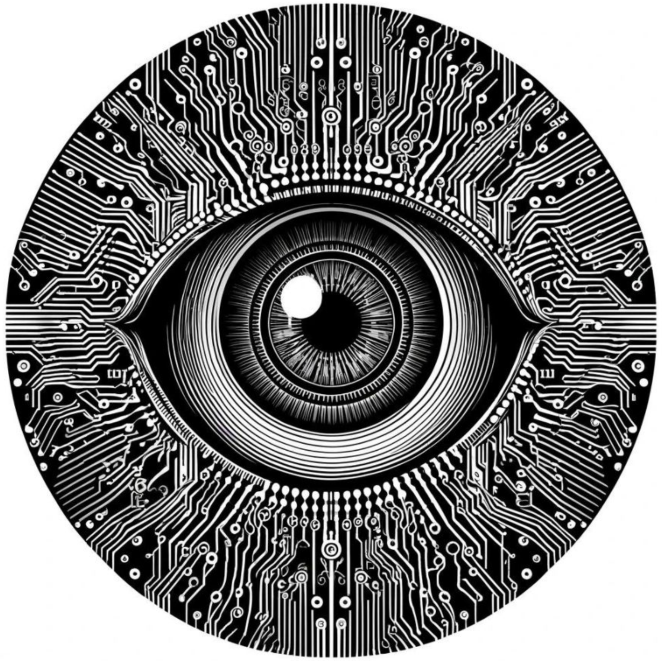
Building a Multilayer Perceptron for Regression in PyTorch



Antonio Rueda-Toicen

KI Service Zentrum by Hasso-Plattner-Institut HPI

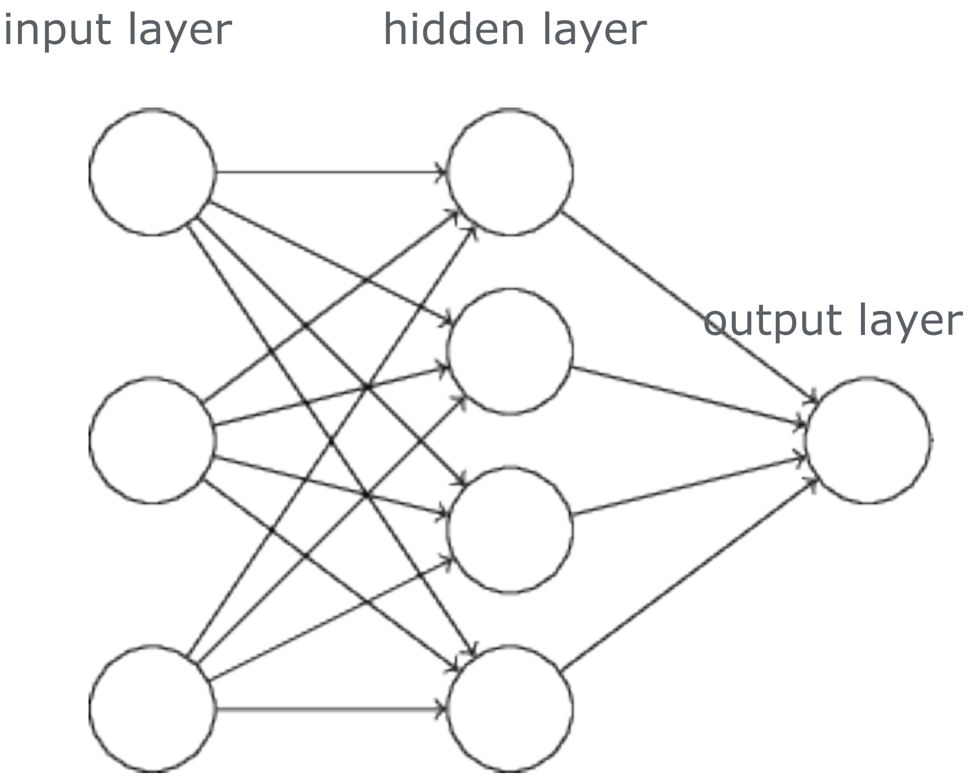
SPONSORED BY THE

Federal Ministry of Education and Research

Learning goals

- Implement a fully connected network in PyTorch with `nn.Linear` and `nn.Sequential` syntax
- Explore the necessary agreements between processing units and input size
- Understand the motivations for image input resizing
- Code input units, hidden layers, and output units
- Examine the representation of weights, gradients, and loss in PyTorch

