Discovering activation functions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Limitations of the sigmoid and softmax function

Sigmoid functions:

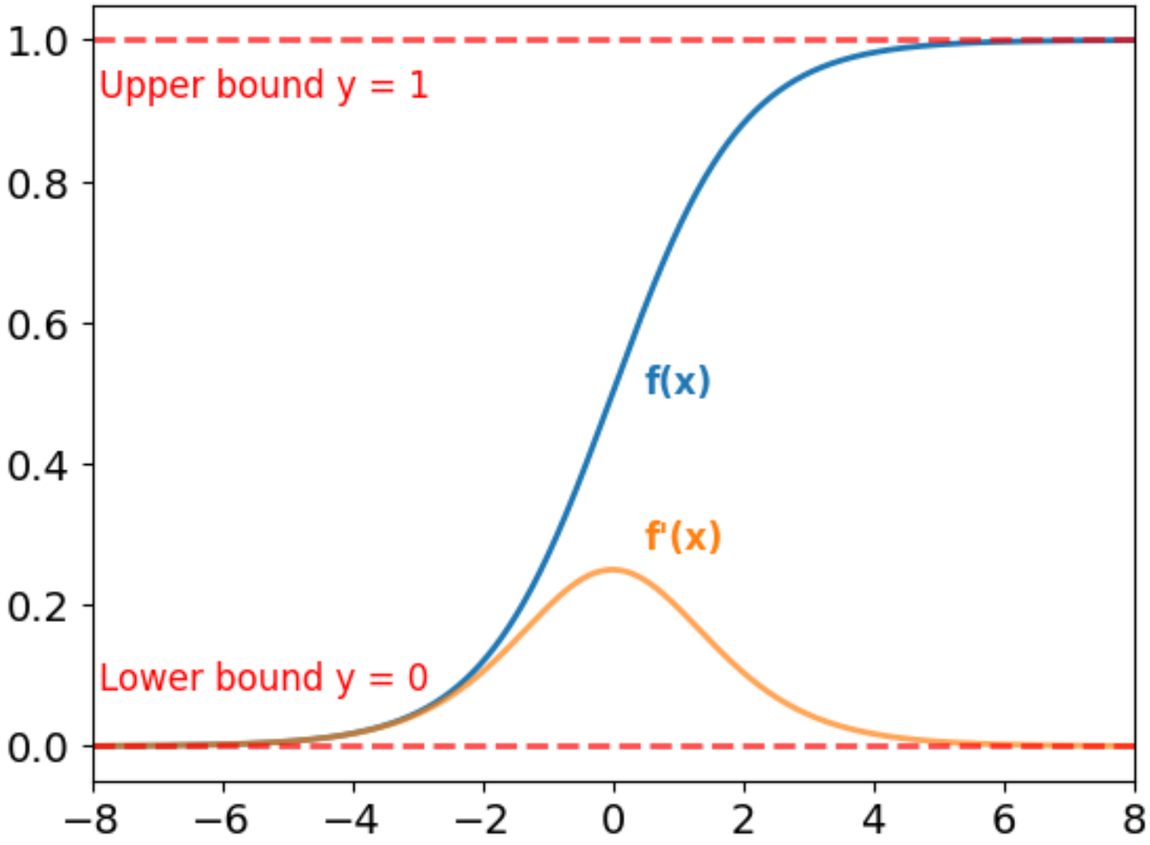
- Bounded between 0 and 1
- Can be used anywhere in the network

Gradients:

- Approach zero for low and high values of x
- Cause function to **saturate**

Sigmoid function saturation can lead to **vanishing gradients** during backpropagation.

This is also a problem for **softmax**.



