A deeper dive into loading data

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf

Senior Data Science Content Developer, DataCamp

Our animals dataset

```
import pandas as pd
animals = pd.read_csv('animal_dataset.csv')
```

animal_name	hair	feathers	eggs	milk	predator	legs	tail	type
sparrow	0	1	1	0	0	2	1	0
eagle	0	1	1	0	1	2	1	0
cat	1	0	0	1	1	4	1	1
dog	1	0	0	1	0	4	1	1
lizard	0	0	1	0	1	4	1	2

Type categories: **bird** (0), **mammal** (1), **reptile** (2)

Our animals dataset: defining features

```
import numpy as np

# Define input features
features = animals.iloc[:, 1:-1]

X = features.to_numpy()
print(X)
```

```
[[0 1 1 0 0 2 1]
 [0 1 1 0 1 2 1]
 [1 0 0 1 1 4 1]
 [1 0 0 1 0 4 1]
 [0 0 1 0 1 4 1]]
```

Back to our animals dataset: defining target values

```
# Define target values (ground truth)
target = animals.iloc[:, -1]
y = target.to_numpy()
print(y)
```

```
[0 0 1 1 2]
```

TensorDataset

```
import torch
from torch.utils.data import TensorDataset

# Instantiate dataset class
dataset = TensorDataset(torch.tensor(X), torch.tensor(y))

# Access an individual sample
input_sample, label_sample = dataset[0]
print('input sample:', input_sample)
print('label_sample:', label_sample)
```

```
input sample: tensor([0, 1, 1, 0, 0, 2, 1])
label sample: tensor(0)
```

DataLoader

```
from torch.utils.data import DataLoader

batch_size = 2
shuffle = True

# Create a DataLoader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

- **Epoch:** one full pass through the training dataloader
- **Generalization:** model performs will with unseen data

DataLoader

```
# Iterate over the dataloader
for batch_inputs, batch_labels in dataloader:
    print('batch_inputs:', batch_inputs)
    print('batch_labels:', batch_labels)
```

```
batch_inputs: tensor([[1, 0, 0, 1, 1, 4, 1],
        [1, 0, 0, 1, 0, 4, 1]])
batch_labels: tensor([1, 1])
batch_inputs: tensor([[0, 1, 1, 0, 1, 2, 1],
        [0, 0, 1, 0, 1, 4, 1]])
batch_labels: tensor([0, 2])
batch_inputs: tensor([[0, 1, 1, 0, 0, 2, 1]])
batch_labels: tensor([0])
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Writing our first training loop

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf

Senior Data Science Content Developer, DataCamp

Training a neural network

1. Create a model
2. Choose a loss function
3. Define a dataset
4. Set an optimizer
5. Run a training loop:
    - Calculate loss (forward pass)
    - Compute gradients (backpropagation)
    - Updating model parameters

Introducing the Data Science Salary dataset

experience_level	employment_type	remote_ratio	company_size	salary_in_usd
0	0	0.5	1	0.036
1	0	1.0	2	0.133
2	0	0.0	1	0.234
1	0	1.0	0	0.076
2	0	1.0	1	0.170

- Features: categorical, target: salary (USD)
- Final output: linear layer
- Loss: regression-specific

Mean Squared Error Loss

- MSE loss is the mean of the squared difference between predictions and ground truth

```
def mean_squared_loss(prediction, target):
  return np.mean((prediction - target)**2)
```

- in PyTorch:

```
criterion = nn.MSELoss()
# Prediction and target are float tensors
loss = criterion(prediction, target)
```

Before the training loop

```
# Create the dataset and the dataloader
dataset = TensorDataset(torch.tensor(features).float(),
                        torch.tensor(target).float())

dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

# Create the model
model = nn.Sequential(nn.Linear(4, 2),
                      nn.Linear(2, 1))

# Create the loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

The training loop

```
for epoch in range(num_epochs):
    for data in dataloader:
        # Set the gradients to zero
        optimizer.zero_grad()
        # Get feature and target from the data loader
        feature, target = data
        # Run a forward pass
        pred = model(feature)
        # Compute loss and gradients
        loss = criterion(pred, target)
        loss.backward()
        # Update the parameters
        optimizer.step()
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

ReLU activation functions

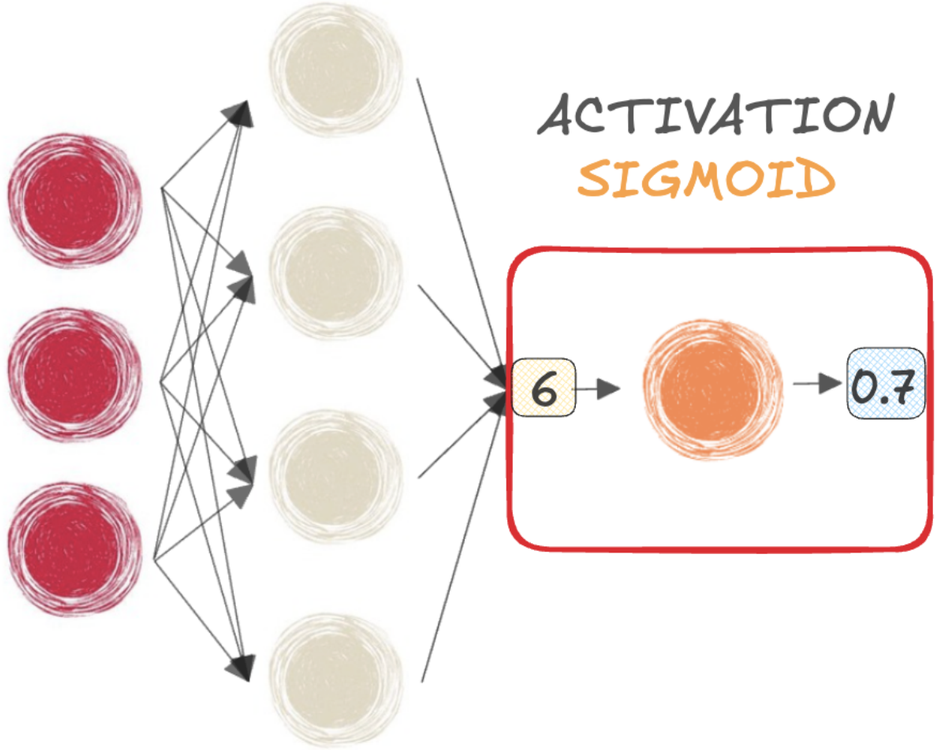
INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf

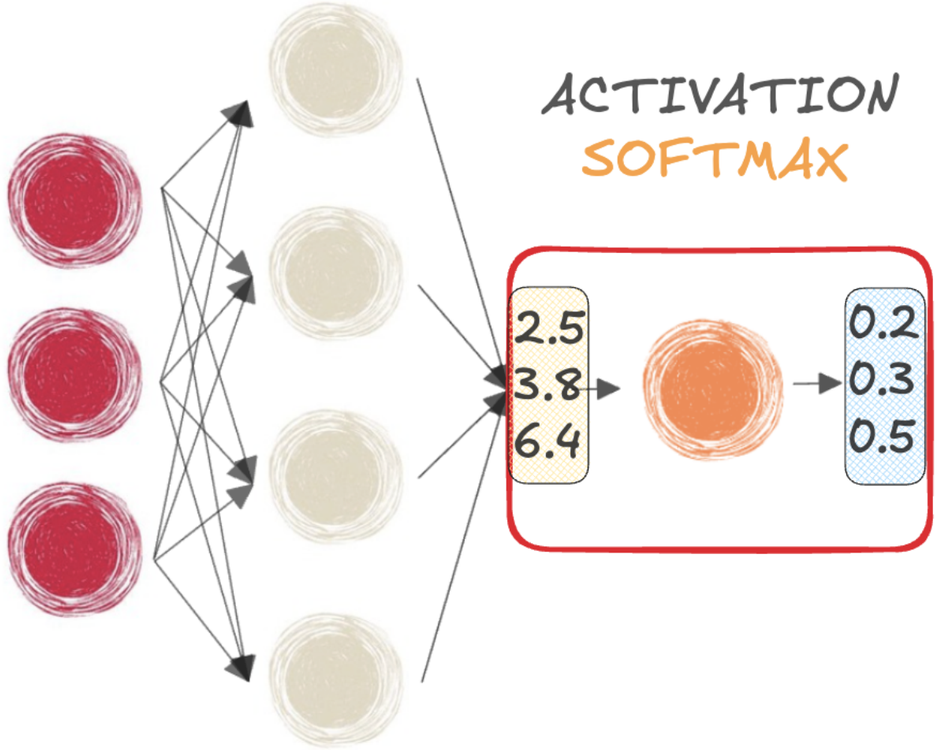
Senior Data Science Content Developer, DataCamp

Sigmoid and softmax functions

- **SIGMOID** for **BINARY** classification



- **SOFTMAX** for **MULTI-CLASS** classification



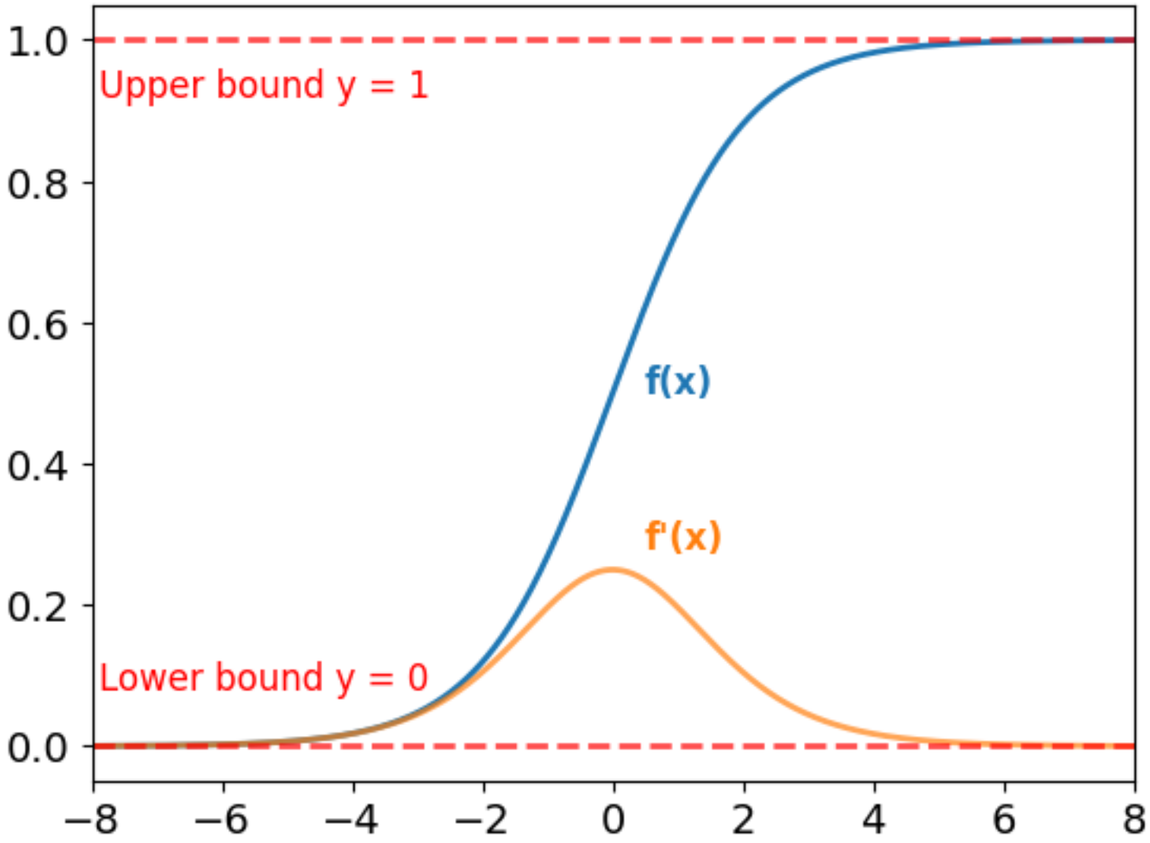
Limitations of the sigmoid and softmax function

