Advanced Data Analysis

Le Trong Ngoc, Ph.D.

Introduction to Deep Learning with PyTorch

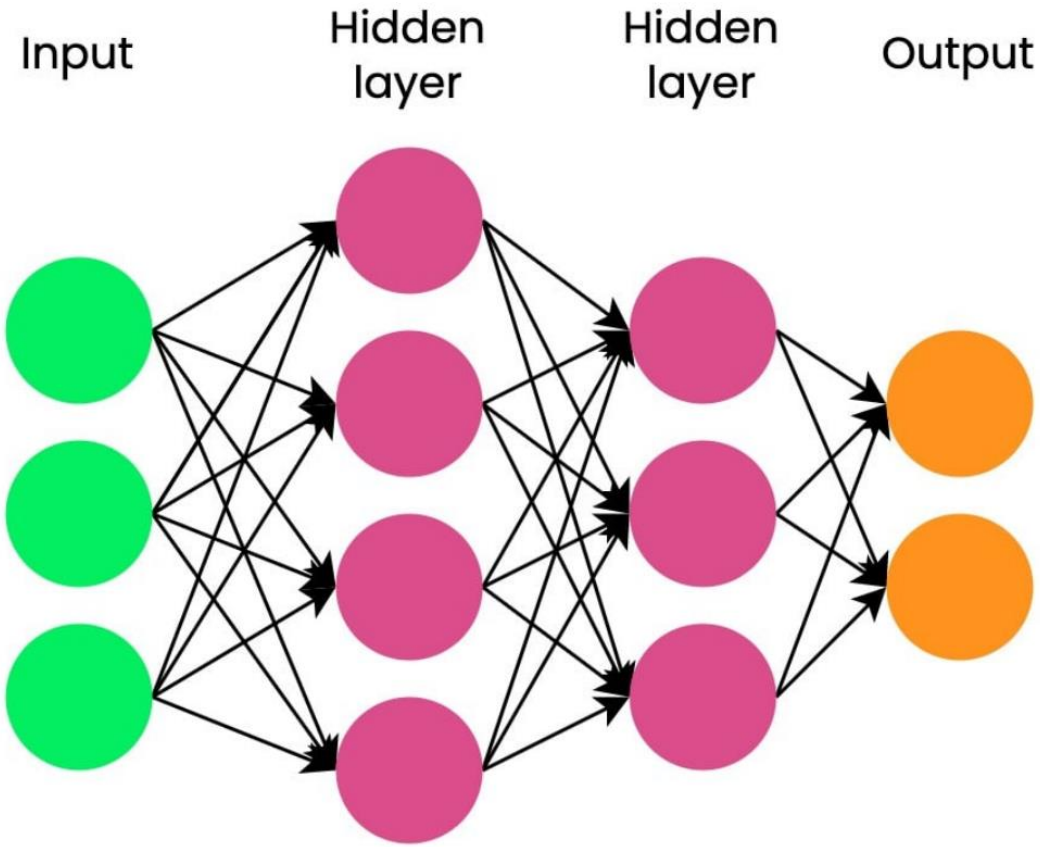
- **Introduction to PyTorch**
- Training Our First Neural Network with PyTorch
- Neural Network Architecture and Hyperparameters
- Evaluating and Improving Models

What is deep learning?

- Deep learning is everywhere:
  - Language translation
  - Self-driving cars
  - Medical diagnostics
  - Chatbots
- Used on multiple data types: **images**, **text** and **audio**
- Traditional machine learning: relies on hand-crafted **feature engineering**
- Deep learning: enables **feature learning** from raw data

What is deep learning?

- Deep learning is a subset of machine learning
- Inspired by connections in the human brain
- Models require large amount of data



PyTorch: a deep learning framework

- PyTorch is
  - one of the most popular deep learning frameworks
  - the framework used in many published deep learning papers
  - intuitive and user-friendly
  - has much in common with NumPy

Importing PyTorch and related packages

- PyTorch import in Python

```
import torch
```

- PyTorch supports
  - image data with `torchvision`
  - audio data with `torchaudio`
  - text data with `torchtext`

Tensors: the building blocks of networks in PyTorch

- **Load from list**

```
import torch

lst = [[1, 2, 3], [4, 5, 6]]
tensor = torch.tensor(lst)
```

- **Load from NumPy array**

```
np_array = np.array(array)
np_tensor = torch.from_numpy(np_array)
```

Like NumPy arrays, tensors are **multidimensional** representations of their elements

Tensor attributes

- **Tensor shape**

```
lst = [[1, 2, 3], [4, 5, 6]]
tensor = torch.tensor(lst)
tensor.shape
```

```
torch.Size([2, 3])
```

- Tensor data type

```
tensor.dtype
```

```
torch.int64
```

Tensor device

```
tensor.device
```

```
device(type='cpu')
```

Deep learning often requires a GPU, which, compared to a CPU can offer:

- parallel computing capabilities
- faster training times
- better performance

Getting started with tensor operations

Compatible shapes

```
a = torch.tensor([[1, 1],
                  [2, 2]])

b = torch.tensor([[2, 2],
                  [3, 3]])
```

- Addition / subtraction

```
a + b
```

```
tensor([[3, 3],
        [5, 5]])
```

Incompatible shapes

```
a = torch.tensor([[1, 1],
                  [2, 2]])

c = torch.tensor([[2, 2, 4],
                  [3, 3, 5]])
```

- Addition / subtraction

```
a + c
```

```
RuntimeError: The size of tensor a
(2) must match the size of tensor b (3)
at non-singleton dimension 1
```

Getting started with tensor operations

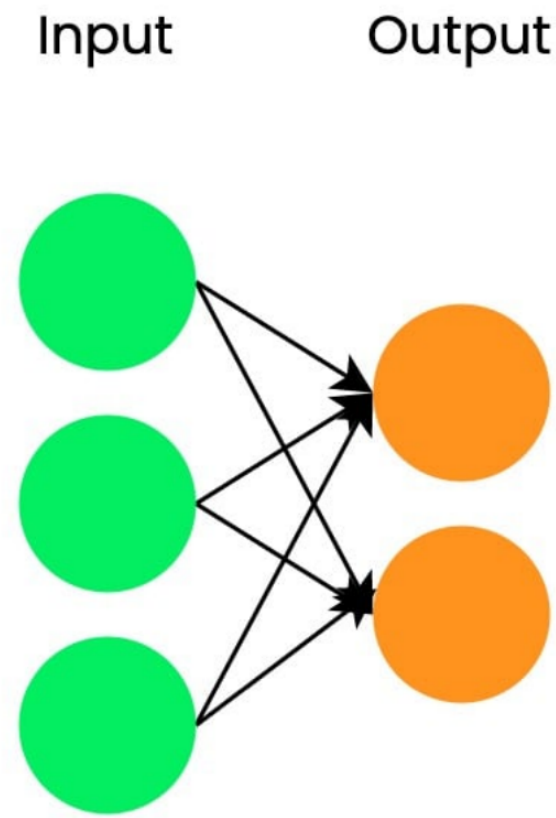
- Element-wise multiplication

```
a = torch.tensor([[1, 1],
                  [2, 2]])
b = torch.tensor([[2, 2],
                  [3, 3]])
a * b
```

```
tensor([[2,  2],
        [6, 6]])
```

- ... and much more
  - Transposition
  - Matrix multiplication
  - Concatenation
- Most NumPy array operations can be performed on PyTorch tensors

Our first neural network




```python
import torch.nn as nn
```

```python
## Create input_tensor with three features
input_tensor = torch.tensor(
    [[0.3471, 0.4547, -0.2356]])
```

```python
# Define our first linear layer
linear_layer = nn.Linear(in_features=3, out_features=2)
```

```python
# Pass input through linear layer
output = linear_layer(input_tensor)
print(output)
```

```
tensor([[-0.2415, -0.1604]],
    grad_fn=<AddmmBackward0>)
```

Getting to know the linear layer operation

Each linear layer has a `.weight` and `.bias` property

```
linear_layer.weight
```

```
Parameter containing:
tensor([[-0.4799,  0.4996,  0.1123],
        [-0.0365, -0.1855,  0.0432]],
        requires_grad=True)
```

```
linear_layer.bias
```

```
Parameter containing:
tensor([0.0310, 0.1537], requires_grad=True)
```

Getting to know the linear layer operation

```
output = linear_layer(input_tensor)
```

For input `X`, weights `W0` and bias `b0`, the linear layer performs

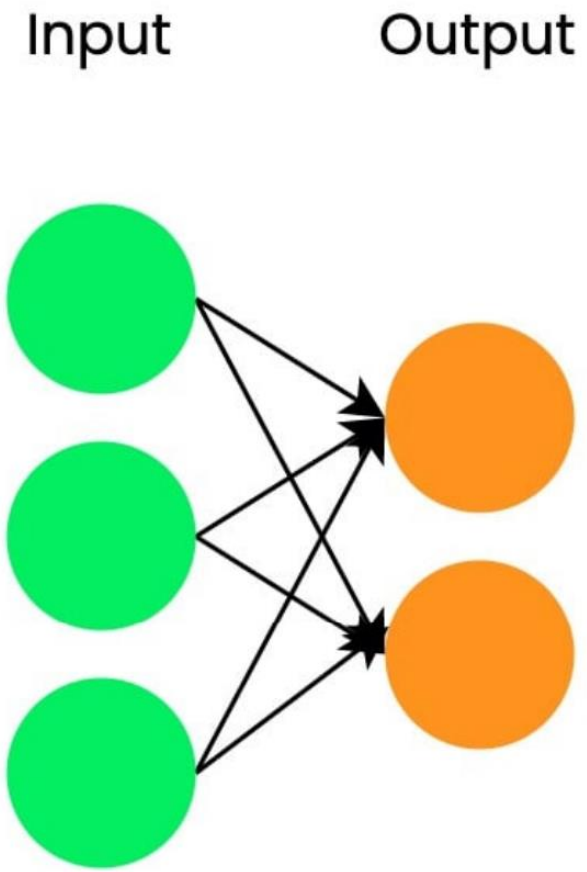
$$y_0 = W_0 \cdot X + b_0$$

In PyTorch: `output = W0 @ input + b0`

- Weights and biases are initialized randomly
- They are not useful until they are tuned

Our two-layer network summary

- Input dimensions: 1×3
- Linear layer arguments:
  - `in_features = 3`
  - `out_features = 2`
- Output dimensions: 1×2
- Networks with only linear layers are called **fully connected**
- Each neuron in one layer is connected to each neuron in the next layer



Stacking layers with nn.Sequential()

```
# Create network with three linear layers
model = nn.Sequential(
    nn.Linear(10, 18),
    nn.Linear(18, 20),
    nn.Linear(20, 5)
)
```

Stacking layers with nn.Sequential()

```
print(input_tensor)
```

```
tensor([[-0.0014,  0.4038,  1.0305,  0.7521, 0.7489, -0.3968,  0.0113, -1.3844, 0.8705, -0.9743]])
```

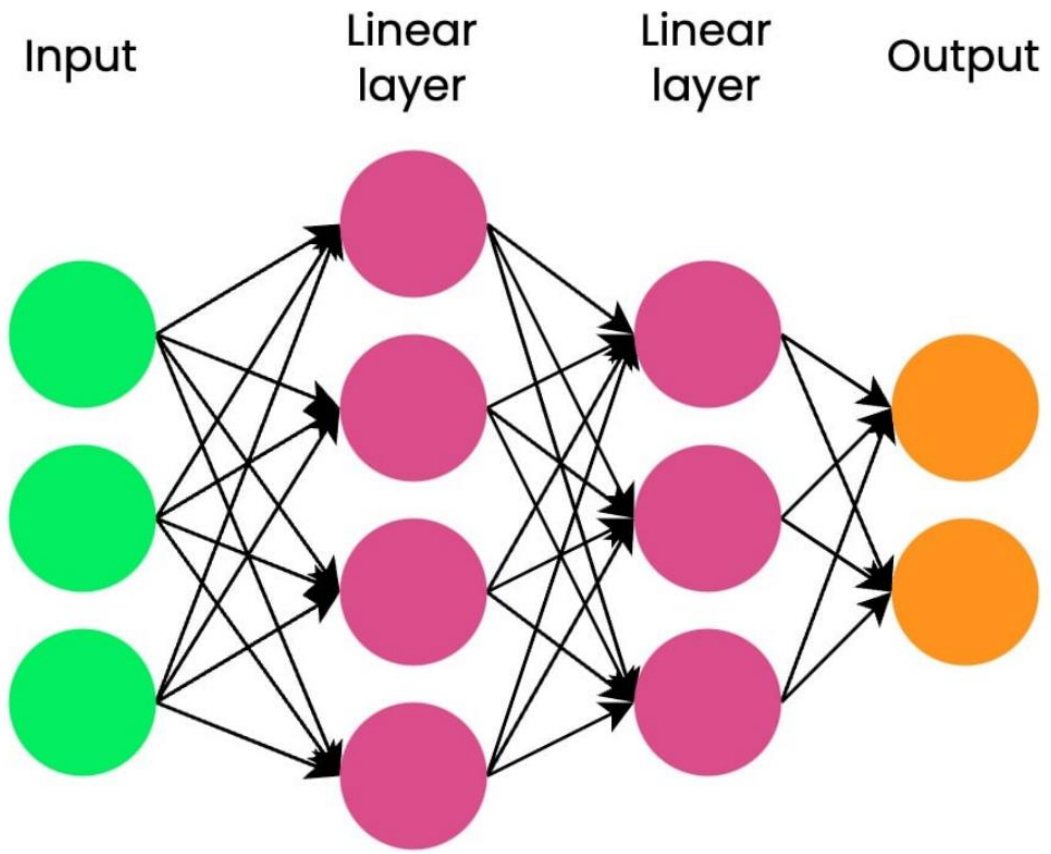
```
# Pass input_tensor to model to obtain output
output_tensor = model(input_tensor)
print(output_tensor)
```

```
tensor([[-0.0254, -0.0673,  0.0763,
0.0008,  0.2561]], grad_fn=<AddmmBackward0>)
```

- We obtain output of 1×5 dimensions
- Output is still not yet meaningful

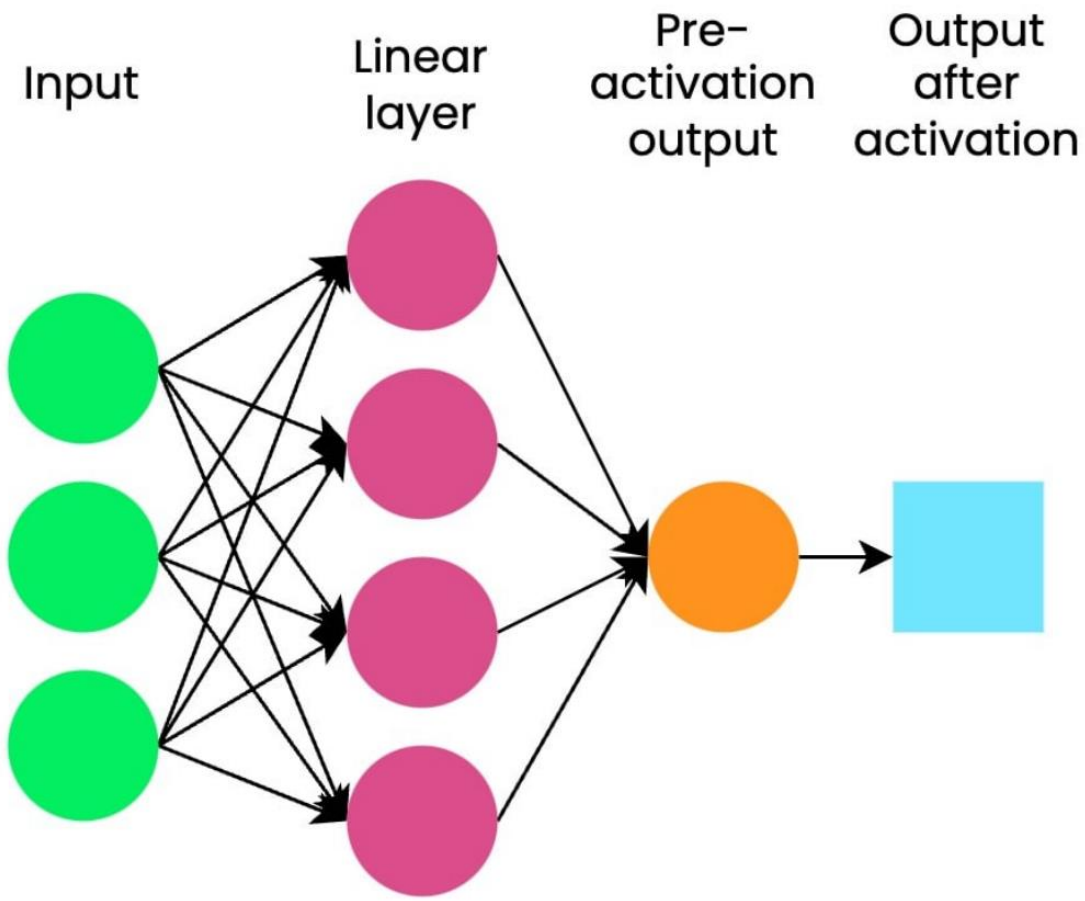
Stacked linear operations

- We have only seen linear layer networks
- Each linear layer multiplies its respective input with layer weights and adds biases
- Even with multiple stacked linear layers, output still has linear relationship with input

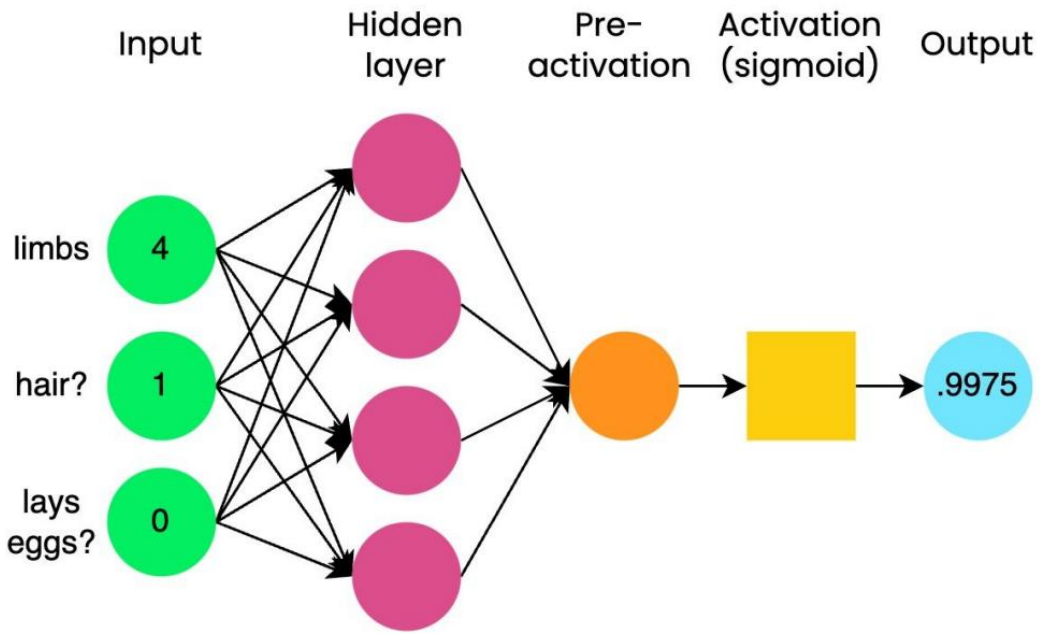


Why do we need activation functions?

- **Activation functions** add **non-linearity** to the network
- A model can learn more **complex** relationships with non-linearity



Meet the sigmoid function



Binary classification task:

- To predict whether animal is **1 (mammal)** or **0 (not mammal)**,
- we take the pre-activation (6),
- pass it to the sigmoid,
- and obtain a value between 0 and 1.

Using the common **threshold** of 0.5:

- If output is > 0.5, class label = 1 (mammal)
- If output is <= 0.5, class label = 0 (not mammal)

Meet the sigmoid function

```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[6.0]])
sigmoid = nn.Sigmoid()
output = sigmoid(input_tensor)
```

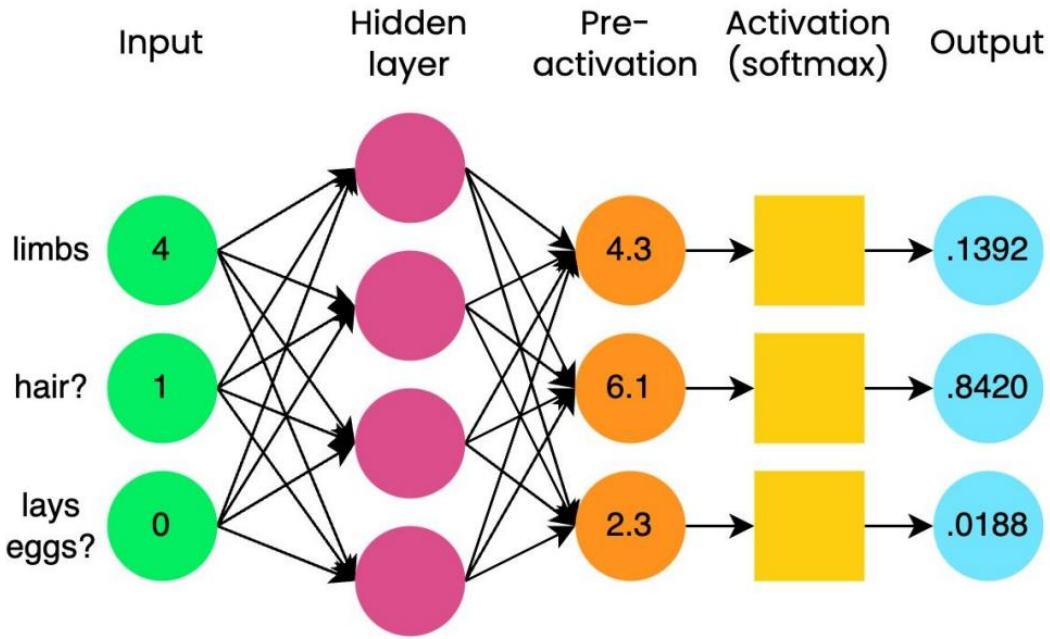
```
tensor([[0.9975]])
```

Activation function as the last layer

```
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1), # Second linear layer
  nn.Sigmoid() # Sigmoid activation function
)
```

Note. Sigmoid as last step in network of linear layers is **equivalent** to traditional logistic regression.

Getting acquainted with softmax



- used for multi-class classification problems
- takes N-element vector as input and outputs vector of same size
- say N=3 classes:
  - bird (0), mammal (1), reptile (2)
  - output has three elements, so softmax has three elements
- outputs a probability distribution:
  - each element is a probability (it's bounded between 0 and 1)
  - the sum of the output vector is equal to 1

Getting acquainted with softmax

```
import torch
import torch.nn as nn

# Create an input tensor
input_tensor = torch.tensor(
    [[4.3, 6.1, 2.3]])

# Apply softmax along the last dimension
probabilities = nn.Softmax(dim=-1)
output_tensor = probabilities(input_tensor)

print(output_tensor)
```

```
tensor([[0.1392, 0.8420, 0.0188]])
```

- dim = -1 indicates softmax is applied to the input tensor's last dimension
- `nn.Softmax()` can be used as last step in `nn.Sequential()`

Introduction to Deep Learning with PyTorch

- Introduction to PyTorch
- **Training Our First Neural Network with PyTorch**
- Neural Network Architecture and Hyperparameters
- Evaluating and Improving Models

What is a forward pass?

- Input data is **passed forward** or **propagated** through a network
- Computations performed at each layer
- Outputs of each layer passed to each subsequent layer
- **Output** of final layer: "prediction"
- Used for both **training** and prediction

Some possible outputs:

- **Binary classification**
  - Single probability between 0 and 1
- **Multiclass classification**
  - Distribution of probabilities summing to 1
- **Regression** values
  - Continuous numerical predictions

Is there also a backward pass?

- **Backward pass**, or **backpropagation** is used to update weights and biases during training
- In the "training loop", we:
  1. **Propagate** data forward
  2. **Compare** outputs to true values (ground-truth)
  3. **Backpropagate** to update model weights and biases
  4. **Repeat** until weights and biases are tuned to produce useful outputs

Binary classification: forward pass

```
# Create input data of shape 5x6
input_data = torch.tensor(
        [[-0.4421,  1.5207,  2.0607, -0.3647,  0.4691,  0.0946],
        [-0.9155, -0.0475, -1.3645,  0.6336, -1.9520, -0.3398],
        [ 0.7406,  1.6763, -0.8511,  0.2432,  0.1123, -0.0633],
        [-1.6630, -0.0718, -0.1285,  0.5396, -0.0288, -0.8622],
        [-0.7413,  1.7920, -0.0883, -0.6685,  0.4745, -0.4245]])
```

```
# Create binary classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1), # Second linear layer
  nn.Sigmoid() # Sigmoid activation function
)

# Pass input data through model
output = model(input_data)
```

Binary classification: forward pass

```
print(output)
```

```
tensor([[0.5188], [0.3761], [0.5015], [0.3718], [0.4663]],
    grad_fn=<SigmoidBackward0>)
```

- **Outputs:**
  - five probabilities between zero and one
  - one value for each sample (row) in data
- **Classification:**
  - Class = 1 for first and third values: `0.5188` , `0.5015`
  - Class = 0 for second, fourth and fifth values: `0.3761` , `0.3718` , `0.4633`

Multi-class classification: forward pass

```
# Specify model has three classes
n_classes = 3

# Create multiclass classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, n_classes), # Second linear layer
  nn.Softmax(dim=-1) # Softmax activation
)

# Pass input data through model
output = model(input_data)
print(output.shape)
```

```
torch.Size([5, 3])
```

Multi-class classification: forward pass

```
print(output)
```

```
tensor([[0.4969, 0.3606, 0.1425],
        [0.5105, 0.3262, 0.1633],
        [0.3253, 0.3174, 0.3572],
        [0.5499, 0.3361, 0.1141],
        [0.4117, 0.3366, 0.2517]], grad_fn=<SoftmaxBackward0>)
```

- **Outputs:**
  - The output dimension is 5×3
  - Each row sums to one
  - Value with highest probability is assigned predicted label in each row
  - Row 1 = class 1 (mammal), row 2 = class 1 (mammal), row 3 = class 3 (reptile)

Regression: forward pass

```
# Create regression model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1) # Second linear layer
)

# Pass input data through model
output = model(input_data)

# Return output
print(output)
```

```
tensor([[0.3818],
        [0.0712],
        [0.3376],
        [0.0231],
        [0.0757]],
        grad_fn=<AddmmBackward0>)
```

Why do we need a loss function?

Loss function:

- Gives feedback to model during training
- Takes in model prediction $\hat{y}$ and ground truth y
- Outputs a float

Why do we need a loss function?

hair	feathers	eggs	milk	airborne	aquatic	predator	toothed	backbone	breathes	venomous	fins	legs	tail	domestic	catsize	class
1	0	0	1	0	0	1	1	1	1	0	0	4	0	0	1	0

- Predicted class = 0 -> **correct** = low loss
- Predicted class = 1 -> **wrong** = high loss
- Predicted class = 2 -> **wrong** = high loss

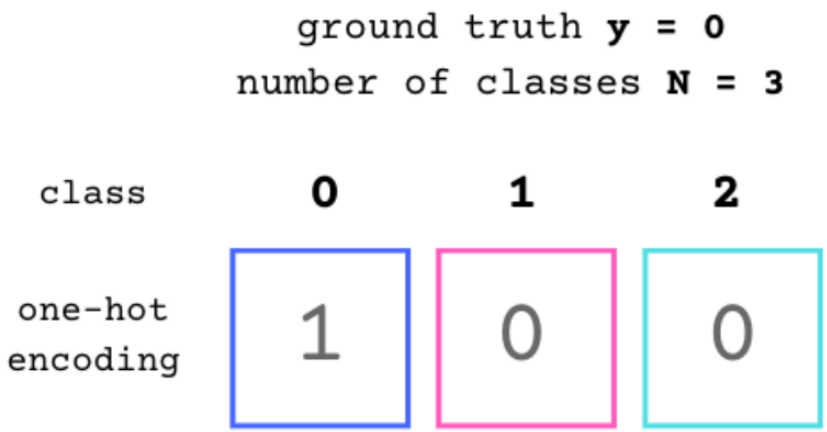
One-hot encoding concepts

- $loss = F(y, \hat{y})$
- y is a single **integer** (class label)
  - e.g. $y = 0$ when y is a mammal
- $\hat{y}$ is a **tensor** (output of softmax)
  - If N is the number of classes, e.g. N = 3
  - $\hat{y}$ is a tensor with N dimensions,
    - e.g. $\hat{y}$ = `[0.57492, 0.034961, 0.15669]`

How do we compare an integer with a tensor?

One-hot encoding concepts

Transforming true label to tensor of zeros and ones




```
one_hot_numpy = np.array([1, 0, 0])
```

Transforming labels with one-hot encoding

```
import torch.nn.functional as F

F.one_hot(torch.tensor(0), num_classes = 3)
```

```
tensor([1, 0, 0])
```

```
F.one_hot(torch.tensor(1), num_classes = 3)
```

```
tensor([0, 1, 0])
```

```
F.one_hot(torch.tensor(2), num_classes = 3)
```

```
tensor([0, 0, 1])
```

Cross entropy loss in PyTorch

```
from torch.nn import CrossEntropyLoss

scores = tensor([[-0.1211,  0.1059]])
one_hot_target = tensor([[1, 0]])

criterion = CrossEntropyLoss()
criterion(scores.double(), one_hot_target.double())
```

```
tensor(0.8131, dtype=torch.float64)
```

Bringing it all together

Loss function takes

- **scores**
  - model predictions **before** the final softmax function
- **one_hot_target**
  - one hot encoded ground truth label

and outputs

- **loss**
  - a single **float.**

Our training goal is to minimize loss.

Using derivatives to update model parameters

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Minimizing the loss

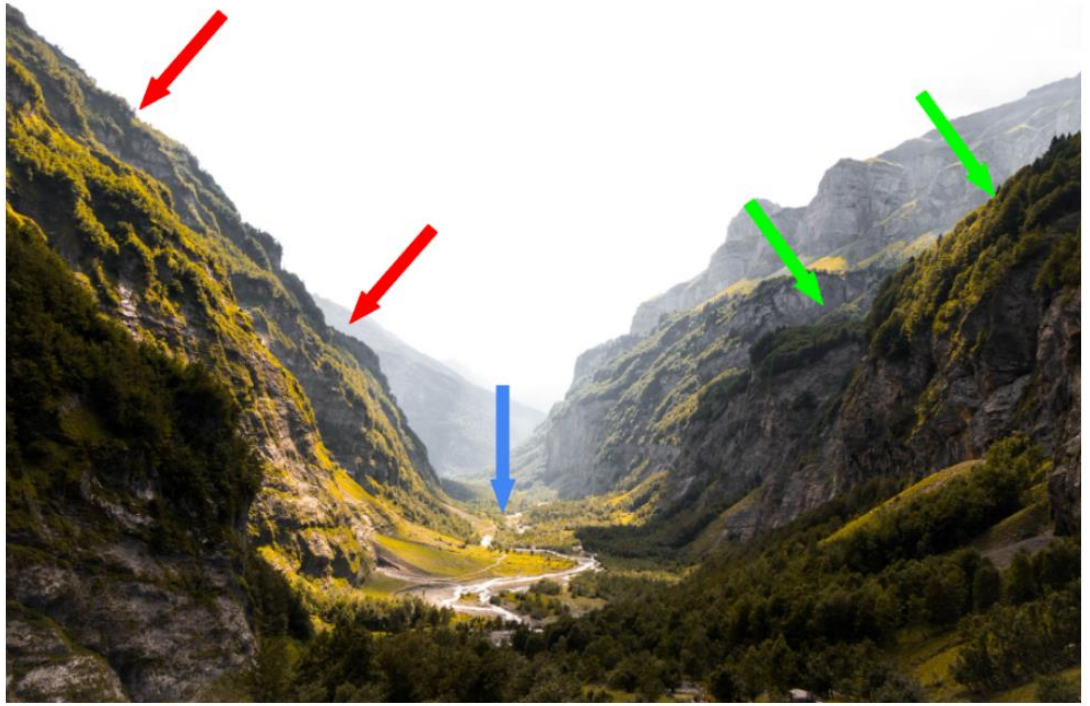
We need to minimize loss

- High loss: model prediction is wrong
- Low loss: model prediction is correct

An analogy for derivatives

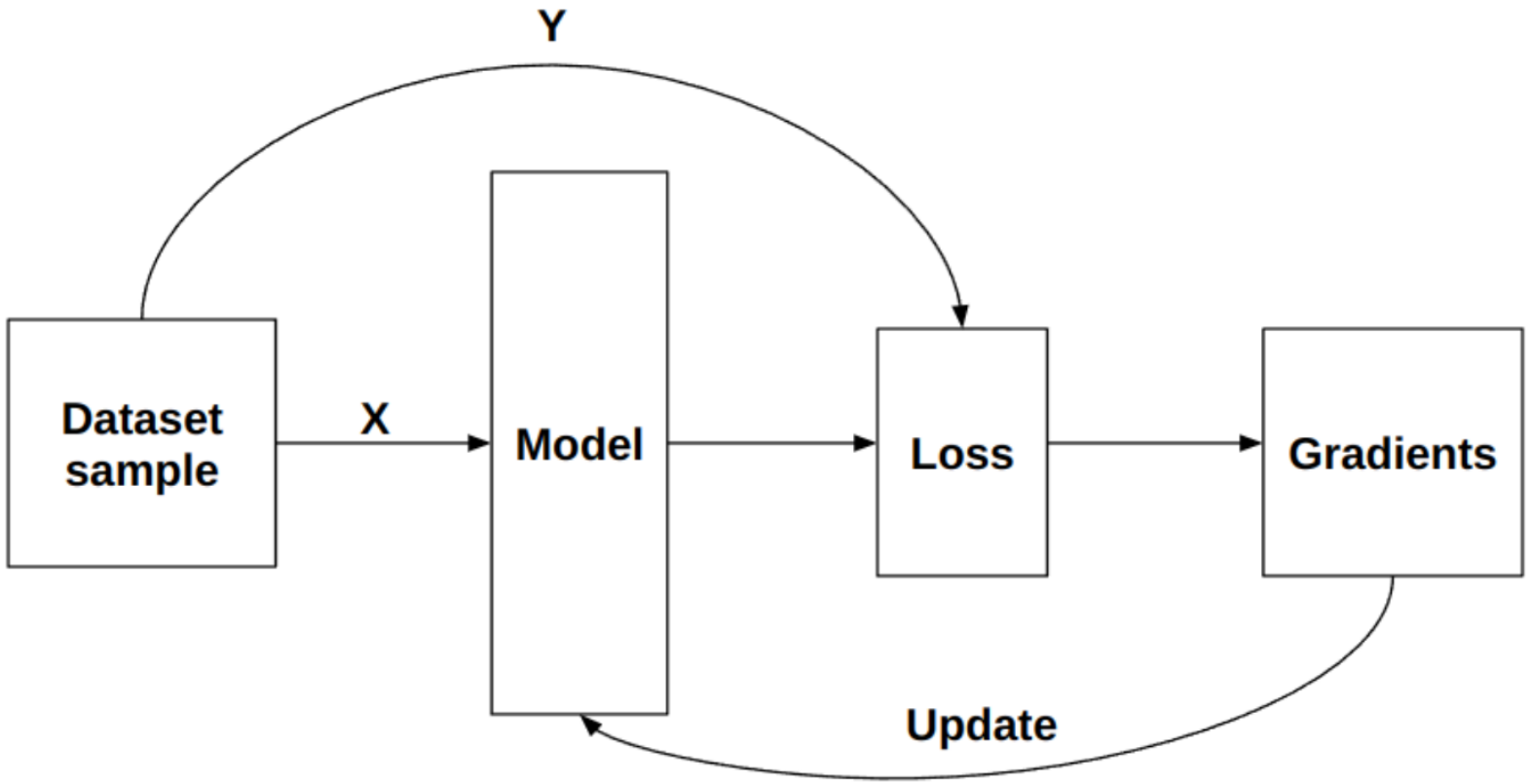
Hiking down a mountain to the valley floor:

- **steep slopes:**
  - a step makes us lose a lot of elevation = derivative is high (red arrows)
- **gentler slopes:**
  - a step makes us lose a little bit of elevation = derivative is low (green arrows)
- **valley floor:**
  - not losing elevation by taking a step = derivative is null (blue arrow)



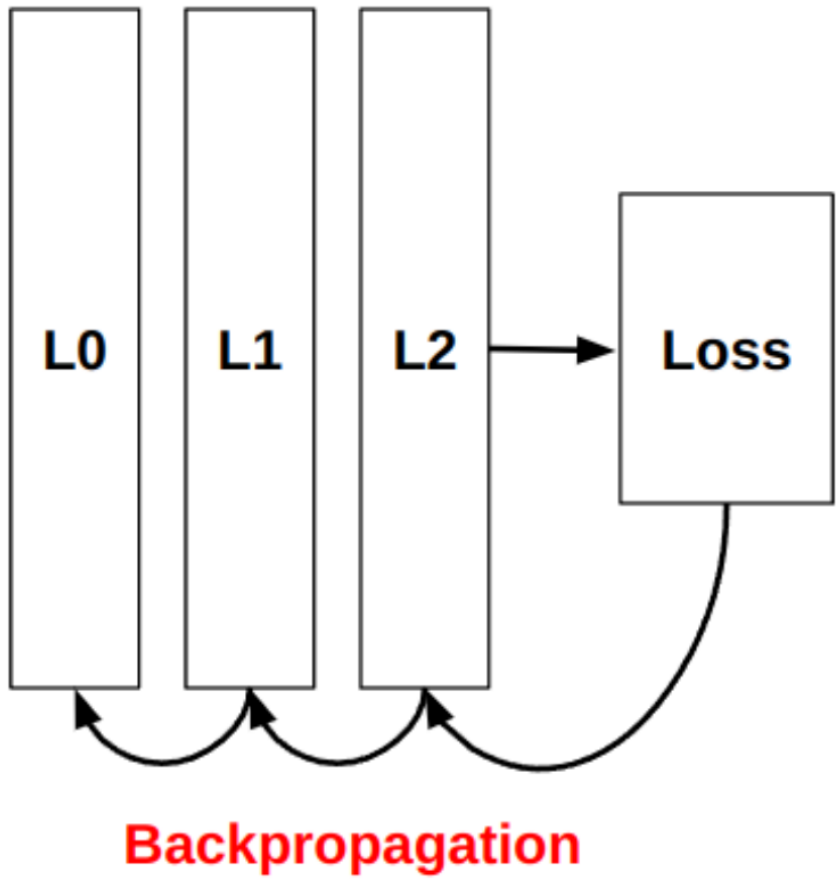
Connecting derivatives and model training

Model training: updating a model's parameters to **minimize** the loss.



Backpropagation concepts

- Consider a network made of three layers, $L0$, $L1$ and $L2$
  - we calculate local gradients for $L0$, $L1$ and $L2$ using **backpropagation**
  - we calculate loss gradients with respect to $L2$, then use $L2$ gradients to calculate $L1$ gradients, and so on



Backpropagation in PyTorch

```
# Create the model and run a forward pass
model = nn.Sequential(nn.Linear(16, 8),
                      nn.Linear(8, 4),
                      nn.Linear(4, 2))
prediction = model(sample)
```

```
# Calculate the loss and compute the gradients
criterion = CrossEntropyLoss()
loss = criterion(prediction, target)
loss.backward()
```

```
# Access each layer's gradients
model[0].weight.grad, model[0].bias.grad
model[1].weight.grad, model[1].bias.grad
model[2].weight.grad, model[2].bias.grad
```

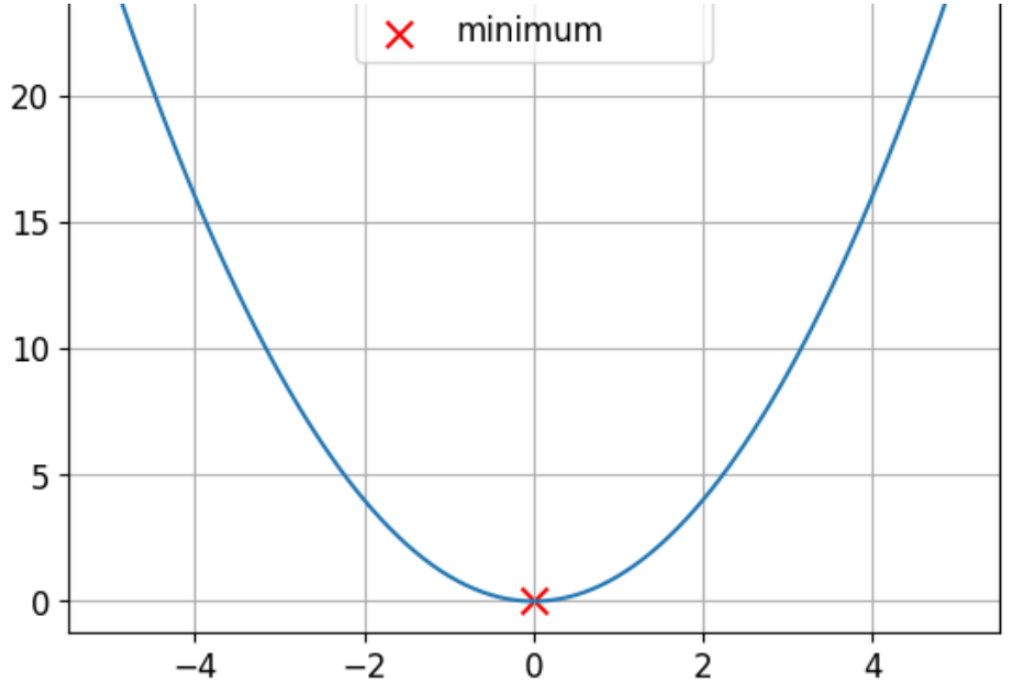
Updating model parameters

- Update the weights by subtracting local gradients scaled by the **learning rate**

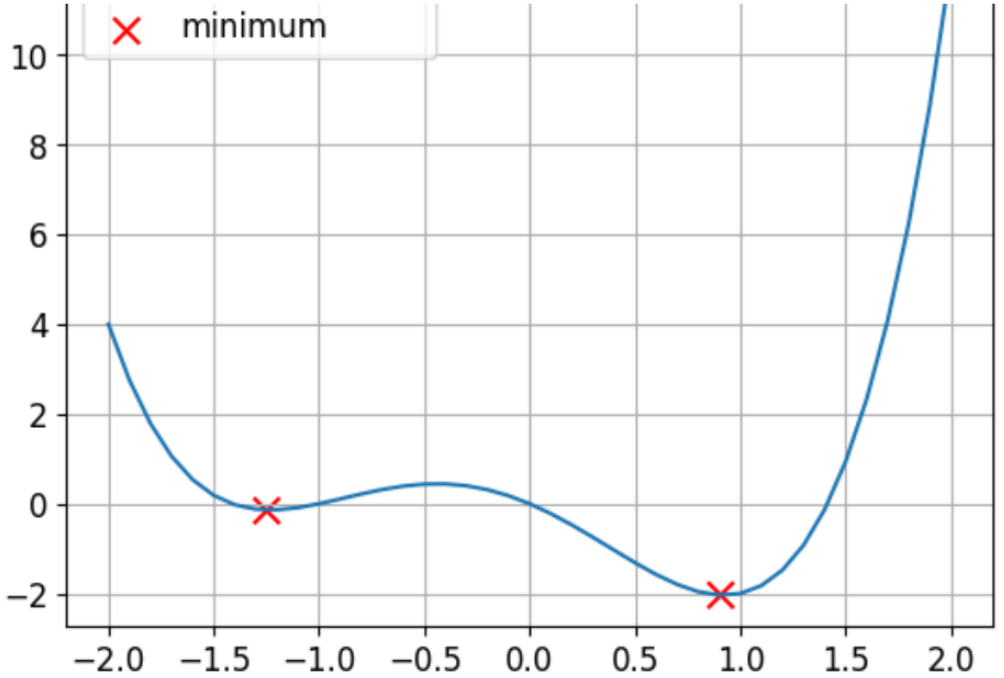
```
# Learning rate is typically small
lr = 0.001

# Update the weights
weight = model[0].weight
weight_grad = model[0].weight.grad
weight = weight - lr * weight_grad

# Update the biases
bias = model[0].bias
bias_grad = model[0].bias.grad
bias = bias - lr * bias_grad
```

Convex and non-convex functions

This is a **convex function.**



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



Gradient descent

- For non-convex functions, we will use an iterative process such as **gradient descent**
- In PyTorch, an **optimizer** takes care of weight updates
- The most common **optimizer** is stochastic gradient descent (SGD)

```
import torch.optim as optim

# Create the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

- Optimizer handles updating model parameters (or weights) after calculation of local gradients

```
optimizer.step()
```

Writing our first training loop

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Training a neural network

1. Create a model
2. Choose a loss function
3. Create a dataset
4. Define an optimizer
5. Run a training loop, where for each sample of the dataset, we repeat:
    - Calculating loss (forward pass)
    - Calculating local gradients
    - Updating model parameters

Introducing the Data Science Salary dataset

- This dataset contains salary data for data science-related jobs.
- The features are: `experience_level`, `employment_type`, `remote_ratio` and `company_size`. They were turned into categories.

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

- The target is salary in US dollars; it is **not a category but a continuous quantity**
- For regression problems, we cannot use softmax or sigmoid as last activation function
- We need a different loss function than cross-entropy

Introducing the Mean Squared Error Loss

- The mean squared error loss (MSE loss) is the squared difference between the prediction and the 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)
```

- This loss is used for regression problems (e.g., when trying to fit a linear regression model).

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

```
# Loop through the dataset multiple times
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()
```

Introduction to Deep Learning with PyTorch

- Introduction to PyTorch
- Training Our First Neural Network with PyTorch
- **Neural Network Architecture and Hyperparameters**
- Evaluating and Improving Models

Discovering activation functions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Limitations of the sigmoid and softmax function

Sigmoid functions:

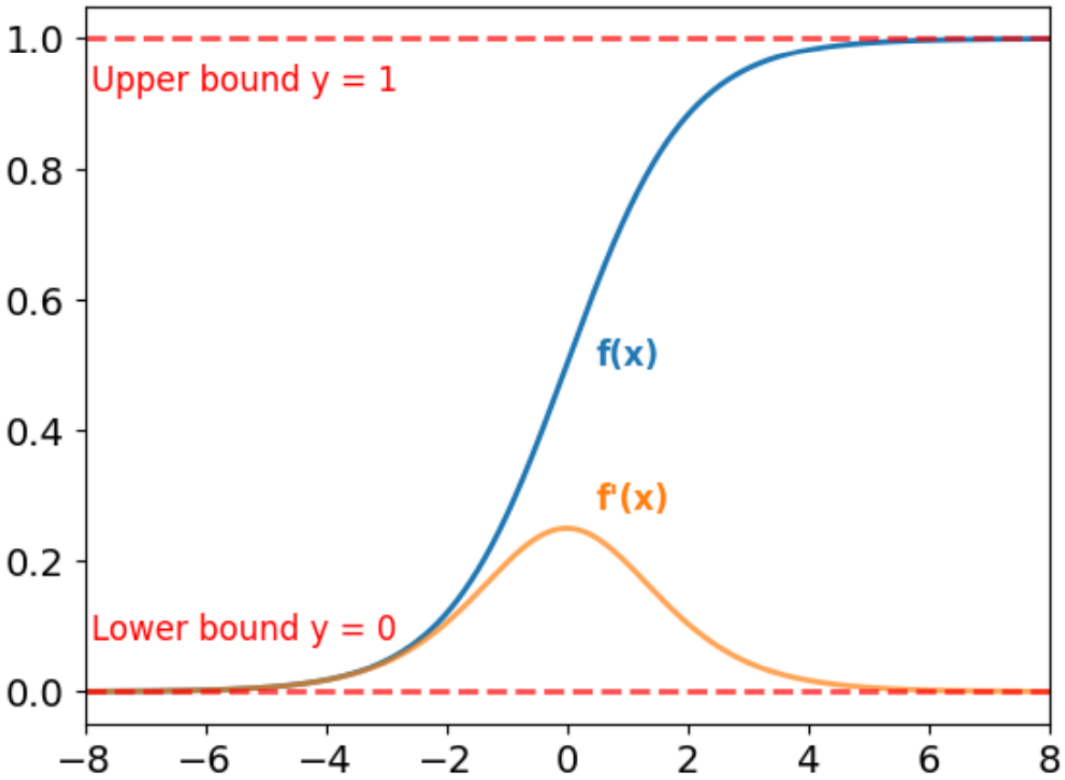
- Bounded between 0 and 1
- Can be used anywhere in the network

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**.



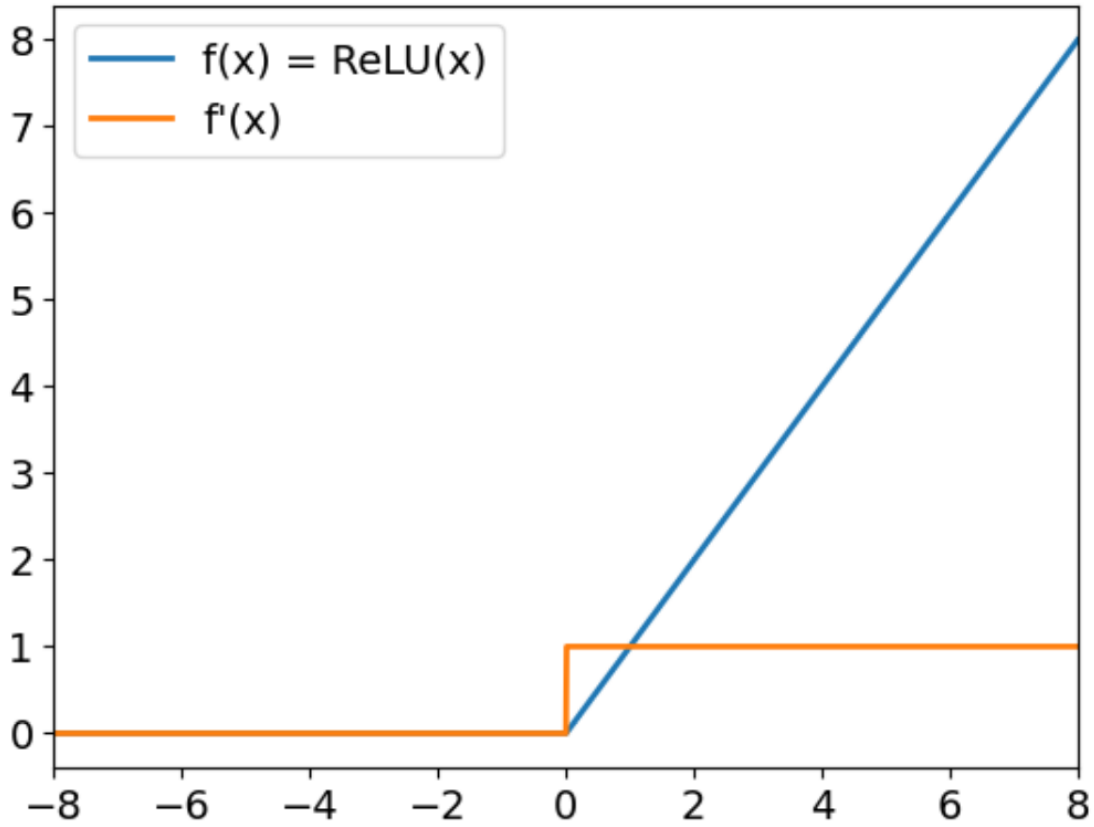
Introducing ReLU

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()
```



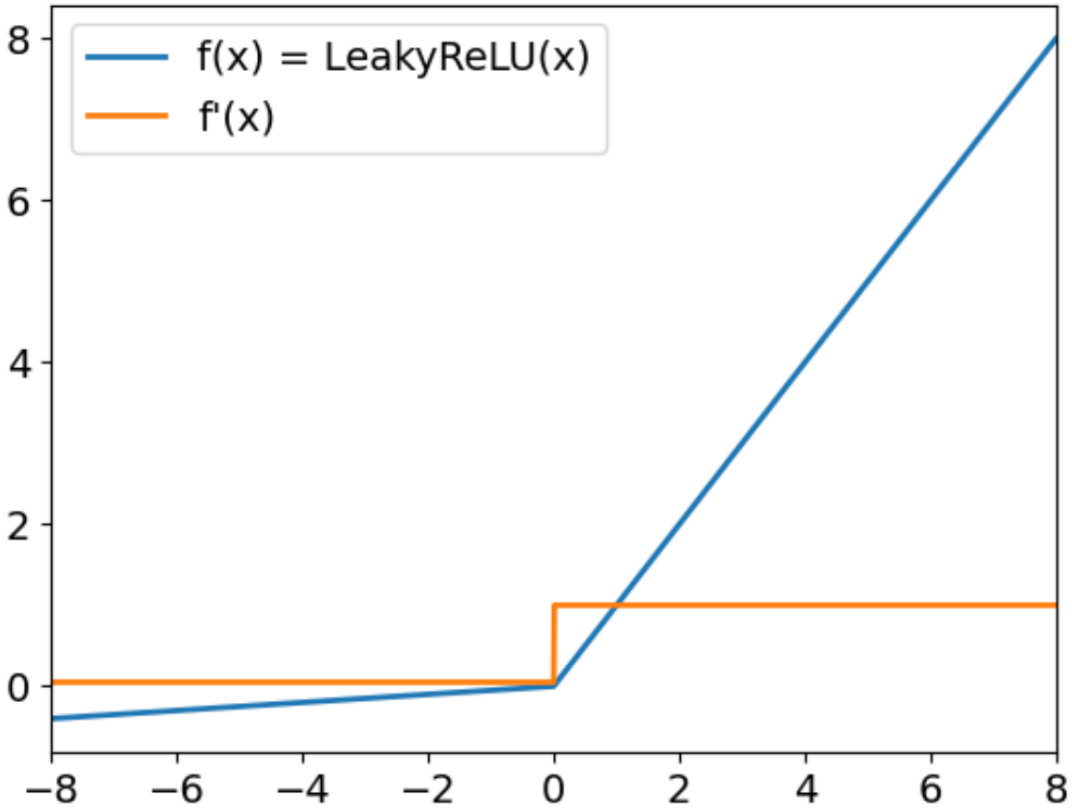
Introducing Leaky ReLU

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)
```



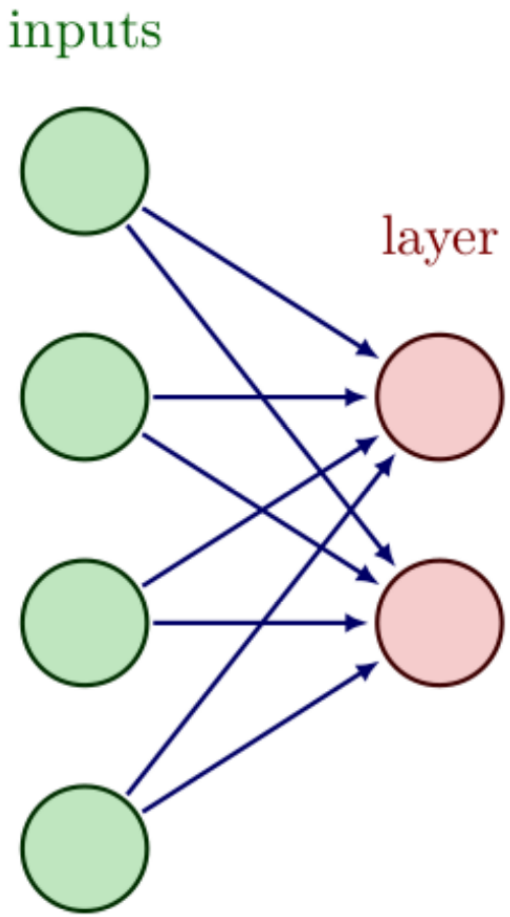
A deeper dive into neural network architecture

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

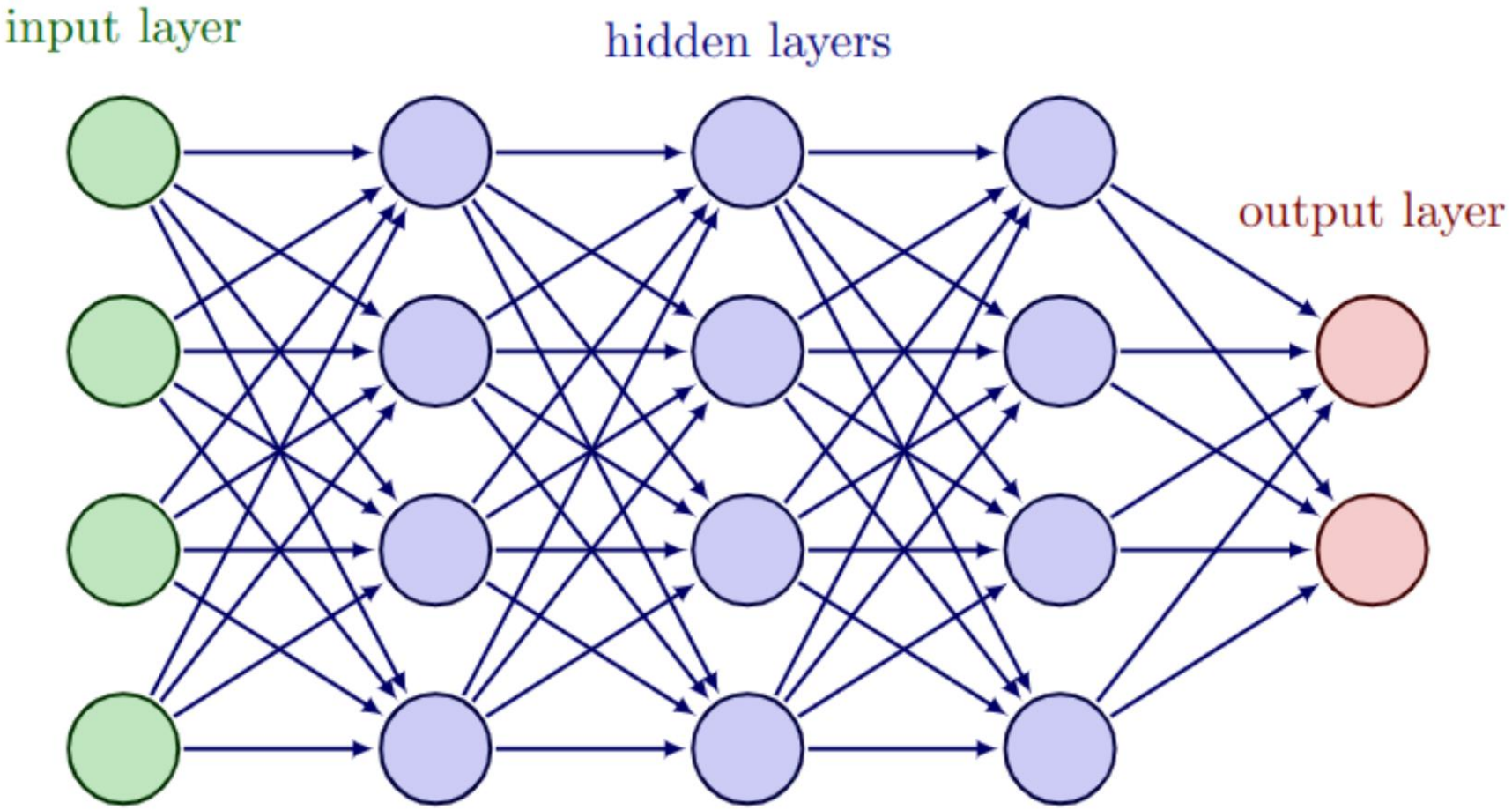
Layers are made of neurons

- 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

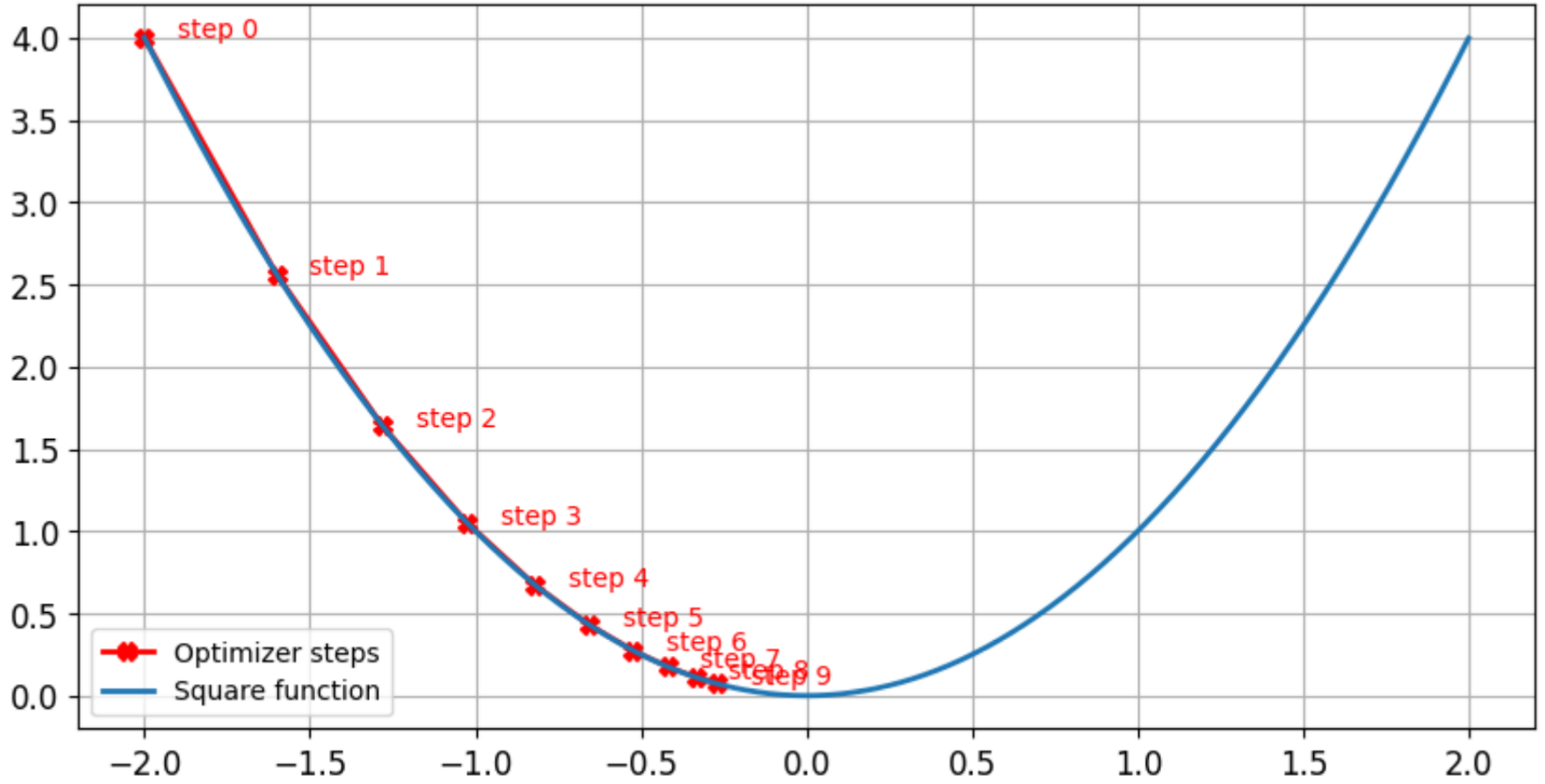
- Training a neural network = solving an **optimization problem.**

Stochastic Gradient Descent (SGD) optimizer

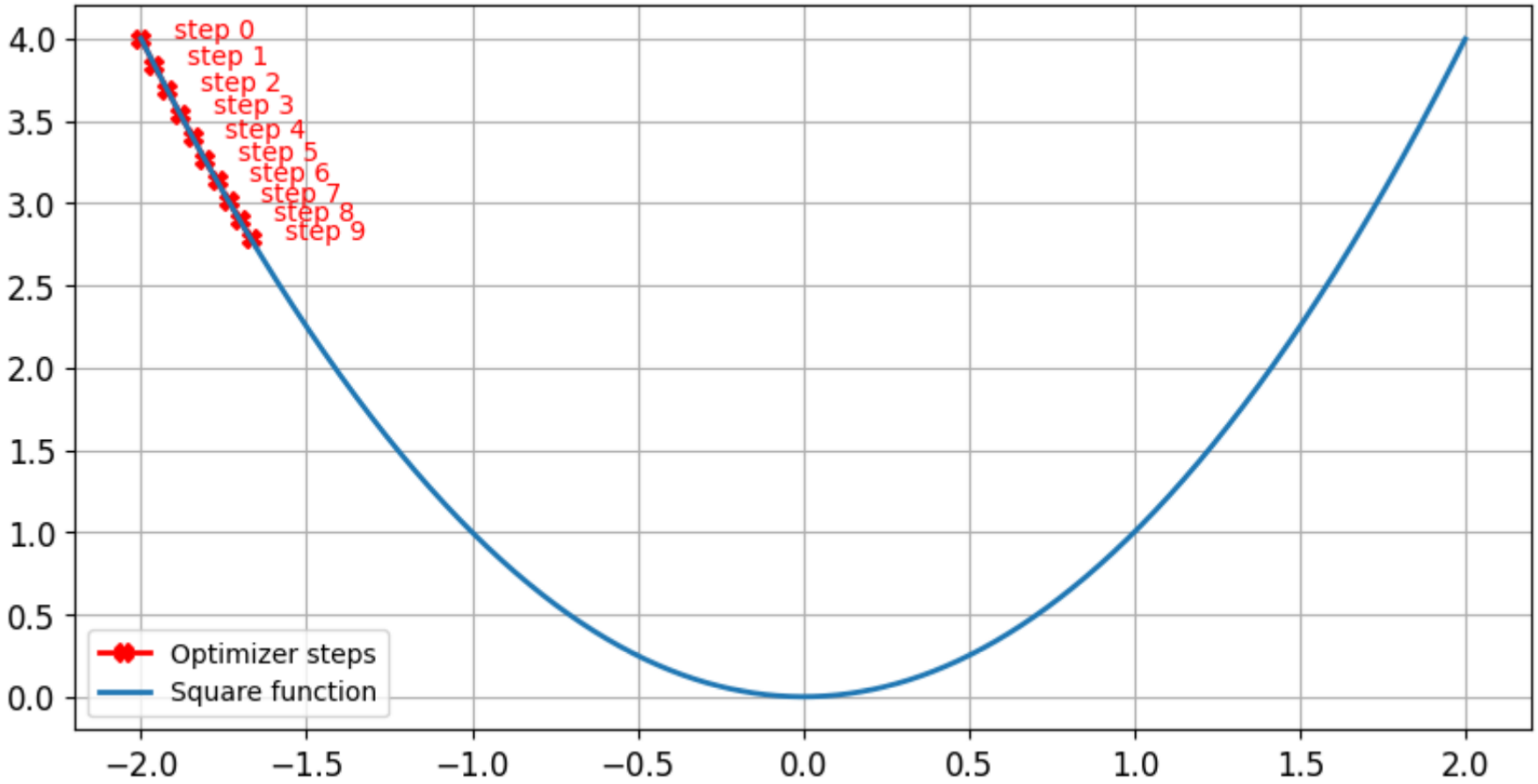
```
sgd = optim.SGD(model.parameters(), lr=0.01, momentum=0.95)
```

- 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)

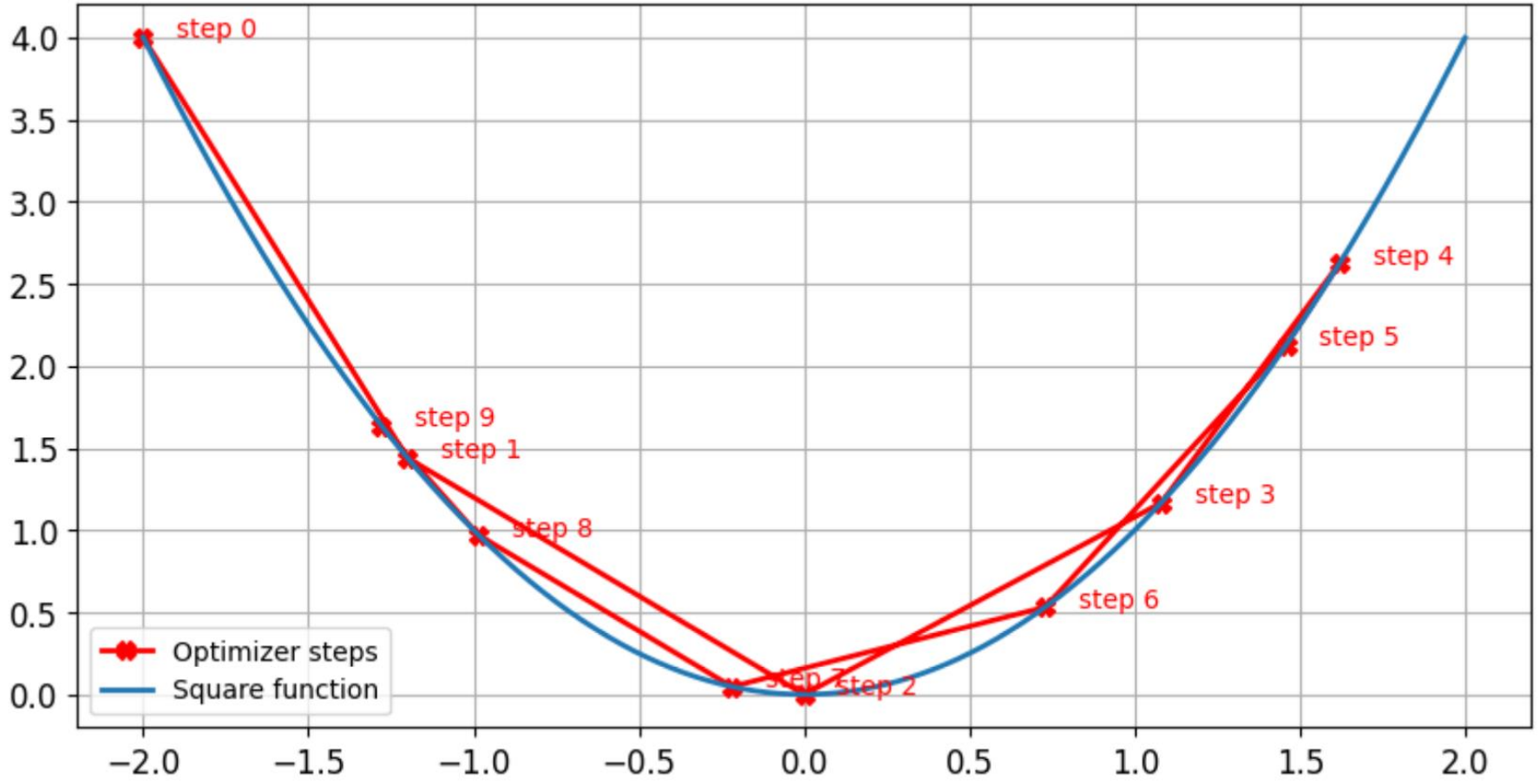
Impact of the learning rate: optimal learning rate



Impact of the learning rate: small learning rate

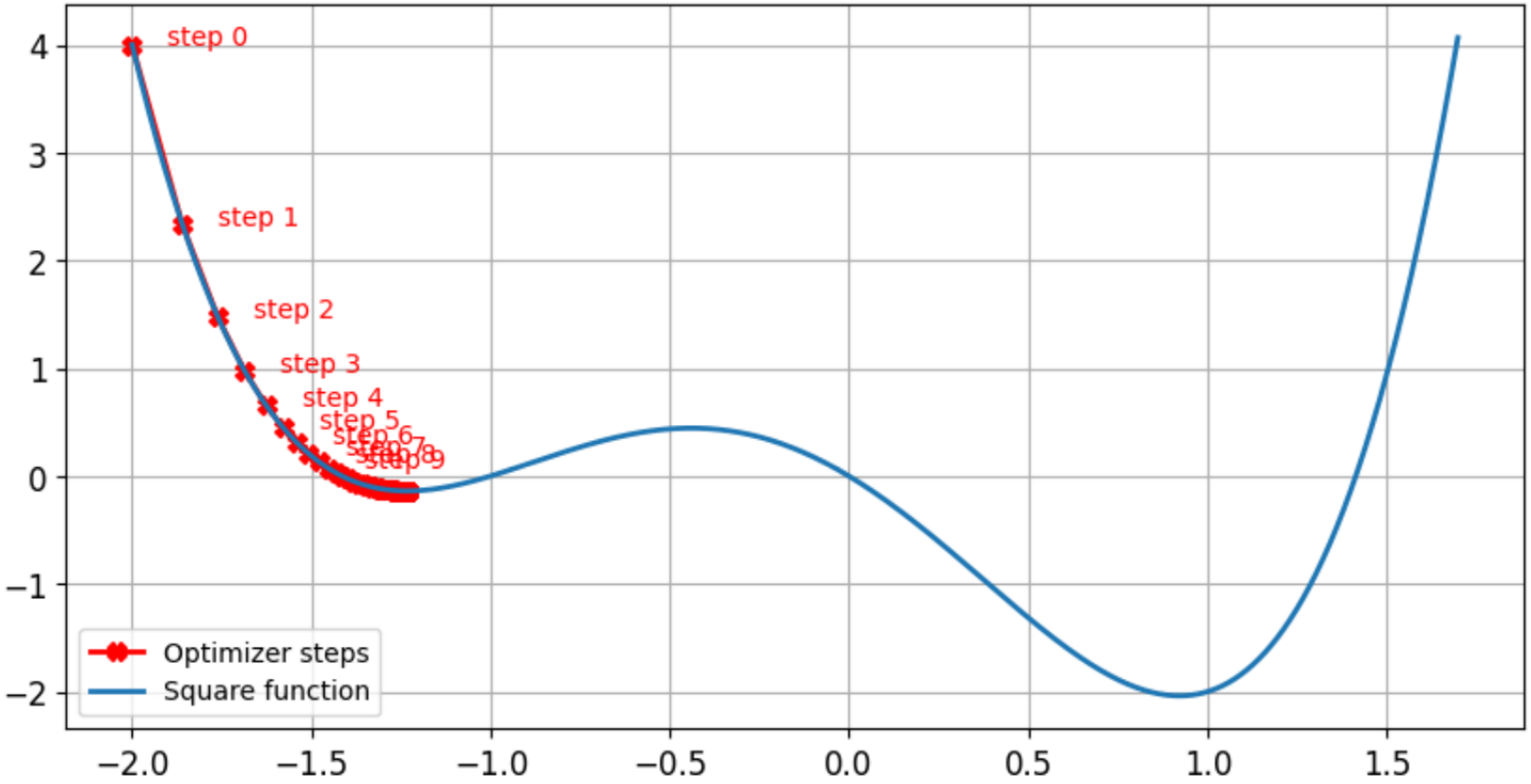


Impact of the learning rate: high learning rate



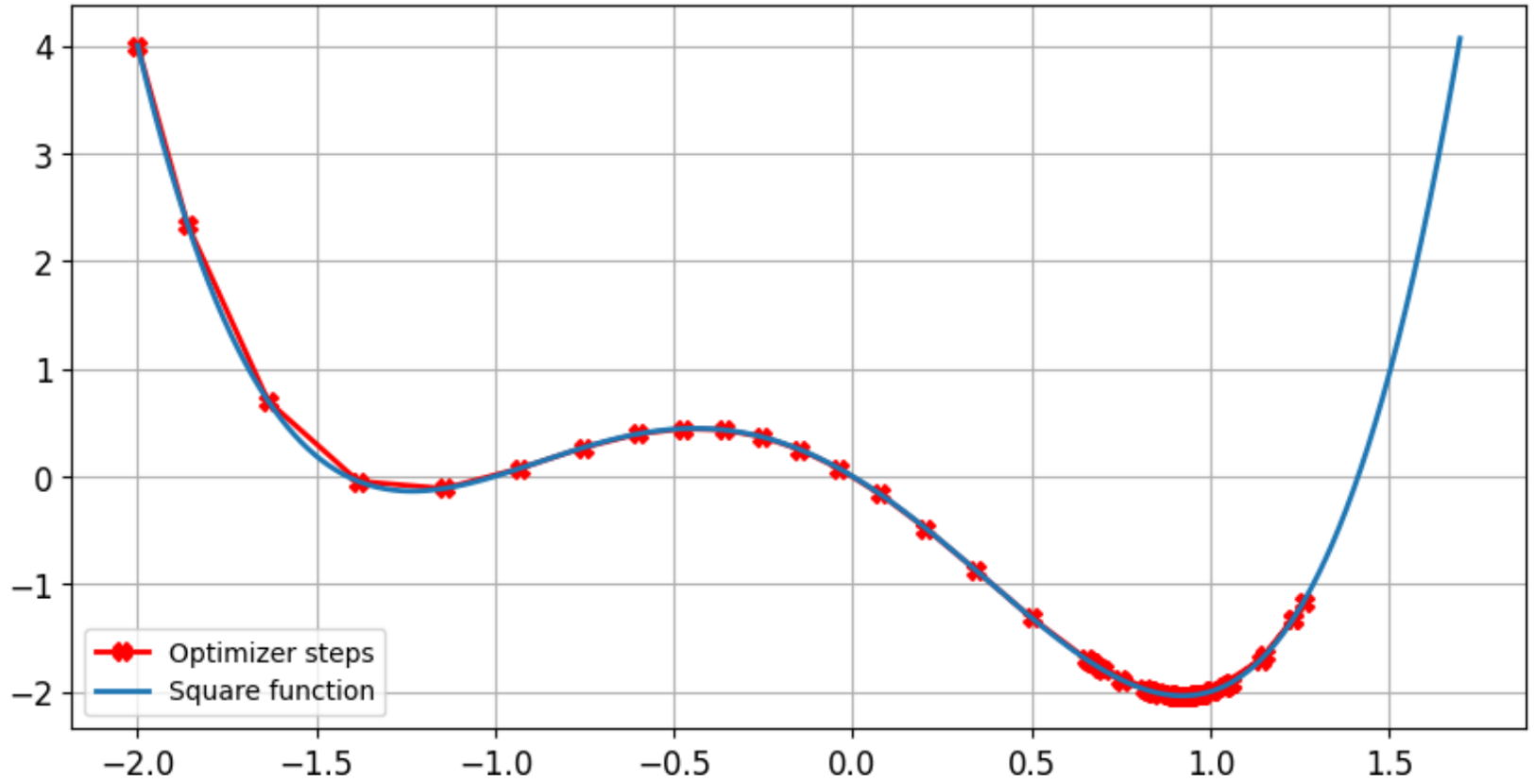
Without momentum

- `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

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
```

Introduction to Deep Learning with PyTorch

- Introduction to PyTorch
- Training Our First Neural Network with PyTorch
- Neural Network Architecture and Hyperparameters
- **Evaluating and Improving Models**

A deeper dive into loading data

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Back to our animals dataset

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

animal_name	hair	feathers	eggs	milk	predator	fins	legs	tail	type
skimmer	0	1	1	0	1	0	2	1	2
gull	0	1	1	0	1	0	2	1	2
seahorse	0	0	1	0	0	1	0	1	4
tuatara	0	0	1	0	1	0	4	1	3
squirrel	1	0	0	1	0	0	2	1	1

Type key: **mammal** (1), **bird** (2), **reptile** (3), **fish** (4), **amphibian** (5), **bug** (6), **invertebrate** (7).

Back to our animals dataset: defining features

```
import numpy as np
# Define input features
features = animals.iloc[:, 1:-1]
X = features.to_numpy()
print(X)
```

```
array([[0, 1, 1, 0, 1, 0, 2, 1],
       [0, 1, 1, 0, 1, 0, 2, 1],
       [0, 0, 1, 0, 0, 1, 0, 1],
       [0, 0, 1, 0, 1, 0, 4, 1],
       [1, 0, 0, 1, 0, 0, 2, 1]])
```

Back to our animals dataset: defining target values

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

```
array([2, 2, 4, 3, 1])
```

Recalling TensorDataset

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

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

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

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

Recalling DataLoader

```
from torch.utils.data import DataLoader
```

```
batch_size = 2
shuffle = True

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

Recalling 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([[0., 0., 1., 0., 0., 1., 0., 1.],
                      [0., 1., 1., 0., 1., 0., 2., 1.]])
batch labels: tensor([4., 2.])
batch inputs: tensor([[0., 1., 1., 0., 1., 0., 2., 1.],
                      [1., 0., 0., 1., 0., 0., 2., 1.]])
batch labels: tensor([2., 1.])
batch inputs: tensor([[0., 0., 1., 0., 1., 0., 4., 1.]])
batch labels: tensor([3.])
```

Evaluating model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Training, validation and testing

- Raw dataset is usually split in three subsets:

	Percent of data	Role
Training	80-90%	Used to adjust the model's parameters
Validation	10-20%	Used for hyperparameter tuning
Testing	5-10%	Only used once to calculate final metrics

Model evaluation metrics

- In this video, we'll focus on evaluating:
  - **Loss**
    - Training
    - Validation
  - **Accuracy**
    - Training
    - Validation

- In classification, **accuracy** measures how well model correctly predicts ground truth labels

Calculating training loss

- For each epoch:
  - we sum up the loss for each iteration of the training set dataloader
  - at the end of the epoch, we calculate the mean training loss

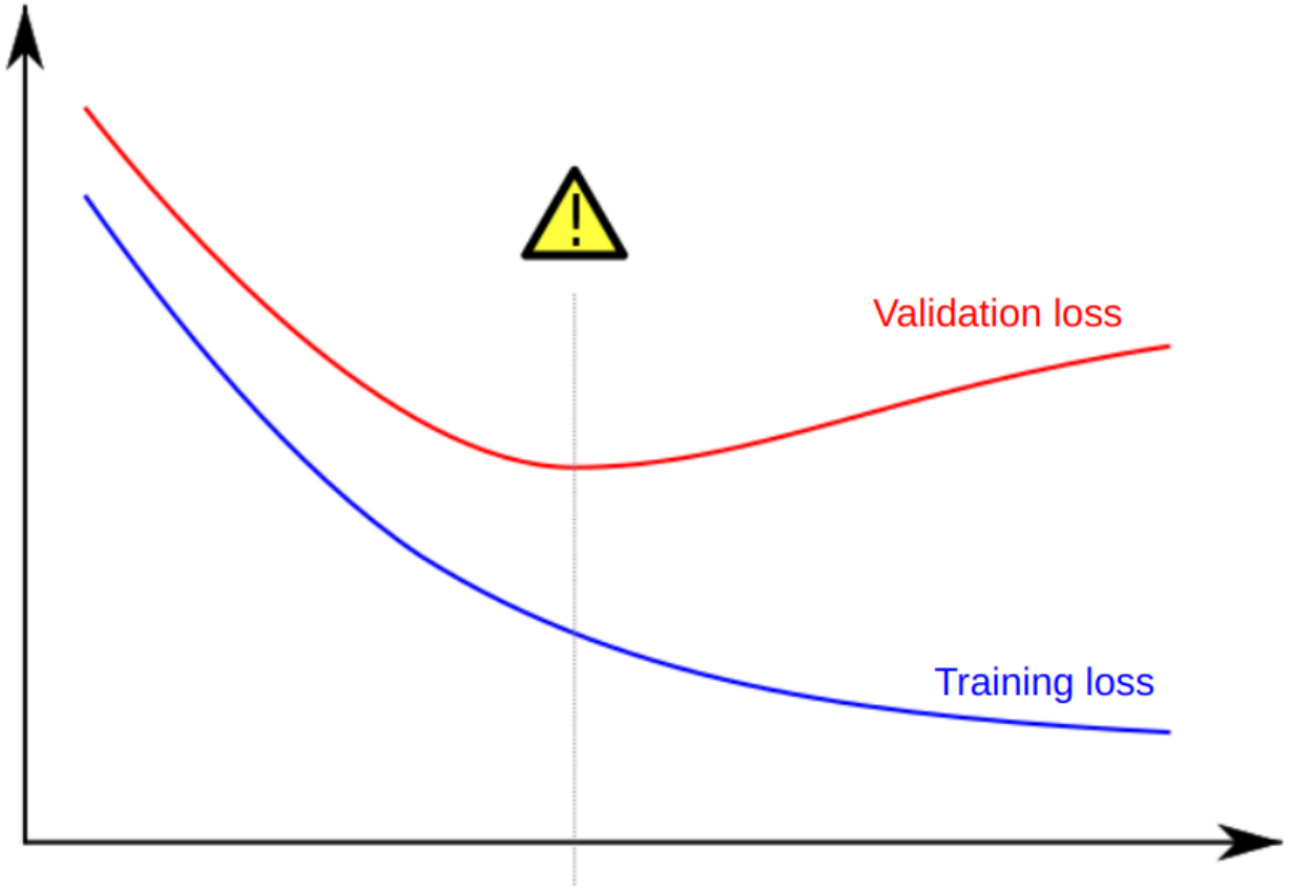
```
training_loss = 0.0
for i, data in enumerate(trainloader, 0):
    # Run the forward pass
    ...

    # Calculate the loss
    loss = criterion(outputs, labels)
    # Calculate the gradients
    ...
    # Calculate and sum the loss
    training_loss += loss.item()
epoch_loss = training_loss / len(trainloader)
```

Calculating validation loss

- After the training epoch, we iterate over the validation set and calculate the average validation loss

```
validation_loss = 0.0
model.eval() # Put model in evaluation mode
with torch.no_grad(): # Speed up the forward pass
  for i, data in enumerate(validationloader, 0):
      # Run the forward pass
      ...
      # Calculate the loss
      loss = criterion(outputs, labels)
      validation_loss += loss.item()
epoch_loss = validation_loss / len(validationloader)
model.train()
```

Overfitting



Calculating accuracy with torchmetrics

```
import torchmetrics

# Create accuracy metric using torch metrics
metric = torchmetrics.Accuracy(task="multiclass", num_classes=3)
for i, data in enumerate(dataloader, 0):
    features, labels = data
    outputs = model(features)
    # Calculate accuracy over the batch
    acc = metric(outputs, labels.argmax(dim=-1))
# Calculate accuracy over the whole epoch
acc = metric.compute()
print(f"Accuracy on all data: {acc}")
# Reset the metric for the next epoch (training or validation)
metric.reset()
```

Fighting overfitting

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Reasons for overfitting

- **Overfitting:** the model does not generalize to unseen data.
  - model memorizes training data
  - good performances on the training set / poor performances on the validation set
- Possible causes:

Problem	Solutions
Dataset is not large enough	Get more data / use data augmentation
Model has too much capacity	Reduce model size / add dropout
Weights are too large	Weight decay

Fighting overfitting

Strategies:

- Reducing model size or adding dropout layer
- Using weight decay to force parameters to remain small
- Obtaining new data or augmenting data

"Regularization" using a dropout layer

- Randomly zeroes out elements of the input tensor **during training**

```
model = nn.Sequential(nn.Linear(8, 4),
                      nn.ReLU(),
                      nn.Dropout(p=0.5))
features = torch.randn((1, 8))
model(i)
```

```
tensor([[1.4655, 0.0000, 0.0000, 0.8456]], grad_fn=<MulBackward0>)
```

- Dropout is added **after** the activation function
- Behaves differently during training and evaluation; we must remember to switch modes using `model.train()` and `model.eval()`

Regularization with weight decay

```
optimizer = optim.SGD(model.parameters(), lr=1e-3, weight_decay=1e-4)
```

- Optimizer's `weight_decay` parameter takes values between zero and one
  - Typically small value, e.g. 1e-3
- Weight decay adds penalty to loss function to discourage large weights and biases
- The higher the parameter, the less likely the model is to overfit

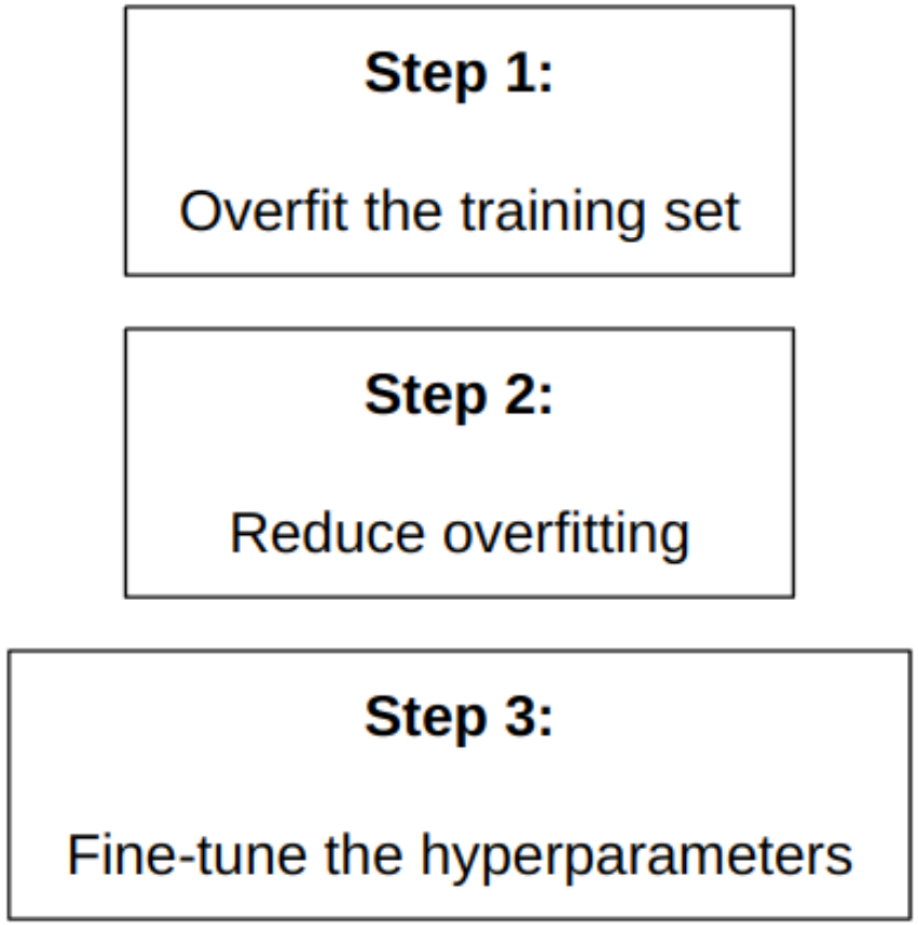
Data augmentation

Improving model performance

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Steps to maximize performance

- Overfit the training set
  - can we solve the problem?
  - sets a performance baseline
- Reduce overfitting
  - improve performances on the validation set
- Fine-tune hyperparameters



Step 1: overfit the training set

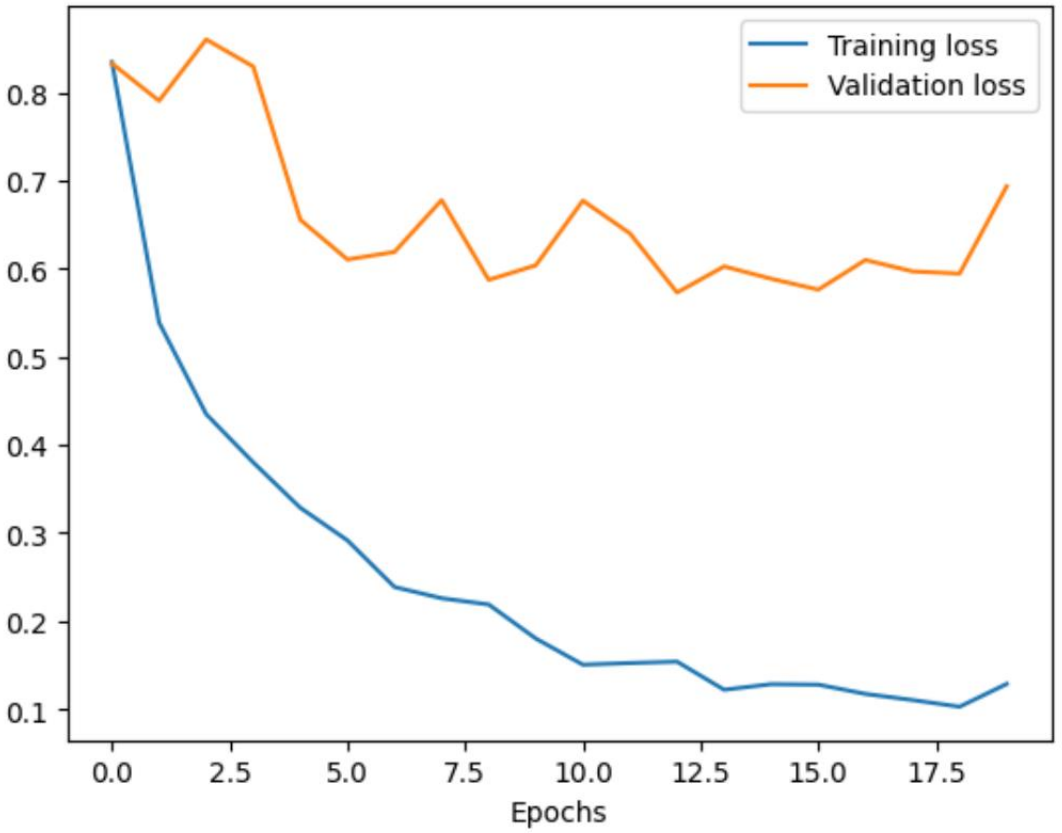
- Modify the training loop to overfit a single data point (batch size of 1)

```
features, labels = next(iter(trainloader))
for i in range(1e3):
  outputs = model(features)
  loss = criterion(outputs, labels)
  loss.backward()
  optimizer.step()
```

  - should reach 1.0 accuracy and 0 loss
  - helps findings bugs in the code
- **Goal**: minimize the training loss
  - create large enough model
  - use a default learning rate

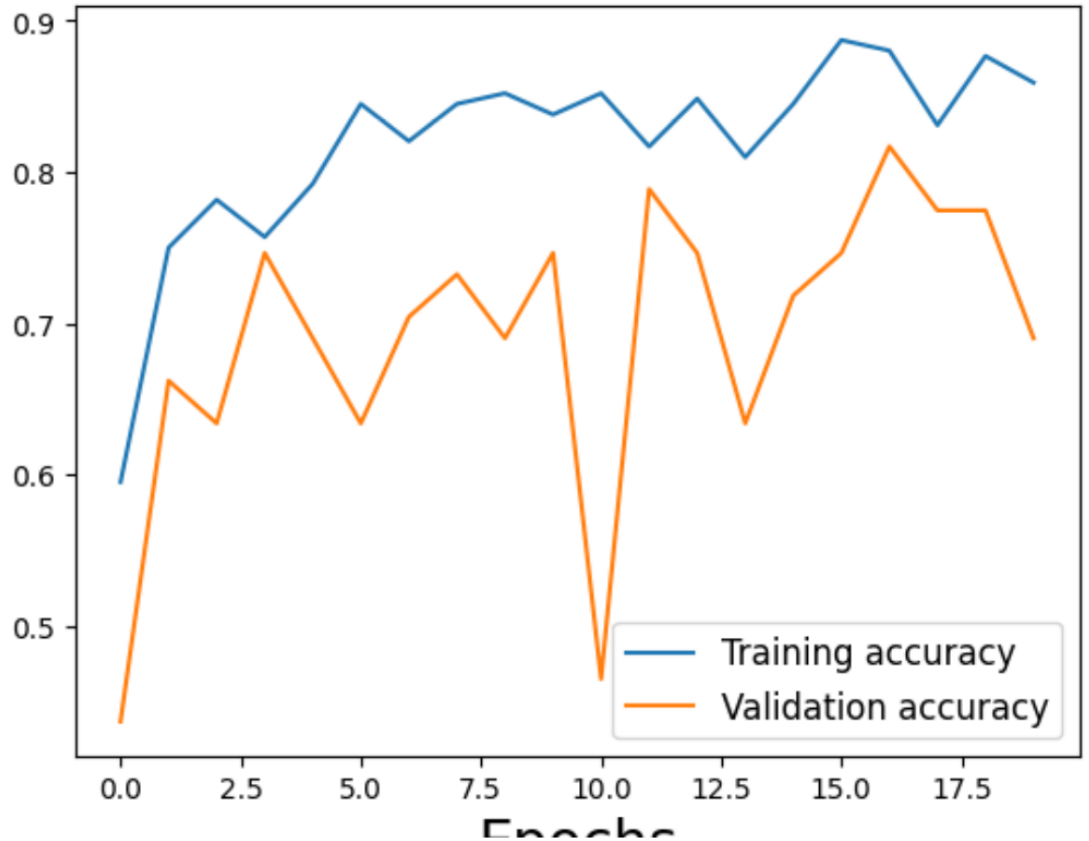
Step 2: reduce overfitting

- **Goal**: maximize the validation accuracy
- Experiment with:
  - Dropout
  - Data augmentation
  - Weight decay
  - Reducing model capacity
- Keep track of each hyperparameter and report maximum validation accuracy

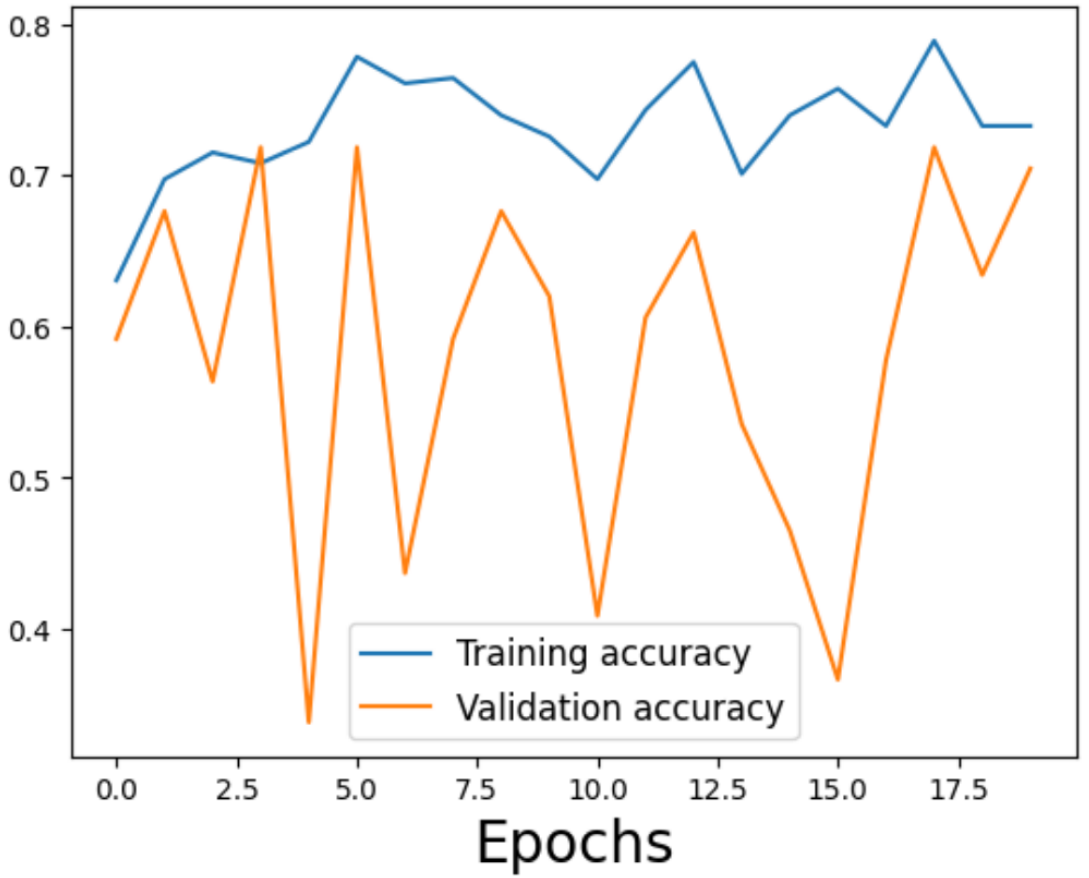


Step 2: reduce overfitting

Original model overfitting the training data



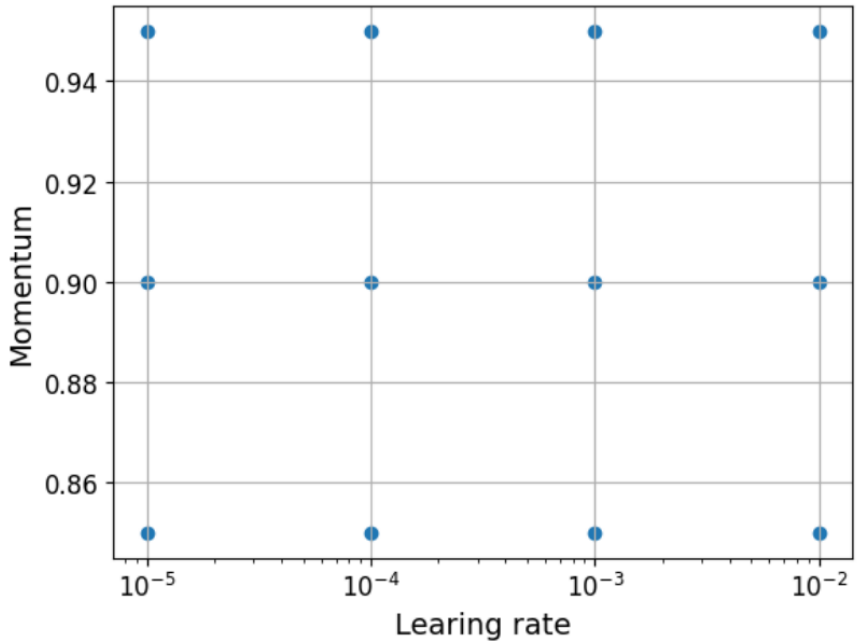
Model with too much regularization



Step 3: fine-tune hyperparameters

- Grid search

```
for factor in range(2, 6):
    lr = 10 ** -factor
```



- Random search

```
factor = np.random.uniform(2, 6)
lr = 10 ** -factor
```