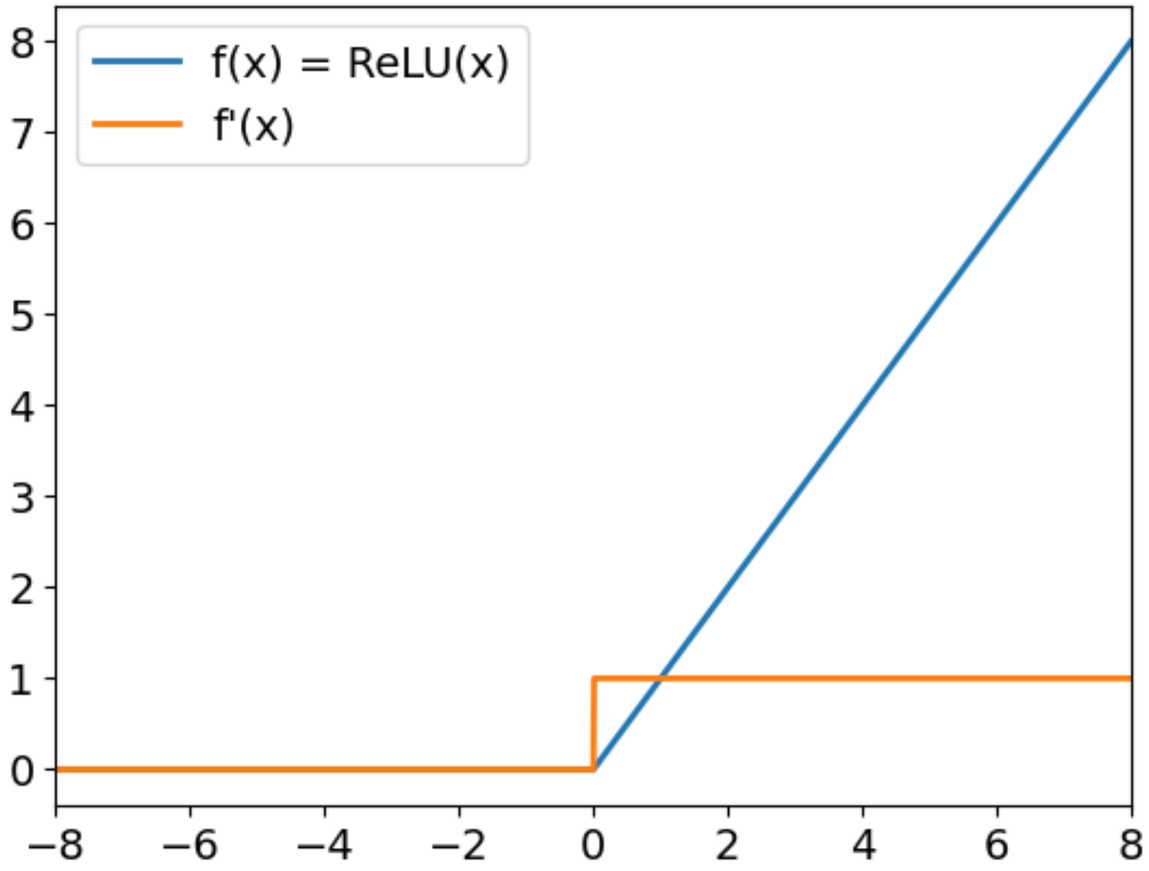
Introducing ReLU

Rectified Linear Unit (ReLU):

- `f(x) = max(x, 0)`
- for positive inputs, the output is equal to the input
- for strictly negative inputs, the output is equal to zero
- overcomes the vanishing gradients problem

In PyTorch:

```
relu = nn.ReLU()
```



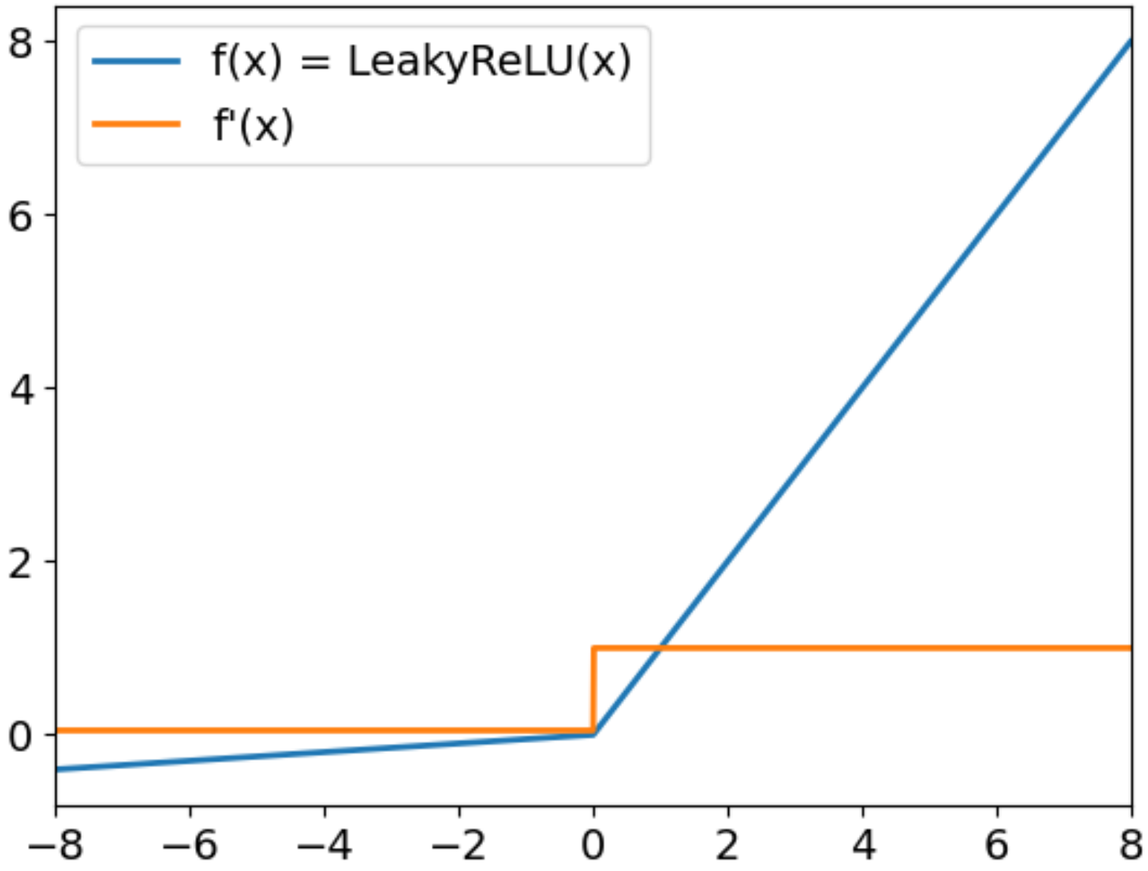
Introducing Leaky ReLU

Leaky ReLU:

- For positive inputs, it behaves similarly to ReLU
- For negative inputs, it multiplies the input by a small coefficient (defaulted to 0.01)
- The gradients for negative inputs are never null

In PyTorch:

```
leaky_relu = nn.LeakyReLU(negative_slope = 0.05)
```



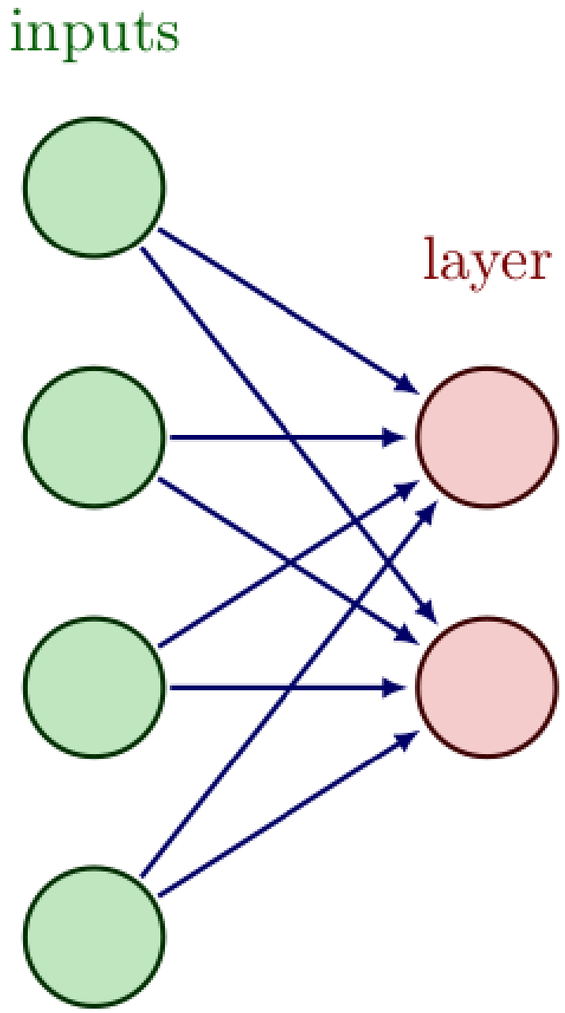
A deeper dive into neural network architecture

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

datacamp

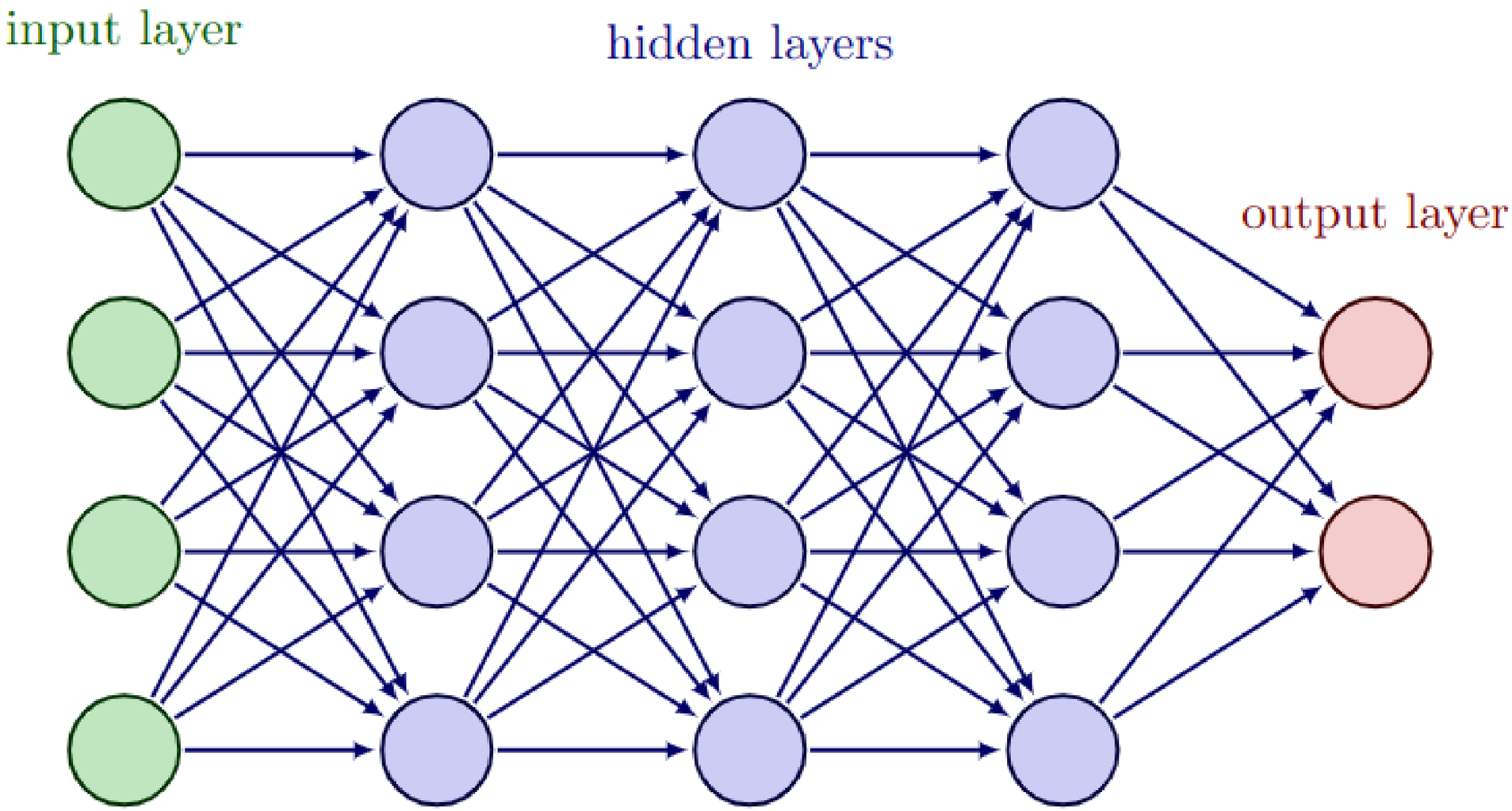
Layers are made of neurons

- Linear layers are **fully connected**
- Each neuron of a layer connected to each neuron of previous layer



- A neuron of a linear layer:
  - computes a linear operation using all neurons of previous layer
  - contains N+1 learnable parameters
  - where N = dimension of previous layer's outputs

Layer naming convention



Tweaking the number of hidden layers

- Input and output layers dimensions are fixed.
  - input layer depends on the number of features `n_features`
  - output layer depends on the number of categories `n_classes`

```
model = nn.Sequential(nn.Linear(n_features, 8),
                      nn.Linear(8, 4),
                      nn.Linear(4, n_classes))
```

- We can use as many hidden layers as we want
- Increasing the number of hidden layers = increasing the number of parameters = increasing the **model capacity**

Counting the number of parameters

Given the following model:

```
model = nn.Sequential(nn.Linear(8, 4),
                      nn.Linear(4, 2))
```

Manually calculating the number of parameters:

- first layer has 4 neurons, each neuron has 8+1 parameters = 36 parameters
- second layer has 2 neurons, each neuron has 4+1 parameters = 10 parameters
- total = 46 learnable parameters

Using PyTorch:

- `.numel()` : returns the number of elements in the tensor

```
total = 0
for parameter in model.parameters():
    total += parameter.numel()
print(total)
```

```
46
```

Learning rate and momentum

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Updating weights with SGD

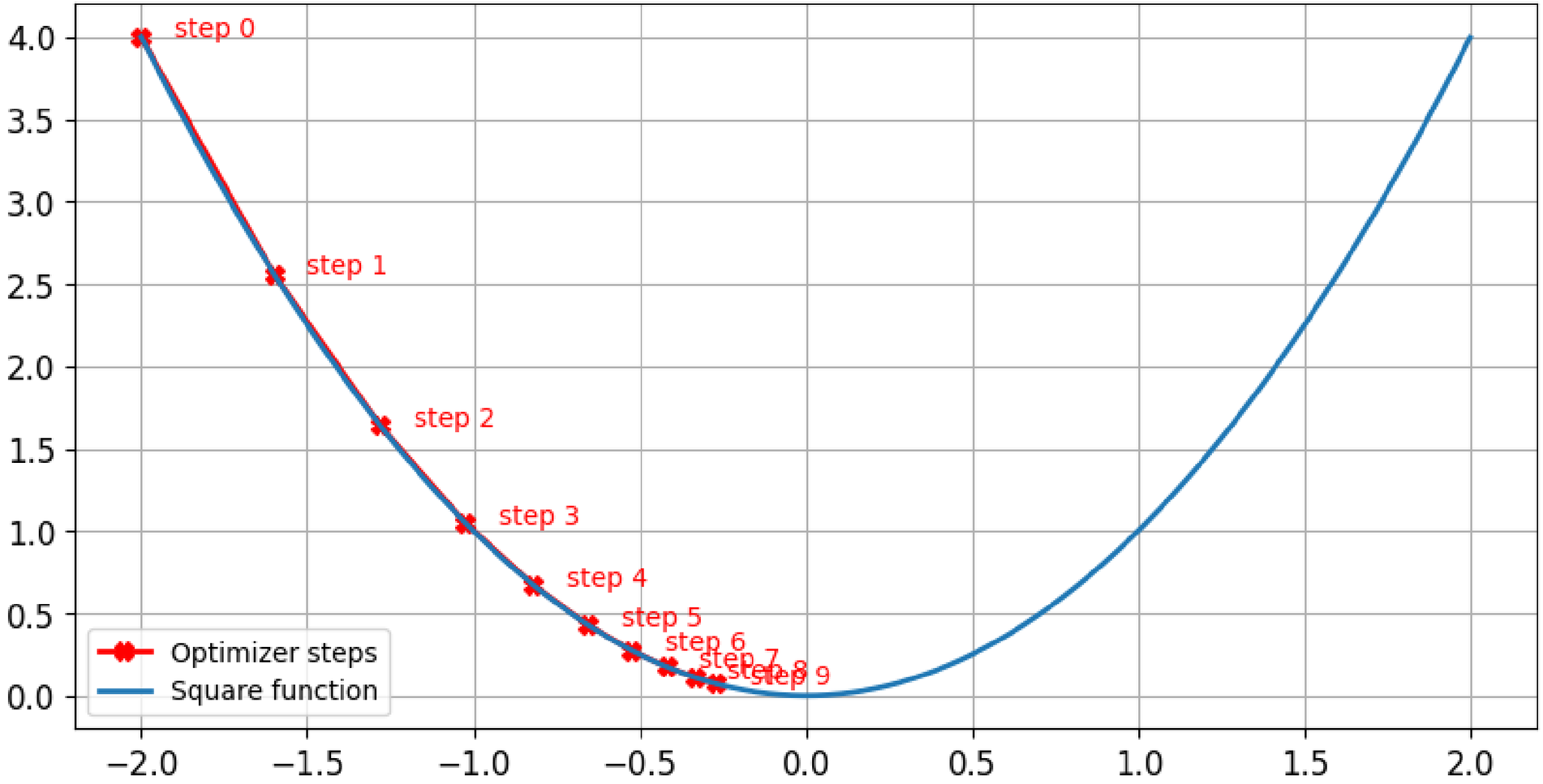
- Training a neural network = solving an **optimization problem.**