Sigmoid function:

- Outputs bounded between 0 and 1
- Usable anywhere in a network

Gradients:

- Very small for large and small values of x
- Cause **saturation**, leading to the **vanishing gradients problem**

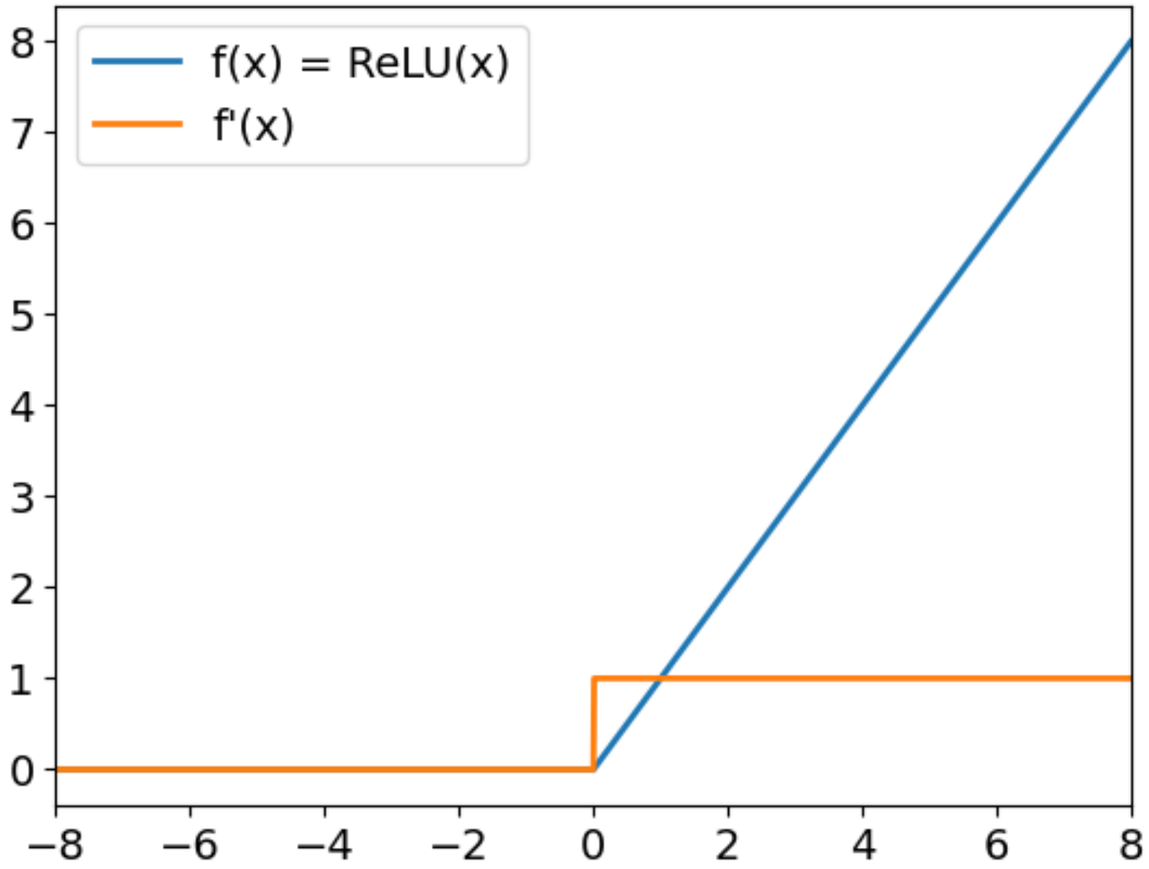


The softmax function also suffers from **saturation**

ReLU

Rectified Linear Unit (ReLU):

- `f(x) = max(x, 0)`
- For **positive** inputs: output equals input
- For **negative** inputs: output is 0
- Helps overcome **vanishing gradients**



