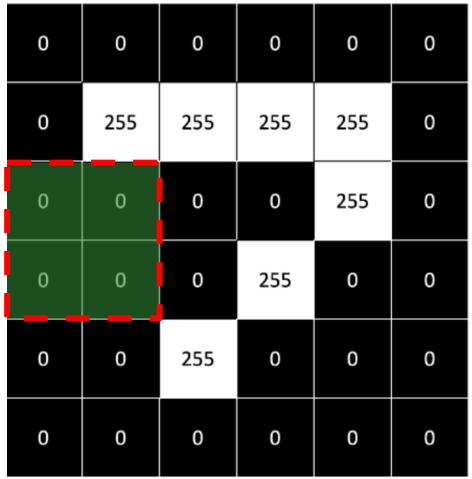
255	255	

Max Pooling



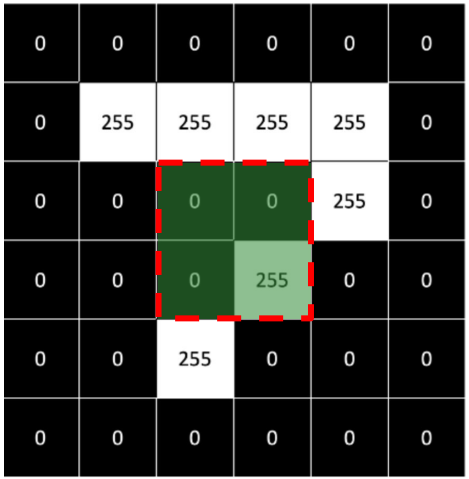
255	255	255

Max Pooling



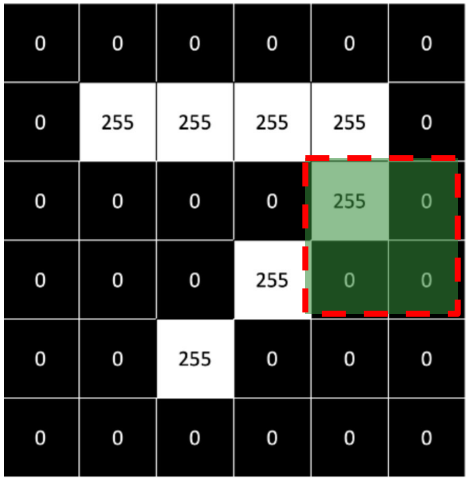
255	255	255
0		

Max Pooling



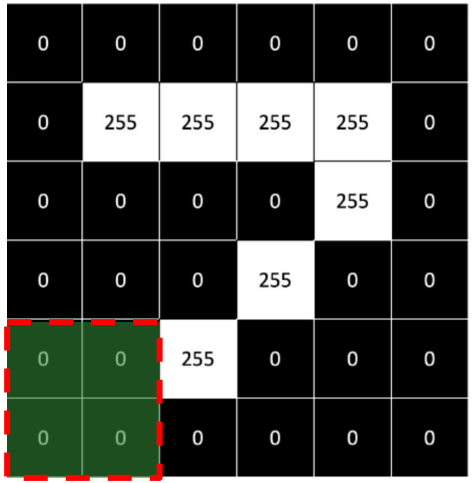
255	255	255
0	255	

Max Pooling



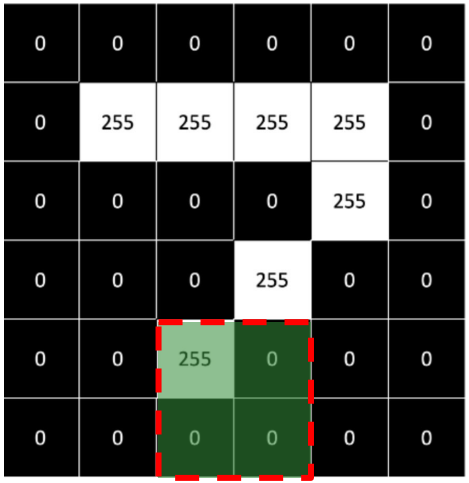
255	255	255
0	255	255

Max Pooling



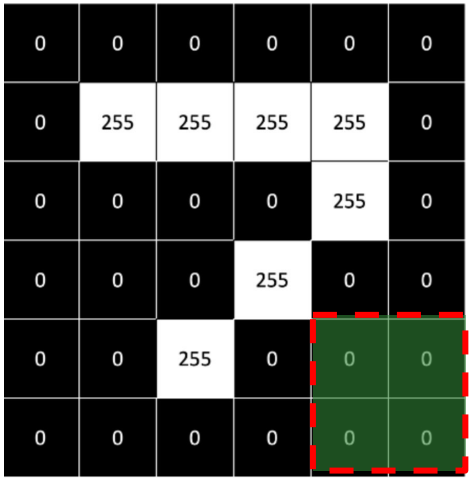
255	255	255
0	255	255
0		

Max Pooling



255	255	255
0	255	255
0	255	

Max Pooling



255	255	255
0	255	255
0	255	**0**

$Z = W / 2$

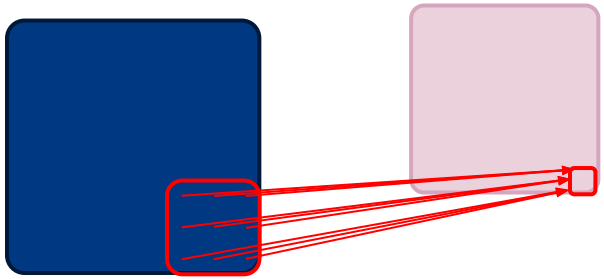
Convolutional Neural Networks (CNN)

Then, we can combine convolutions and downsampling to extract higher-level information from the images. Until now we worked with predefined filters, but what if we could **learn the best filters** to extract features that lead to lower loss values?

In deep learning, the purpose is to let the model learn by itself. This also includes the filters!

Convolutional Neural Networks (CNN)

$$W_1^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix}$$

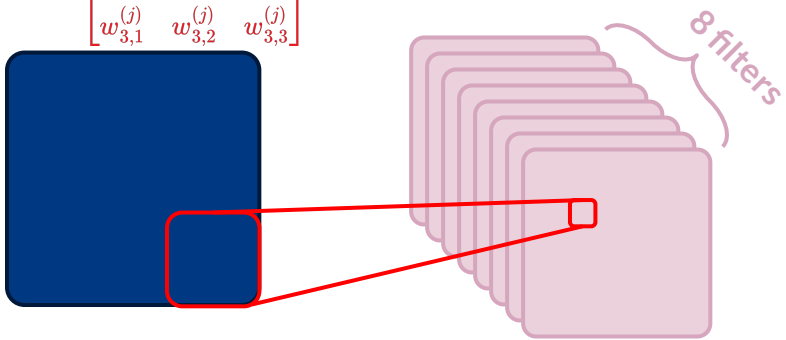


3x3 filter, replicated for each pixel

CNN: **9+1** parameters!
FC: ~**2300** parameters!

Convolutional Neural Networks (CNN)

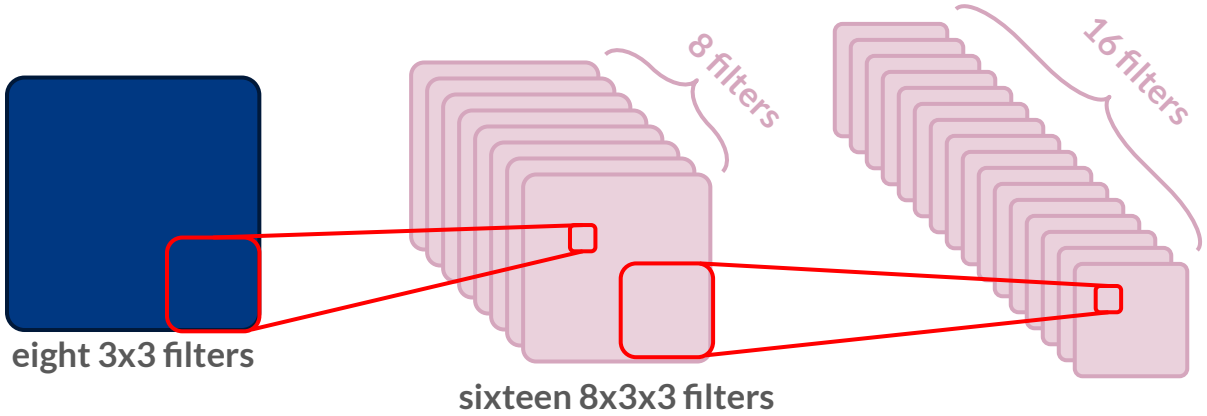
$$W_1^{(j)} = \begin{bmatrix} w_{1,1}^{(j)} & w_{1,2}^{(j)} & w_{1,3}^{(j)} \\ w_{2,1}^{(j)} & w_{2,2}^{(j)} & w_{2,3}^{(j)} \\ w_{3,1}^{(j)} & w_{3,2}^{(j)} & w_{3,3}^{(j)} \end{bmatrix}$$



eight 3x3 filters

CNN: **9+1** x 8 = **80** parameters
FC: ~**2300** x 8 ~=**18500** parameters!