Stochastic Gradient Descent (SGD) optimizer

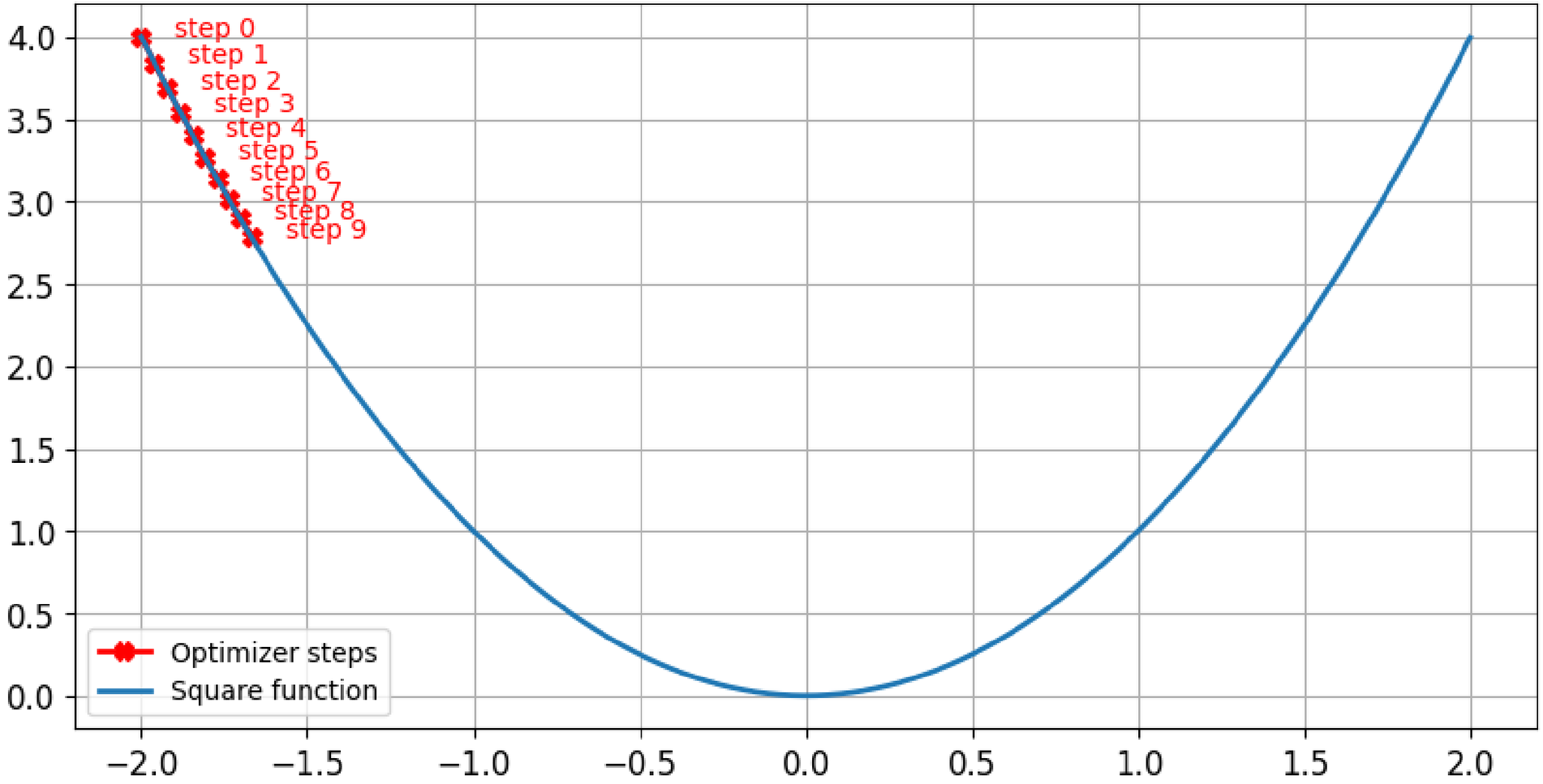
```
sgd = optim.SGD(model.parameters(), lr=0.01, momentum=0.95)
```