A minimal network




```python
import torch.nn as nn

# Input layer to hidden layer (3 -> 4)
i_h_layer = nn.Linear(in_features=3, out_features=4, bias = False)

# Hidden layer to output layer (4 -> 1)
h_o_layer = nn.Linear(in_features=4, out_features=1, bias = False)


# Connecting the layers, notice that 'hidden layer' is implicit
model = nn.Sequential(
            i_h_layer,
            nn.ReLU(),
            h_o_layer,
        )

# The input has as many entries as we have input units
input_data = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float32)

# Feedforward pass
prediction = model(input_data)

# The value that we want to match
target = torch.tensor([11.], dtype=torch.float32)

# Our measure of error
loss_function = nn.L1Loss()

# Evaluate how good our prediction is
loss_value = loss_function(target, prediction)
```

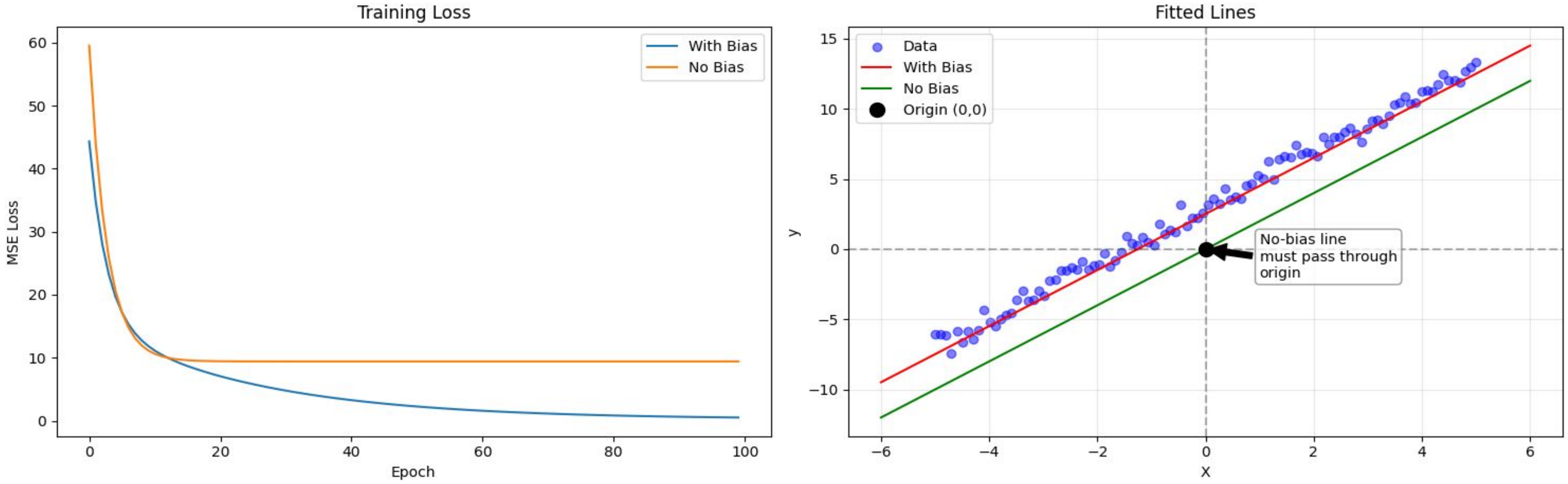
Inspecting nn.Linear

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

# Define a linear layer
layer = nn.Linear(in_features=4, out_features=1, bias=True)


# Examine parameters, weights and bias
for name, param in layer.named_parameters():
    print(f"{name}: {param.shape}")
    print(param)