Convolutional Neural Networks (CNN)



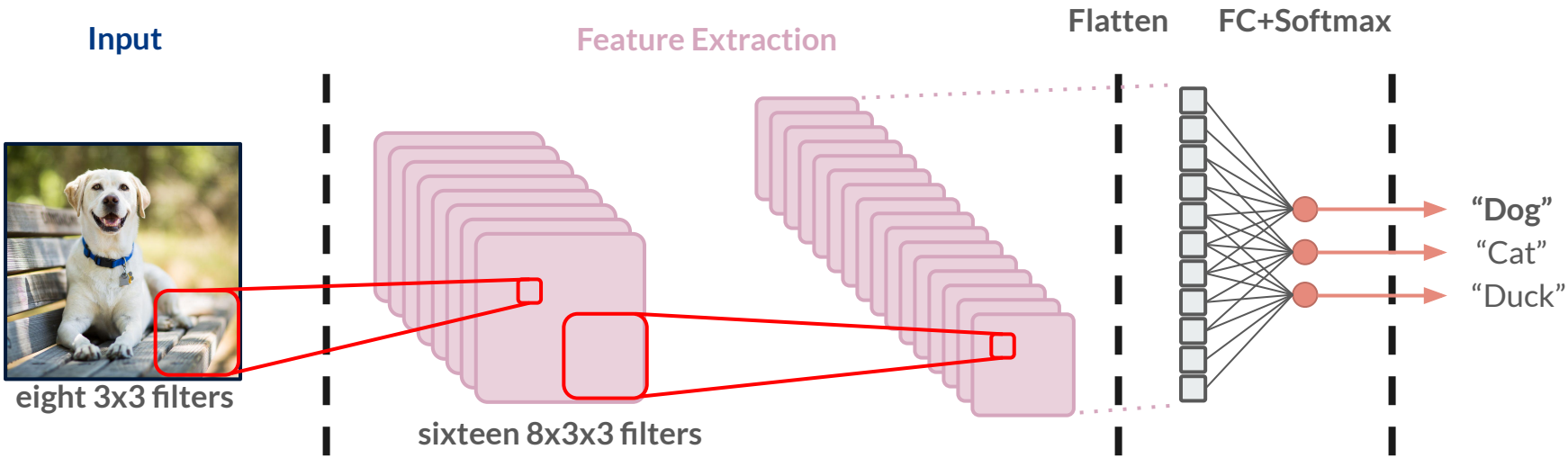
Conv Layer 1

CNN: **9+1** x 8 = **80** parameters
FC: ~**2300** x 8 ~=**18500** parameters!

Conv Layer 1

CNN: **8x3x3+1** x 16 = **1168** parameters
FC: ~**4600** x 16 ~=**73700** parameters!

Convolutional Neural Networks (CNN)



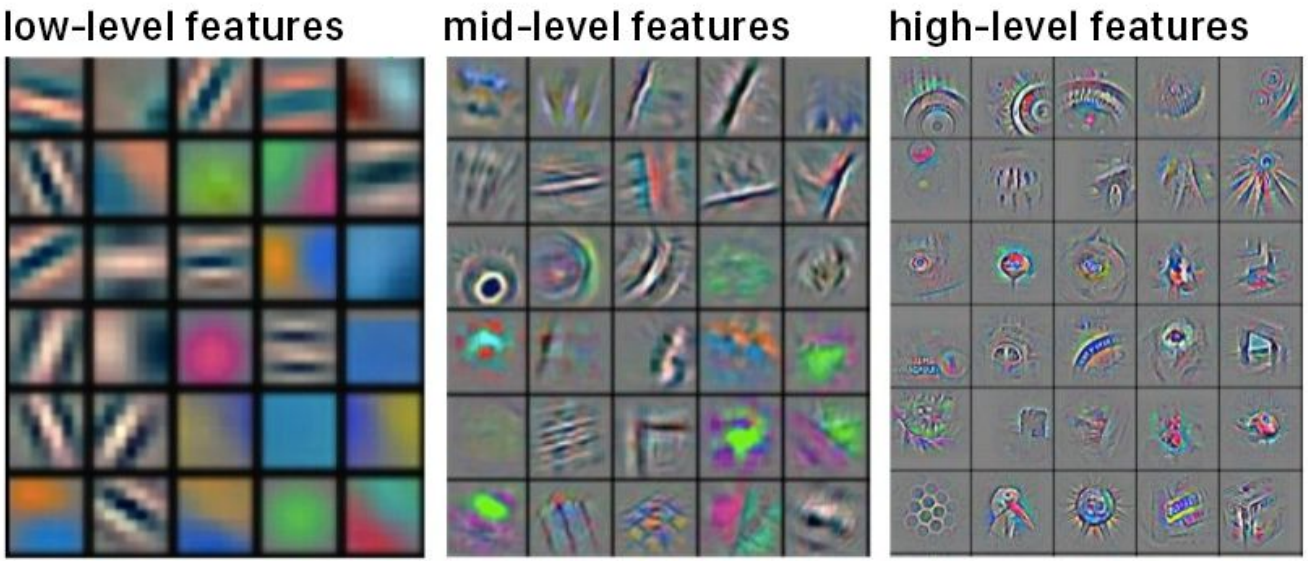
Conv Layer 1

CNN: **9+1** x 8 = **80** parameters
FC: ~**2300** x 8 ~=**18500** parameters!