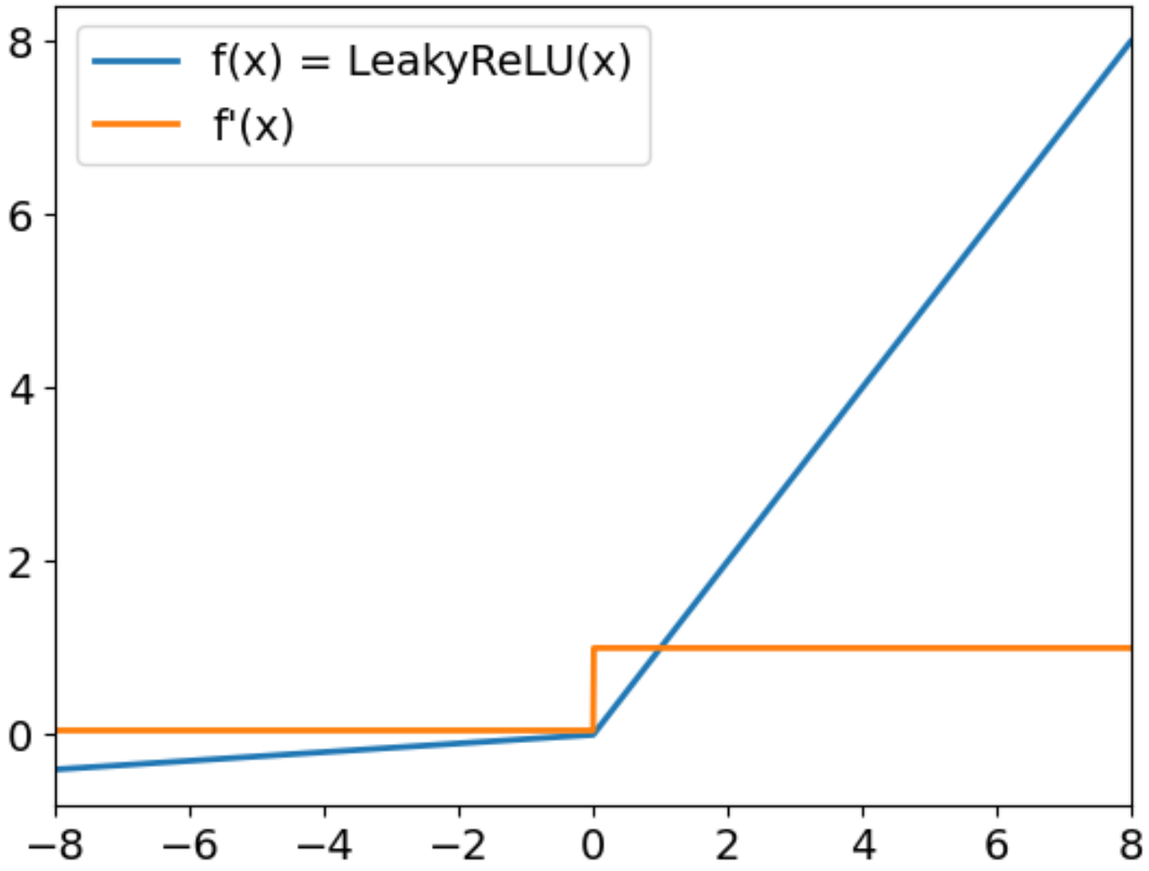
In PyTorch:

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

Leaky ReLU

Leaky ReLU:

- **Positive** inputs behave like ReLU
- **Negative** inputs are scaled by a small coefficient (default 0.01)
- Gradients for negative inputs are **non-zero**



In PyTorch:

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

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Learning rate and momentum

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf

Senior Data Science Content Developer, DataCamp

datacamp

Updating weights with SGD

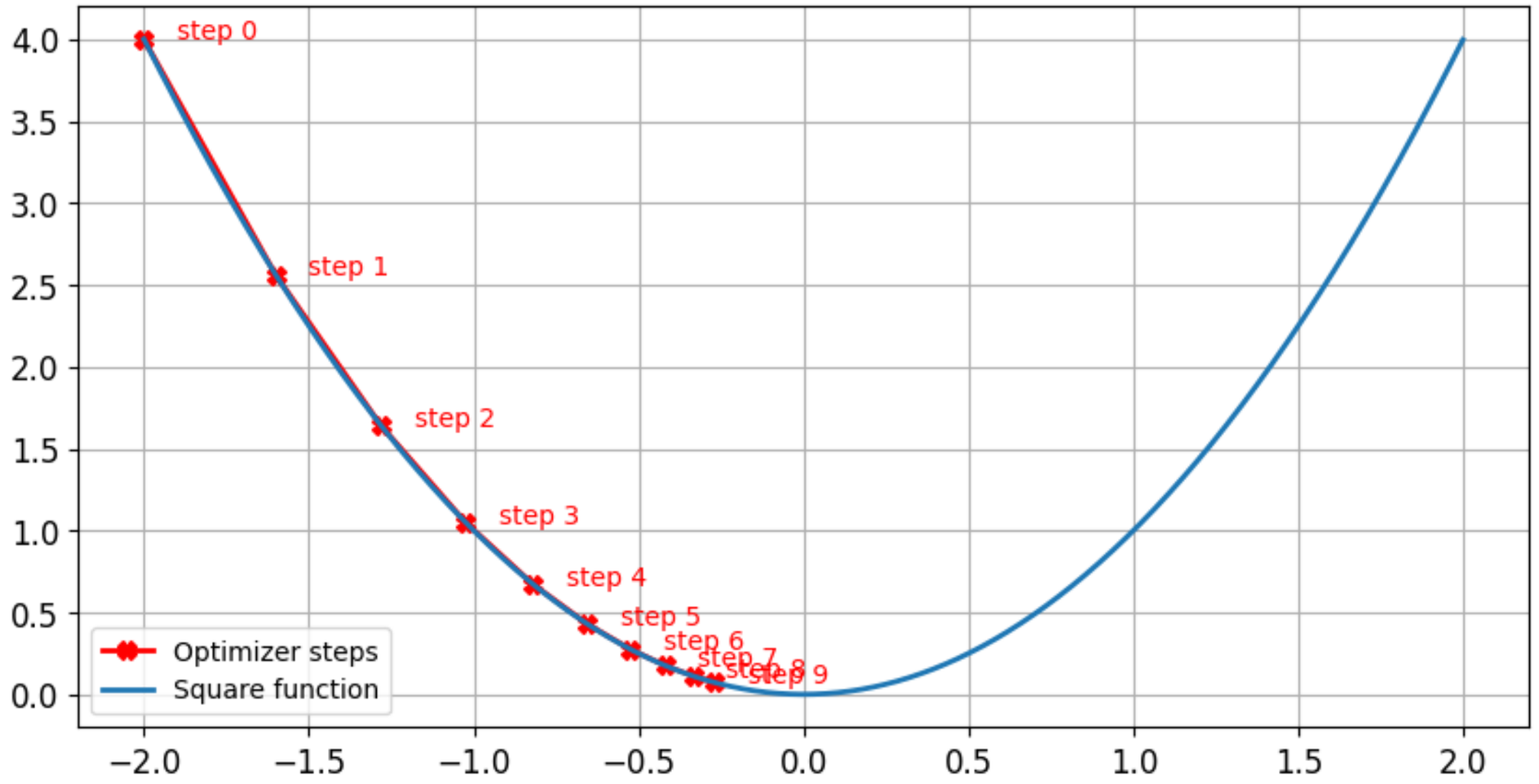
- Training a neural network = solving an **optimization problem.**

