Part 04 - Learning from Tensors: Gradient Descent and Backpropagation

- 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)

Representing input data

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 $[100 \times 3]$ represents a dataset of 100 samples with 3 features each.

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

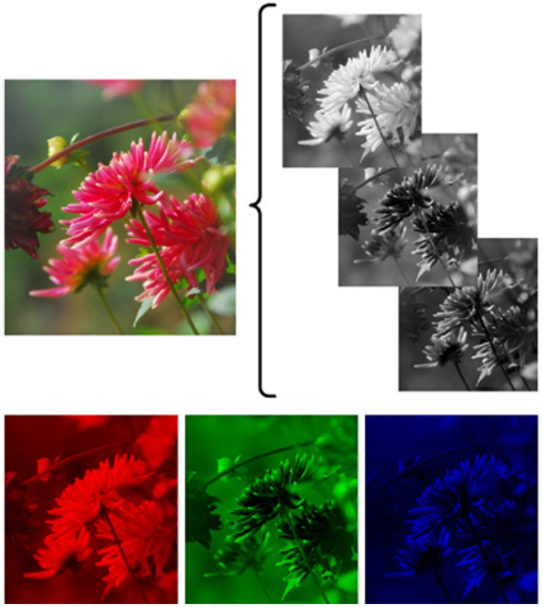
Images are represented as collections of scalars arranged in a specific grid with height x width grid points (pixels)

- 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, ...)

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 $[0, 255]$



Loading an image

```python
from PIL import Image
import numpy as np
import torchvision

with Image.open("dog.jpg") as im:

    im.show()

    # convert to numpy
    numpy_image = np.array(im)
    print(numpy_image)

    tensor = torchvision.transforms.ToTensor()(im)
    # the transform already normalizes the data in [0, 1]
    # we will soon know what it means
    print(tensor)

    print(tensor.shape)
```

https://pytorch.org/vision/0.9/transforms.html#torchvision.transforms.ToTensor

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 (flatten) 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:

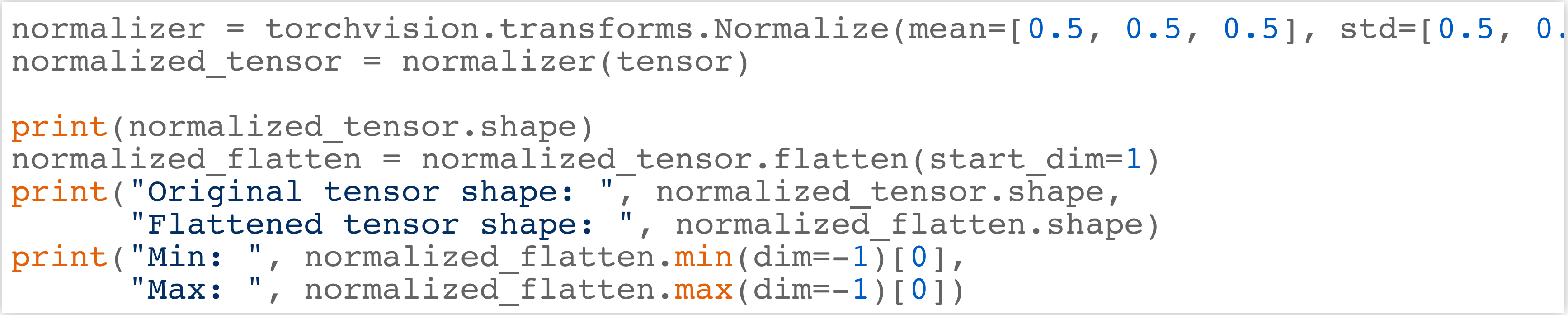
```
[num_samples, colors(=3), width, height]
```

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 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 s_2, and this is often called the **prediction** of the machine-learning model

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

```python
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.

```python
import torch
# let's create a column tensor with 10 scores for 4 samples
scores = torch.randn(4, 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.

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

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

Learning

As seen in Part 3, Learning is just Parameter Estimation

We have to find the parameters to approximate the unknown function

$f(\mathbf{x}, \boldsymbol{\theta}) = y$

To find the good parameters $\boldsymbol{\theta}$, we define a loss (*a.k.a.* cost) function that we want to minimize.

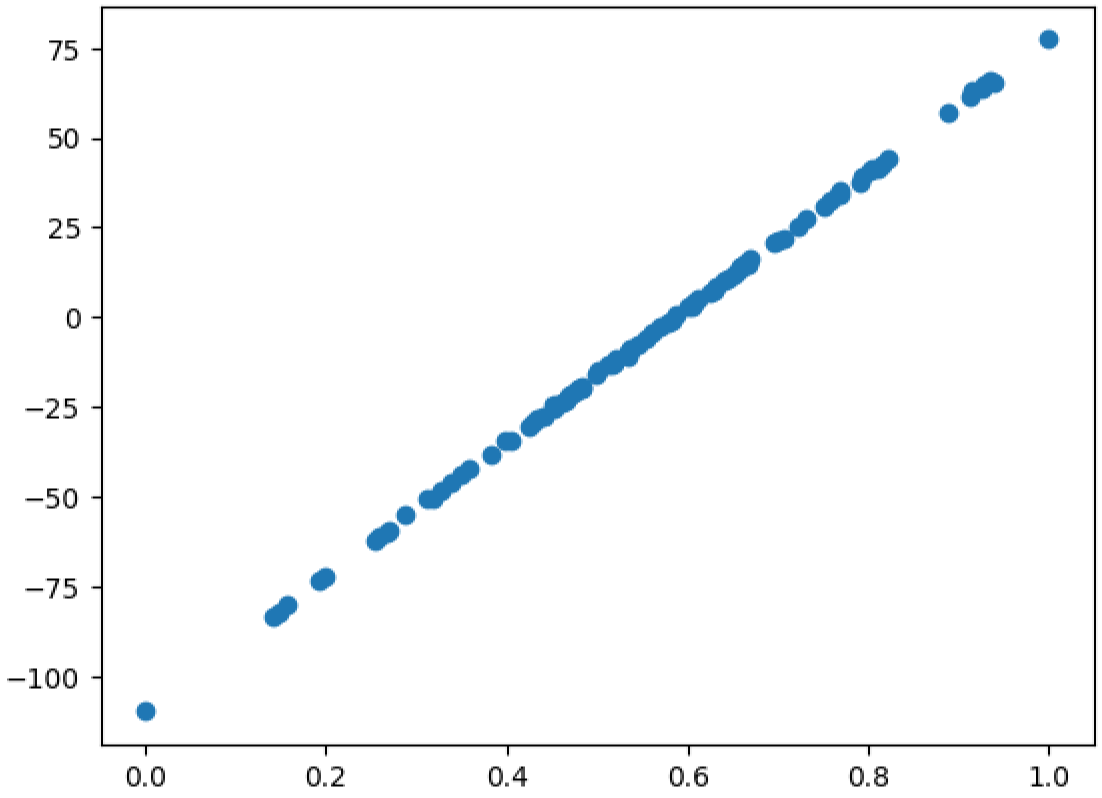
Learning

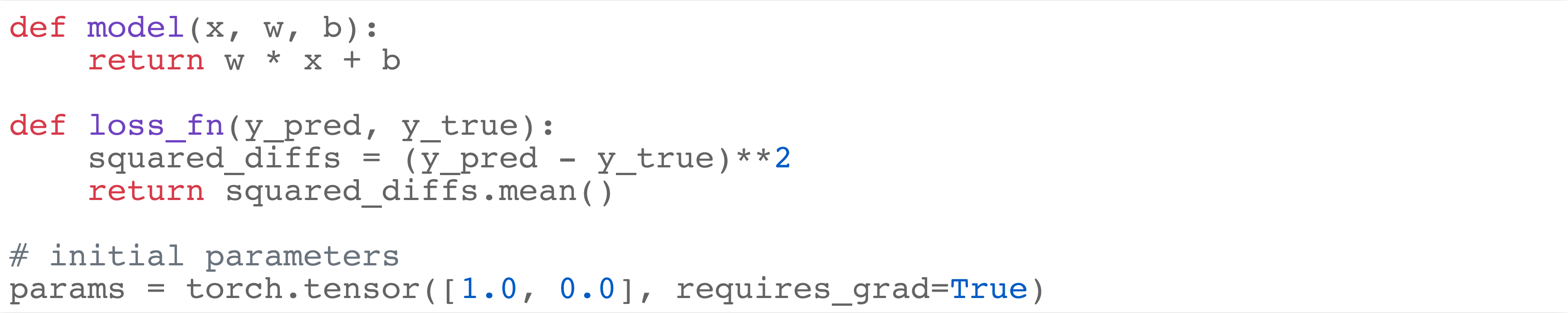
As we saw in previous chapters, learning involves using a loss and adjusting the parameters with gradients.