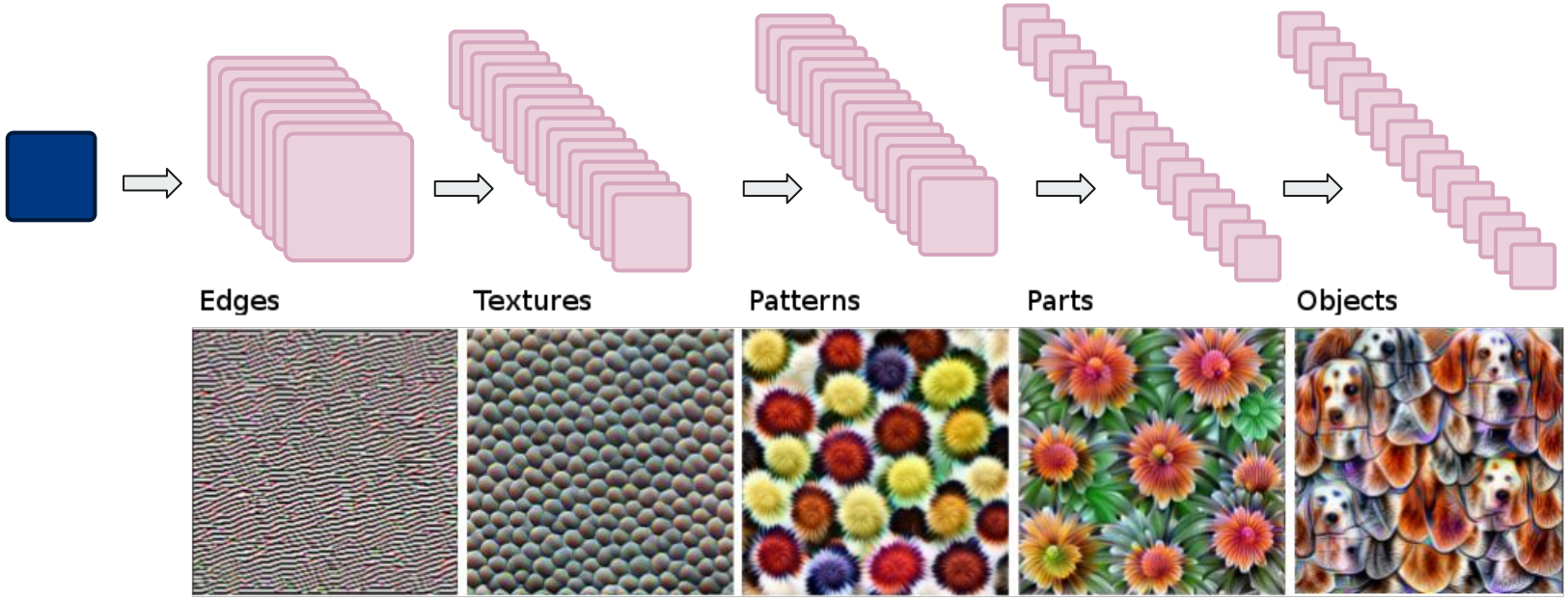
Conv Layer 1

CNN: **8x3x3+1** x 16 = **80** parameters
FC: ~**4600** x 16 ~=**73700** parameters

Feature Extraction using CNNs



Feature Extraction using CNNs



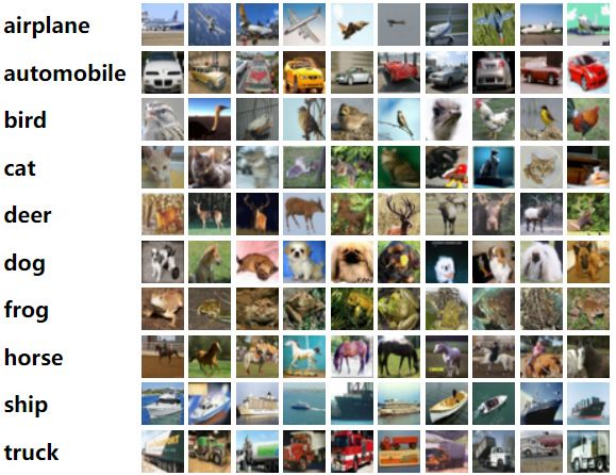
Implementing and Training a CNN

Moving to more complex images...

Let's put all items together and create our first CNN.

This time we use the CIFAR10 dataset, composed of **RGB images** of size **32 × 32**.

The training and testing set contains respectively 50.000 and 10.000 images, each belonging to 10 different classes.



```python
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
transf = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.4914, 0.4822, 0.4465),
                         std=(0.2023, 0.1994, 0.2010))
    ])
data_path = 'data'
cifar_train = datasets.CIFAR10(data_path,
                               train=True,
                               download=True,
                               transform=transf)
cifar_validation = datasets.CIFAR10(data_path,
                              train=False,
                              download=True,
                              transform=transf)
train_loader = torch.utils.data.DataLoader(cifar_train,
                                           batch_size=64,
                                           shuffle=True)
val_loader = torch.utils.data.DataLoader(cifar_validation,
                                         batch_size=64,
                                         shuffle=False)

label_to_class = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                  'dog', 'frog', 'horse', 'ship', 'truck']
print(f"Samples in training set: {len(cifar_train)}")
print(f"Samples in validation set: {len(cifar_validation)}")
```