# Forward pass
x = torch.randn(5, 4)  # Batch of 5 samples, 4 features each
output = layer(x)

# Batch of 5 samples, single output value
print(output.shape)
```

$$z = \mathbf{WX} = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^{m} w_i x_i$$

```
weight: torch.Size([1, 4])
Parameter containing:
tensor([[-0.0576,  0.3385, -0.3604, -0.1795]], requires_grad=True)
bias: torch.Size([1])
Parameter containing:
tensor([0.3461], requires_grad=True)
```

Bias vs no bias

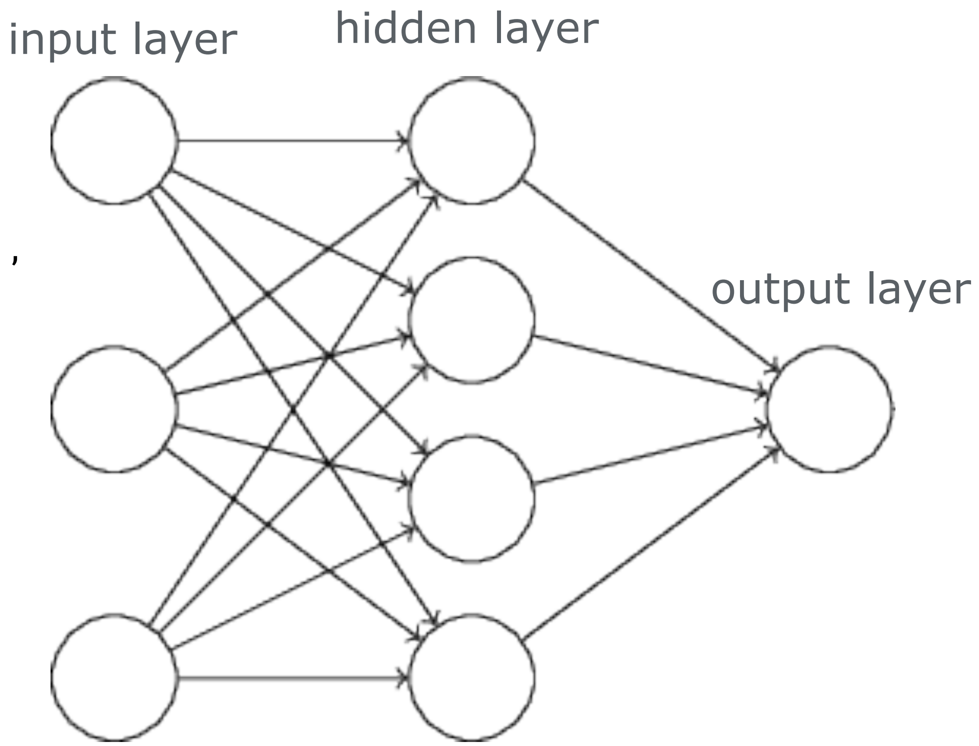


$$\text{Mean Squared Error (MSE) Loss} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

A Colab demo notebook

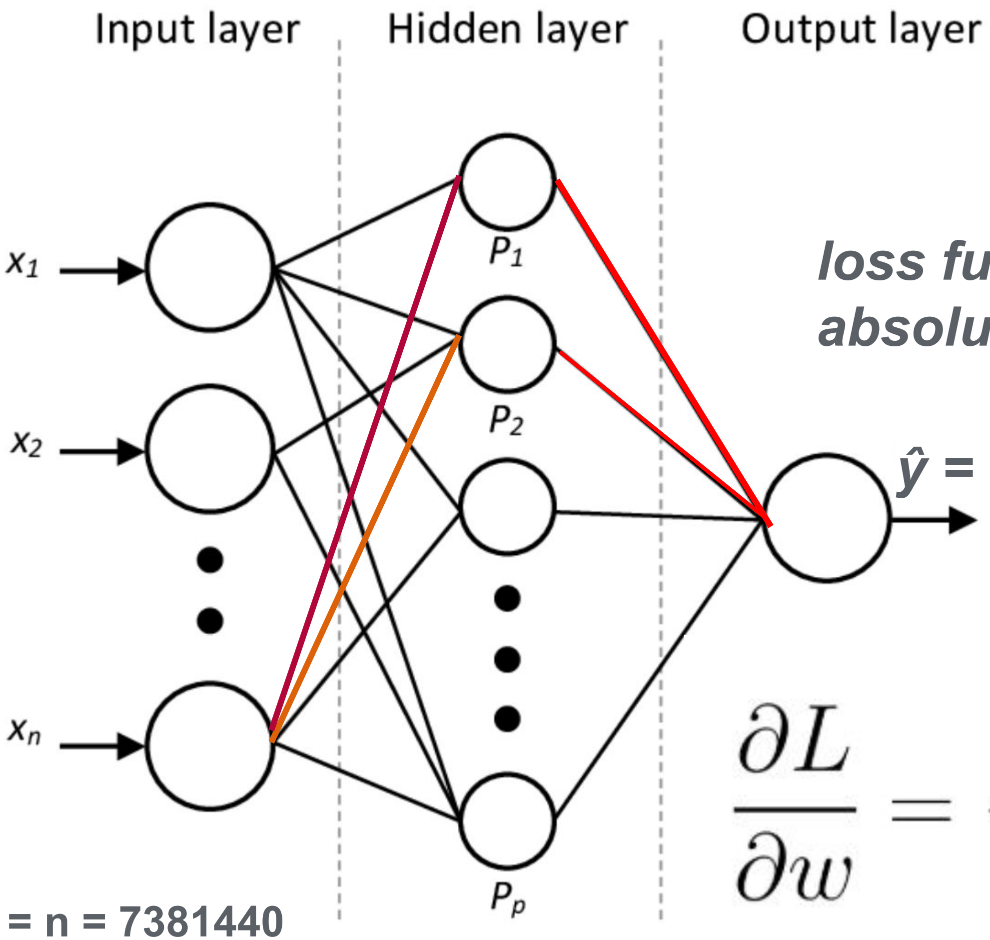
Inspecting nn.Sequential