- Two parameters:
  - **learning rate:** controls the step size
  - **momentum:** controls the inertia of the optimizer
- Bad values can lead to:
  - long training times
  - bad overall performances (poor accuracy)

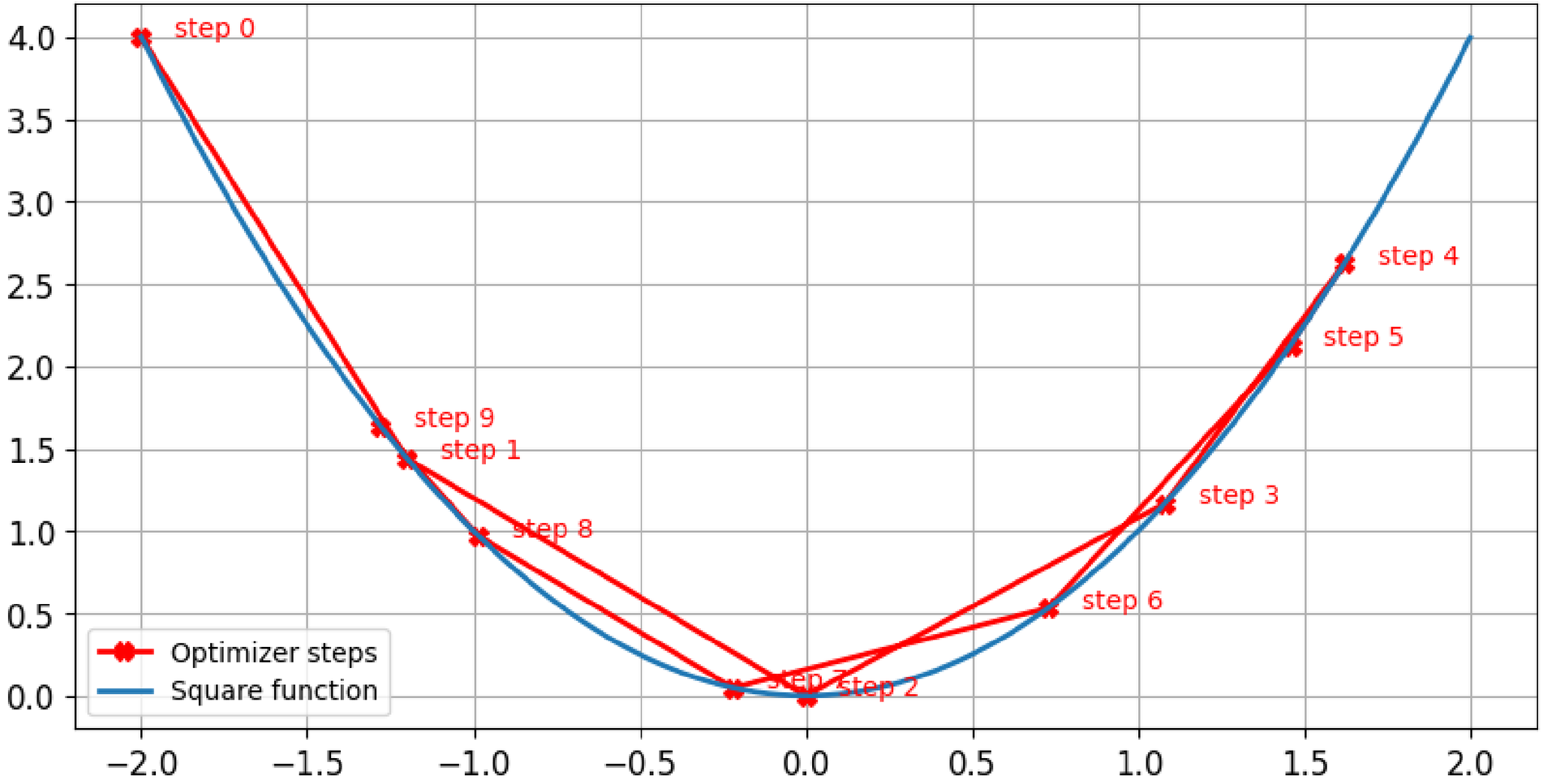
Impact of the learning rate: optimal learning rate



