Introduction to PyTorch

Maura Pintor (maura.pintor@unica.it)

Part 01 - Introduction

The PyTorch Library

- Most of PyTorch is written in C++ and CUDA
- Python is just used for exposing the APIs
- Tensors keep track of operations and analytically compute derivatives

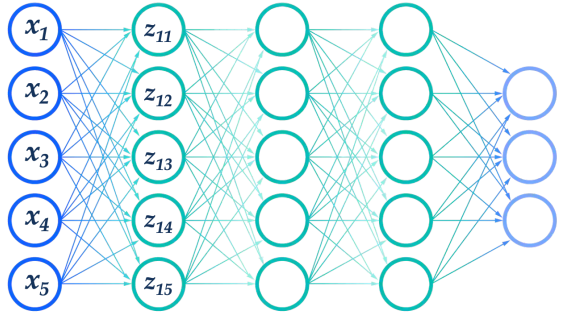
Hardware and software requirements for this lecture

- It is not required to have a GPU
- We won't train large models in our local machines
- The only requirement is a browser

Part 02 - Deep Learning Foundations

Deep learning

Deep Neural Networks (DNNs) stack **layers** of linear classifiers and non-linear activation functions



The output of each layer l is computed as a function of the product of $\mathbf{w}_l$ and the output of the previous layer $\mathbf{z}_{l-1}$

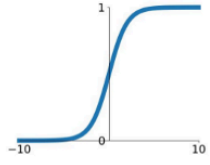
The function is called **activation function** and it is usually non-linear

Activation functions

Choices of the activation function a are in general:

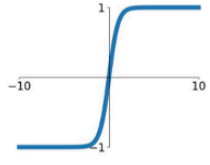
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



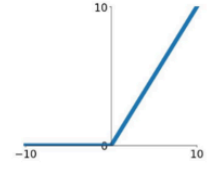
tanh

$\tanh(x)$



ReLU

$\max(0, x)$



Introducing non-linearities ensures learning non-linear decision functions, which in general fit better non-linearly-separable data

Activation functions

... **Are nonlinear**

Repeated applications of (w*x+b) without an activation function results in a function of the same (affine linear) form. The nonlinearity allows the overall network to approximate more complex functions.

... **Are differentiable**

Gradients can be computed through them. Point discontinuities, as we can see in the ReLU, are fine. We can still compute the derivatives for the different areas.

How to find the best model?

To evaluate the goodness of each candidate function (each value of the parameters $\boldsymbol{\theta}$), we define a **loss function** ℓ

$$L(D, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

Where:

- y_i is the true label of sample $\mathbf{x}_i$
- $f(\mathbf{x}_i; \boldsymbol{\theta})$ is the decision function
- $\frac{1}{n} \sum_{i=1}^{n} (\dots)$ computed over the training set

By computing the loss over the whole training set, we get an estimate of the quality of the predictor

How to get the minimum loss

We define the learning problem as an optimization problem

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{n} \sum_{1=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

Some simple problems have a closed form solution (you can find $\boldsymbol{\theta}^*$ directly by solving the problem analytically)

For others we have to rely on **solvers** that find an approximate solution.

Gradient descent

The most popular algorithm for solving the optimization problem is Gradient Descent

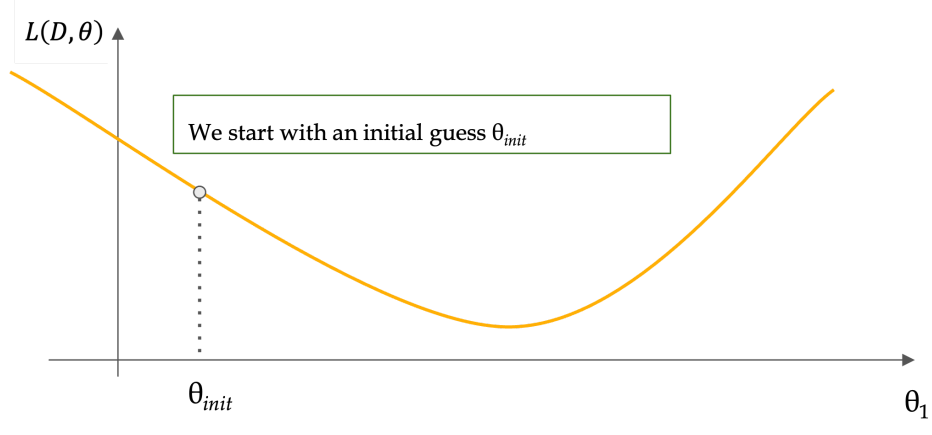
We use the **gradient** of the loss function

$$L(D, \theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \theta))$$

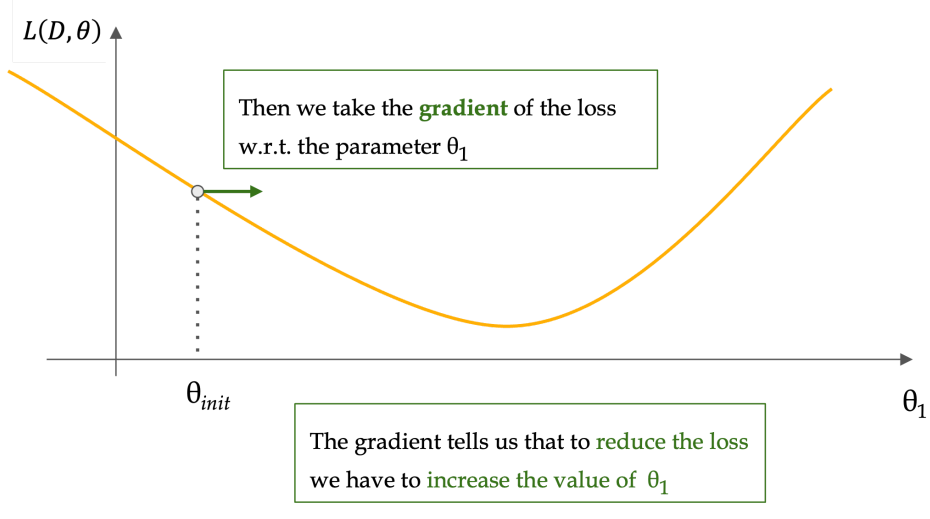
$$\nabla_\theta L = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(y_i, f(\mathbf{x}_i; \theta))$$

And **iteratively** update the parameters to find the optimal ones.

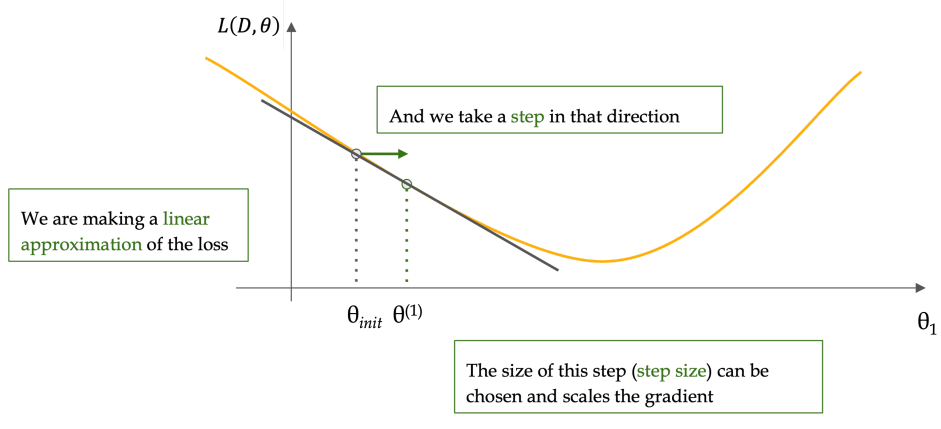
Let's assume that we fix all parameters but one (e.g., θ_1)



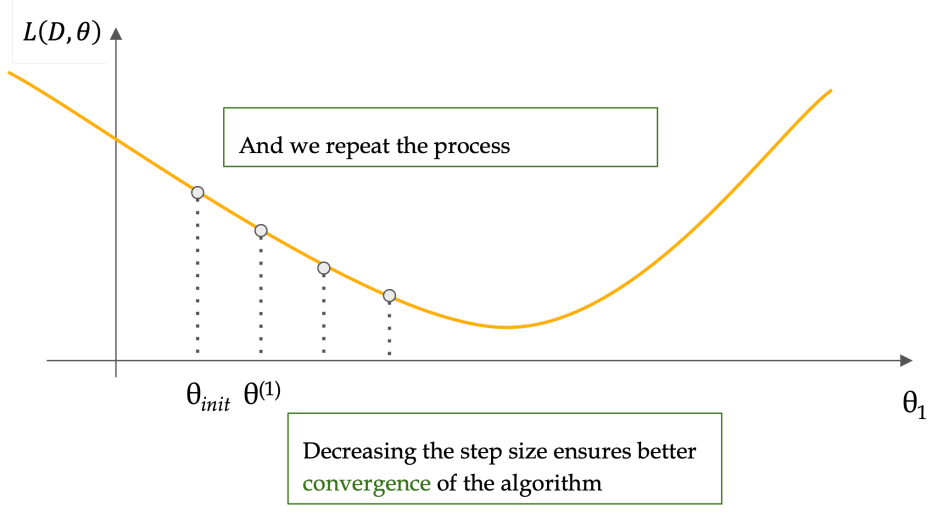
Let's assume that we fix all parameters but one (e.g., θ_1)



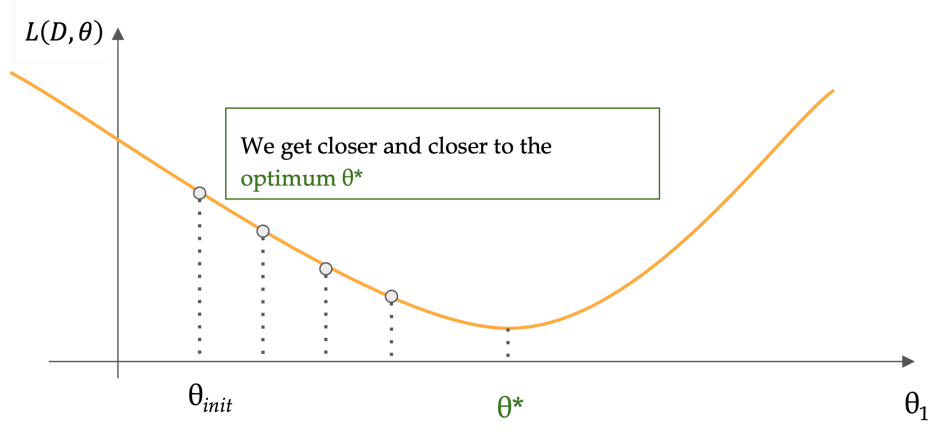
Let's assume that we fix all parameters but one (e.g., θ_1)



Let's assume that we fix all parameters but one (e.g., θ_1)

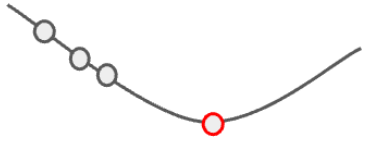


Let's assume that we fix all parameters but one (e.g., θ_1)



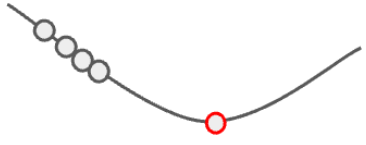
What influences the progress (and results) of the optimization?

- number of steps
  - if we don’t take enough steps we can stop too early and far from the optimum



What influences the progress (and results) of the optimization?

- step size
  - if the step size is too small, we need many steps to reach convergence

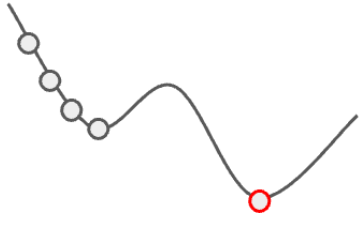


  - if the step size is too big, we might overshoot the optimum

  - the decay of the step size is also important (how much we reduce the step size)

What influences the progress (and results) of the optimization?

- function that we are optimizing
  - there might be local minima and our optimization can get stuck in them



Try out gradient descent with this online tool:
https://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

In general, we can compute the gradient of the loss w.r.t. multiple parameters and update the parameters simultaneously

The gradient is a vector with one component for each parameter

We can optimize the parameters of a DNN all simultaneously, by computing the gradient of the loss w.r.t. each single parameter

Chain rule and backpropagation

Let's introduce briefly the chain rule, as it will be useful later to find out how to automatically update all the parameters of a DNN with a few lines of code.

Consider two functions of a single independent variable $f(x)$ and $g(x)$. The composite function is defined as $h = g(f(x))$.

You can think of f and g as functions applied in cascade, as the output of $f(x)$ goes as the argument of $g(x)$

Chain rule and backpropagation

The chain rule is useful to find the derivative of the composite function

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x}$$

If $u = f(x)$ is the output of $f(x)$, then we can compute the derivative of the composed function as the product of the derivative of the "external" function w.r.t. u and the derivative of the "internal" function w.r.t. x.

Let's see it with an example:

$$f(x) = 2x - 1 = u$$

$$g(x) = x^2$$

$$h(x) = g(f(x)) = (2x - 1)^2$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x} = 2u \cdot 2 = 2(2x - 1) \cdot 2 = 4(2x - 1)$$

Application in machine learning

A neural network can be represented as a nested composite function

$$y = f_K(f_{K-1}(\cdots(f_1(\mathbf{x}))))$$

Here, $\mathbf{x}$ are the inputs to the neural network, whereas y are the outputs

Every function, f_i, for $i = 1, \ldots, K$, is characterized by its own weights

Application in machine learning

Applying the chain rule to such a composite function allows us to work backwards through all of the hidden layers and efficiently calculate the **error gradient** of the loss function **w.r.t. each weight**, w_i, of the network until we arrive at the input

$$\nabla_{x_i} L = \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial w_1}\frac{\partial w_1}{\partial x_i} + \cdots + \frac{\partial L}{\partial w_M}\frac{\partial w_M}{\partial x_i}$$

Thus, we can update each weight so that the loss decreases

This is the basic parameter update for gradient descent. By re-iterating multiple times and using a small enough learning rate, the model will converge to better values of parameters (lower loss)

$$w_i \leftarrow w_i - \alpha \nabla L_{w_i}$$

Autograd

We can get the gradient of all parameters of our models by propagating derivatives backward using the chain rule

The basic requirement is that all functions can be differentiated analytically

If this is the case, we can compute the gradient with respect to the parameters in one sweep (even with highly-complicated models!)

Autograd

PyTorch creates operations that can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs

This means we won't need to derive our model by hand

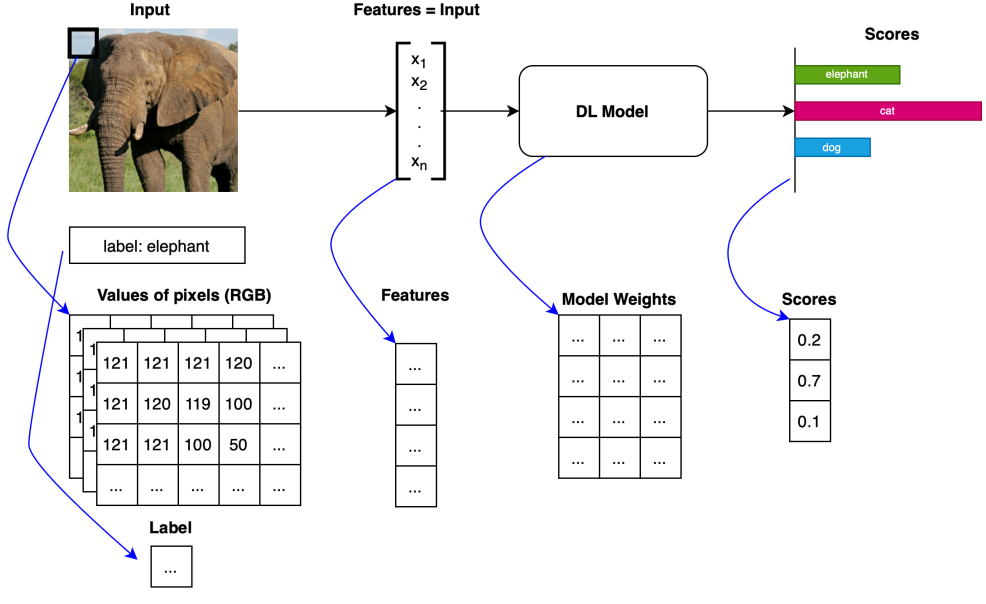
Given a "forward" expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

Part 03 - PyTorch basics

Representing data

Deep-learning systems have to be able to map input data to the outputs. To do so, they usually represent the input data with a specific format.

Deep Learning as Floating Point Numbers



We will understand soon what are these "boxes".

Deep Learning as Floating Point Numbers

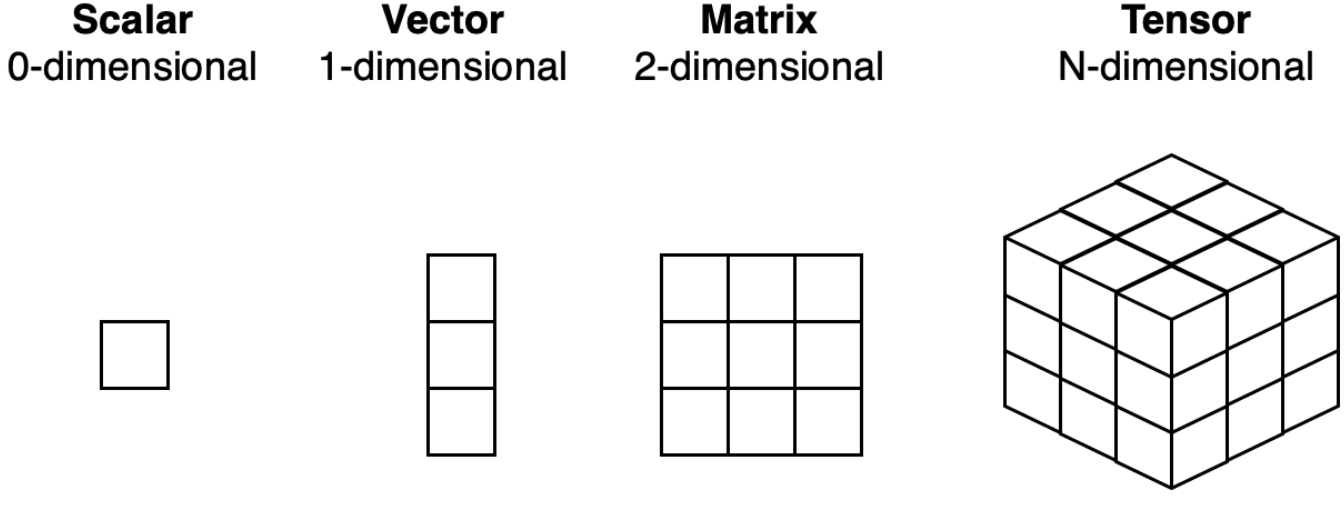
- **Pixels**: representations of the input images
- **Features**: internal representations of the model, similar inputs should have close representations
- **Model Weights**: matrices used to prioritize the inputs and extract the deep features
- **Scores**: output representations

How are these data units represented and stored?

Tensors of Floating Point Numbers

In the context of deep learning, a tensor is simply the extension of a vector that has an arbitrary numbers of dimensions.

tensors == multi-dimensional arrays



Tensors in PyTorch

`PyTorch` is not the only library that deals with n-dimensional arrays. `NumPy`, `SciPy`, `Scikit-learn`, `Pandas`, and other deep-learning libraries such as `Tensorflow` also support n-dimensional arrays.

However, in `PyTorch`, the `Tensor` class is more powerful than standard numeric libraries.

- GPU support
- Parallel operations on multiple devices or machines
- Autograd!!!

All these features, especially the last one, are of utmost importance when dealing with deep learning!

Tensor API

As an approximation, think of PyTorch tensors as numpy arrays with augmented functionalities (mostly, the APIs are similar).

Your first source of information should be the PyTorch documentation.

- more complete
- more updated
- (it might also say something different than these slides!).

Basic Tensor operations

Creation operations and mutations

```
a = torch.ones(3, 2)  # 3x2 tensor of only ones
b = torch.zeros(3, 1)  # 3x1 tensor of only zeros
c = torch.zeros_like(a)  # same shape and type as a
a_t = a.t()  # 2x3 tensor (transpose of a)
print(a.shape)  # prints the shape (i.e., all the sizes of the dimensions)
```

Math operations

```
absolute_values = torch.abs(a)  # pointwise operations
mean_value = torch.mean(a)  # reduction operations
s = a + c  # element-wise sum
p = a * c  # element-wise product
z = torch.mm(a, c.t())  # matrix multiplication (careful with shapes!)
broadcasting = a + torch.tensor([1, 2])  # torch tries to match shapes
```

Moving tensors to the GPU

Moving tensors to the GPU can make computations massively parallel and fast. Then, all the operations will be performed with GPU operations, while the API remains the same.

PyTorch supports all GPUs that have support for CUDA (Compute Unified Device Architecture), a software layer created by Nvidia.

An accelerated version of PyTorch is also available for Apple Silicon, but it is still not very stable.

Moving tensors to the GPU

Every PyTorch tensor has the attribute `device`, which says where the tensor data is placed in storage. Tensors can be "moved" (rather, copied) to another device by using the method `to`.

```
gpu_tensor = torch.zeros(1, device='cuda')  # created on the GPU
cpu_tensor = torch.zeros(1)
to_gpu = cpu_tensor.to(device='cuda')  # this creates a copy of the tensor!
to_gpu_another = cpu_tensor.cuda()  # shorthand for the previous command
again_to_cpu = to_gpu.cpu()  # shorthand for copying the tensor to cpu
```

If your machine has more GPUs, you can also specify which one to use, e.g., `cuda:0`. Note that operations can be performed only between tensors located on the same device.

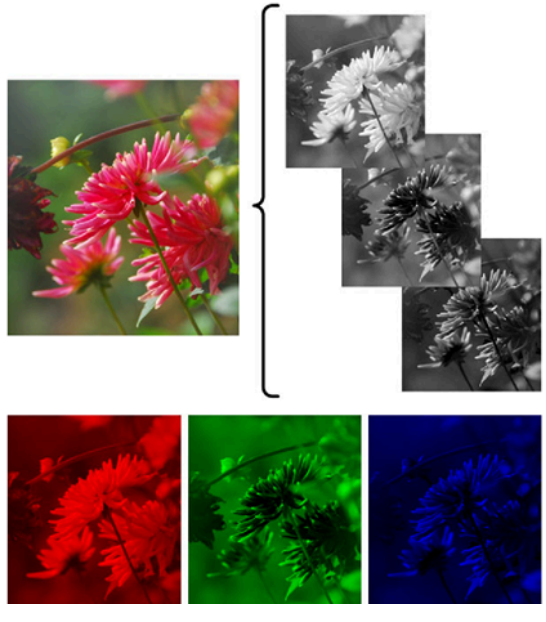
Part 04 - Learning from Tensors: Gradient Descent and Backpropagation

- Notebook with the code used in these slides (follow along)
- Notebook with the code used in these slides (solution)

RGB representation

Each pixel is represented by three values, corresponding to the colors Red, Green and Blue (RGB)

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



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 representation follows the standard `[num_samples, num_features]`, where each sample is a (flatten) row vector, and we stack multiple samlpes in the first dimension.

By convention, the first dimension in PyTorch is always the sample index, the other dimensions are the features.

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

```
[num_samples, colors(=3), width, height]
```

Normalizing the data

A typical thing for machine learning is to normalize the data so that the values of the pixels 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.
normalized_tensor = normalizer(tensor)

print(normalized_tensor.shape)
normalized_flatten = normalized_tensor.flatten(start_dim=1)
print("Original tensor shape: ", normalized_tensor.shape,
      "Flattened tensor shape: ", normalized_flatten.shape)
print("Min: ", normalized_flatten.min(dim=-1)[0],
      "Max: ", normalized_flatten.max(dim=-1)[0])
```

For our case:

- the minimum value 0 will be converted to $\frac{0-0.5}{0.5} = -1$
- the maximum value of 1 will be converted to $\frac{1-0.5}{0.5} = 1$

Representing the scores

When dealing with any system, we should also take care of what the outputs are. In machine learning, the output is represented with a set of **scores**.

For example, in classification, we have one score (i.e., continuous variable) for each output class

We can think of them of the "probability of classes" (but keep in mind that this is not a very well-defined concept)

The predicted class is assigned to the class with the highest score.

The output of the machine learning model is the matrix of N scores. Assuming we have 3 classes, one example is:

$$f(\boldsymbol{x}) = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.9 \\ -0.5 \end{pmatrix} = \begin{pmatrix} - \\ * \\ - \end{pmatrix}$$

The winner class is , and this is often called the **prediction** of the machine-learning model (OVA classification)

Even better, we can use the `argmax` function to retrieve the best score

```
import torch
# let's create a column tensor with 10 scores
scores = torch.randn(1, 10)
print(scores.argmax(dim=1))
```

And if we want to compute the scores for a batch, let's remember that the first dimension is the sample index.

```
1 import torch
2 # let's create a column tensor with 10 scores for 4 samples
3 scores = torch.randn(4, 10)
4 print(scores.argmax(dim=1))
```

And if we want to compute the scores for a batch, let's remember that the first dimension is the sample index.

```
1 import torch
2 # let's create a column tensor with 10 scores for 4 samples
3 scores = torch.randn(4, 10)
4 print(scores.argmax(dim=1))
```

What are now the dimensions of the printed tensor? What do they represent?

Learning

As seen earlier, Learning is just Parameter Estimation.

We have to find the parameters to approximate the unknown function:

To find the good parameters , we define a loss (*a.k.a.* cost) function that we want to minimize.

Loss functions for training

We are almost ready to train a deep neural network on the MNIST dataset. We are only missing a proper loss function.

We need a function , where are some input-output pair.

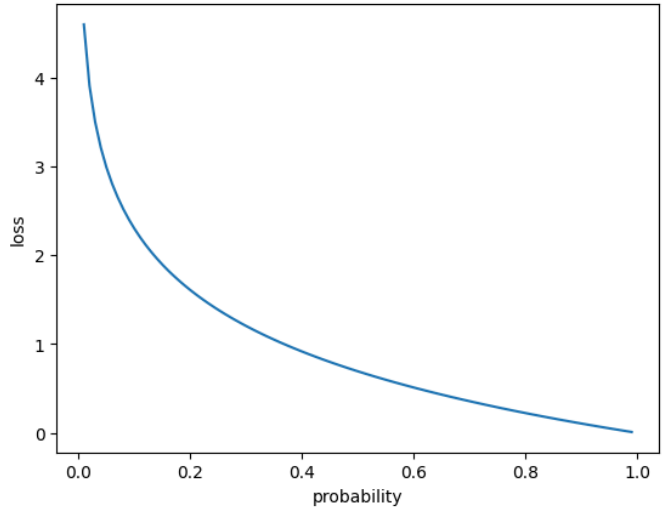
A loss function is used to help the model determine how "wrong" it is and, based on that error signal improve itself.

Loss functions for training

The cross-entropy loss measures the error (or difference) between two probability distributions. In the binary case, the cross-entropy is computed as:

Where is the predicted probability and is the label.

```
import numpy as np
import matplotlib.pyplot as plt
def ce_loss_binary(p, y):
    return - y * np.log(p) + (1-y) * np.log(1-p)

y = torch.tensor(1)
p = np.linspace(0, 1, 100)
losses = [ce_loss_binary(_, y) for _ in p]

plt.xlabel("probability")
plt.ylabel("loss")
plt.plot(p, losses)
```



Cross-entropy Loss

In the multi-class classification case, the cross-entropy loss is separate for each class label and we sum the result:

It is useful when training a classification model with C classes

The input is expected to contain the unnormalized logits for each class (which do not need to be positive or sum to 1, in general)

In PyTorch we can use the class `torch.nn.CrossEntropyLoss`

Optimizers

Once the loss is picked, we have to **minimize** it.

There are several optimization strategies and tricks that can assist convergence, especially when models get complicated

PyTorch abstracts the optimization strategy away from the DNN code

This saves us from having to update each and every parameter to our model ourselves

The torch module has an optim submodule where we can find classes implementing different optimization algorithms

... to see available optimizers:

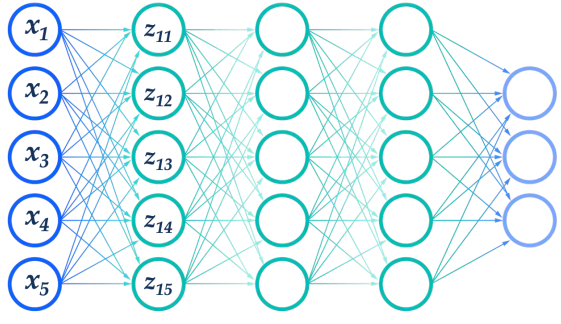
```
import torch.optim as optim

print(dir(optim))
```

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 hyper-parameter? Parameters are **trainable**, which means that are parameters, whereas the step size is an hyper-parameter.

Deep Neural Networks (DNNs) stack **layers** of neurons



The output of each layer is computed as a function of the product of and the output of the previous layer

Composing a DNN

A multi-layer network can be written as follows:

```
x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
y = f(w_n * x_n + b_n)
```

where the output of a layer of neurons is used as an input for the following layer.

Composing a DNN

However, PyTorch has a whole submodule dedicated to neural networks, called `torch.nn`.

It contains the building blocks needed to create all sorts of neural network architectures.

Those building blocks are called modules. A PyTorch module is a Python class deriving from the `nn.Module` base class.

A module can have one or more `Parameter` instances as attributes, which are tensors whose values are optimized during the training process.

A module can also have one or more *submodules* (subclasses of `nn.Module`) as attributes, and it will be able to track their parameters as well.

Composing a DNN

we can find a subclass of `nn.Module` called `nn.Linear`, which applies an affine transformation to its input (via the parameter attributes weight and bias) and is equivalent to a linear classifier.

We will compose these units with the activation functions to obtain a DNN.

A dataset of tiny images

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

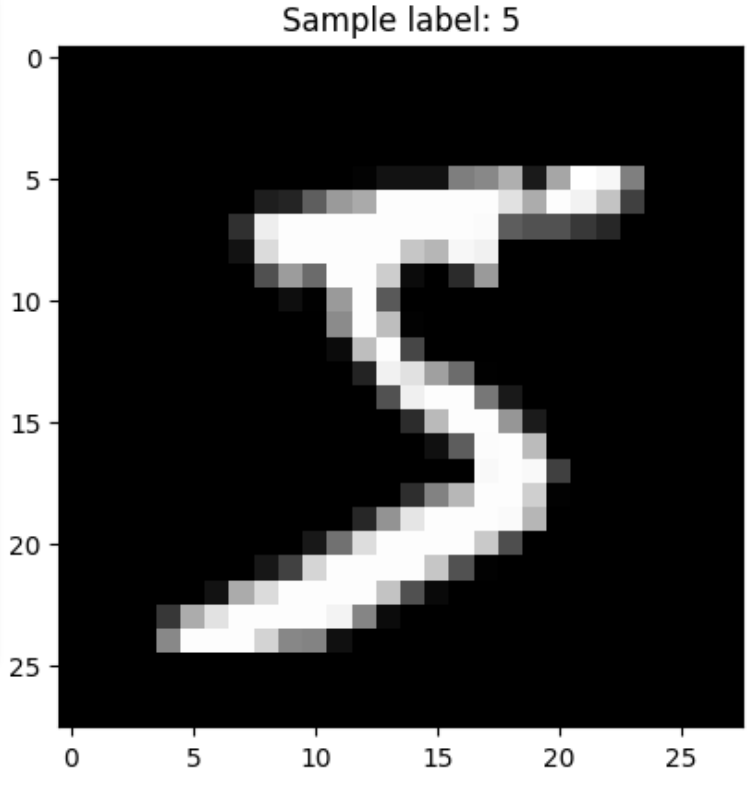
Each of these images has a shape of and represents a digit from to

The labels are, as well, numbers from to

Downloading MNIST

The easiest way to download the MNIST dataset is to use the `torchvision` library

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

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

print("samples in training dataset: ",
      len(mnist_train))

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



Transformations

That's all very nice, but we'll likely need a way to convert the PIL image to a PyTorch tensor before we can do anything with it

```
from torchvision import transforms

to_tensor = transforms.ToTensor()
image_as_tensor = to_tensor(image)
print(image_as_tensor.shape)
```

We can also define other tranformations, for example random rotations. These are useful to obtain data augmentations, *i.e.*, to have artificial variations of the images:

```
# random rotation between 0 an 90 degrees
rotation = transforms.RandomRotation(90)
rotated = rotation(image_as_tensor)

def display_image(image, label):
    # permute required to transform back in the PIL format
    plt.imshow(image.permute(1, 2, 0), cmap='gray')
    plt.title(f"Sample label: {label}")
    plt.show()

display_image(rotated, label)
```

And we can apply the transform to the whole dataset at loading time:

```
tensor_mnist_train = datasets.MNIST(data_path,
                                    train=True,
                                    download=False,
                                    transform=transforms.ToTensor())

sample, label = tensor_mnist_train[0]
print(type(sample), sample.dtype, sample.shape)
```

Behavior of the `ToTensor` transformation

Whereas the values in the original PIL image ranged from to (bits per channel), the `ToTensor` transform turns the data into a 32-bit floating point per channel, scaling the values down from to . Let's verify that:

```
import numpy as np
print(f"image min: {np.array(image).min()}, image max: {np.array(image).max()}")
print(f"tensor min: {sample.min()}, tensor max: {sample.max()}")
```

Normalizing the data

Let's now normalize the data so that they have mean and standard deviaion . First, we compute the mean and standard deviation:

```
dataset = torch.stack([sample for sample, _ in tensor_mnist_train], dim=3)
print(dataset.shape)
means = dataset.view(1, -1).mean(dim=1)
stds = dataset.view(1, -1).std(dim=1)
print(means)
print(stds)
normalize = transforms.Normalize(means, stds)
```

Normalizing the data

Then we can apply now the `ToTensor`, chained with a normalization operation:

```
transformed_mnist_train = datasets.MNIST(data_path, train=True,
                                         download=False,
                                         transform=transforms.Compose([
                                         transforms.ToTensor(),
                                         normalize,]))

dataset = torch.stack([sample for sample, _ in transformed_mnist_train], dim=3)

# let's verify that the mean and standard deviation are now as we want them
means = dataset.view(1, -1).mean(dim=1)
stds = dataset.view(1, -1).std(dim=1)
print(f"mean: {means}")
print(f"std: {stds}")
```

A fully connected model

Let's create a model able to process the MNIST dataset

The important part is that we have an input of features and an output of classes

```
class MNISTModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = torch.nn.Linear(784, 512)
        self.fc2 = torch.nn.Linear(512, 10)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        # we have to flatten the samples that are 28x28
        x = x.view(-1, 784)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Now let's create the model and get the prediction for one sample:

```
net = MNISTModel()

sample, label = transformed_mnist_train[0]

# prediction from untrained model
out = net(sample)
print("scores:", out)

# predicted class
pred = out.argmax(dim=1)
print("predicted class:", pred.item())
print("original label: ", label)
```

Loading our data in batches

To properly load data in batches, we create data loaders, that are classes to load data dynamically from a dataset. They load batches of a specified batch size and optionally shuffle the samples.

```
transformed_mnist_validation = datasets.MNIST(data_path, train=False,
                                              download=False,
                                              transform=transforms.Compose([
                                              transforms.ToTensor(),
                                              normalize,]))  # same normalize as

train_loader = torch.utils.data.DataLoader(transformed_mnist_train,
                                           batch_size=64,
                                           shuffle=True)
val_loader = torch.utils.data.DataLoader(transformed_mnist_validation,
                                         batch_size=64,
                                         shuffle=False)
```

Data loaders

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

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

Or also by putting it in a `for` loop:

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

Training our model

We can finally write the code for training our model:

```
learning_rate = 1e-2
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
loss_fn = torch.nn.CrossEntropyLoss()
epochs = 2

for epoch in range(epochs):
    for samples, labels in train_loader:
        outputs = net(samples)
        loss = loss_fn(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch: {epoch + 1}, Loss: {loss}")
```

Evaluating our model

And for testing it:

```
accuracy = 0.0
total = 0
for samples, labels in train_loader:
    outputs = net(samples)
    predictions = outputs.argmax(dim=1)
    # we have to compute the total number of samples every time
    # because the last batch can be smaller than the batch size
    total += samples.shape[0]
    accuracy += (predictions.type(labels.dtype) == labels).float().sum()
accuracy = accuracy / total

print(f"Accuracy: {accuracy}")
```

An improved training loop

It't best to isolate the training and evaluation loops in functions and to keep track of the training and validation losses to monitor the training function:

```
train_losses, val_losses = [], []

for epoch in range(epochs):
    train_loss = train(model, train_loader, optimizer, loss_fn)
    val_loss, accuracy = validate(model, test_loader)
    train_losses.append(train_loss)
    val_losses.append(val_loss)

import matplotlib.pyplot as plt

plt.plot(train_losses, label='train loss')
plt.plot(val_losses, label='validation loss')
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
```

Limits of going fully connected

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

In the next lecture we will see how to make the model focus on 2D representations rather than vectorized images.

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

# let's see how the network generalizes
augmented_mnist_validation = datasets.MNIST(data_path, train=False,
                                            download=False,
                                            transform=transforms.Compose([
                                            transforms.ToTensor(),
                                            transforms.Lambda(shift_pixels),
                                            normalize,]))
augmented_val_loader = torch.utils.data.DataLoader(augmented_mnist_validation,
                                           batch_size=64,
                                           shuffle=False)

images, labels = augmented_mnist_validation[0]
display_image(images, labels)

val_loss_augmented, accuracy_augmented = valid_epoch(net, augmented_val_loader,

print("accuracy: ", accuracy, "accuracy after augmentation:", accuracy_augmented
```

What are the solutions to this problem?

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)

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) **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 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 version for now -

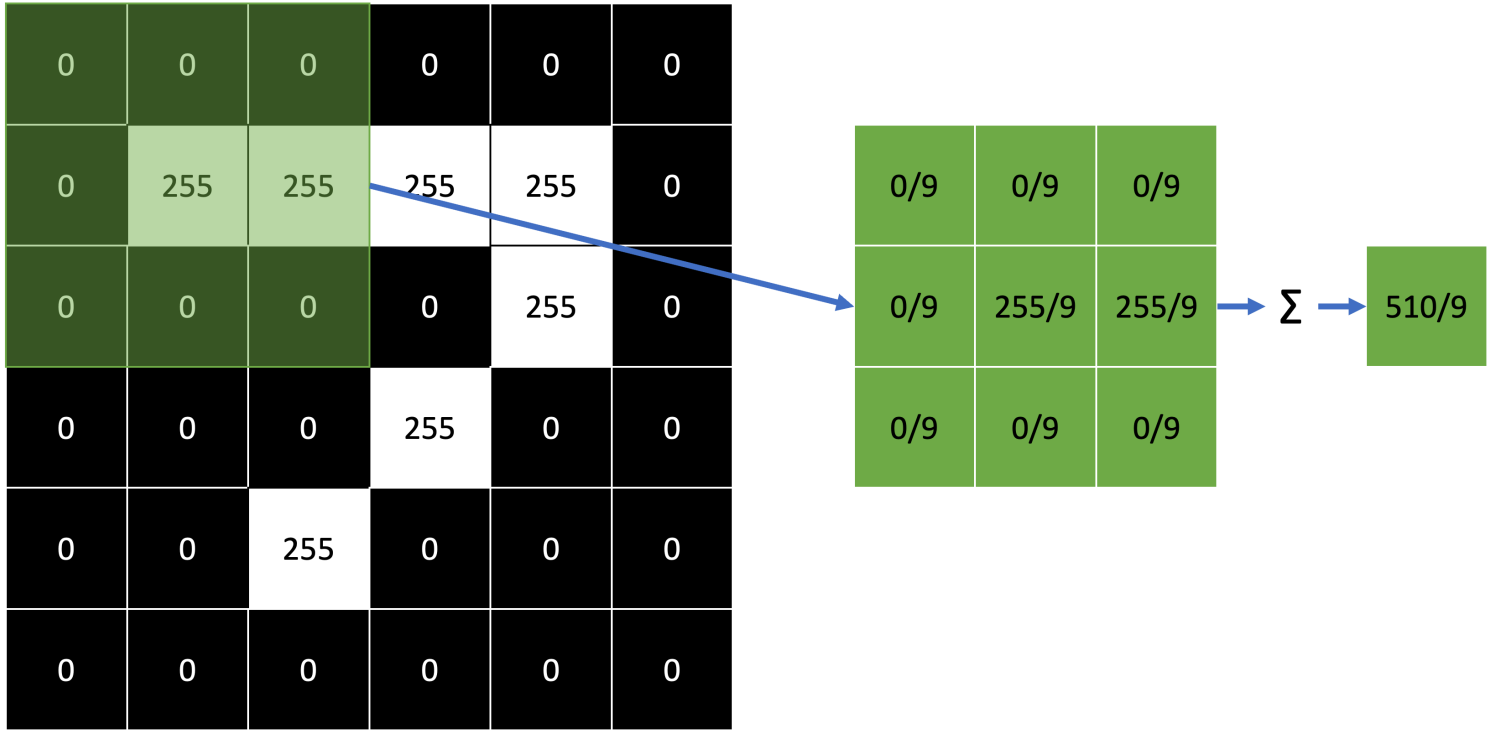
1. simply multiplies every pixel by ;
2. and sums all results, placing them in the resulting pixel.

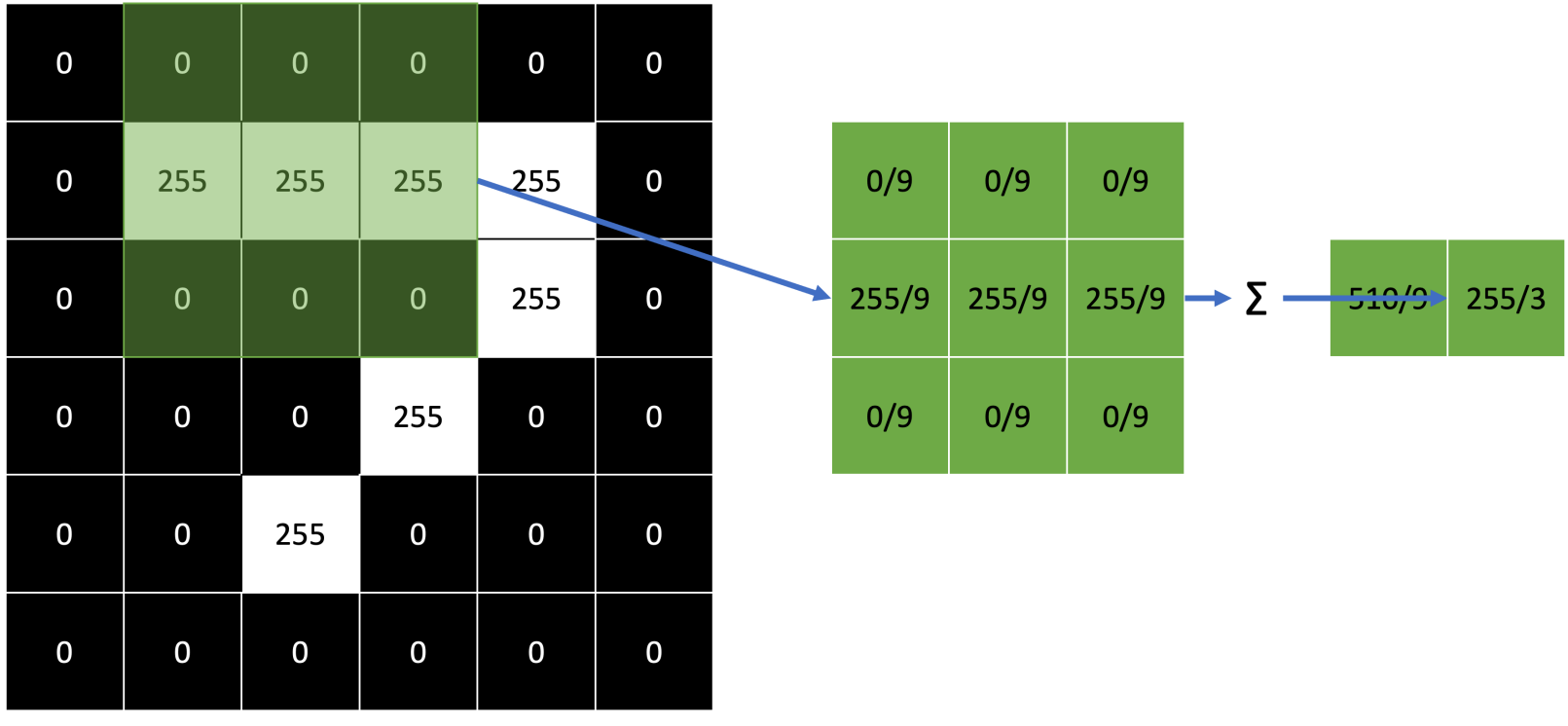
This corresponds to the averagin operation.

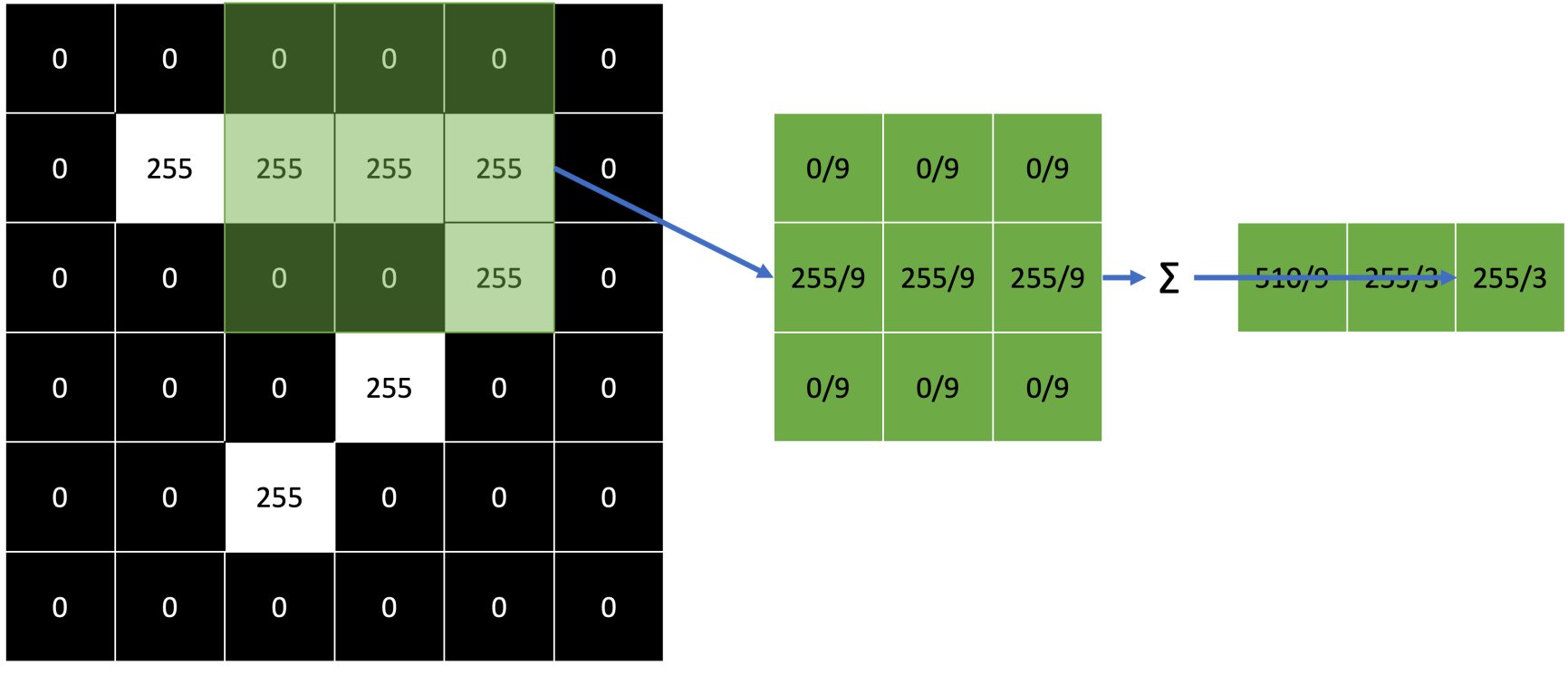
Then, the filter slides of one position and computes again the average.

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
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

Let's implement this with 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 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 ()
- 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:

- = Ouput size
- = Input size
- = Filter size
- = Stride
- = 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

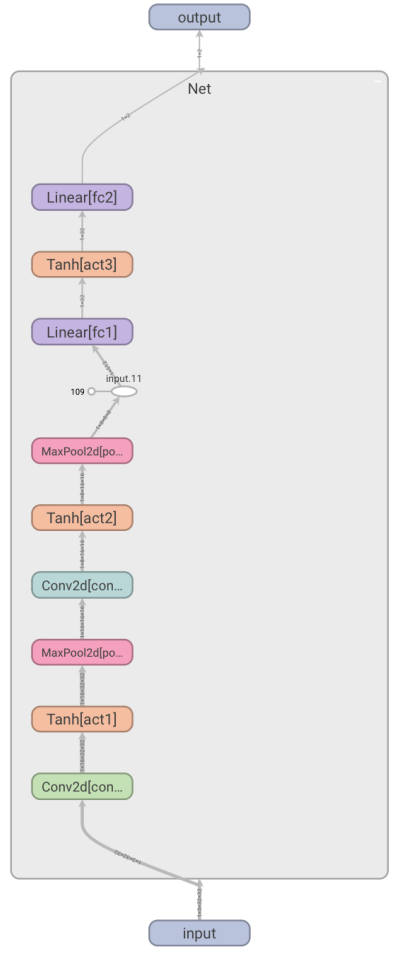
```
import torch
from torch import nn

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

- 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
- MaxPool, `K=2` -> output = 16 channels of size
- Conv layer, `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
- Linear (fully-connected) Layer of input size and output units
- Linear (fully-connected) Layer of input size and output as the 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
Net
Linear[fc2]
Tanh[act3]
Linear[fc1]
input.11
109
MaxPool2d[po...
Tanh[act2]
Conv2d[con...
MaxPool2d[po...
Tanh[act1]
Conv2d[con...
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.

```
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.

Where is the norm. In general, the or 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.

```
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:

```
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:

```
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:

```
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)
```