```python
# Connecting the layers,
# notice that 'hidden layer' is implicit

model = nn.Sequential(
          nn.Linear(in_features=3, out_features=4, bias = False),
          nn.ReLU(),
          nn.Linear(in_features=4, out_features=1, bias = False)
        )

# Forward pass
x = torch.randn(5, 3) # Batch of 5 samples, 3 features each
output = model(x)


# Batch of 5 outputs, each one having one regression target value
print(output.shape) # prints torch.Size([5, 1])
```



Fully connected networks (aka multilayer perceptrons)

y = 83999 EUR



loss function =
absolute error = |ŷ - y| = 12188

$\hat{y} = f(x_1, \ldots, x_n) = 71811$

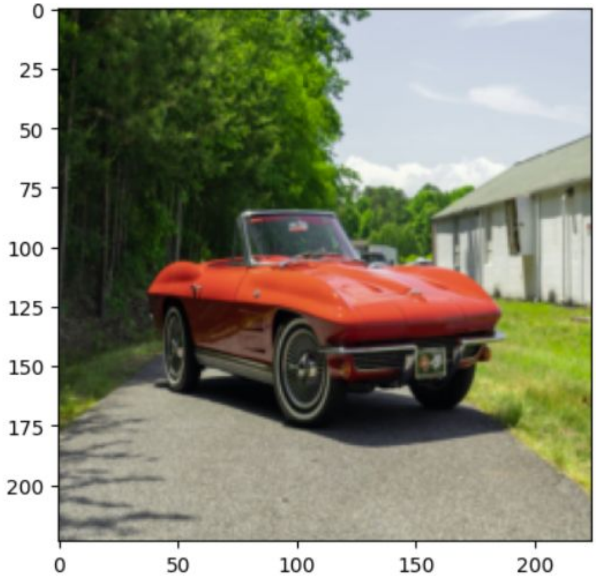
$$\frac{\partial L}{\partial w} = -\,\mathrm{sign}(y - f(x))\frac{\partial f(x)}{\partial w}$$