Learning

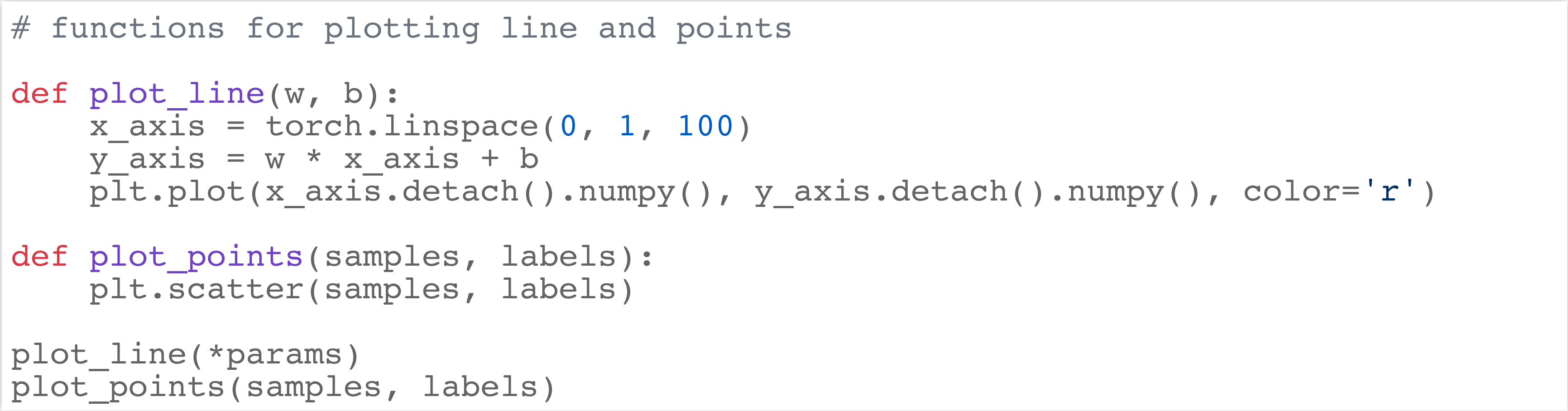
Let's start with a simple linear example. Let's fit a line to a distribution of points (regression).

```python
import torch
from sklearn import datasets
import matplotlib.pyplot as plt

samples, labels = datasets.make_regression(n_samples=100,
                                           n_features=1, noise=0.5,
                                           random_state=42)

samples, labels = torch.from_numpy(samples), torch.from_numpy(labels)

# normalization
samples -= samples.min()
samples /= samples.max()
samples = samples.ravel()

print(samples[:5], labels[:5])
plt.scatter(samples, labels)
```

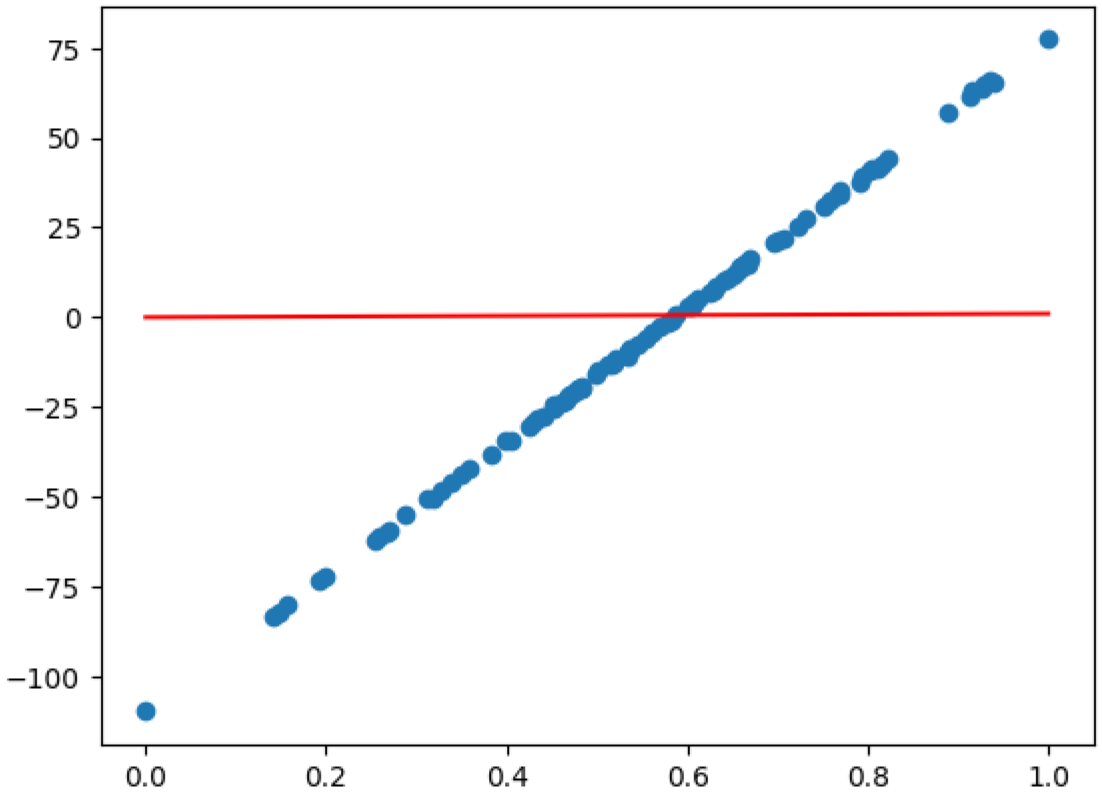

75
50
25
0
−25
−50
−75
−100
0.0
0.2
0.4
0.6
0.8
1.0

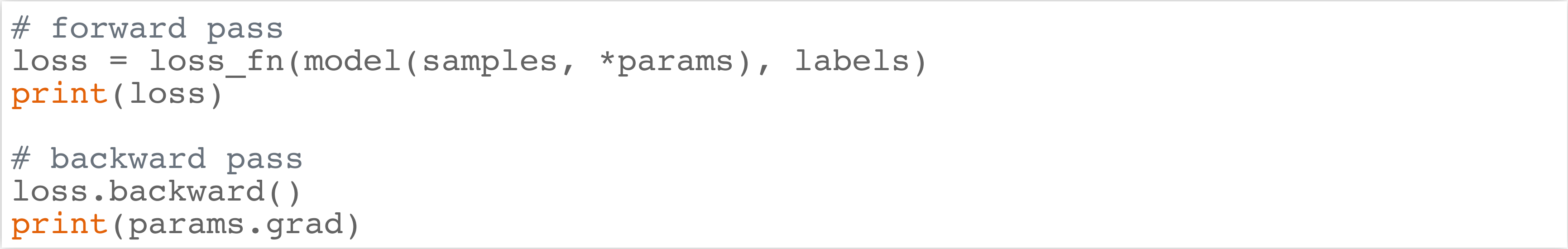


```
def model(x, w, b):
    return w * x + b

def loss_fn(y_pred, y_true):
    squared_diffs = (y_pred - y_true)**2
    return squared_diffs.mean()

# initial parameters
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

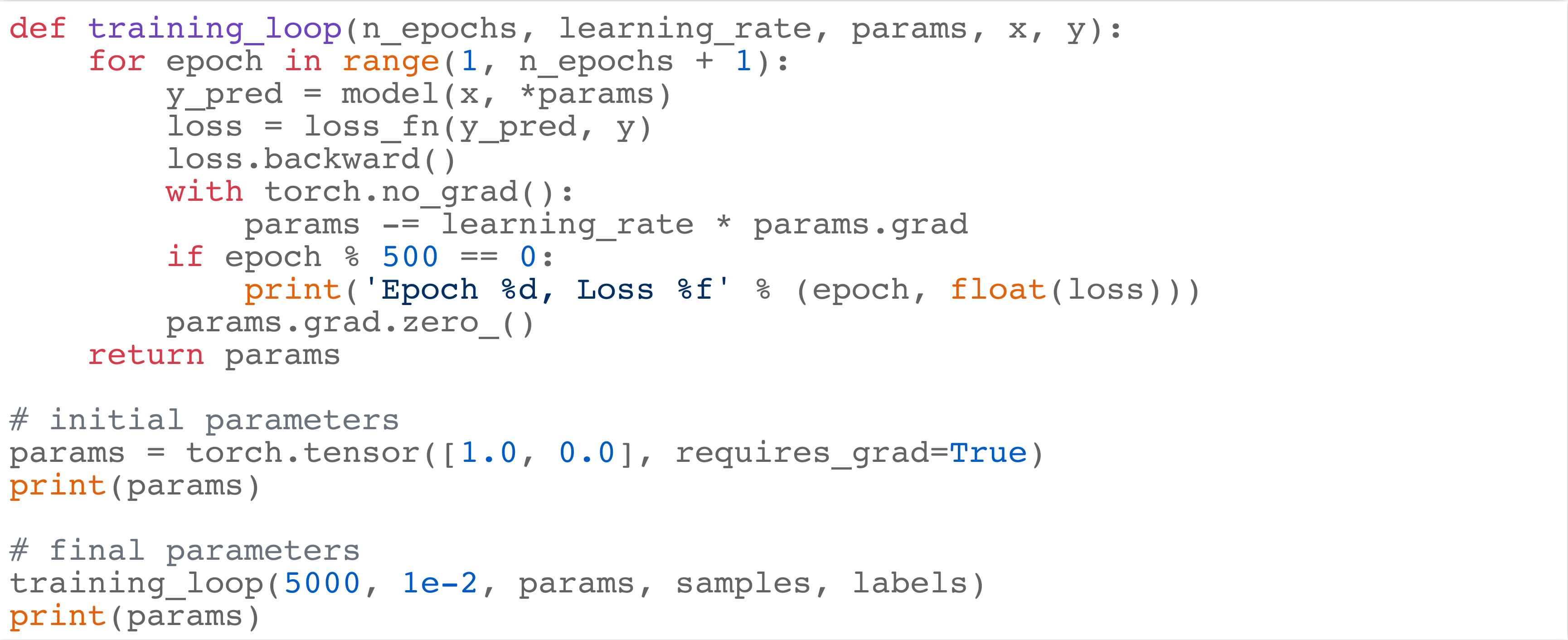


```
# functions for plotting line and points

def plot_line(w, b):
    x_axis = torch.linspace(0, 1, 100)
    y_axis = w * x_axis + b
    plt.plot(x_axis.detach().numpy(), y_axis.detach().numpy(), color='r')

def plot_points(samples, labels):
    plt.scatter(samples, labels)

plot_line(*params)
plot_points(samples, labels)
```

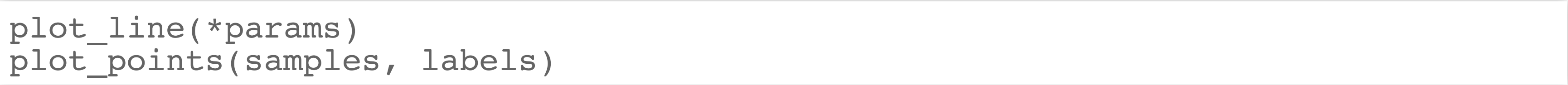

75
50
25
0
−25
−50
−75
−100
0.0
0.2
0.4
0.6
0.8
1.0



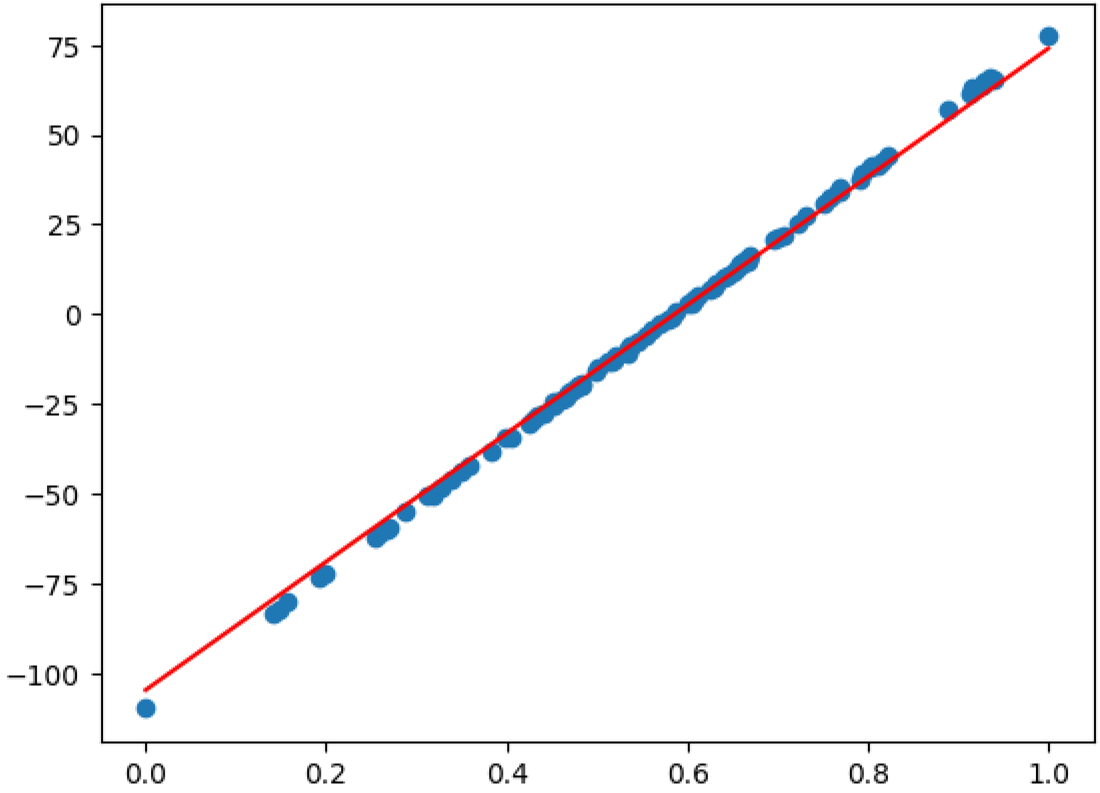
```
# forward pass
loss = loss_fn(model(samples, *params), labels)
print(loss)

# backward pass
loss.backward()
print(params.grad)
```



```python
def training_loop(n_epochs, learning_rate, params, x, y):
    for epoch in range(1, n_epochs + 1):
        y_pred = model(x, *params)
        loss = loss_fn(y_pred, y)
        loss.backward()
        with torch.no_grad():
            params -= learning_rate * params.grad
        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
        params.grad.zero_()
    return params

# initial parameters
params = torch.tensor([1.0, 0.0], requires_grad=True)
print(params)

# final parameters
training_loop(5000, 1e-2, params, samples, labels)
print(params)
```



```
plot_line(*params)
plot_points(samples, labels)
```


75
50
25
0
−25
−50
−75
−100
0.0
0.2
0.4
0.6
0.8
1.0

Note that our code updating params is not quite as straightforward as we might have expected.

There are two particularities.

1. We are encapsulating the update in a `no_grad` context using the Python `with` statement. This means within the with block, the PyTorch autograd mechanism should look away that is, **not add edges to the forward graph**.
2. We update params in place. This means **we keep the same params tensor** around but subtract our update from it.

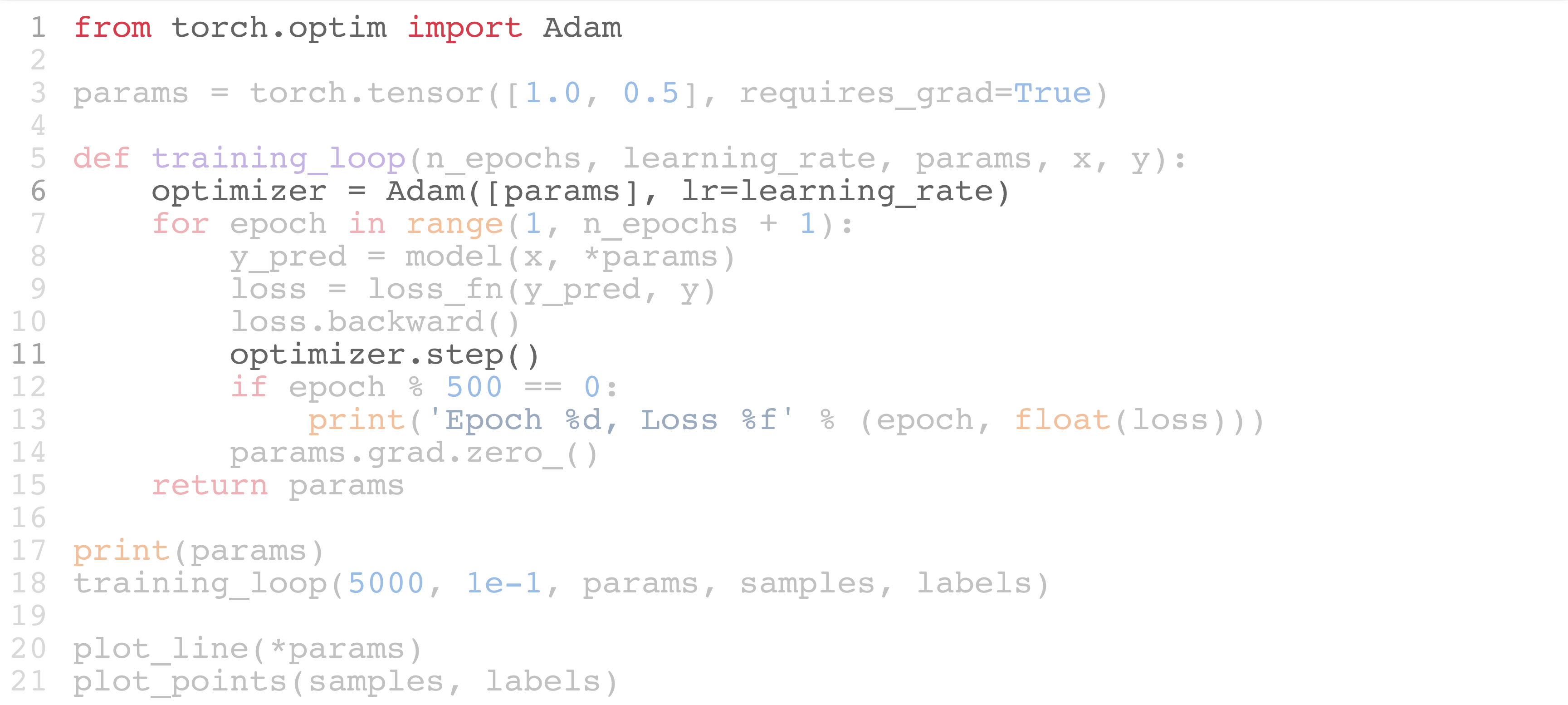
Optimizers

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



```python
from torch.optim import Adam

params = torch.tensor([1.0, 0.5], requires_grad=True)

def training_loop(n_epochs, learning_rate, params, x, y):
    optimizer = Adam([params], lr=learning_rate)
    for epoch in range(1, n_epochs + 1):
        y_pred = model(x, *params)
        loss = loss_fn(y_pred, y)
        loss.backward()
        optimizer.step()
        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
        params.grad.zero_()
    return params

print(params)
training_loop(5000, 1e-1, params, samples, labels)

plot_line(*params)
plot_points(samples, labels)
```

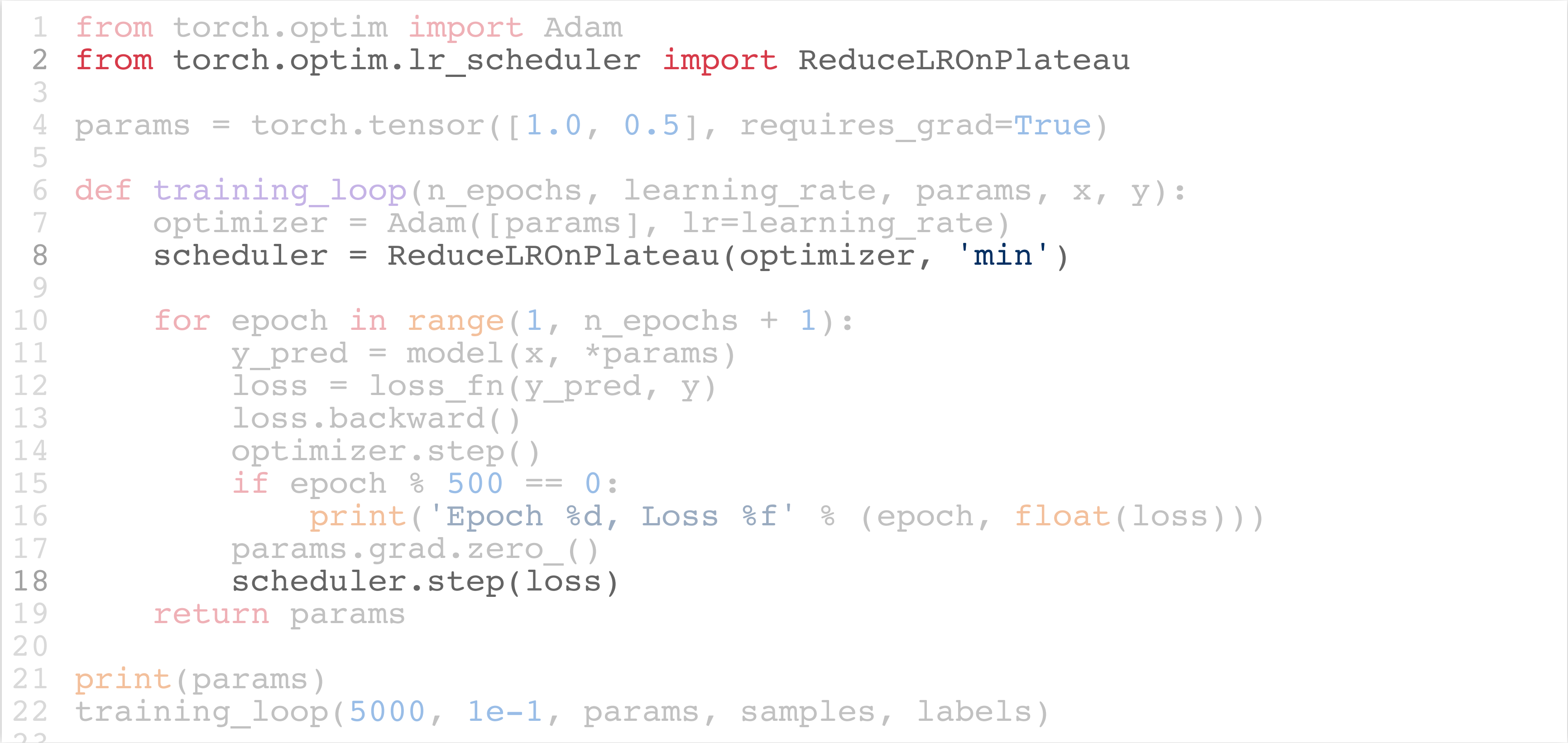
… and to see available optimizers:

```
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 w and b from the example above are parameters.



```python
from torch.optim import Adam
from torch.optim.lr_scheduler import ReduceLROnPlateau

params = torch.tensor([1.0, 0.5], requires_grad=True)

def training_loop(n_epochs, learning_rate, params, x, y):
    optimizer = Adam([params], lr=learning_rate)
    scheduler = ReduceLROnPlateau(optimizer, 'min')

    for epoch in range(1, n_epochs + 1):
        y_pred = model(x, *params)
        loss = loss_fn(y_pred, y)
        loss.backward()
        optimizer.step()
        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
        params.grad.zero_()
        scheduler.step(loss)
    return params

print(params)
training_loop(5000, 1e-1, params, samples, labels)
```

From linear units to DNNs

Note that the optimizer is not the only flexible part of our training loop

For example, the model is also flexible. In order to train a neural network on the same data and the same loss, all we would need to change is the model function

From linear units to DNNs

At the core of deep learning are neural networks: mathematical entities capable of representing complicated functions through a composition of simpler functions

They combine linear functions (as the one that we just trained, basically a line) and non-linear functions (often called *activations*)

The basic building block of these complicated functions is the **neuron**

The neurons

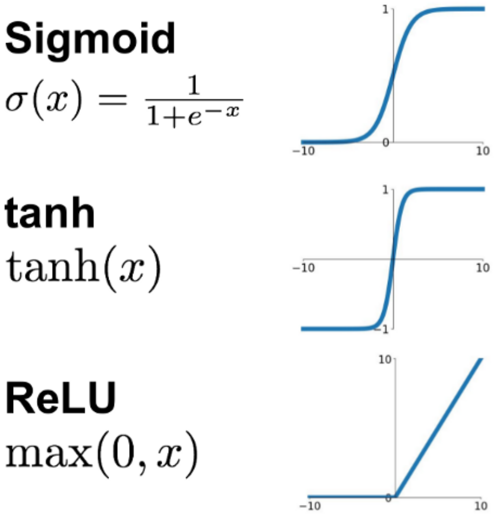
At its core, a neuron is a **linear transformation of the input** (for example, multiplying the input by a number [the *weight*] and adding a constant [the *bias*]) followed by the application of a **fixed nonlinear function** (referred to as the *activation* function).

Mathematically, we can write the neuron as

$$\mathbf{z}_l = a(\mathbf{w}_l \mathbf{z}_{l-1})$$

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

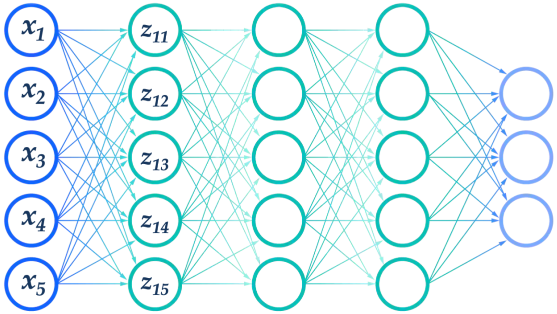
… **Are nonlinear**

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.

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



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 $\mathbf{z}_{l-1}$

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

Building models out of stacks of linear transformations followed by differentiable activations leads to models that can approximate **highly nonlinear processes** and whose parameters we can estimate surprisingly well through gradient descent.

Composing a DNN

With a deep neural network model, we have a universal approximator and a method to estimate its parameters.

Starting from a generic, untrained model, we specialize it on a task by providing it with a set of inputs and outputs and a loss function from which to backpropagate (**train**)

Composing a DNN

Our first step will be to replace our *naive* linear model with a neural network unit.

This will be a somewhat useless step for now, but it will still be instrumental for starting on a sufficiently simple problem and scaling up later.

Composing a DNN

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.

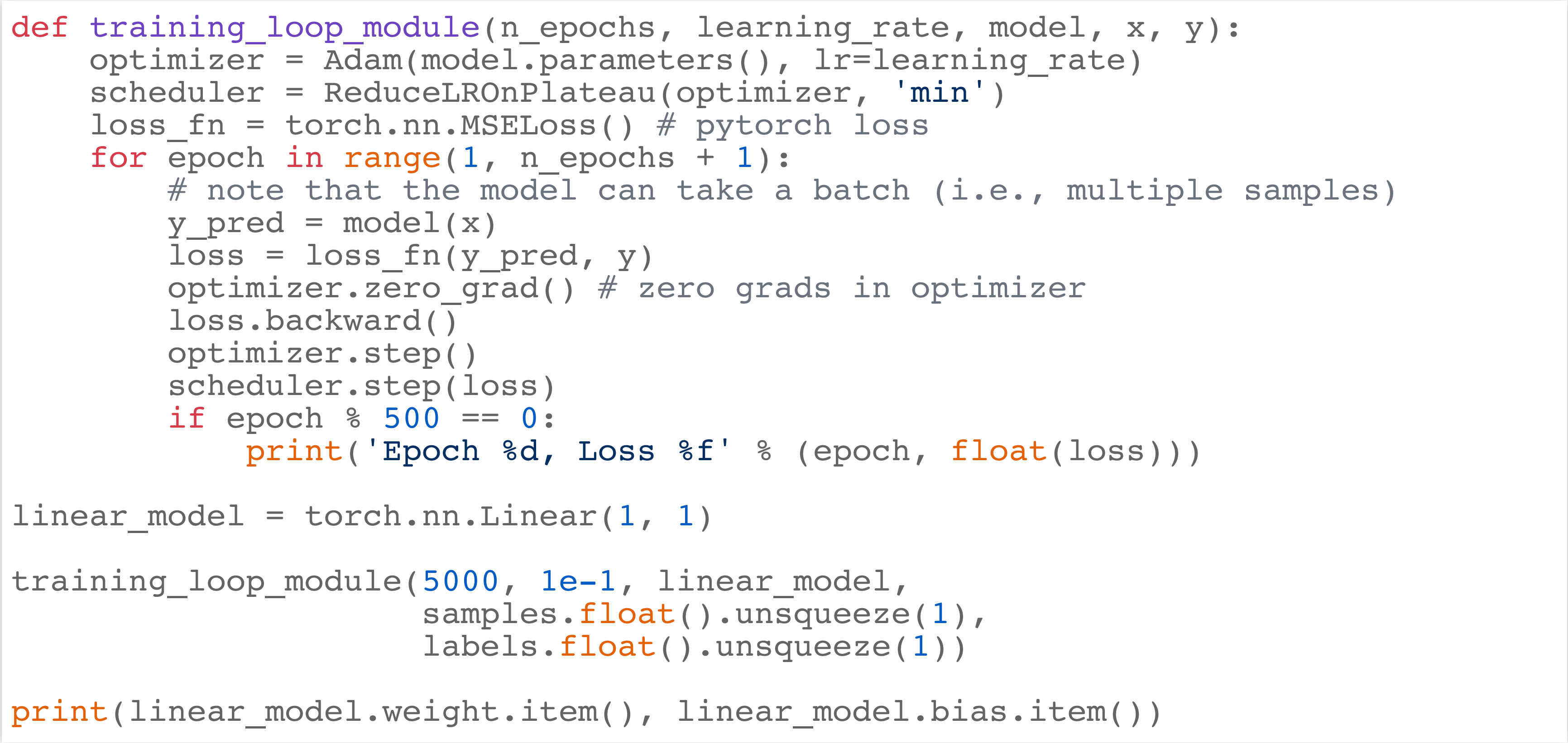
Those building blocks are called modules. A PyTorch module is a Python class deriving from the `nn.Module` base class.

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 what we implemented earlier without the module.



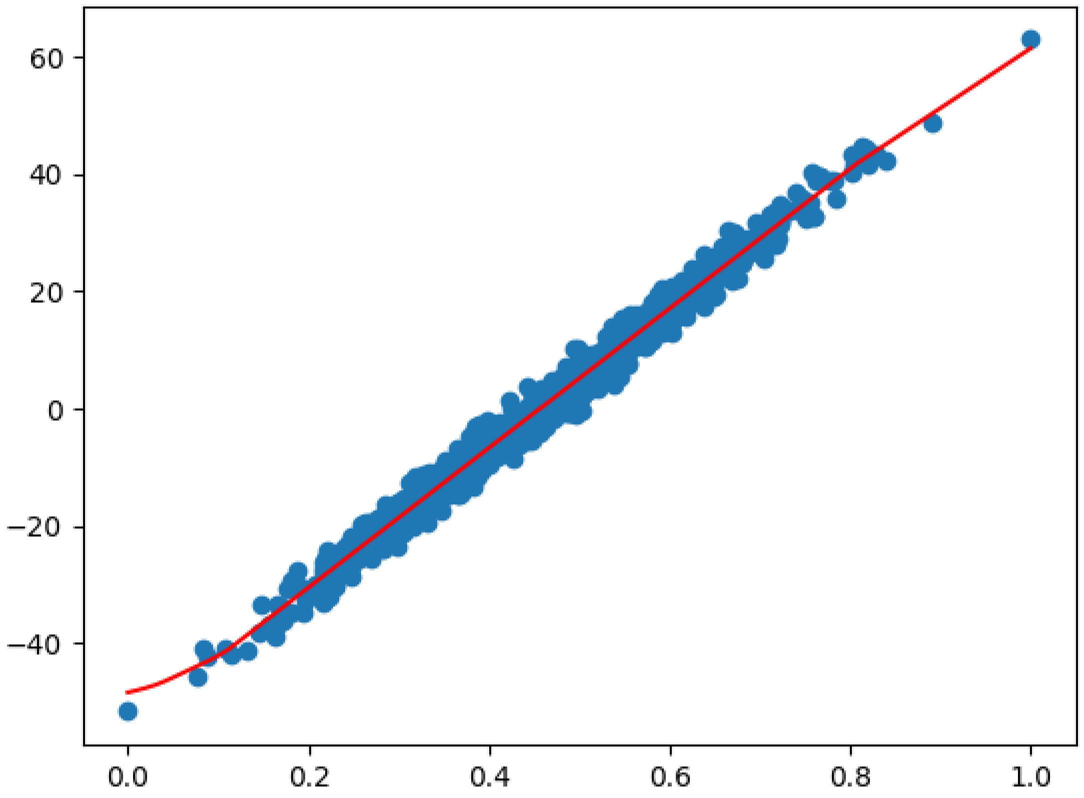
```
def training_loop_module(n_epochs, learning_rate, model, x, y):
    optimizer = Adam(model.parameters(), lr=learning_rate)
    scheduler = ReduceLROnPlateau(optimizer, 'min')
    loss_fn = torch.nn.MSELoss() # pytorch loss
    for epoch in range(1, n_epochs + 1):
        # note that the model can take a batch (i.e., multiple samples)
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        optimizer.zero_grad() # zero grads in optimizer
        loss.backward()
        optimizer.step()
        scheduler.step(loss)
        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

linear_model = torch.nn.Linear(1, 1)

training_loop_module(5000, 1e-1, linear_model,
                     samples.float().unsqueeze(1),
                     labels.float().unsqueeze(1))

print(linear_model.weight.item(), linear_model.bias.item())
```

We can also define now the following function to plot the output of the DNN:

```python
def plot_module(model):
    x_axis = torch.linspace(0, 1, 100).unsqueeze(1)
    y_axis = model(x_axis)
    plt.plot(x_axis.detach().numpy(), y_axis.detach().numpy(), color='r')

plot_module(linear_model)
```

Finally, a DNN

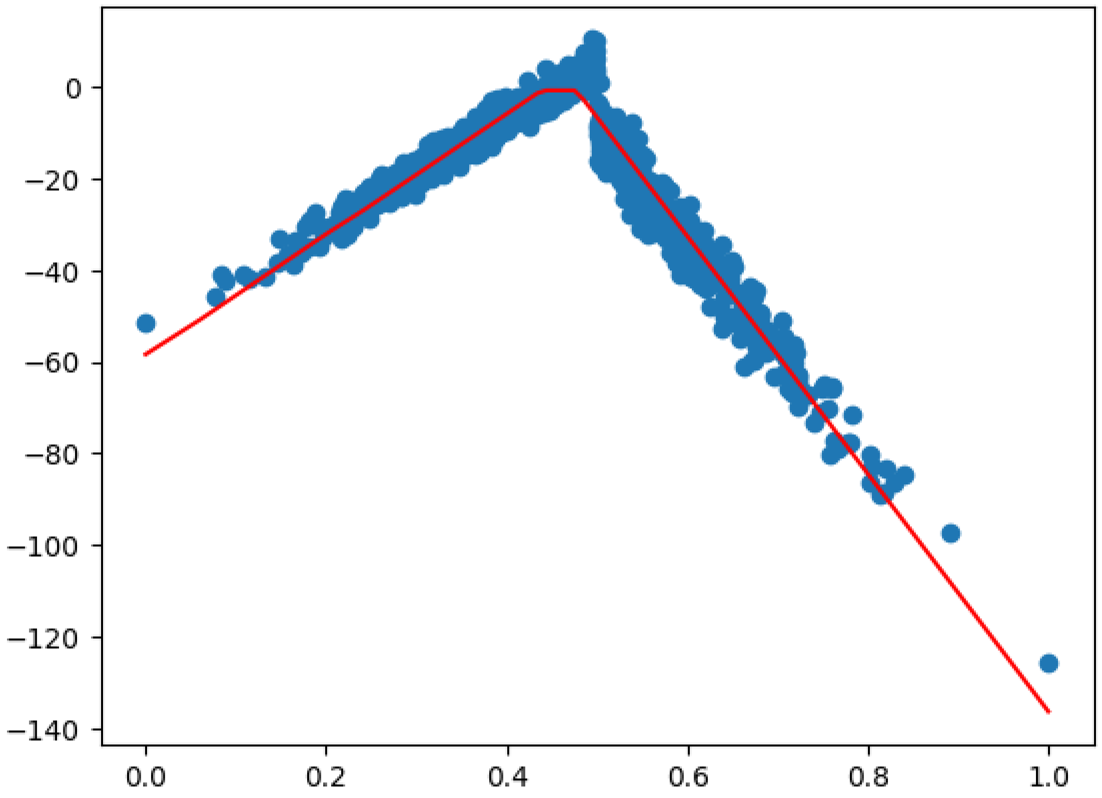
```
dnn = torch.nn.Sequential(
    torch.nn.Linear(1, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1),
)

training_loop_module(5000, 1e-1, dnn, samples.float().unsqueeze(1), labels.float

def plot_module(model):
    x_axis = torch.linspace(0, 1, 100).unsqueeze(1)
    y_axis = model(x_axis)
    plt.plot(x_axis.detach().numpy(), y_axis.detach().numpy(), color='r')

plot_module(dnn)
plot_points(samples, labels)
```



60
40
20
0
−20
−40
0.0
0.2
0.4
0.6
0.8
1.0

Let's make the data highly non-linear

```
y_new = labels.clone()
y_new[samples>0.5] *= -2  # this makes the labels flip at x=0.5

dnn = torch.nn.Sequential(
    torch.nn.Linear(1, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1),
)

training_loop_module(5000, 1e-1, dnn, samples.float().unsqueeze(1),
                     y_new.float().unsqueeze(1))

def plot_module(model):
    x_axis = torch.linspace(0, 1, 100).unsqueeze(1)
    y_axis = model(x_axis)
    plt.plot(x_axis.detach().numpy(), y_axis.detach().numpy(), color='r')

plot_module(dnn)
plot_points(samples, y_new)
```


0
−20
−40
−60
−80
−100
−120
−140
0.0
0.2
0.4
0.6
0.8
1.0

Building more complicated models

Models can also be created by defining a class that inherits from `nn.Module`

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        """Defines the layers"""
        super().__init__()
        self.fc1 = nn.Linear(1, 10)
        self.fc2 = nn.Linear(10, 10)
        self.out = nn.Linear(10, 1)
        # no need to define the relu twice
        # we can call it in the forward
        self.relu = nn.ReLU()

    def forward(self, x):
        """Defines the operations applied to x"""
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        out = self.out(x)
        return out
```

Using `__call__` rather than `forward`

All PyTorch-provided subclasses of `nn.Module` have their `__call__` method defined

Calling an instance of `nn.Module` produces the same output of calling the `forward`

```
y = model(x)
y = model.forward(x)
```

However, there is a silent error here, for which it is always recommended to use `__call__`

Using `__call__` rather than `forward`

```python
def __call__(self, *input, **kwargs):
    for hook in self._forward_pre_hooks.values():
        hook(self, input)

    result = self.forward(*input, **kwargs)

    for hook in self._forward_hooks.values():
        hook_result = hook(self, input, result)
        ...
    for hook in self._backward_hooks.values():
        ...
    return results
```

There are hidden functions that are called inside the `__call__` before the `forward`, and calling the `forward` alone would skip all these operations

Hooks are used to customize the call with user-defined functions, *e.g.*, logging functions

Learning from images

We can now move to more complicated that than the linear data

We will use a dataset of small images at first, then move to more complex data

A dataset of tiny images

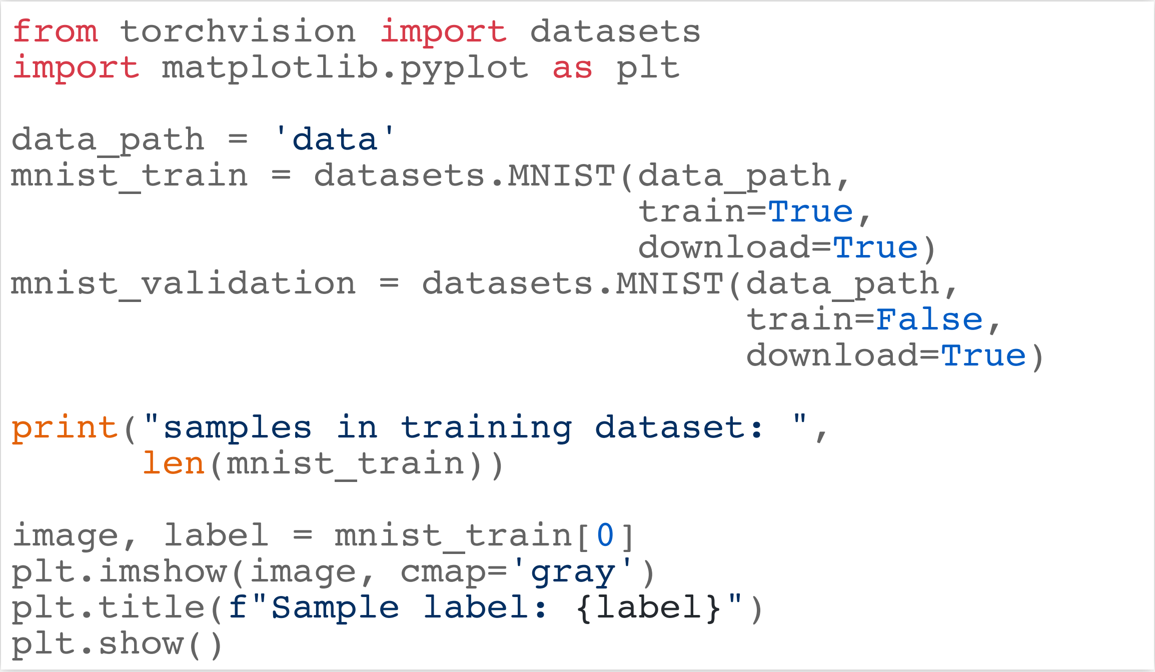
The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples

Each of these images has a shape of $[28 \times 28]$ and represents a digit from 0 to 9

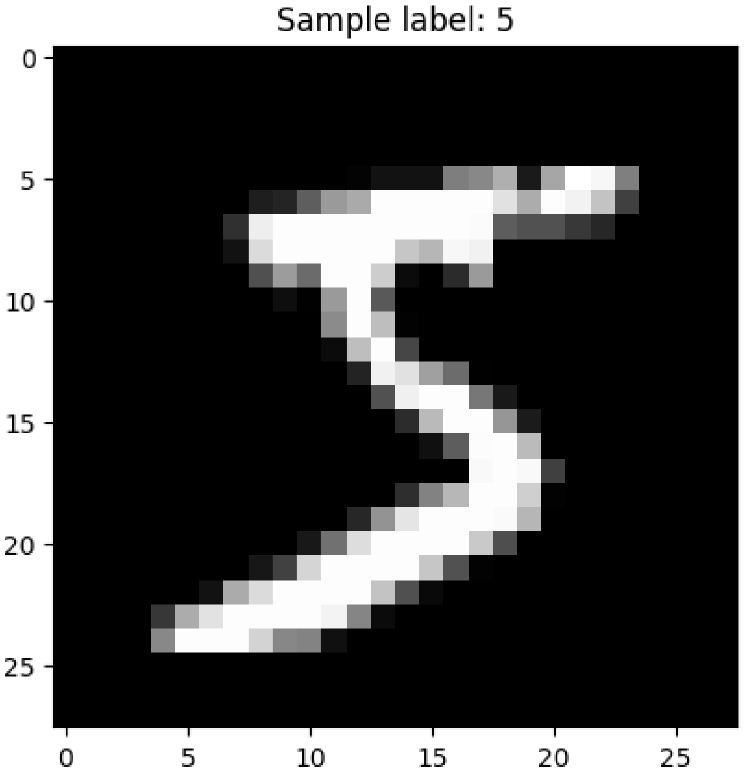
The labels are, as well, numbers from 0 to 9

Downloading MNIST

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



```python
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()
```

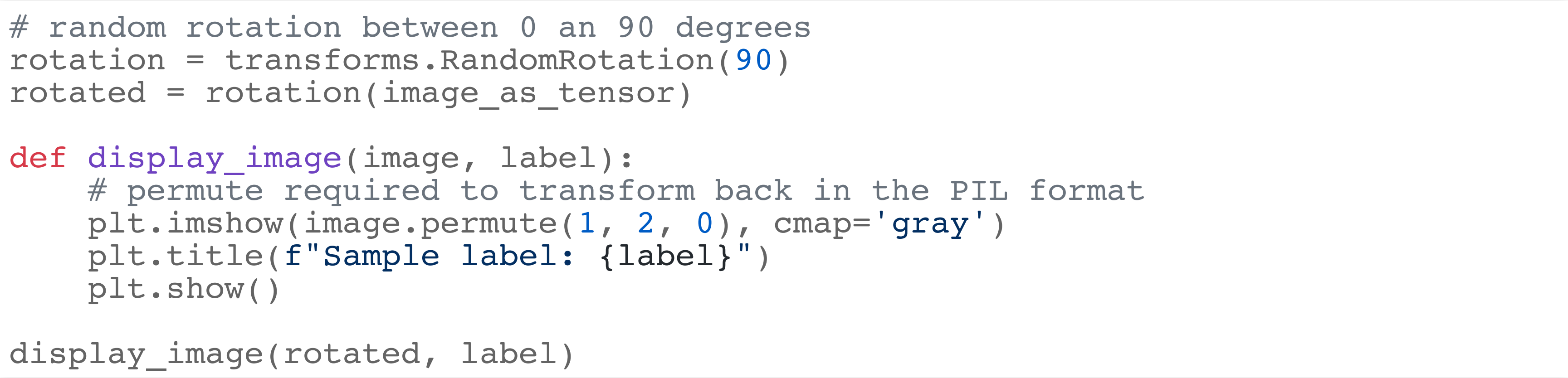


Transformations

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

```python
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 0 to 255 (8 bits per channel), the `ToTensor` transform turns the data into a 32-bit floating point per channel, scaling the values down from 0.0 to 1.0. Let's verify that:

```python
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 $\mu = 0$ and standard deviaion $\sigma = 1$. 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 $28x28 = 784$ features and an output of 10 classes

```python
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
```

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 $l(f(\mathbf{x}_i, y_i))$, where $(\mathbf{x}_i, y_i)$ 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.

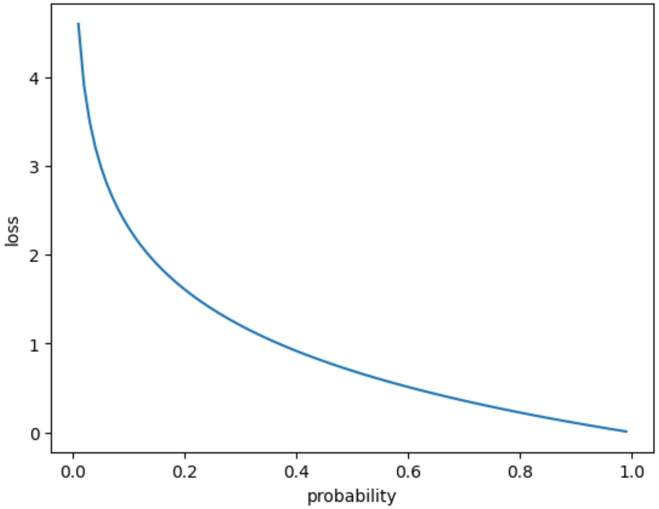
Loss functions for training

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

$$l = -(y \log(p) + (1 - y) \log(1 - p))$$

Where p is the predicted probability and y 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:

$$l(\mathbf{x}, y) = \sum_{c=1}^{C} y_c \log(f(\mathbf{x}))$$

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`

Training our model

We can finally write the code for training our model:

```python
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:

```python
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()
```

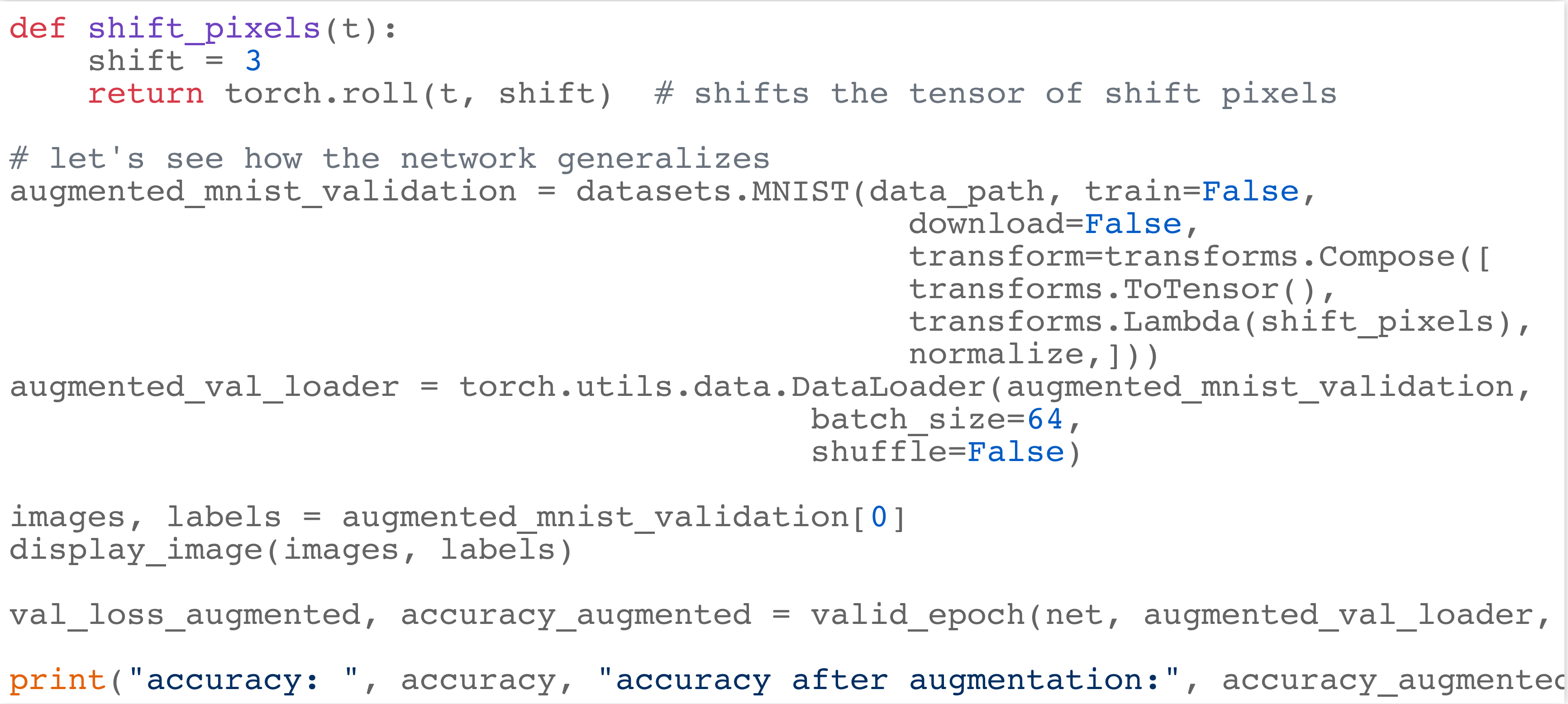
Limits of going fully connected

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?

End of part 4

Summary:

- Autograd with PyTorch
- Simple regression model
- Building a DNN
- DNN Classification (MNIST)

End of part 4

In the next chapter:

- Convolutional neural networks
- Designing a more complicated model
- Tracking results and experiments

- 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)