Impact of the learning rate: small learning rate

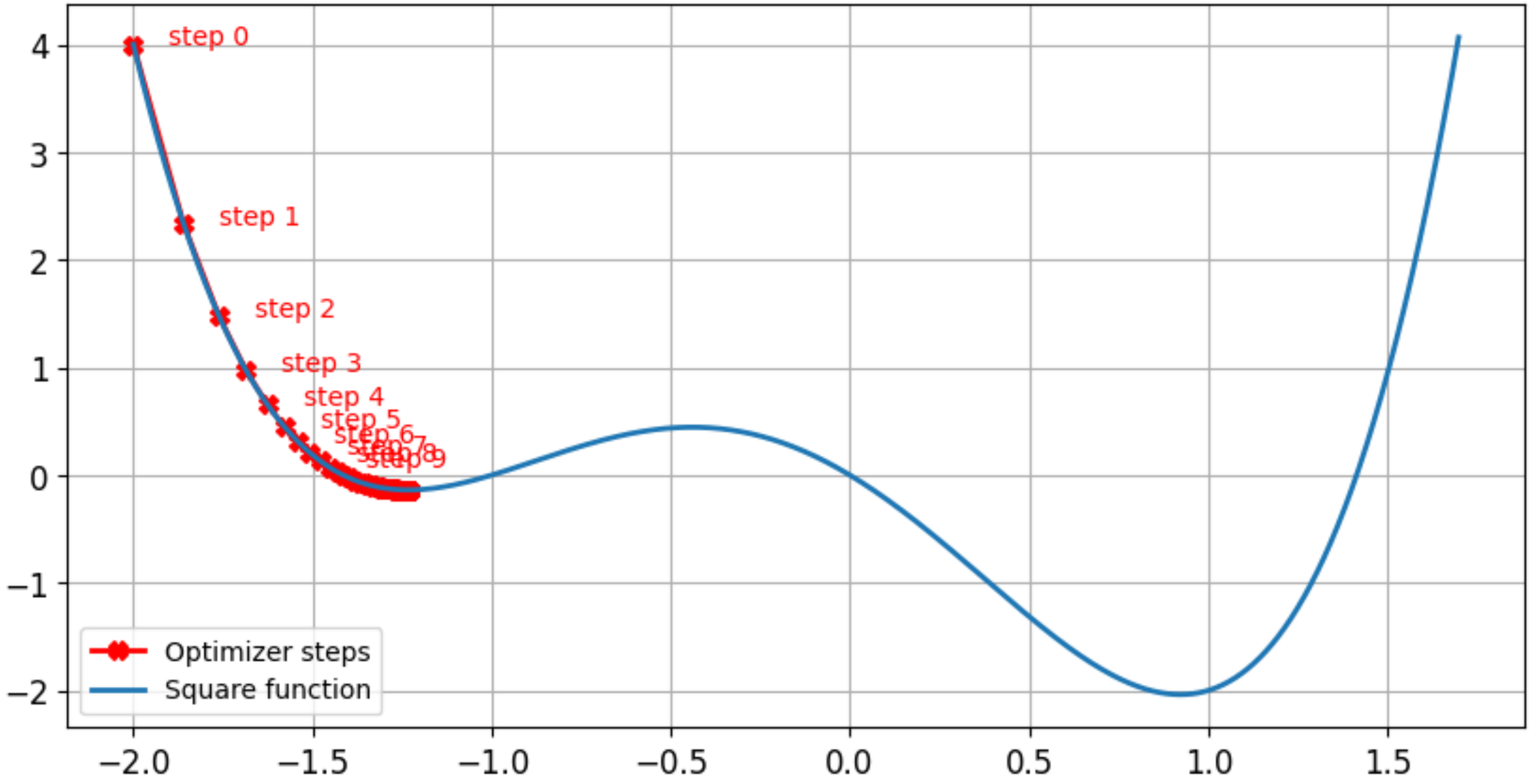


Impact of the learning rate: high learning rate



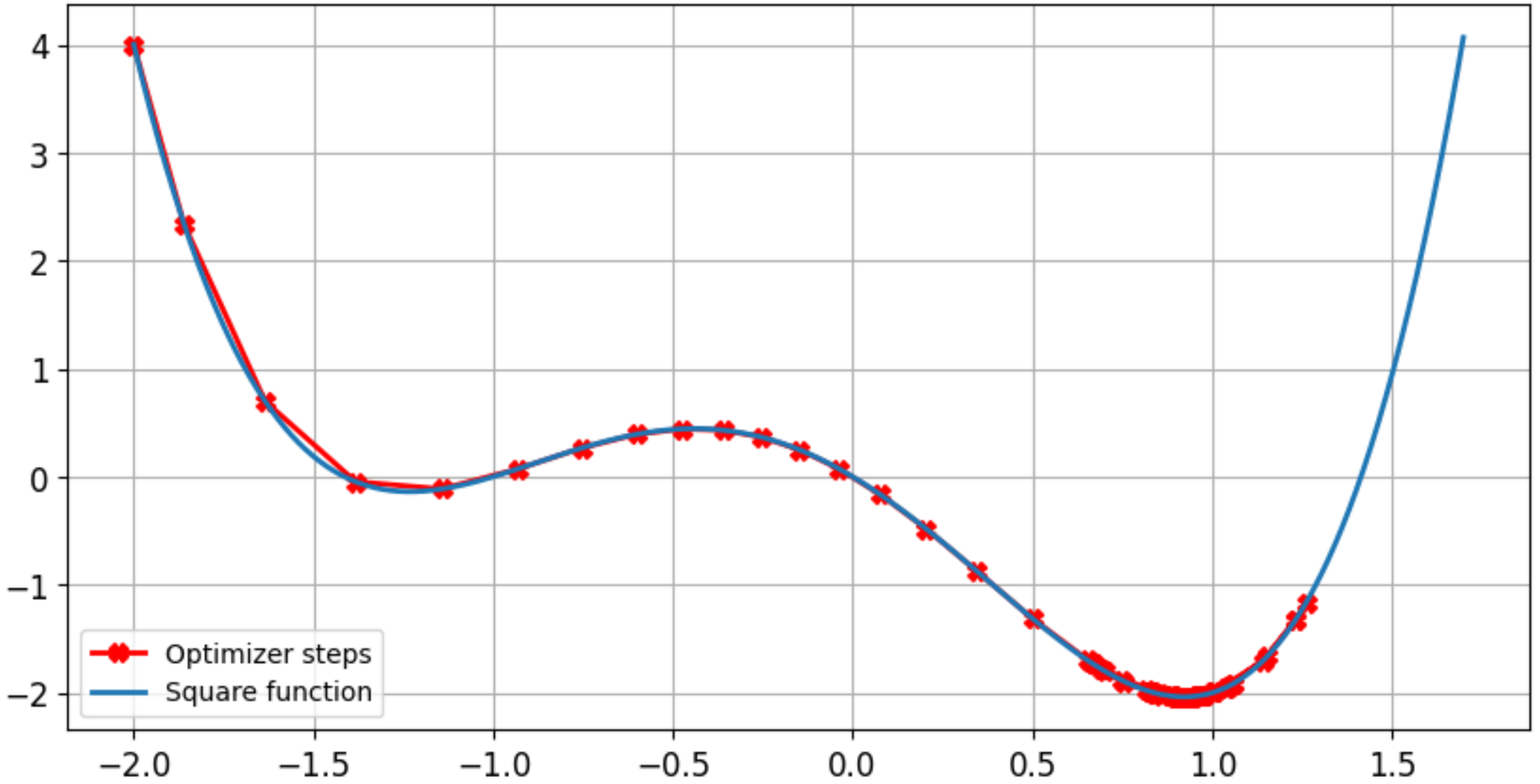
Without momentum

- `lr = 0.01` `momentum = 0` , after 100 steps minimum found for `x = -1.23` and `y = -0.14`



With momentum

- `lr = 0.01` `momentum = 0.9` , after 100 steps minimum found for `x = 0.92` and `y = -2.04`



Summary

Learning rate	Momentum
Controls the step size	Controls the inertia
Too small leads to long training times	Null momentum can lead to the optimizer being stuck in a local minimum
Too high leads to poor performances	Non-null momentum can help find the function minimum
Typical values between 10^{-2} and 10^{-4}	Typical values between 0.85 and 0.99

Layers initialization, transfer learning and fine tuning

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

datacamp

Layer initialization (1)

```
import torch.nn as nn
layer = nn.Linear(64, 128)
print(layer.weight.min(), layer.weight.max())
```

```
(tensor(-0.1250, grad_fn=<MinBackward1>), tensor(0.1250, grad_fn=<MaxBackward1>))
```

- A layer weights are initialized to small values
- The outputs of a layer would explode if the inputs and the weights are not normalized.