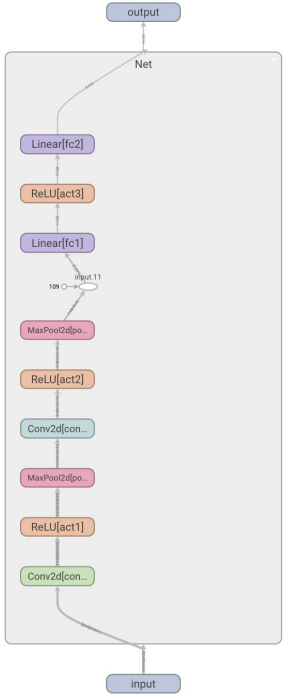
Building the CNN

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

Tensorboard

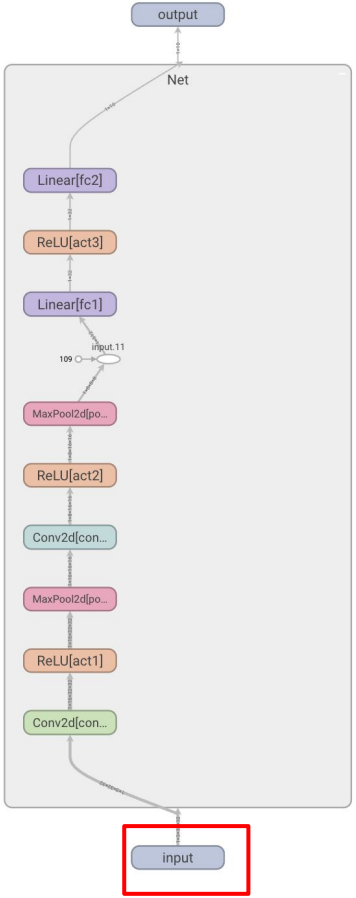
It is also possible to inspect the network by using tensorboard or other similar tools. Here is a snippet to visualize the network:

```
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



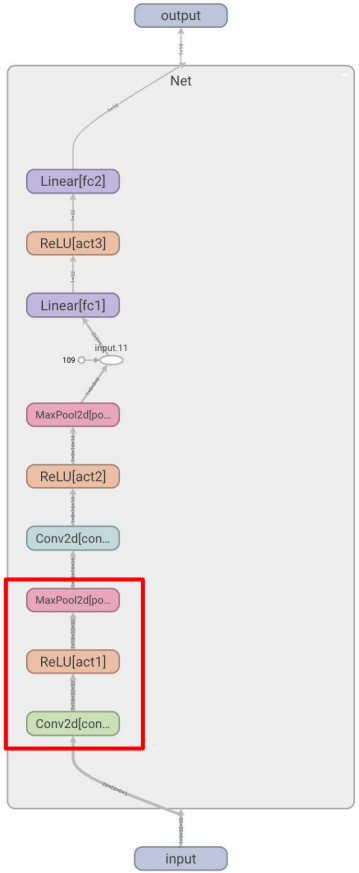
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



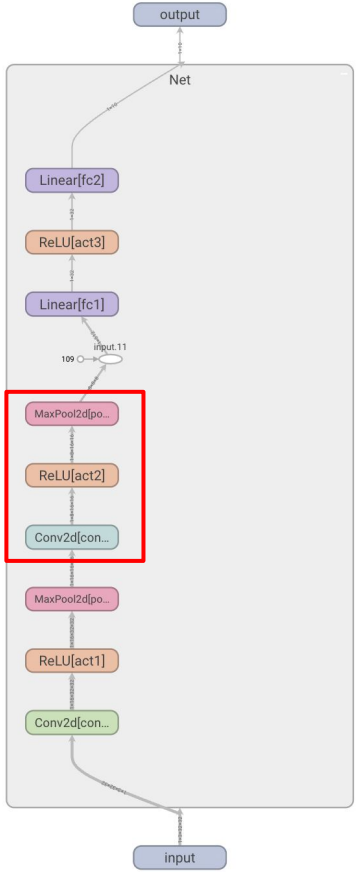
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



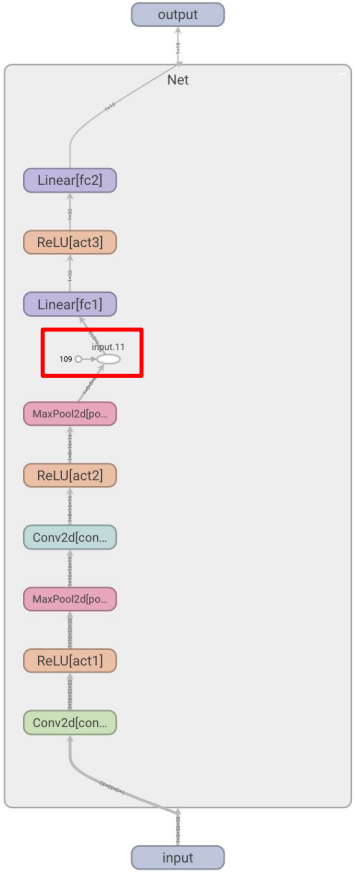
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



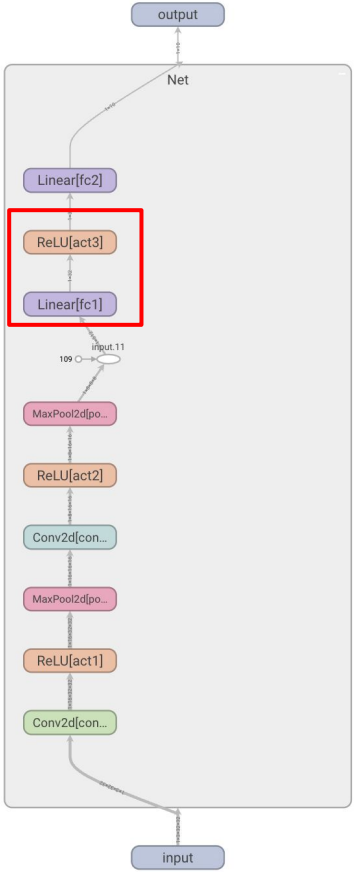
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



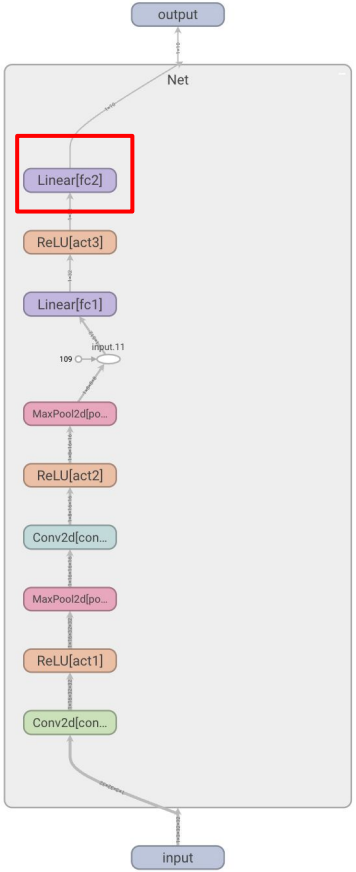
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



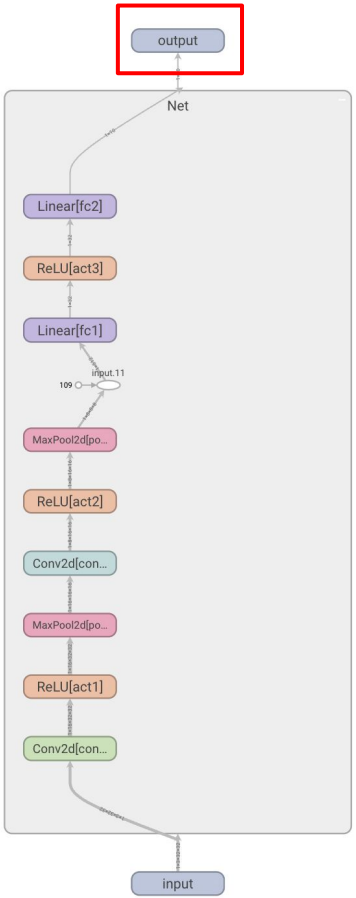
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)
    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

```python
%load_ext tensorboard
from torch.utils.tensorboard import SummaryWriter
x = torch.rand(size=(1, 3, 32, 32))
writer = SummaryWriter("logs/")
model = Net()
writer.add_graph(model, x)
writer.close()
%tensorboard --logdir logs
```



Layer Math

$$Z = \frac{W - K + 2P}{S} + 1$$

- **Input** (batch_size, 3, 32, 32)
- **Conv layer 1**, in_channels=3, K=3, P=1, out_channels=16
  - output = 16 channels of of width and height
- **MaxPool**, K=2 -> output = 16 channels of size 32/2 = 16
- **Conv layer 2**, in_channels=16, K=3, P=1, out_channels=8
  - output = 8 channels of of width and height
- **MaxPool**, K=2 -> output = 8 channels of width and height 16/2 = 8
- **Linear** 1 (fully-connected) Layer of input size $w \times h \times c = 8 \times 8 \times 8$ and 32 output units
- **Linear** 2 (fully-connected) Layer of input size 32 and 10 output as the 10 output classes
- **Output** (batch_size, 10)

Training our CNN

Now, whatever we did for the MLP before can be reused - we just have to make sure to use the other dataset and customize the parts relative to the different format of the data (and different network).

Since we introduced Tensorboard, it's time to use it for its primary role, as experiment tracker. We will add a few lines to make sure we track the results while training.

First, let's set the optimizers and hyperparameters for training

```
learning_rate = 1e-2
epochs = 10
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = Net()
net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)
loss_fn = torch.nn.CrossEntropyLoss()
```

Let’s define the sequence of operation to perform for each epoch:

```python
def train_epoch(model, train_loader, optimizer, loss_fn):
    train_loss = 0.0
    total = 0
    model.train()   # set training mode
    for samples, labels in train_loader:
        samples, labels = samples.to(device), labels.to(device)
        outputs = model(samples)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        total += samples.shape[0]
        train_loss += loss.sum().detach()
    train_loss /= total
    return train_loss.item()
```

Let’s also define a procedure to check the performance on the validation set while training:

```
def valid_epoch(model, val_loader, loss_fn):
    accuracy = 0.0
    validation_loss = 0.0
    total = 0
    model.eval()    # set eval mode
    with torch.no_grad():
        for samples, labels in val_loader:
            samples, labels = samples.to(device), labels.to(device)
            outputs = net(samples)
            loss = loss_fn(outputs, labels)
            predictions = outputs.argmax(dim=1)
            accuracy += (predictions.type(labels.dtype) == labels).float().sum()
            total += samples.shape[0]
    validation_loss += loss.sum()
    validation_loss /= total
    accuracy = accuracy / total
    return validation_loss.item(), accuracy.item()
```

Then, similarly to what we already did, let's write the loop:

```
train_losses, val_losses = [], []
for epoch in range(epochs):
    train_loss = train_epoch(net, train_loader, optimizer, loss_fn)
    val_loss, accuracy = valid_epoch(net, val_loader, loss_fn)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(f"Epoch [{epoch + 1}/{epochs}], "\
          f"Train loss: {train_loss:.4f}, "\
          f"Val. loss: {val_loss:.4f}, "\
          f"Val. accuracy: {accuracy:.4f}")
```

```
Epoch [1/10], Train loss: 0.0249, Val. loss: 0.0001, Val. accuracy: 0.5425
Epoch [2/10], Train loss: 0.0193, Val. loss: 0.0001, Val. accuracy: 0.5790
Epoch [3/10], Train loss: 0.0176, Val. loss: 0.0001, Val. accuracy: 0.6170
Epoch [4/10], Train loss: 0.0167, Val. loss: 0.0001, Val. accuracy: 0.6232
Epoch [5/10], Train loss: 0.0160, Val. loss: 0.0001, Val. accuracy: 0.6332
Epoch [6/10], Train loss: 0.0155, Val. loss: 0.0001, Val. accuracy: 0.6296
Epoch [7/10], Train loss: 0.0150, Val. loss: 0.0001, Val. accuracy: 0.6143
Epoch [8/10], Train loss: 0.0148, Val. loss: 0.0001, Val. accuracy: 0.6433
Epoch [9/10], Train loss: 0.0145, Val. loss: 0.0001, Val. accuracy: 0.6338
Epoch [10/10], Train loss: 0.0142, Val. loss: 0.0001, Val. accuracy: 0.6382
```

Saving and loading the models

We can store the model parameters in a le and reload them in a different session.

Saving and loading in PyTorch can be done with the Torch APIs, but we have to be careful.

The torch.save API uses pickle to save the Python object in memory. In theory, we could issue torch.save(net) and we can store the object somewhere in our memory.

However, this has some issues.

Saving and loading the models

If we save the model directly, we risk problems when reloading the model (as we cannot save torch.nn inside a pickle). You can find the issue well described in the PyTorch documentation.

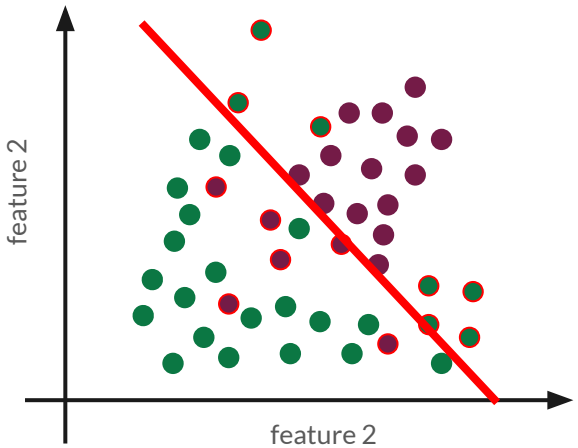
The correct way of saving the model is to save the model code in a .py file, and then use torch.save to **store the parameters** in a file. Here are the correct steps for saving and loading a model.

```
1  model_path = 'cifar_model.pt'
2  torch.save(net.state_dict(), model_path)
3
4  new_model = Net()
5  new_model.load_state_dict(torch.load(model_path))
6
7  new_model.eval()
8  val_loss, accuracy = valid_epoch(new_model, val_loader, loss_fn)
9  print("Accuracy of the loaded model: ", accuracy)
```

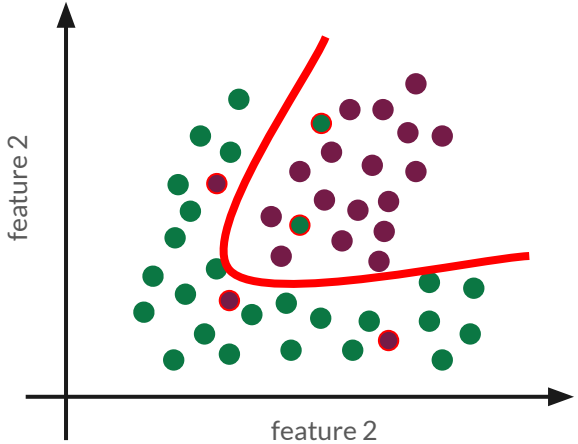
```
Accuracy of the loaded model:  0.6381999850273132
```

Overfitting

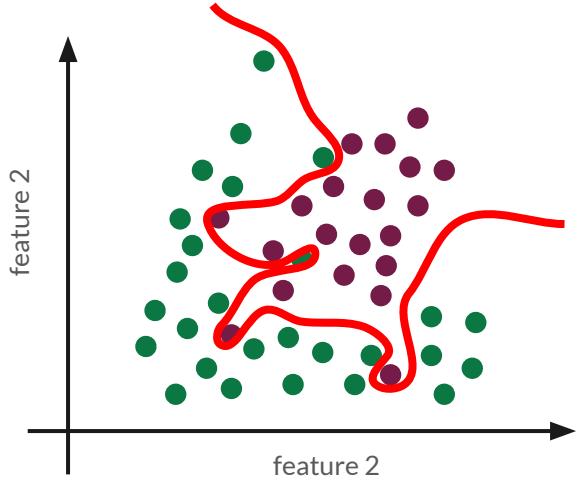
Neural Networks Suffers from high risk of Overfitting!



Underfitting



Good Fit



Overfitting

Helping the model to converge: Regularization

Training a model involves two critical steps:

- **optimization**, when we need the loss to decrease on the training set; and
- **generalization**, when the model has to work not only on the training set but also on data it has not seen before, like the validation set.

The mathematical tools aimed at easing these two steps are sometimes subsumed under the label **regularization**.

Weight Penalities

The first way to stabilize generalization is to add a regularization term to the loss.

This term is crafted so that the weights of the model tend to be small on their own, limiting how much training makes them grow. In other words, it is a penalty on larger weight values. This makes the loss have a smoother topography, and there's relatively less to gain from tting individual samples.

$$L(\mathbf{x}, y; \theta) + \lambda ||\theta||_p$$

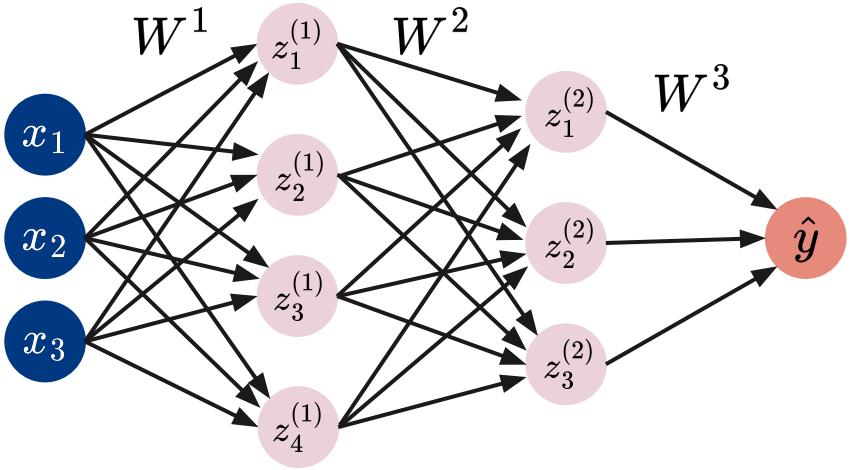
Where $||\cdot||_p$ is the ℓ_p norm. In general, the ℓ_2 or ℓ_1 norm are used.

Weight Penalities

To use, for example, an ℓ_2 penalty, one can specify the hyperparameter λ setting the weight_decay parameter in a given PyTorch optimizer.

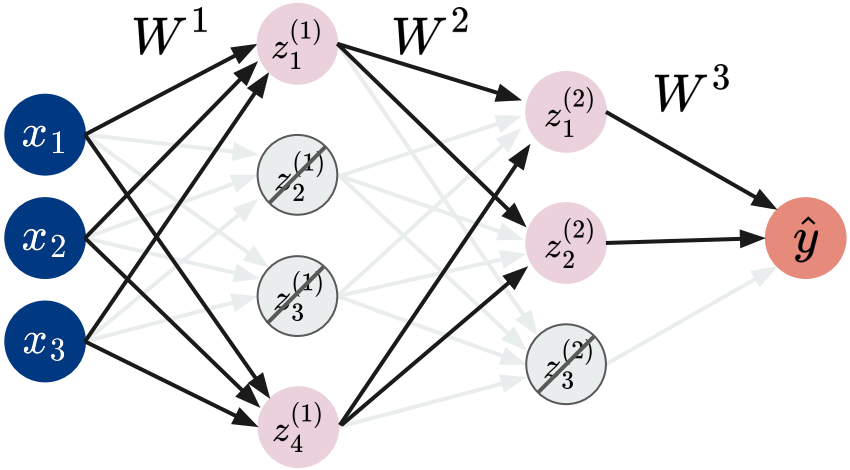
```
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                            weight_decay=0.01)
```

Dropout Regularization



- In standard training we use all the weights during training…

Dropout Regularization



- When training with **dropout**, instead, we "**turn off**" some of the neurons **randomly at each iteration**, in order to reinforce each connection of the network

Dropout Regularization

The idea behind dropout is indeed simple: zero out a random fraction of outputs from neurons across the network, where the randomization happens at each training iteration.

This procedure effectively generates slightly different models with different neuron topologies at each iteration, giving neurons in the model less chance to coordinate in the memorization process that happens during overtting.

Find out more: *Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overtting." The journal of machine learning research 15.1 (2014): 1929-1958.*

Dropout in PyTorch

In PyTorch, we can implement dropout in a model by adding an `nn.Dropout` module between the nonlinear activation function and the linear or convolutional module of the subsequent layer.

As an argument, we need to specify the probability with which inputs will be zeroed out. In case of convolutions, we'll use the specialized `nn.Dropout2d`, which zero out entire channels of the input.

model.train() **VS** model.eval()

Note that dropout is normally active **during training**, while during the evaluation of a trained model in production, dropout is bypassed or, equivalently, assigned a probability equal to zero.

We finally know what the model.train() and model.eval() are for!

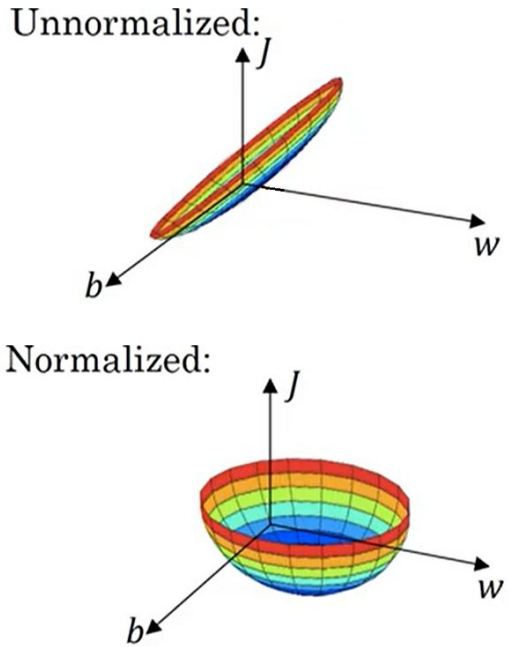
In the case of dropout, the model.eval() assigns a zero probability of dropout for the model's neurons.

Keeping activation in check: Batch Normalization

The main idea behind batch normalization is to rescale the inputs to the activations of the network so that minibatches have a certain desirable distribution.

Recalling the mechanics of learning and the role of nonlinear activation functions, this helps avoid the inputs to activation functions being too far into the saturated portion of the function, thereby killing gradients and slowing training.

In practical terms, batch normalization shifts and scales an intermediate input using the mean and standard deviation collected at that intermediate location over the samples of the minibatch.



Batch Normalization in PyTorch

Batch normalization in PyTorch is provided through the `nn.BatchNorm1D`, and `nn.BatchNorm2d` modules, depending on the dimensionality of the input.

Just as for dropout, batch normalization needs to behave differently during training and inference. In fact, at inference time, we want to avoid having the output for a specific input depend on the statistics of the other inputs we're presenting to the model.

As such, we need a way to still normalize, but this time fixing the normalization parameters once and for all (the `torch.eval()`).

Batch Normalization in PyTorch

As minibatches are processed, in addition to estimating the mean and standard deviation for the current minibatch, **PyTorch also updates the running estimates for mean and standard deviation** that are representative of the whole dataset, as an approximation.

Find out more: *Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. pmlr, 2015.*

Adding width and depth

To add capacity to our network, we can make it larger. There are two aspects of the network that can be changed.

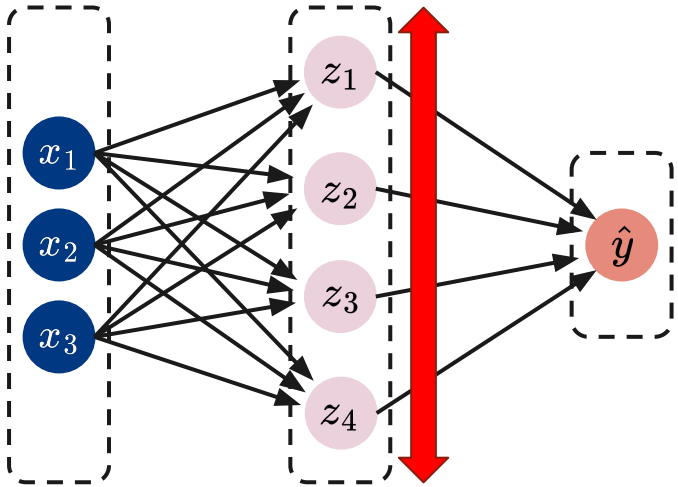
- **Width**: the number of neurons per layer, or channels per convolution.
- **Depth**: the number of layers.

Adding width

We can increase the number of neurons/channels in each linear/convolutional layer, and increase the subsequent layers accordingly

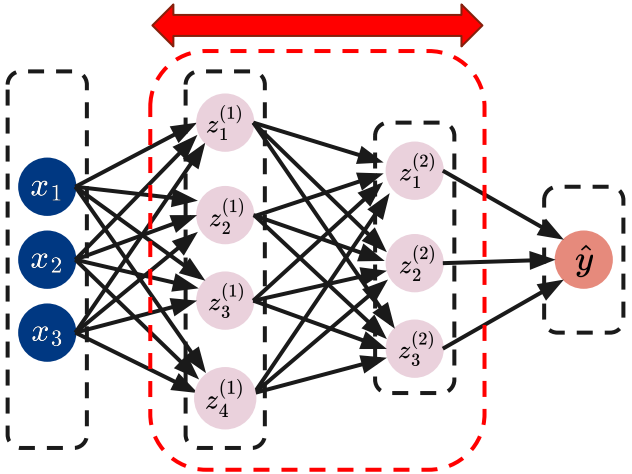
This increase the **capacity** of the model:

- can learn more complex data
- more likely to **overfit** ⚠



Adding depth

- Depth allows to deal with hierarchical information.
- Depth comes with some additional challenges, which prevented deep learning from reaching 20 or more layers until 2015.
- This because more layers are harder to converge due to gradients becoming extremely small.
- This phenomenon is known as **vanishing gradients**, leading to ineffective training since the parameters won't be properly updated



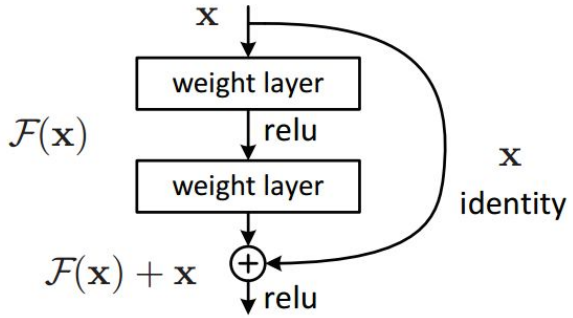
ResNets

Residual networks (ResNets), an architecture that uses a simple trick to allow very deep networks to be successfully trained using a skip connection to short-circuit blocks of layers.

A skip connection is nothing but the addition of the input to the output of a block of layers.

Find out more: *He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.*

```
def forward(self, x):
    ...
    out1 = out
    out = self.maxpool(torch.relu(self.conv(out)) + out1, 2)
    ...
    return out
```



How ResNets have solved vanishing gradients?

Thinking about backpropagation, we can appreciate that a skip connection, or a sequence of skip connections in a deep network, creates a **direct path from the deeper parameters to the loss**.

This makes their contribution to the gradient of the loss more direct, as partial derivatives of the loss with respect to those parameters have a chance not to be multiplied by a long chain of other operations.

Since the advent of ResNets, other architectures have taken skip connections to the next level (e.g., DenseNet).

Building very deep models in PyTorch

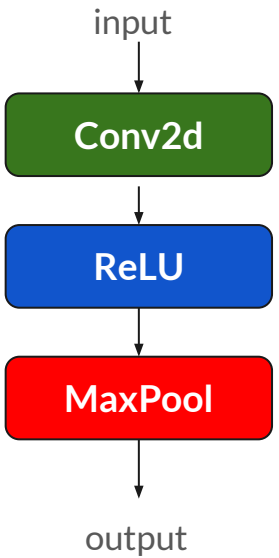
How can we build very big networks in PyTorch without losing our minds in the process?

Building very deep models in PyTorch

How can we build very big networks in PyTorch without losing our minds in the process?

The standard strategy is to dene a building block, e.g.,

(Conv2d, ReLU, MaxPool) + skip connection

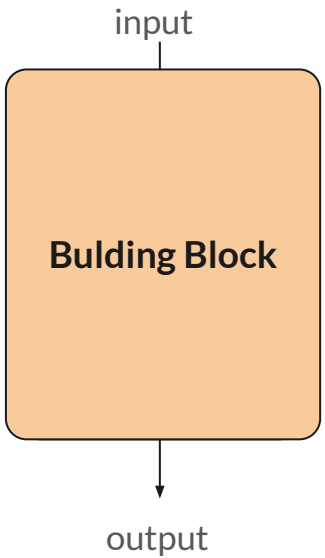


Building very deep models in PyTorch

How can we build very big networks in PyTorch without losing our minds in the process?

The standard strategy is to dene a building block, e.g.,

(Conv2d, ReLU, MaxPool) + skip connection



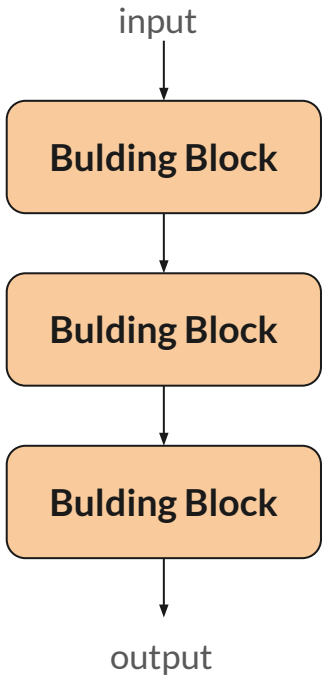
Building very deep models in PyTorch

How can we build very big networks in PyTorch without losing our minds in the process?

The standard strategy is to dene a building block, e.g.,

(Conv2d, ReLU, MaxPool) + skip connection

And re-use it to create a bigger network.



The Residual Block

```
class ResBlock(nn.Module):
    def __init__(self, n_chans, dropout_rate=0):
        super(ResBlock, self).__init__()
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3,
        padding=1, bias=False)
        self.batch_norm = nn.BatchNorm2d(num_features=n_chans)
        self.dropout = nn.Dropout(p=dropout_rate)
    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)
        out = self.dropout(out)
        return out + x # skip connection
```

Putting all together

Now we can create a model with a nn.Sequential containing a list of ResBlock instances. nn.Sequential will ensure that the output of one block is used as input to the next. It will also ensure that all the parameters in the block are visible to Net.

Then, in forward, we just call the sequential to traverse the blocks and generate the output:

```
15  class NetResDeep(nn.Module):
16      def __init__(self, n_chans1=32, n_blocks=10):
17          super().__init__()
18          self.n_chans1 = n_chans1
19          self.conv1 = nn.Conv2d(3, n_chans1,
20                                 kernel_size=3,
21                                 padding=1)
22          self.resblocks = nn.Sequential(
23              *(n_blocks * [ResBlock(n_chans=n_chans1,
24                                     dropout_rate=0.1)]))
25          self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
26          self.fc2 = nn.Linear(32, 10)
27      def forward(self, x):
28          out = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
29          out = self.resblocks(out)
30          out = torch.max_pool2d(out, 2)
31          out = out.view(-1, 8 * 8 * self.n_chans1)
32          out = torch.relu(self.fc1(out))
33          out = self.fc2(out)
34          return out
```

Finally, we can train the deep net same as we already did:

```
learning_rate = 1e-2
epochs = 10
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = NetResDeep()
net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)
loss_fn = torch.nn.CrossEntropyLoss()
train_losses, val_losses = [], []
for epoch in range(epochs):
    train_loss = train_epoch(net, train_loader, optimizer, loss_fn)
    val_loss, accuracy = valid_epoch(net, val_loader, loss_fn)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(f"Epoch [{epoch + 1}/{epochs}], "\
          f"Train loss: {train_loss:.4f}, "\
          f"Val. loss: {val_loss:.4f}, "\
          f"Val. accuracy: {accuracy:.4f}")
```

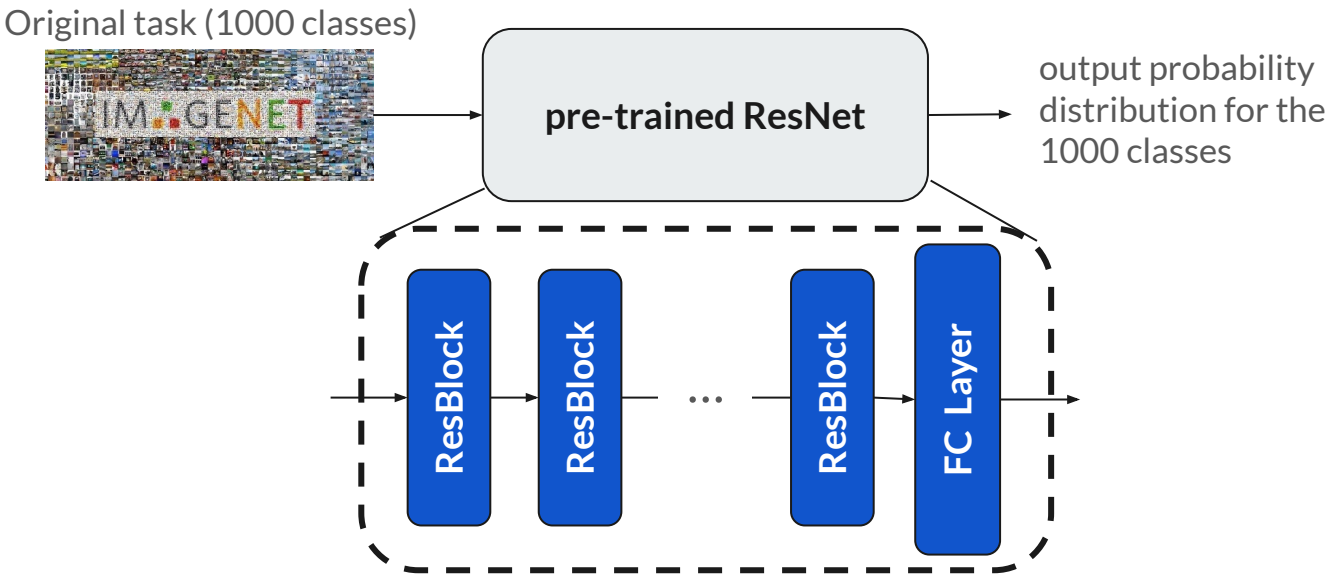
Finetuning a pre-trained model

Even if a model was trained for another tasks, its weight could be useful as a starting point for another one.

Using the torchvision.models package we can load large pre-trained models on ImageNet, a dataset composed of millions of large images.

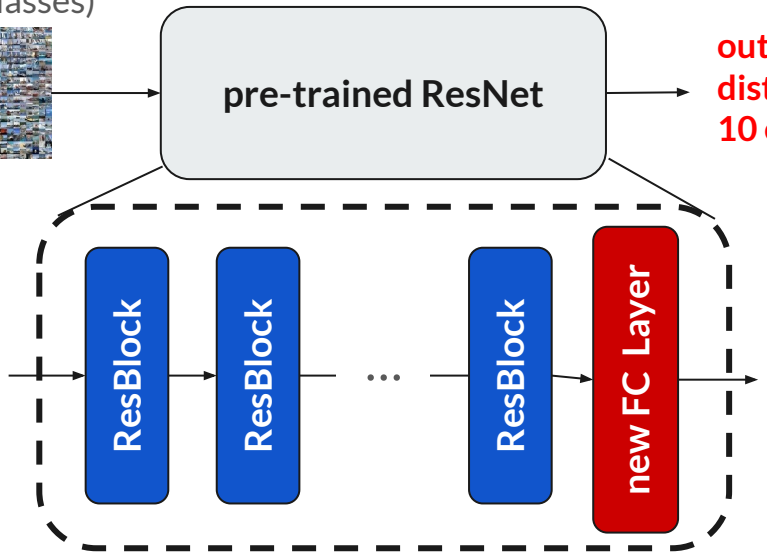
Starting from this pre-trained parameters, we can easily obtain high performances on the CIFAR-10 dataset just by replacing the linear layer at the end of the network.

Finetuning a pre-trained model



Finetuning a pre-trained model

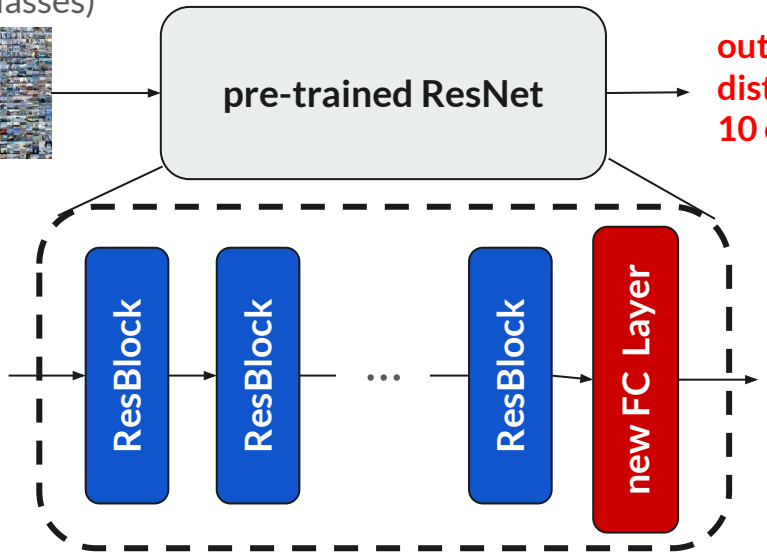
Original task (1000 classes)



- substitute the last fully connected layer (the one responsible for the actual classification)

Finetuning a pre-trained model

Original task (1000 classes)



- substitute the last fully connected layer (the one responsible for the actual classification)
- Train the network from this checkpoint

Finetuning a pre-trained model

```python
from torchvision.models import resnet18, ResNet18_Weights
# Best available weights (currently alias for IMAGENET1K_V2)
# Note that these weights may change across versions
net = resnet18(weights=ResNet18_Weights.DEFAULT)

# replace the last layer for training on CIFAR-10
net.fc = nn.Linear(in_features=512, out_features=10, bias=True)
```

```
learning_rate = 1e-2
epochs = 10
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)
loss_fn = torch.nn.CrossEntropyLoss()
train_losses, val_losses = [], []
for epoch in range(epochs):
    train_loss = train_epoch(net, train_loader, optimizer, loss_fn)
    val_loss, accuracy = valid_epoch(net, val_loader, loss_fn)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(f"Epoch [{epoch + 1}/{epochs}], "\
          f"Train loss: {train_loss:.4f}, "\
          f"Val. loss: {val_loss:.4f}, "\
          f"Val. accuracy: {accuracy:.4f}")

```

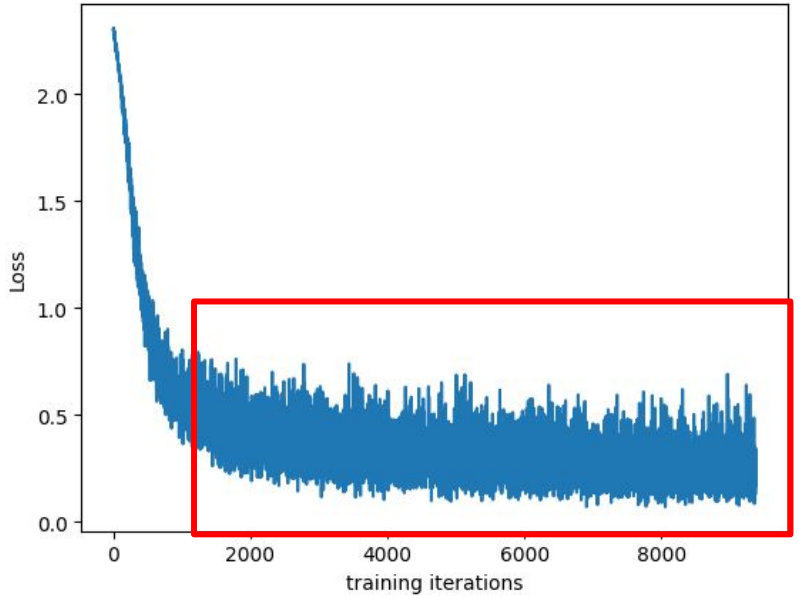
```
Epoch [1/10], Train loss: 0.0157, Val. loss: 0.0001, Val. accuracy: 0.7057
Epoch [2/10], Train loss: 0.0101, Val. loss: 0.0001, Val. accuracy: 0.7767
Epoch [3/10], Train loss: 0.0077, Val. loss: 0.0001, Val. accuracy: 0.8052
Epoch [4/10], Train loss: 0.0062, Val. loss: 0.0001, Val. accuracy: 0.8030
Epoch [5/10], Train loss: 0.0048, Val. loss: 0.0001, Val. accuracy: 0.8106
Epoch [6/10], Train loss: 0.0040, Val. loss: 0.0001, Val. accuracy: 0.8030
Epoch [7/10], Train loss: 0.0033, Val. loss: 0.0001, Val. accuracy: 0.8073
Epoch [8/10], Train loss: 0.0026, Val. loss: 0.0001, Val. accuracy: 0.8104
Epoch [9/10], Train loss: 0.0023, Val. loss: 0.0000, Val. accuracy: 0.8063
Epoch [10/10], Train loss: 0.0019, Val. loss: 0.0001, Val. accuracy: 0.8151
```

End of part 3

Summary:

- Image Representation
- MLP for image classification
- Convolutional Neural Networks for image classification
- Building complex networks
- Fine-tuning

```
1  import matplotlib.pyplot as plt
2  plt.plot(loss_path)
3  plt.xlabel('training iterations')
4  plt.ylabel('Loss')
```



While initially the loss decrease is smooth, after 2000 iterations the loss starts bouncing around 0.5

In practice, the optimization is getting stuck in a local minima.

This can be improved by changing the learning rate during training.

Schedulers

As the most important hyper-parameter* of our optimizer is the learning rate, PyTorch offers learning-rate schedulers to tune it depending on some rules (e.g., every 10 epochs, or if the loss does not improve, or others)

* Do you know what is the difference between a parameter and an hyperparameter? Parameters are **trainable**, which means that are set automatically by the optimization process, while hyper-parameters must be tuned with heuristic approaches.

Schedulers

For example, we can use the ReduceLROnPLateau module that automatically reduce the learning rate when the loss metric stop improving.

It just need few additional lines of code to our training pipeline.

We

```python
from torch.optim import SGD
from torch.nn import CrossEntropyLoss
from torch.optim.lr_scheduler import ReduceLROnPlateau

def training_pipeline(model,
                      train_loader,
                      epochs=10,
                      learning_rate=1e-2
                      ):
    optimizer = SGD(model.parameters(), lr=learning_rate)
    scheduler = ReduceLROnPlateau(optimizer, 'min')
    loss_fn = CrossEntropyLoss()
    model.cuda()        # send model parameters to the GPU
    model.train()       # set the training mode
    loss_path = []      # list to store the running loss values
    # iterations over the whole train_set
    for epoch in range(epochs):
        # iterations over each batch
        for batch_idx, (samples, labels) in enumerate(train_loader):
            # send the data to the GPU
            samples = samples.cuda()
            labels = labels.cuda()

            out = model(samples)         # forward pass
            loss = loss_fn(out, labels) # compute the loss
            loss.backward()              # backpropagate the gradients
            optimizer.step()             # update the parameters
            optimizer.zero_grad()        # reset the gradients

            loss_path.append(loss.item())   # store last loss value
            # Debugging prints every N iterations
            if batch_idx % 100 == 0:
                print(f"Epoch [{epoch + 1}/{epochs}], "\
                      f"Batch [{batch_idx + 1}/{len(train_loader)}], "\
                      f"Loss: {loss.item()}")
        scheduler.step(loss)

    return loss_path
```