Stochastic Gradient Descent (SGD) optimizer

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

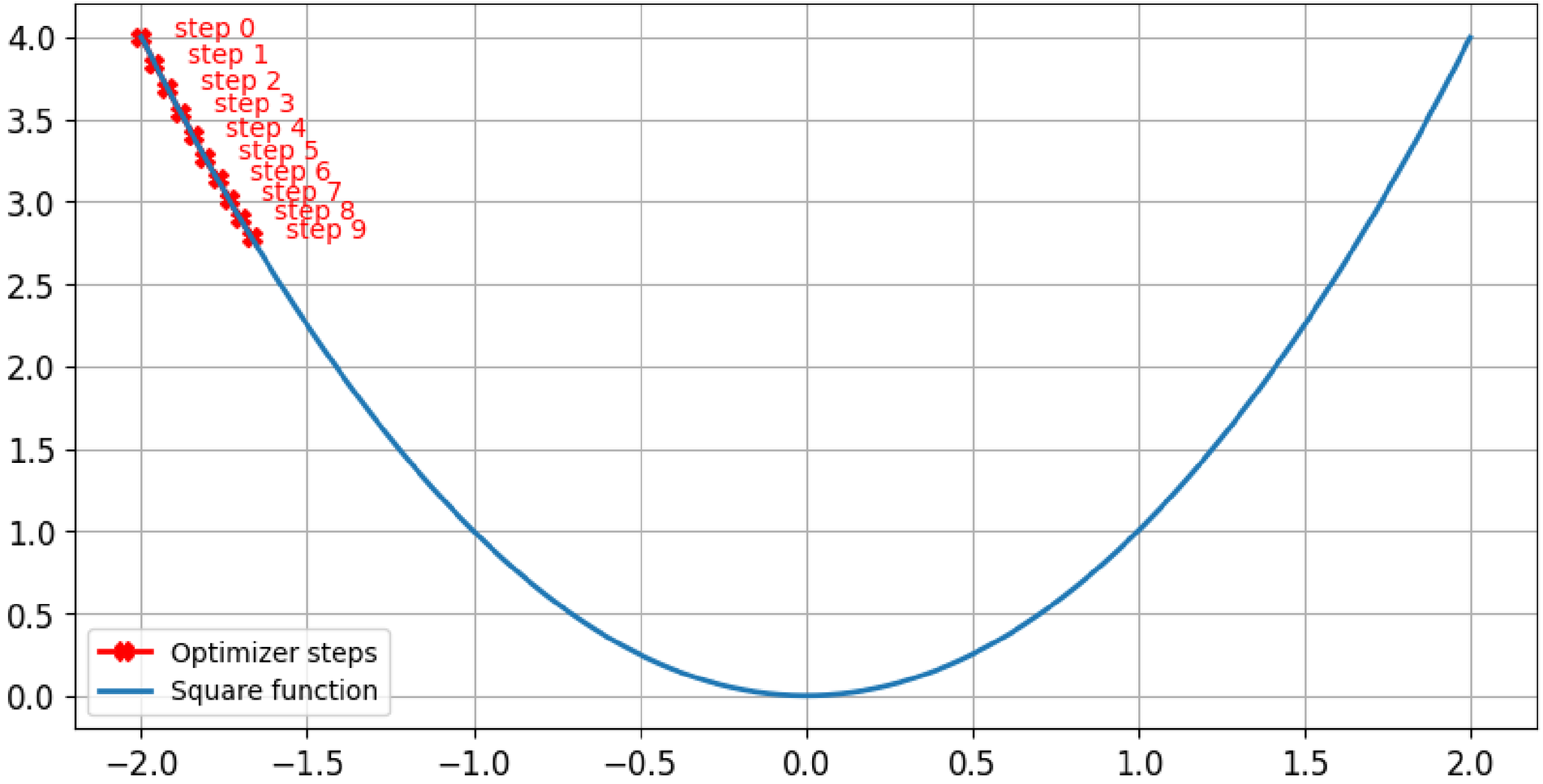
- Two arguments:
  - **learning rate:** controls the step size
  - **momentum:** adds inertia to avoid getting stuck