- The weights can be initialized using different methods (for example, using a uniform distribution)

Layer initialization (2)

```
import torch.nn as nn

layer = nn.Linear(64, 128)
nn.init.uniform_(layer.weight)

print(custom_layer.fc.weight.min(), custom_layer.fc.weight.max())
```

```
(tensor(0.0002, grad_fn=<MinBackward1>), tensor(1.0000, grad_fn=<MaxBackward1>))
```

Transfer learning and fine tuning (1)

Transfer learning: reusing a model trained on a first task for a second similar task, to accelerate the training process.

For example, we trained a first model on a large dataset of data scientist salaries across the US and we want to train a new model on a smaller dataset of salaries in Europe.

```
import torch

layer = nn.Linear(64, 128)
torch.save(layer, 'layer.pth')

new_layer = torch.load('layer.pth')
```

Transfer learning and fine-tuning

- **Fine-tuning** = A type of transfer learning
  - Smaller learning rate
  - Not every layer is trained (we **freeze** some of them)
  - Rule of thumb: freeze early layers of network and fine-tune layers closer to output layer

```
import torch.nn as nn

model = nn.Sequential(nn.Linear(64, 128),
                      nn.Linear(128, 256))

for name, param in model.named_parameters():
    if name == '0.weight':
        param.requires_grad = False
```