```
H = 1920
W = 1282
C = 3 (RGB)
```

C * H * W = n = 7381440

Resizing the input to reduce the number of parameters

```
transforms.Compose([
 transforms.ToImage(),
 transforms.Resize((224, 224)),
transforms.ToDtype(torch.float32,scale=True)
])
```



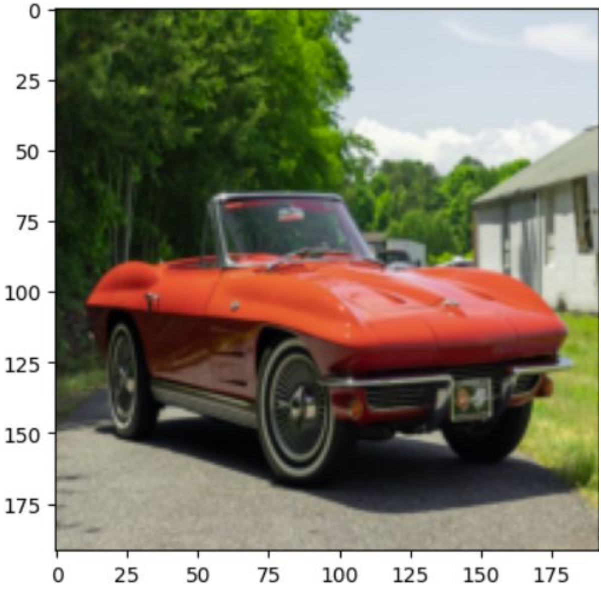
H = 1920
W = 1282
C = 3 (RGB)

H = 224
W = 224
C = 3 (RGB)

CenterCrop to capture more of the object of interest

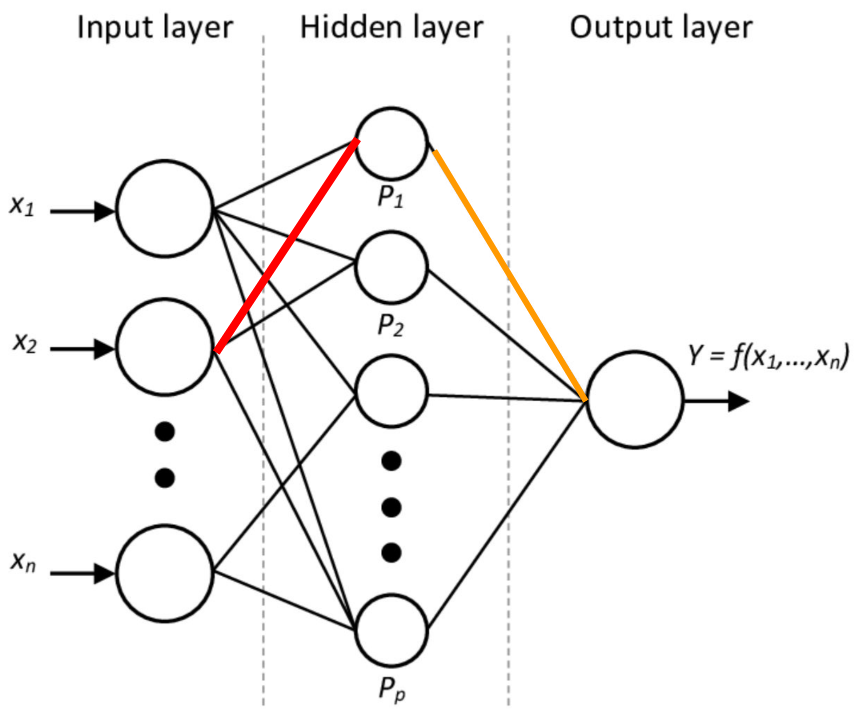
```
H = 1920
W = 1282
C = 3 (RGB)
```

```
transforms.Compose([
 transforms.ToImage(),
 transforms.Resize((256, 256)),
 transforms.CenterCrop(192),
transforms.ToDtype(torch.float32,scale=True)
])
```




```
H = 192
W = 192
C = 3 (RGB)
```

Stochastic gradient descent - the weight update rule



w_{ij} = weight from neuron i to neuron j

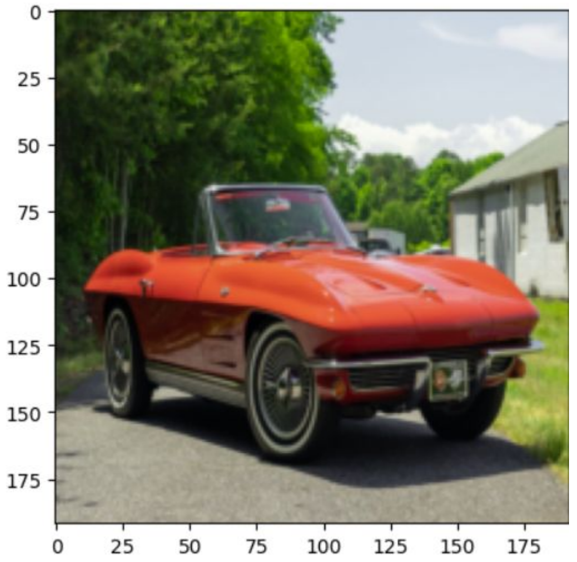
learning_rate = step size for updates, a constant like 0.01

L = loss function

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Implementing the network

```python
# Number of units on the hidden layer, this number is arbitrary

p = 100

# We use nn.Sequential to concatenate PyTorch modules in order

multilayer_perceptron=nn.Sequential(

                      nn.Flatten(), # Flattens the images within a batch

                      nn.Linear(in_features= C * H * W, # This matches the image shape

                                out_features= p,

                                bias=True),

                      nn.ReLU(),

                      nn.Linear(in_features=p,

                                out_features=1, # We are predicting a single value

                                bias=True),

)
```



H = 192
W = 192
C = 3 (RGB)

y = 83999 EUR

Implementing the weight update rule

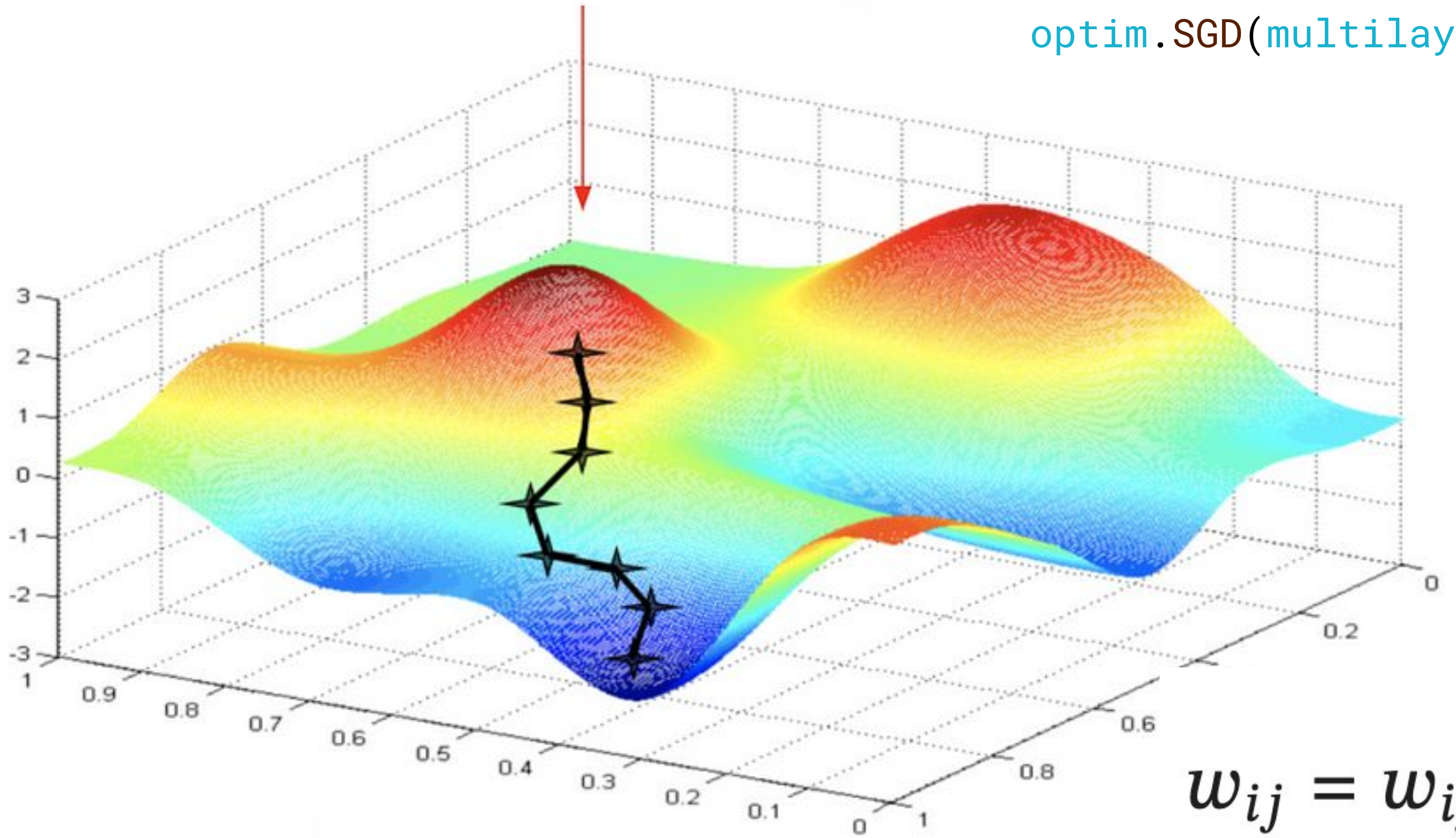
```python
def update_rule(weight, grad, learning_rate):
    return nn.Parameter((weight - learning_rate * grad))

learning_rate = 0.1
new_weights_0 = update_rule(multilayer_perceptron[0].weight,
                            multilayer_perceptron[0].weight.grad,
                            learning_rate)

new_weights_2 = update_rule(multilayer_perceptron[2].weight,
                            multilayer_perceptron[2].weight.grad,
                            learning_rate)

# We reassign the weights to our model
multilayer_perceptron[0].weight = new_weights_0
multilayer_perceptron[2].weight = new_weights_2
```

Image from ideogram.ai

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Training with an optimizer

```python
import torch.optim as optim

optim.SGD(multilayer_perceptron.parameters(), lr=1e-2)
```



$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Training with an optimizer

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

# Define an Stochastic Gradient Descent optimizer
optimizer = optim.SGD(multilayer_perceptron.parameters(), lr=1e-2)

# Define a loss function
criterion = nn.L1Loss()

# Forward pass
pred = multilayer_perceptron(x)

# Compute loss
loss = criterion(pred, y)

# Backprop
optimizer.zero_grad()
loss.backward()

# Update weights
optimizer.step()
```

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

The importance of the learning rate

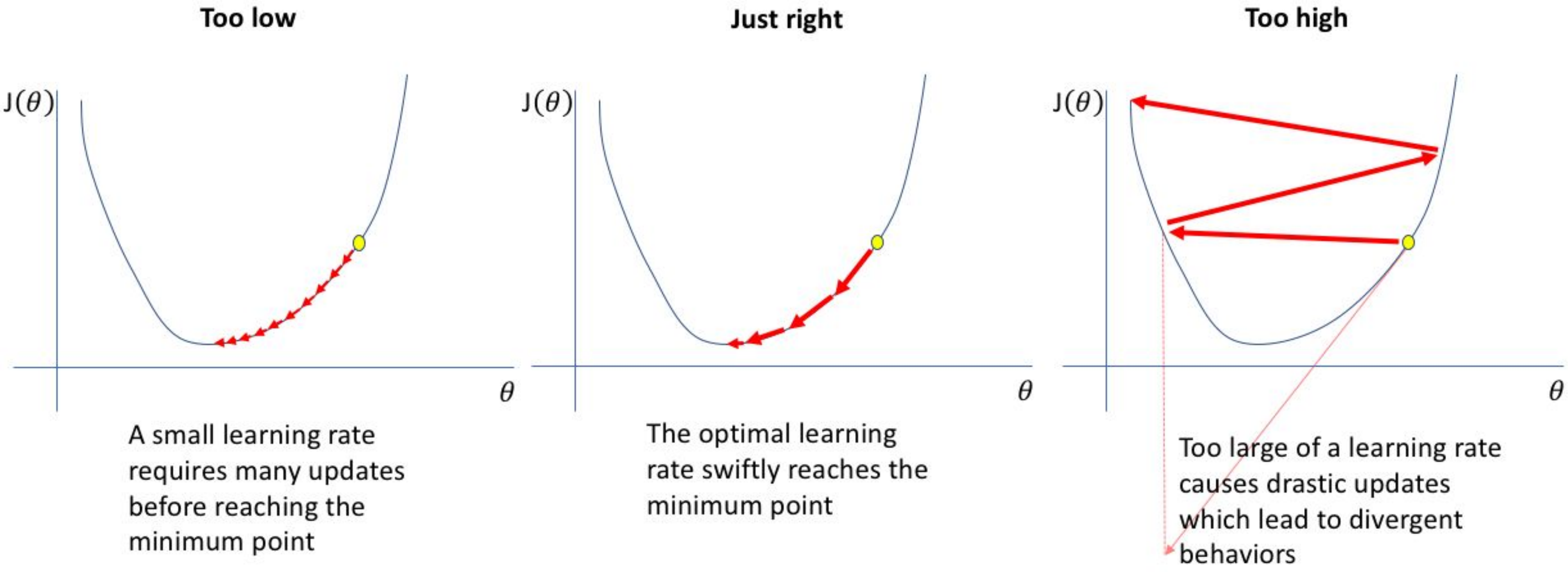


Image from Deep Learning for Coders

Summary

PyTorch building blocks

- Multilayer perceptrons in PyTorch combine `Linear` layers, activation functions, and loss functions. The framework handles gradient computation through autograd.
- We can use the `Sequential` syntax to wrap up components, this is an alternative to subclassing `nn.Module`

Input processing

- Image input requires preprocessing through resizing and normalization to create manageable feature tensors. The input size determines the network architecture.

Model training

- Training combines forward passes, loss computation, backpropagation, and weight updates. The learning rate controls optimization stability and convergence speed.

SPONSORED BY THE
Federal Ministry
of Education
and Research

Further reading

What is torch.nn really?

- https://pytorch.org/tutorials/beginner/nn_tutorial.html

Tensorflow playground

- https://playground.tensorflow.org/

SPONSORED BY THE
Federal Ministry
of Education
and Research