Impact of the learning rate: optimal learning rate

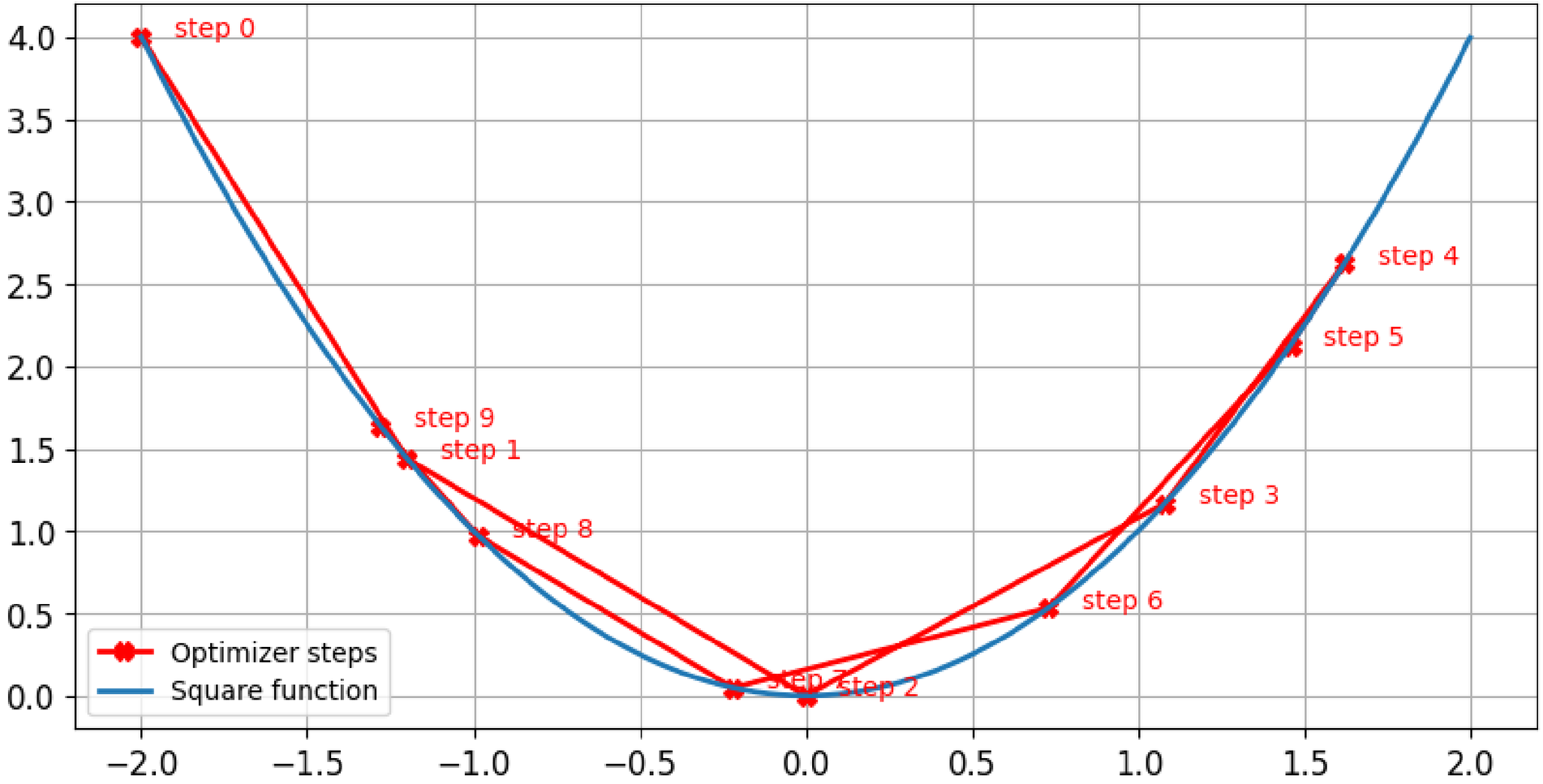


- Step size **decreases** near zero as the gradient gets smaller

Impact of the learning rate: small learning rate

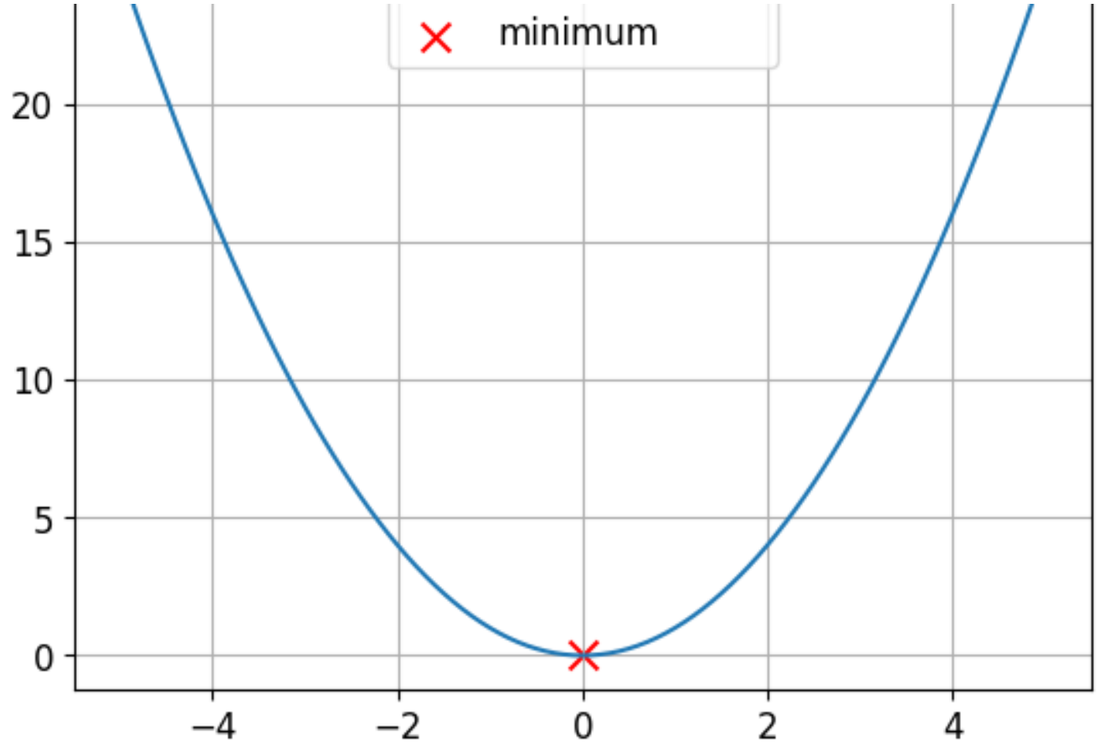


Impact of the learning rate: high learning rate

