Part 05 - Designing and Improving Deep Learning Models

- Notebook with the code used in these slides (part 1)
- Notebook with the code used in these slides (part 2)

Maura Pintor (maura.pintor@unica.it)

We saw that the fully connected structure has its limits. For example, having a simple shift in the pixels makes our network loose most of its performance.

- We could augment the training data and the number of parameters, to force the network in learning more meaningful representations
- Or we could use structures that learn 2D representations of the data (thus considering neighboring pixels!)

Convolutions

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

Of course, this approach is more than impractical. Fortunately, there is a readily available, local, translation-invariant linear operation on the image: a (discrete)[1] **convolution**.

[1] 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 the scalar product of a weight matrix, the kernel, with every neighborhood in the input

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

Convolutions

Let's use the **mean filter**. This filter - we will use the 3×3 version for now -

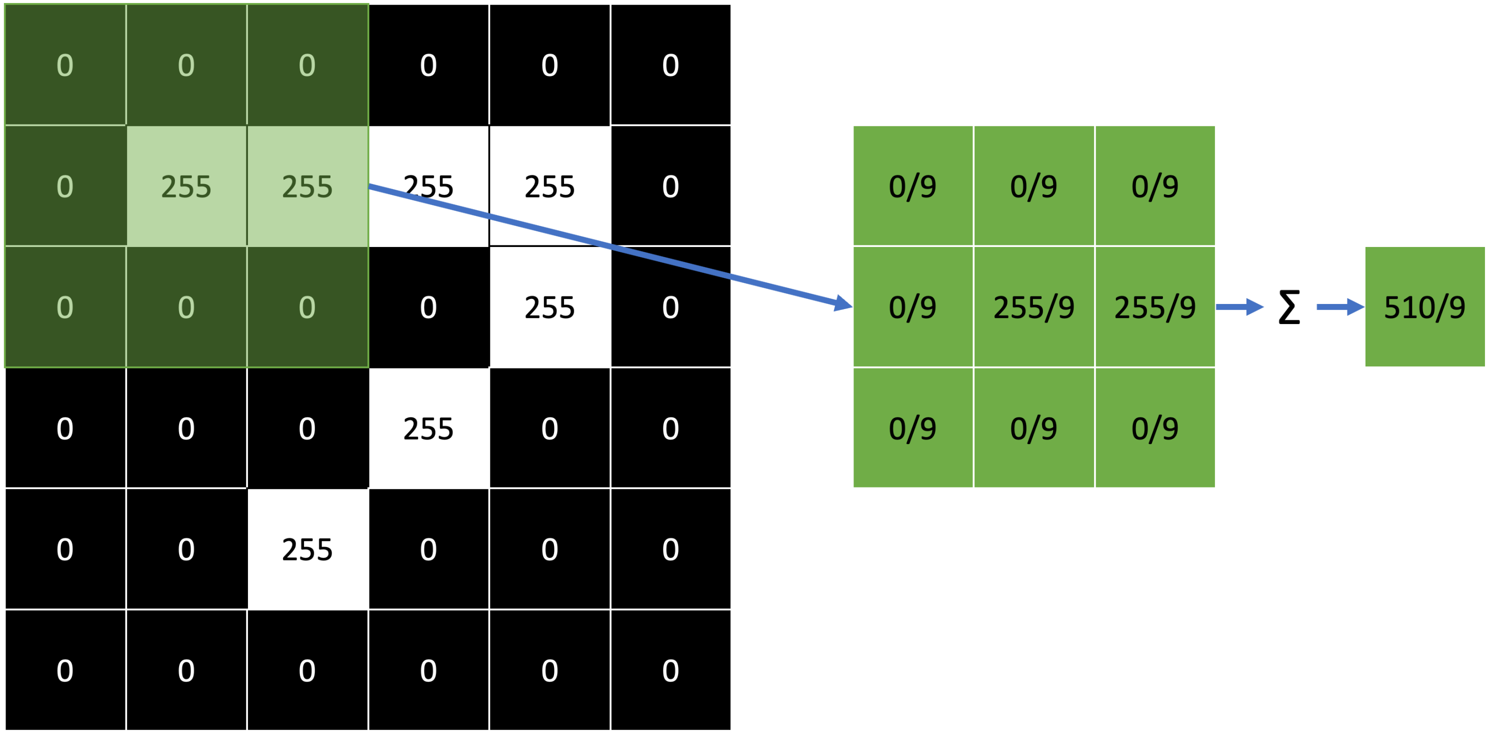
1. simply multiplies every pixel by $\frac{1}{9}$;
2. and sums all results, placing them in the resulting pixel.

This corresponds to the averagin 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

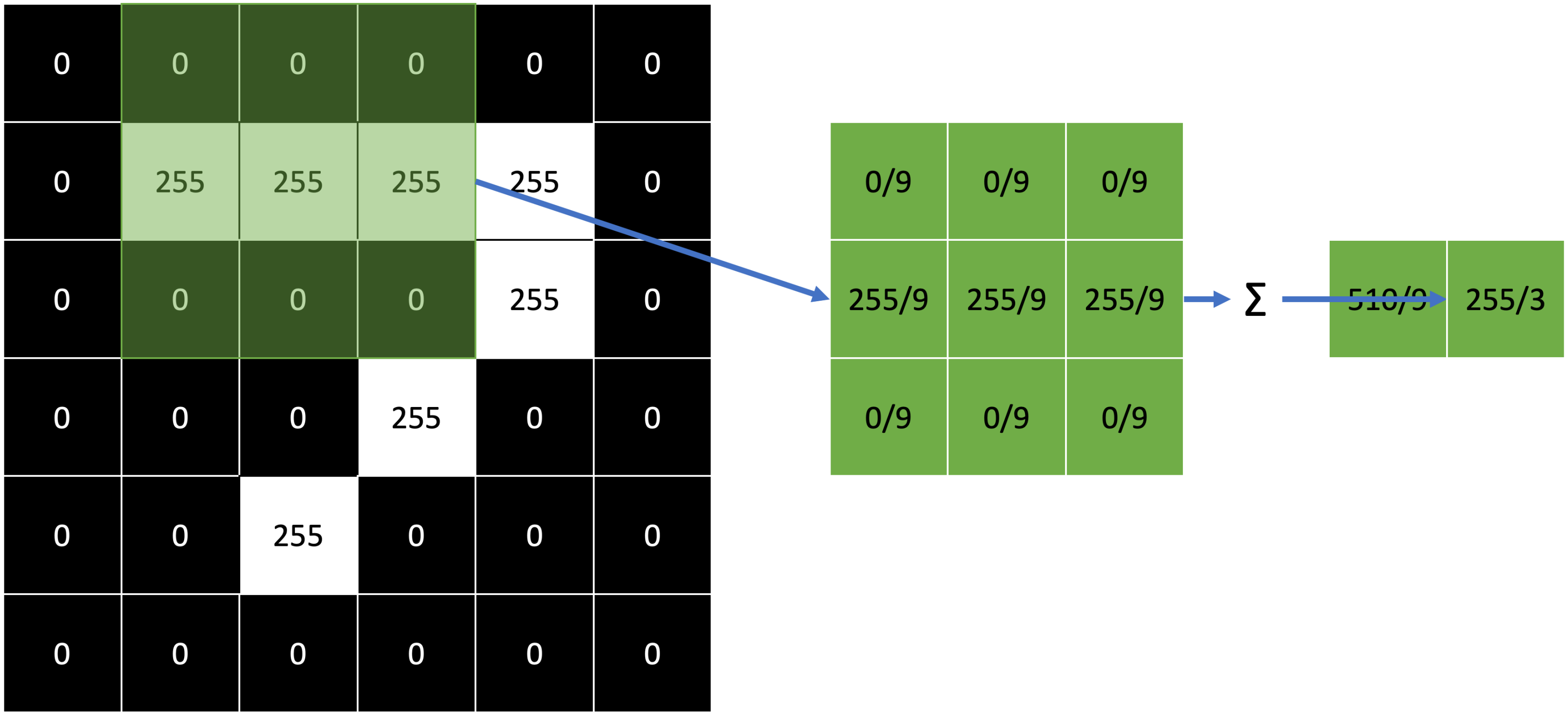


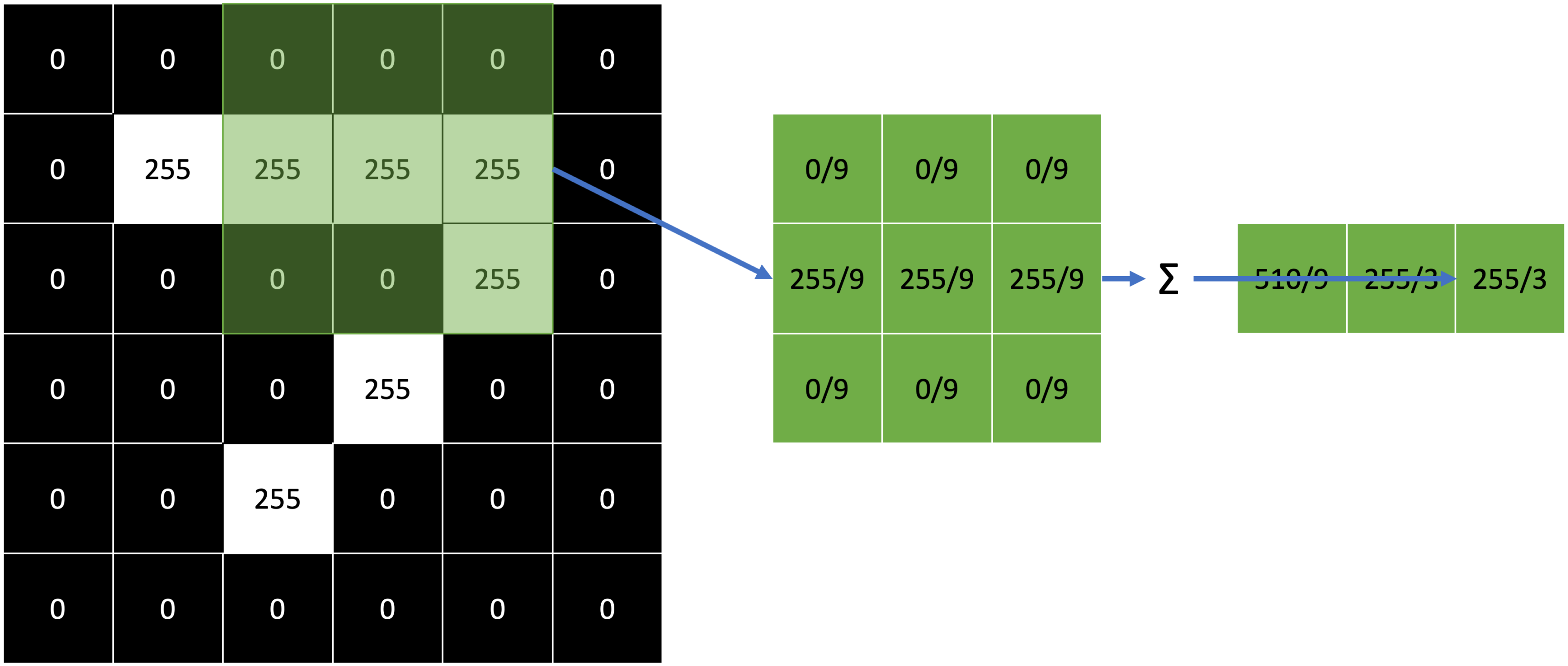
0	0	0	0	0	0
0	255	255	255	255	0
0	0	0	0	255	0
0	0	0	255	0	0
0	0	255	0	0	0
0	0	0	0	0	0

0/9	0/9	0/9
0/9	255/9	255/9
0/9	0/9	0/9

$\sum$

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

510/9	255/3	255/3

Let's implement this with Python

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

data_path = 'data'
mnist_val = datasets.MNIST(data_path, train=False, download=True,)

image, label = mnist_val[42]
plt.imshow(image, cmap='gray')
```

First, let's implement the mean filter:

```
import numpy as np
import cv2

weight = np.array([[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])

filtered = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)

plt.figure()
plt.imshow(image, cmap='gray')
plt.figure()
plt.imshow(filtered, cmap='gray')
```

Then, we can customize the filter to highlight different parts of the image (e.g., the vertical shapes):

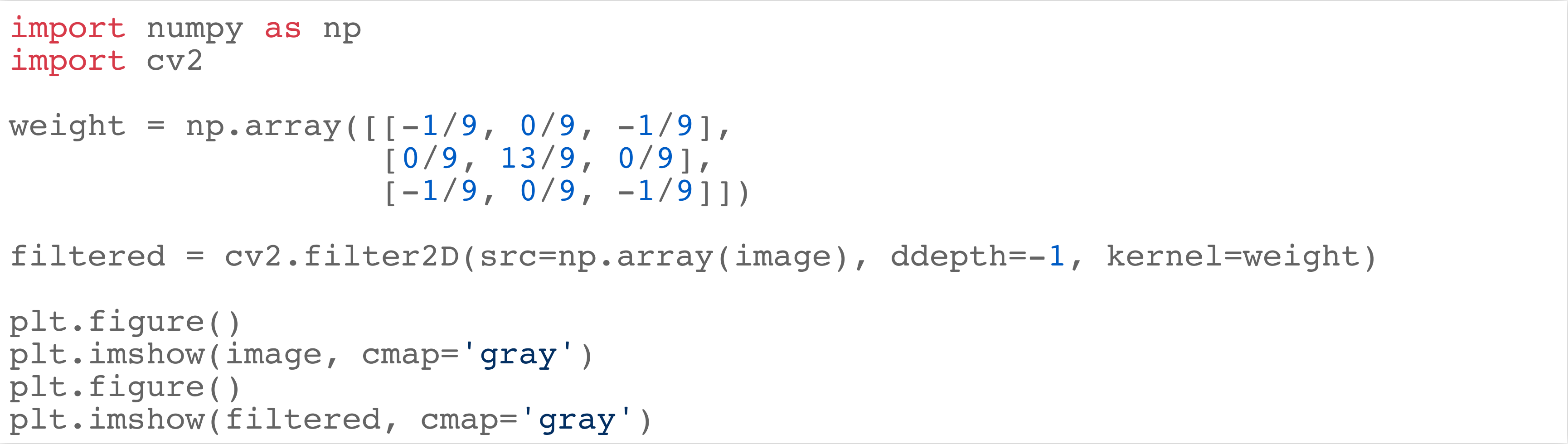
```
import numpy as np
import cv2

weight = np.array([[0/9, 3/9, 0/9],
                   [0/9, 3/9, 0/9],
                   [0/9, 3/9, 0/9]])

filtered = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)

plt.figure()
plt.imshow(image, cmap='gray')
plt.figure()
plt.imshow(filtered, cmap='gray')
```

Or, we can create custom filters:



```
import numpy as np
import cv2

weight = np.array([[-1/9, 0/9, -1/9],
                   [0/9, 13/9, 0/9],
                   [-1/9, 0/9, -1/9]])

filtered = cv2.filter2D(src=np.array(image), ddepth=-1, kernel=weight)

plt.figure()
plt.imshow(image, cmap='gray')
plt.figure()
plt.imshow(filtered, cmap='gray')
```

Note that this is a convolution with only one channel. To have convolution for RGB images, the filter 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 3×3 neighborhood requires that there are neighbors in all directions. To control better the behavior of our filters, we can specify a few parameters:

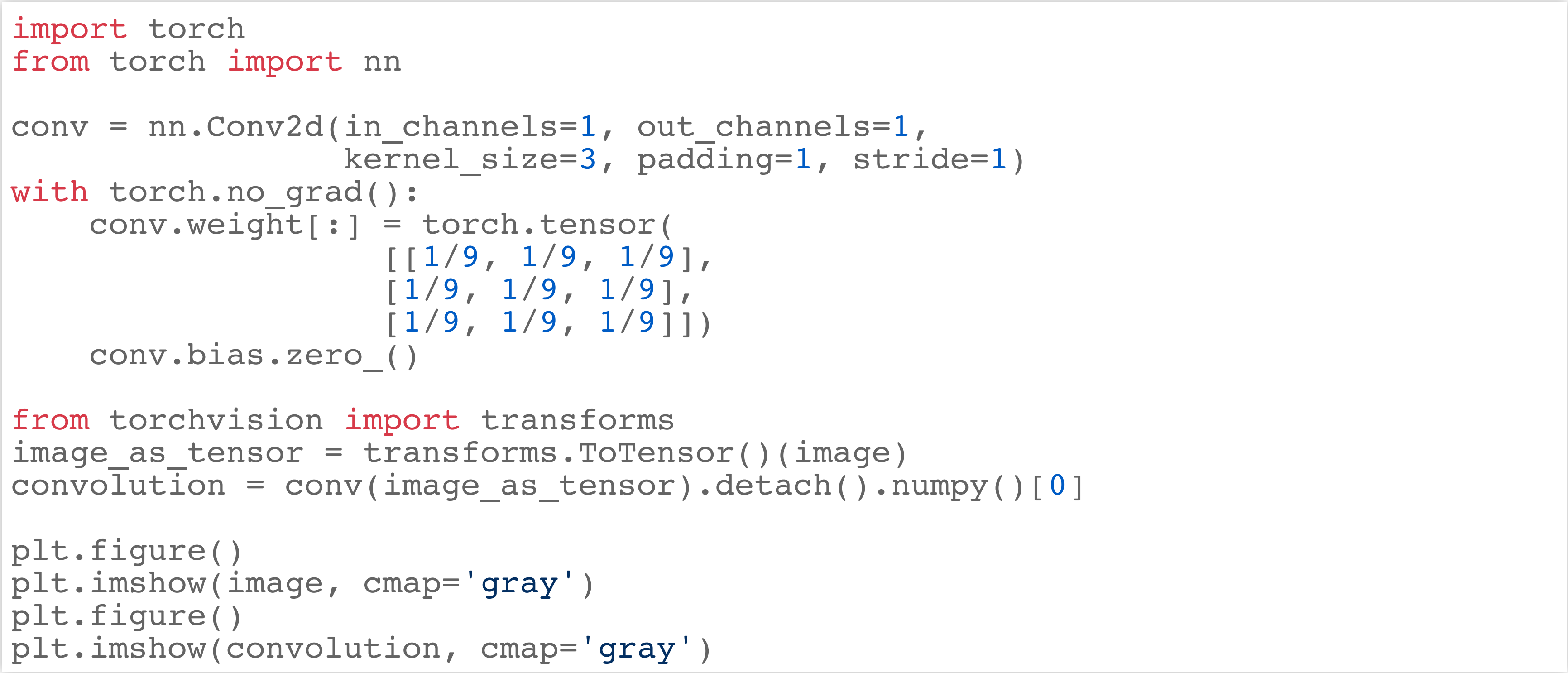
- stride: how much the filter moves between one convolution and the other
- kernel size: size of the kernel filter ($k \times k$)
- padding: how much padding to add to the image (to apply convolutions to the border) - adds zeros on the outside

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

And to use PyTorch and Torchvision:



```
import torch
from torch import nn

conv = nn.Conv2d(in_channels=1, out_channels=1,
                 kernel_size=3, padding=1, stride=1)
with torch.no_grad():
    conv.weight[:] = torch.tensor(
                   [[1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9],
                   [1/9, 1/9, 1/9]])
    conv.bias.zero_()

from torchvision import transforms
image_as_tensor = transforms.ToTensor()(image)
convolution = conv(image_as_tensor).detach().numpy()[0]

plt.figure()
plt.imshow(image, cmap='gray')
plt.figure()
plt.imshow(convolution, cmap='gray')
```

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.

Convolutional Neural Networks (ConvNets)

Then, we can combine convolutions and downsampling to extract higher-level information from the images. Until now we worked with pre-defined 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 (ConvNets)

Let's put all items together and create our fist ConvNet. This time we use the CIFAR10 dataset, composed of RGB images of size 32×32

```python
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.Tanh()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.Tanh()
        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
```

Layer math

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

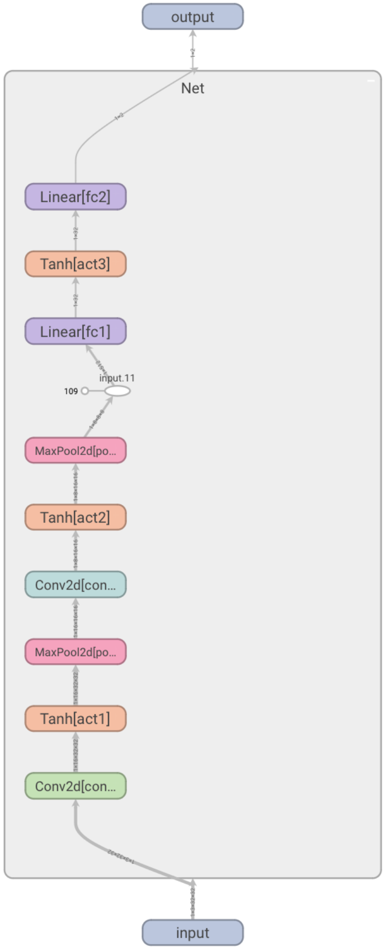
- Input `(batch_size, 3, 32, 32)`
- Conv layer, `in_channels=3`, `K=3`, `P=1`, `out_channels=16`
    - output = 16 channels of of width and height $\frac{32-3+2(1)}{1} + 1 = 32$
- MaxPool, `K=2` -> output = 16 channels of size $\frac{32}{2} = 16$
- Conv layer, `in_channels=16`, `K=3`, `P=1`, `out_channels=8`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

In jupyter notebooks, 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
```

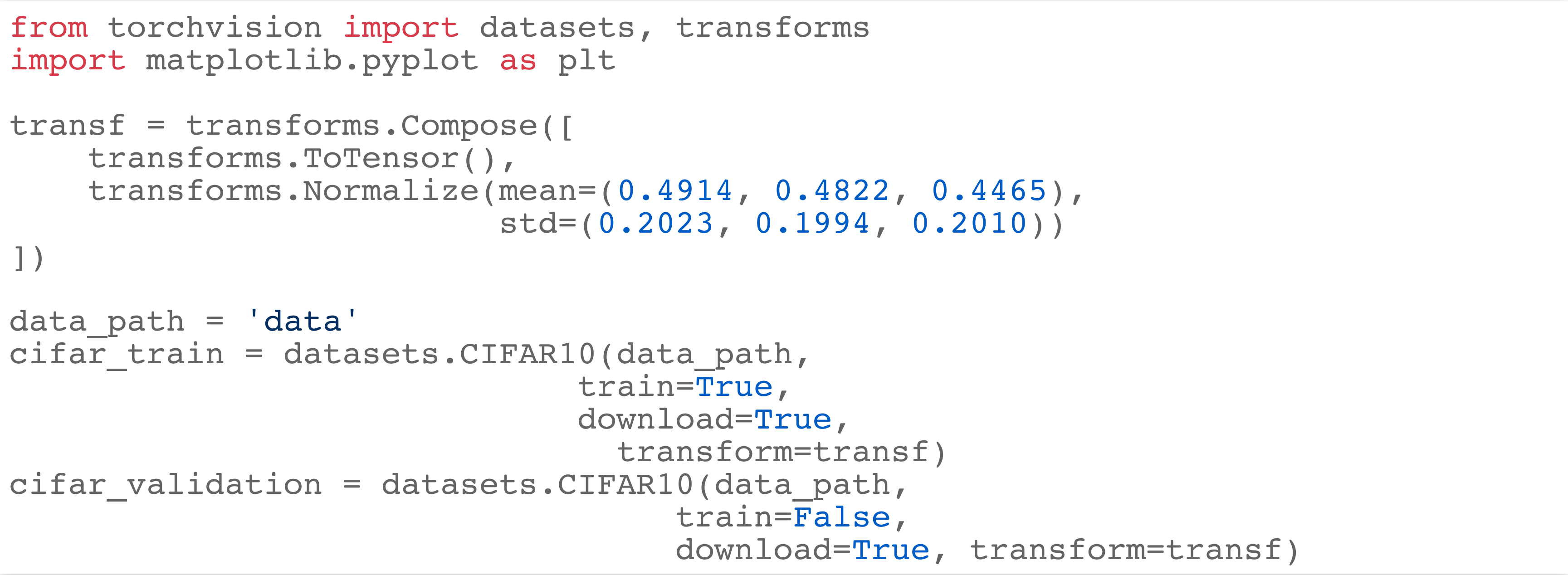

output
1×2
Net
1×2
Linear[fc2]
1×32
Tanh[act3]
1×32
Linear[fc1]
1×512
input.11
109
1×8×8×8
MaxPool2d[po...
1×8×16×16
Tanh[act2]
1×8×16×16
Conv2d[con...
1×16×16×16
MaxPool2d[po...
1×16×32×32
Tanh[act1]
1×16×32×32
Conv2d[con...
1×3×32×32
1×3×32×32
input

Training our ConvNet

Now, whatever we did for the DNN 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 load the CIFAR10 data and normalize them as the previous data.



```
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)
```

Then, we need to load the batches:

```
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)
```

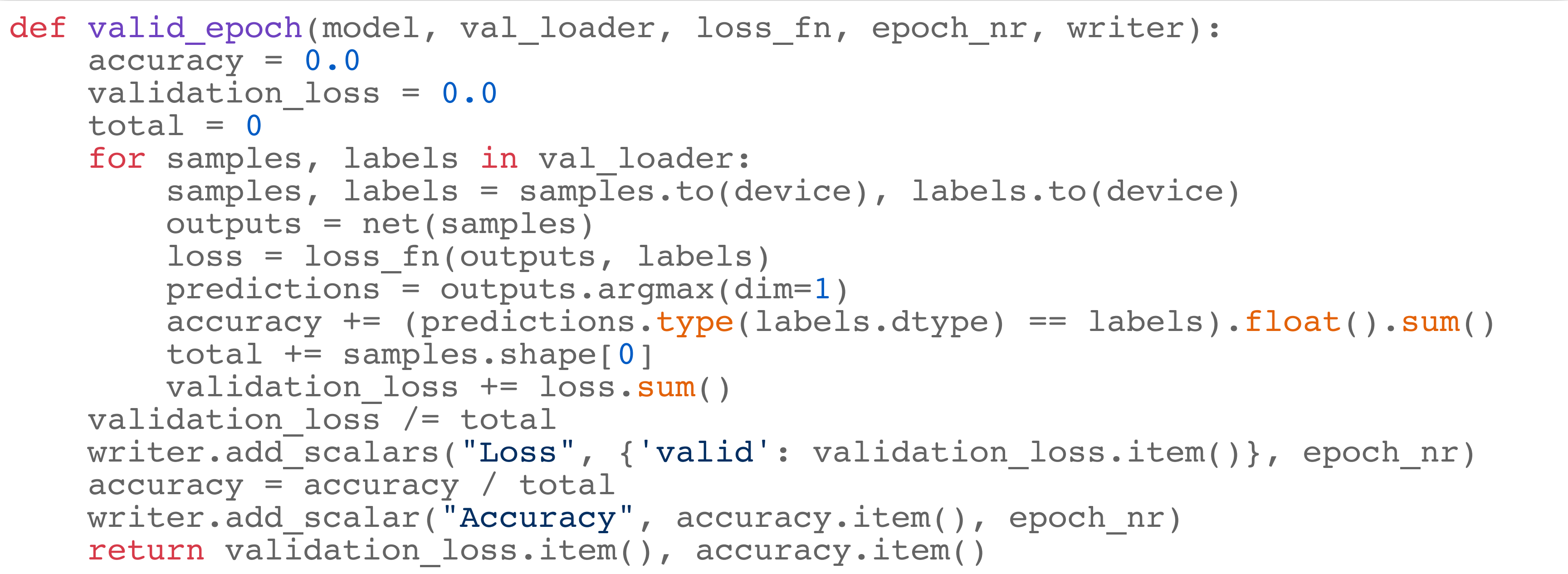
Then, let's set the optimizers and hyperparameters for training:

```
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)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
loss_fn = torch.nn.CrossEntropyLoss()
```

Let's write the training loop. Note the writer commands to track variables in `Tensorboard`:

```
def train_epoch(model, train_loader, optimizer, scheduler,
                loss_fn, epoch_nr, writer):
    train_loss = 0.0
    total = 0
    for samples, labels in train_loader:
        samples, labels = samples.to(device), labels.to(device)
        outputs = model(samples)
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total += samples.shape[0]
        train_loss += loss.sum().detach()
    train_loss /= total
    scheduler.step(train_loss)
    writer.add_scalars("Loss", {'train': train_loss.item()}, epoch_nr)
    return train_loss.item()
```

The validation loop as well uses the summary writer to track validation loss (in the same plot of the training loss) and validation accuracy.



```python
def valid_epoch(model, val_loader, loss_fn, epoch_nr, writer):
    accuracy = 0.0
    validation_loss = 0.0
    total = 0
    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
    writer.add_scalars("Loss", {'valid': validation_loss.item()}, epoch_nr)
    accuracy = accuracy / total
    writer.add_scalar("Accuracy", accuracy.item(), epoch_nr)
    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, scheduler,
                             loss_fn, epoch, writer)
    val_loss, accuracy = valid_epoch(net, val_loader, loss_fn, epoch, writer)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(epoch, accuracy)
```

And let's have a look at the board now:

```
%tensorboard --logdir logs
```

Explore the interface and check the training and validation loss, and the accuracy.

Saving and loading the models

We can store the model parameters in a file 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.

```
model_path = 'cifar_model.pt'
torch.save(net.state_dict(), model_path)

new_model = Net()
new_model.load_state_dict(torch.load(model_path))

new_model.eval()
val_loss, accuracy = valid_epoch(new_model, val_loader, loss_fn, epoch, writer)
print("accuracy of the loaded model: ", accuracy)
```

Helping our model to converge and generalize: 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*.

Keeping the parameters in check: weigth penalties

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 fitting individual samples.

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

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

Keeping the parameters in check: weigth penalties

We can implement that in Torch by using a penalty on the loss:

```
loss = loss_fn(outputs, labels)
            l2_lambda = 0.001
            l2_norm = sum(p.pow(2.0).sum()
                          for p in model.parameters())
            loss = loss + l2_lambda * l2_norm
```

Or alternatively, by using the parameter in the optimizer:

```
optimizer = optim.SGD(model.parameters(), weight_decay=0.001)
```

Not relying too much on a single input: Dropout

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 overfitting.

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

Not relying too much on a single input: Dropout

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.

Not relying too much on a single input: Dropout

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.

Keeping activation in check: Batch Normalization

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()`).

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

To increase the **width**, we just specify a larger number of output channels in the first convolution and increase the subsequent layers accordingly.

The numbers specifying channels and features for each layer are directly related to the number of parameters in a model; all other things being equal, they increase the capacity of the model.

The greater the capacity, the more variability in the inputs the model will be able to manage; but at the same time, the more likely **overfitting** will be, since the model can use a greater number of parameters to memorize unessential aspects of the input. Here, regularization can help reduce the memorization effect.

Adding depth

The second dimension that we can increase is obviously depth. Depth allows a model to deal with **hierarchical information** when we need to understand the context in order to say something about some input.

Adding depth

Depth comes with some additional challenges, which prevented deep learning models from reaching 20 or more layers until late 2015. Adding depth to a model generally makes training **harder to converge** due to gradients becoming extremely small.

The bottom line is that a long chain of multiplications will tend to make the contribution of the parameter to the **gradient vanish**, leading to ineffective training of that layer since that parameter and others like it 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.*

ResNets

It is easy to implement it in PyTorch.

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

In other words, we're using the output of the first activations as inputs to the last, in addition to the standard feed-forward path.

ResNets

How does this solve the problem of 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?

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

```
(Conv2d, ReLU, Conv2d) + skip connection
```

And re-use it to create a bigger network.

Let's first create the ResBlock:

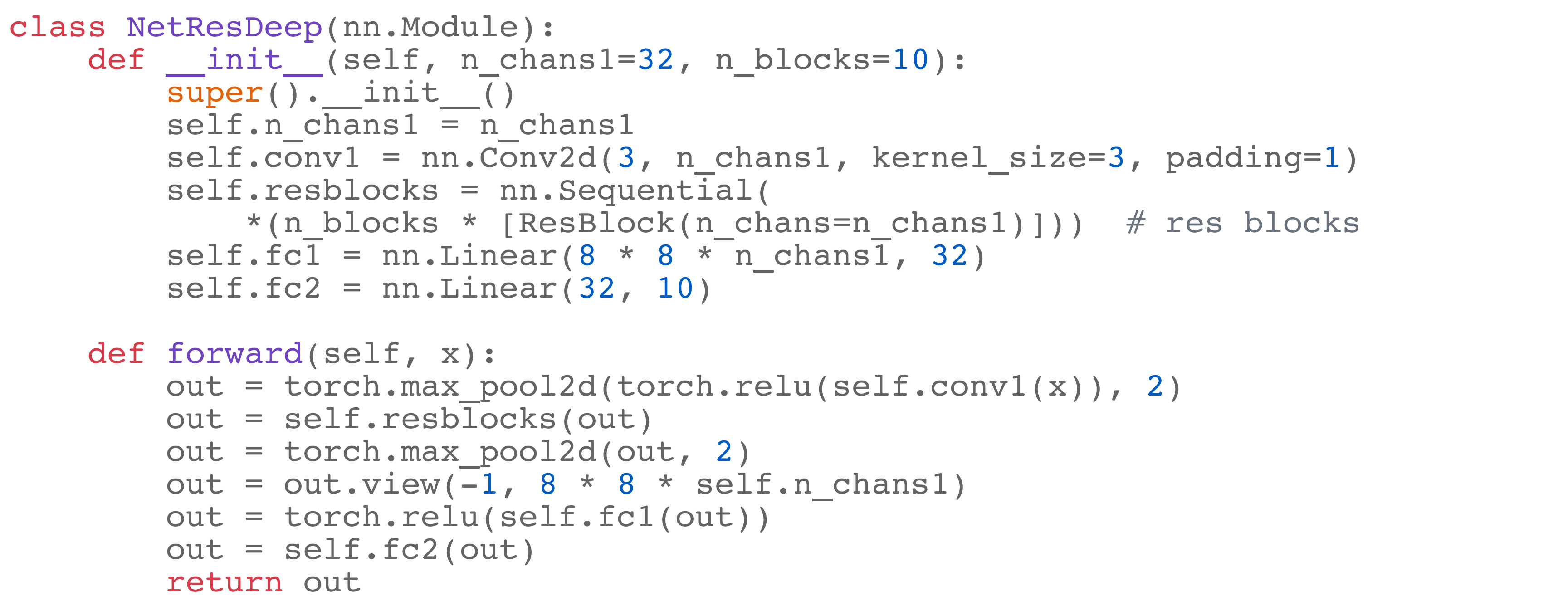
```python
class ResBlock(nn.Module):
    def __init__(self, n_chans):
        super(ResBlock, self).__init__()
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3,
                                padding=1, bias=False)
        self.batch_norm = nn.BatchNorm2d(num_features=n_chans)

    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)
        return out + x  # skip connection
```

Note the batch normalization and the skip connection in the end of the `forward`, in the return

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.

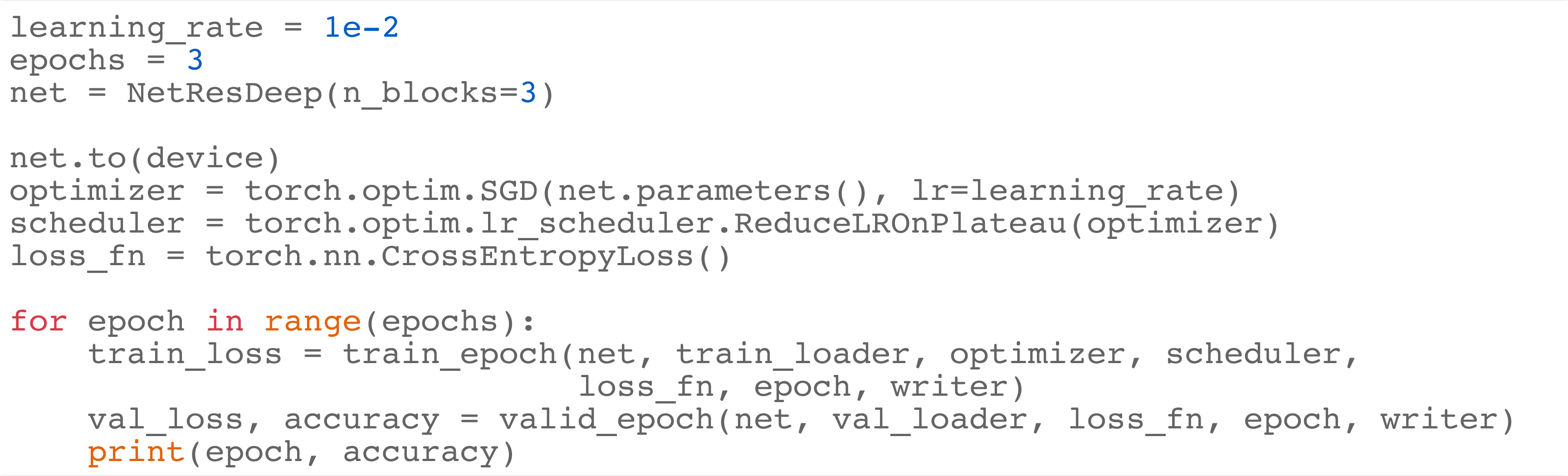
Then, in forward, we just call the sequential to traverse the blocks and generate the output:



```python
class NetResDeep(nn.Module):
    def __init__(self, n_chans1=32, n_blocks=10):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.resblocks = nn.Sequential(
            *(n_blocks * [ResBlock(n_chans=n_chans1)]))  # res blocks
        self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        out = torch.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = self.resblocks(out)
        out = torch.max_pool2d(out, 2)
        out = out.view(-1, 8 * 8 * self.n_chans1)
        out = torch.relu(self.fc1(out))
        out = self.fc2(out)
        return out
```

Then, we can train with the standard loop that we used earlier:



```python
learning_rate = 1e-2
epochs = 3
net = NetResDeep(n_blocks=3)

net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
loss_fn = torch.nn.CrossEntropyLoss()

for epoch in range(epochs):
    train_loss = train_epoch(net, train_loader, optimizer, scheduler,
                             loss_fn, epoch, writer)
    val_loss, accuracy = valid_epoch(net, val_loader, loss_fn, epoch, writer)
    print(epoch, accuracy)
```

End of part 5

Summary:

- Convolutional neural networks
- Tricks to help convergence
- Building complex networks
- Tracking experimental results

End of part 5

In the next chapter:

- Beyond classification: creating an object detection model

- Notebook with the code used in these slides (part 1)
- Notebook with the code used in these slides (part 2)

Maura Pintor (maura.pintor@unica.it)