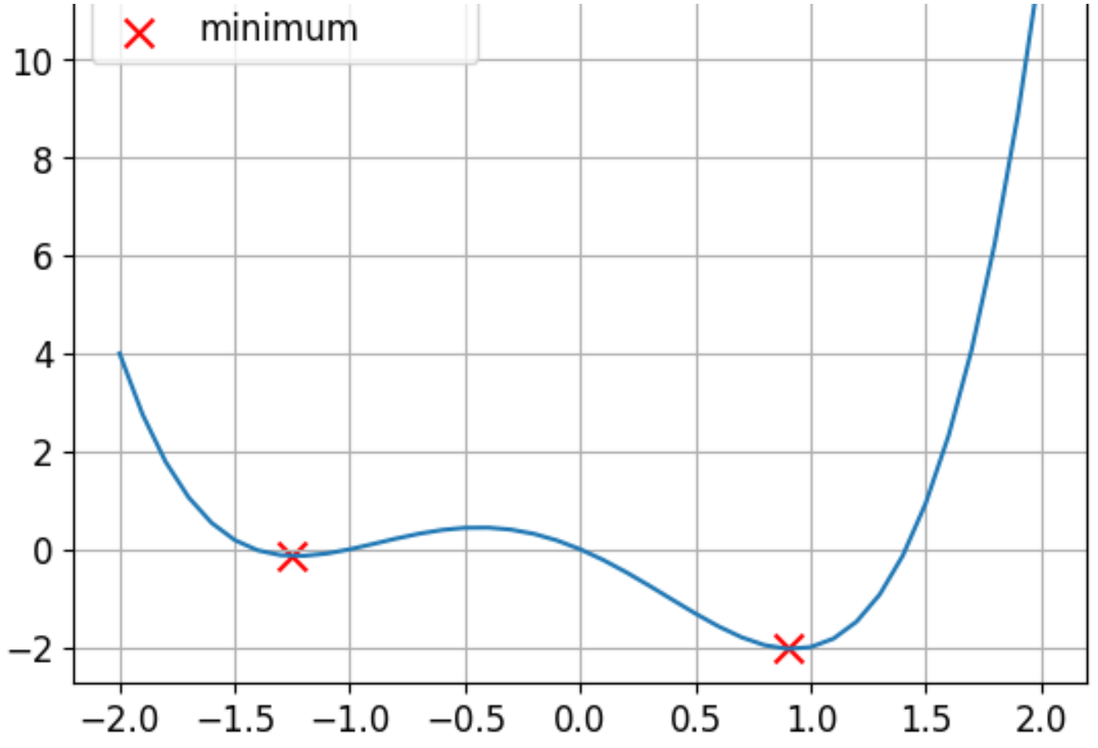


Convex and non-convex functions

This is a **convex function.**



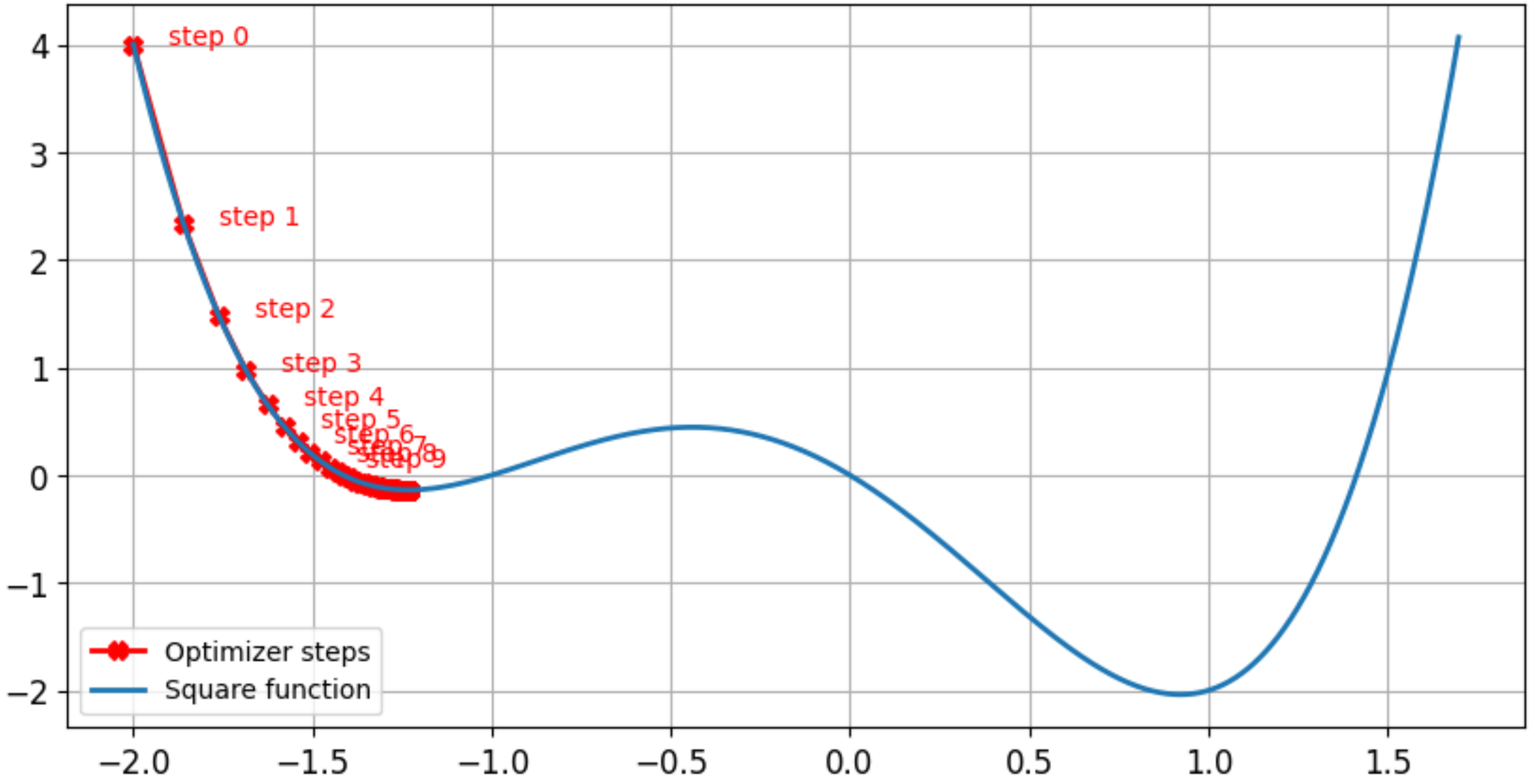
This is a **non-convex function.**



- Loss functions are **non-convex**

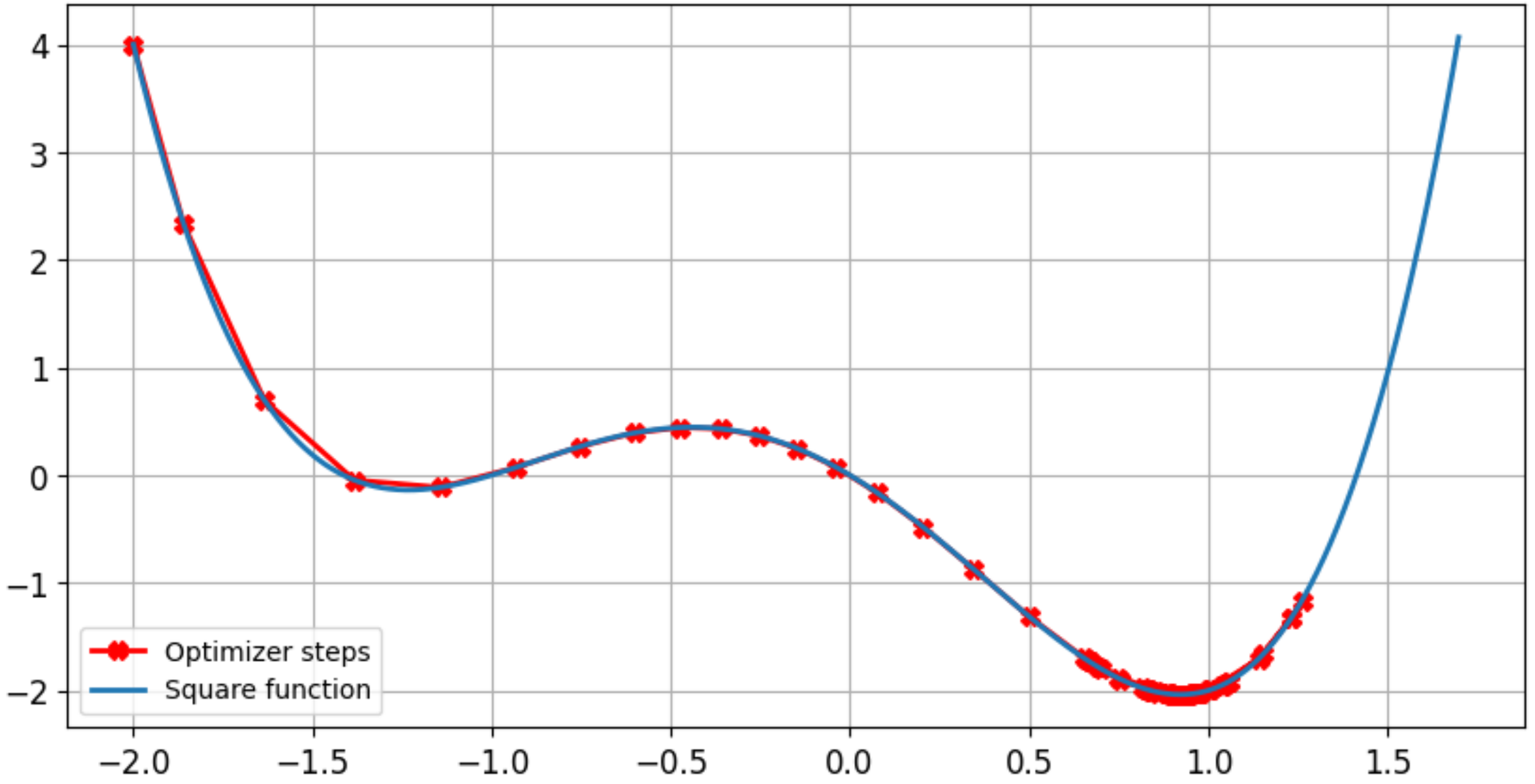
Without momentum

- `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 high → poor performance	Helps escape local minimum
Too low → slow training	Too small → optimizer gets stuck
Typical range: 0.01 (10^{-2}) and 0.0001 (10^{-4})	Typical range: **0.85 to 0.99**

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH