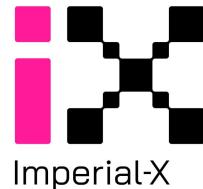


# Deep Learning in Python

Lecture 1: PyTorch and Deep Neural Networks

Dr Andreas C. S. Jørgensen

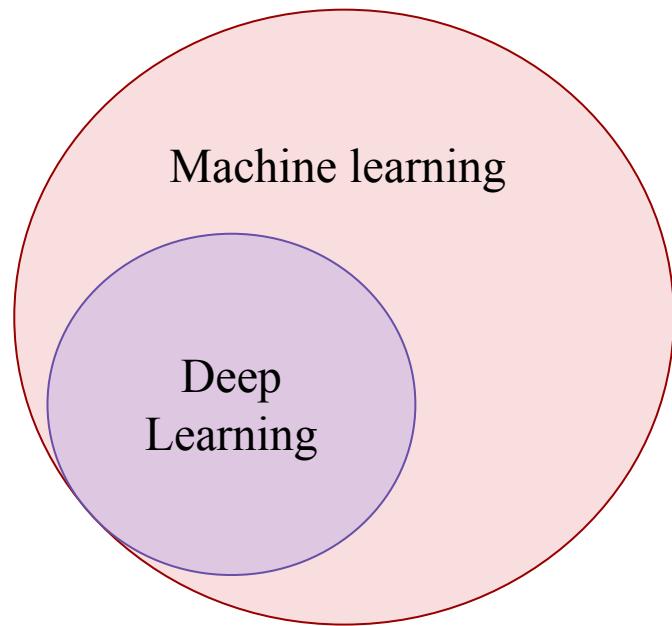
Imperial College  
London



# What is Deep Learning?

Machine-learning algorithms derive patterns from data and generalise these to perform predictions.

Deep Learning is a machine-learning method that draws on **artificial neural networks** with multiple layers.



# Outline

**Hands-on course** in Deep Learning using Python (PyTorch) with focus on computer vision.

<b>Lecture 1</b>	Deep Learning concepts and terminology. PyTorch commands, syntax and libraries (torch, torchvision). Building deep neural networks.
<b>Lecture 2</b>	Convolutional Neural Networks (CNN). Building CNN in PyTorch. Using GPUs with PyTorch.
<b>Lecture 3</b>	Optimisation: How to train and test neural networks robustly. Regularisation and hyper-parameter tuning.

# Outline



1) Introduce concepts  
and aims

2) Become familiar  
with toolkit (PyTorch)

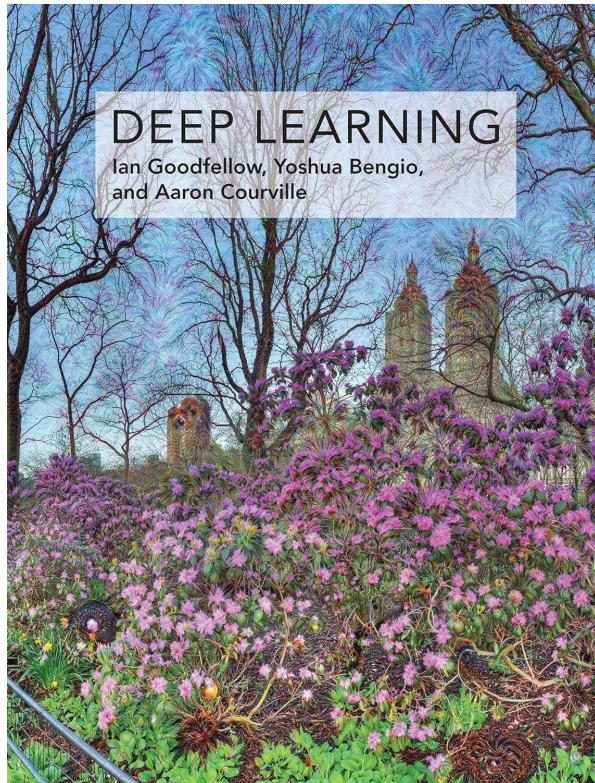
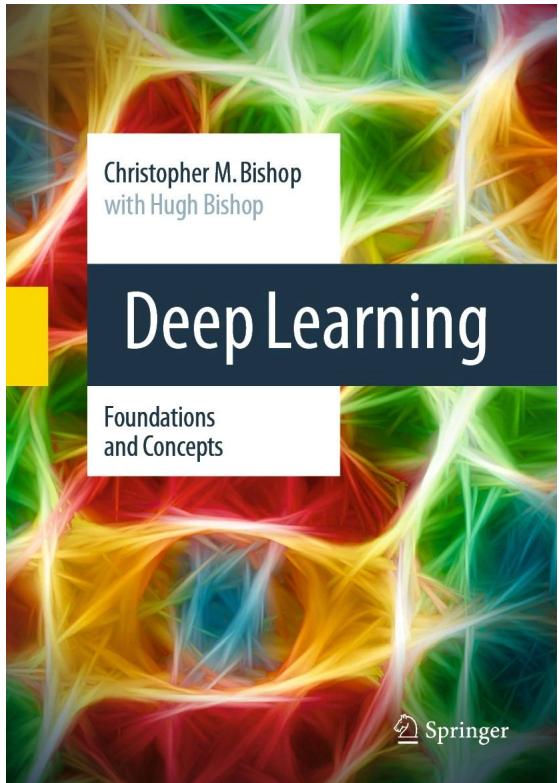
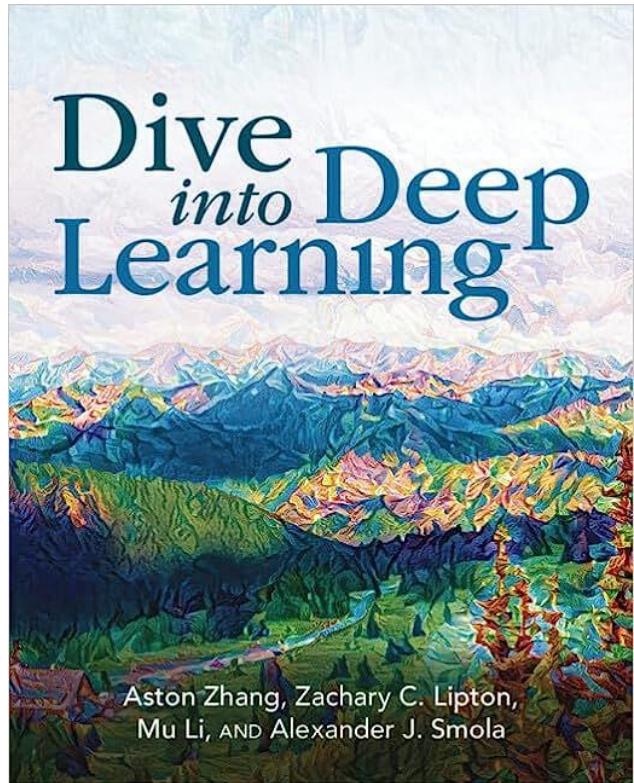
3) Combine 1 and 2 to  
build neural networks

# Preliminaries

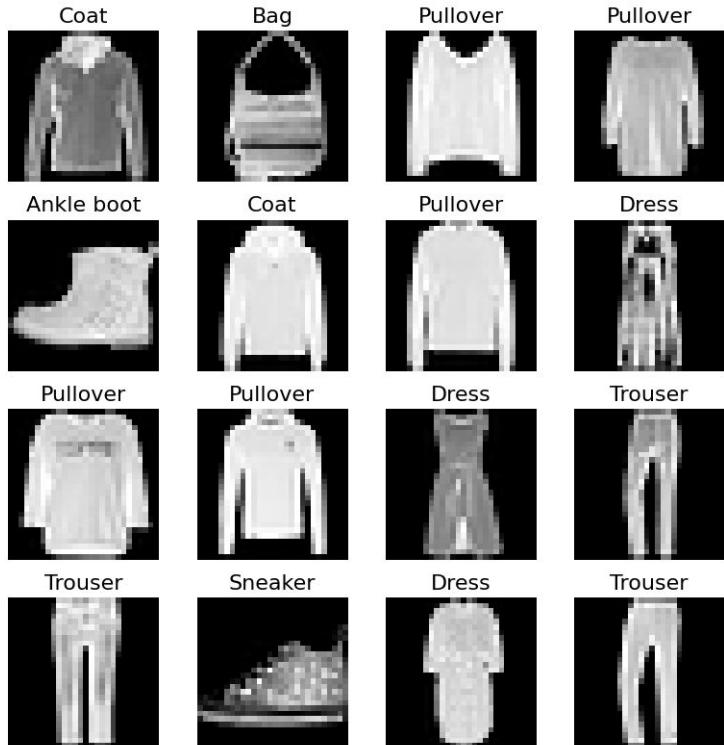
You should be familiar with

- Python3 programming (numpy, matplotlib).
- Jupyter notebooks (we run code on Google Colab).
- Basic machine-learning and statistics terminology.

# External resources

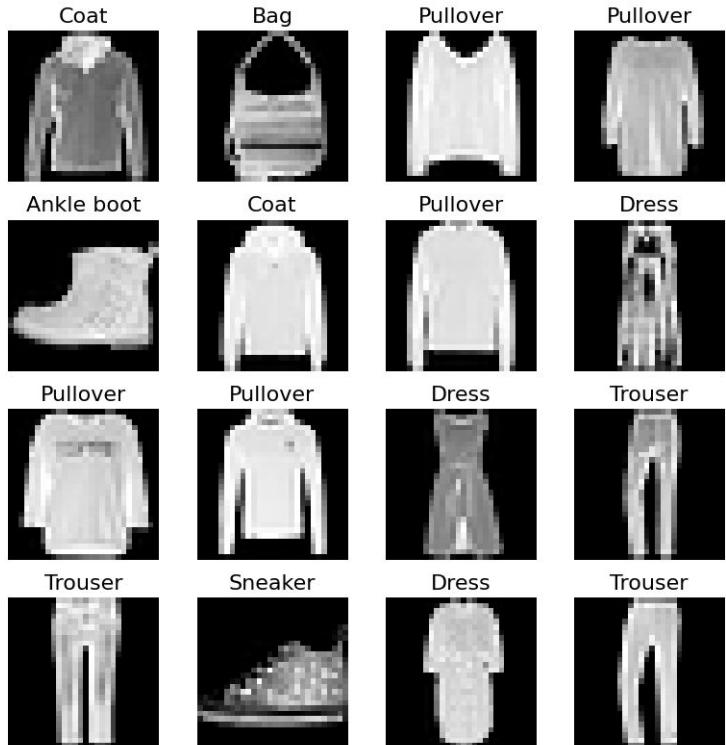


# Classification, regression

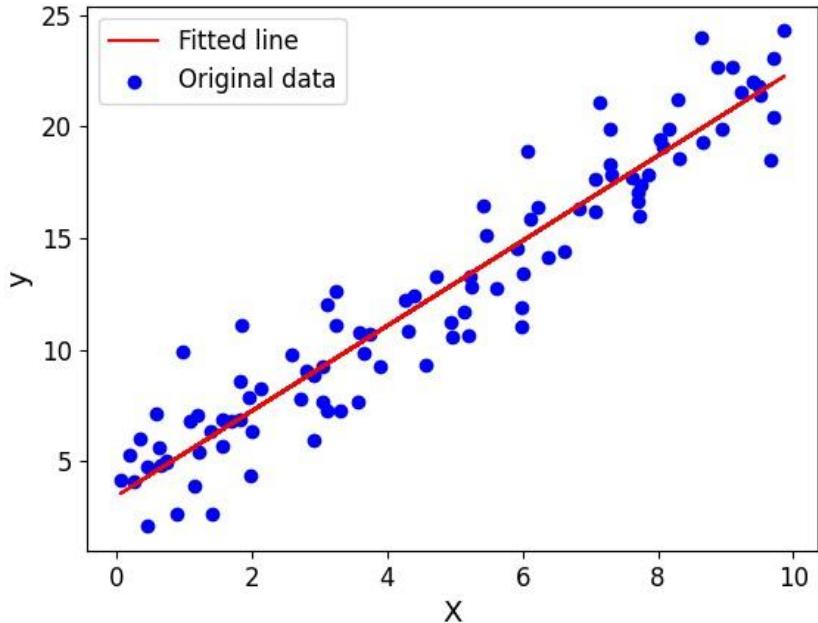


**Classification:** Assigning categorical labels

# Classification, regression

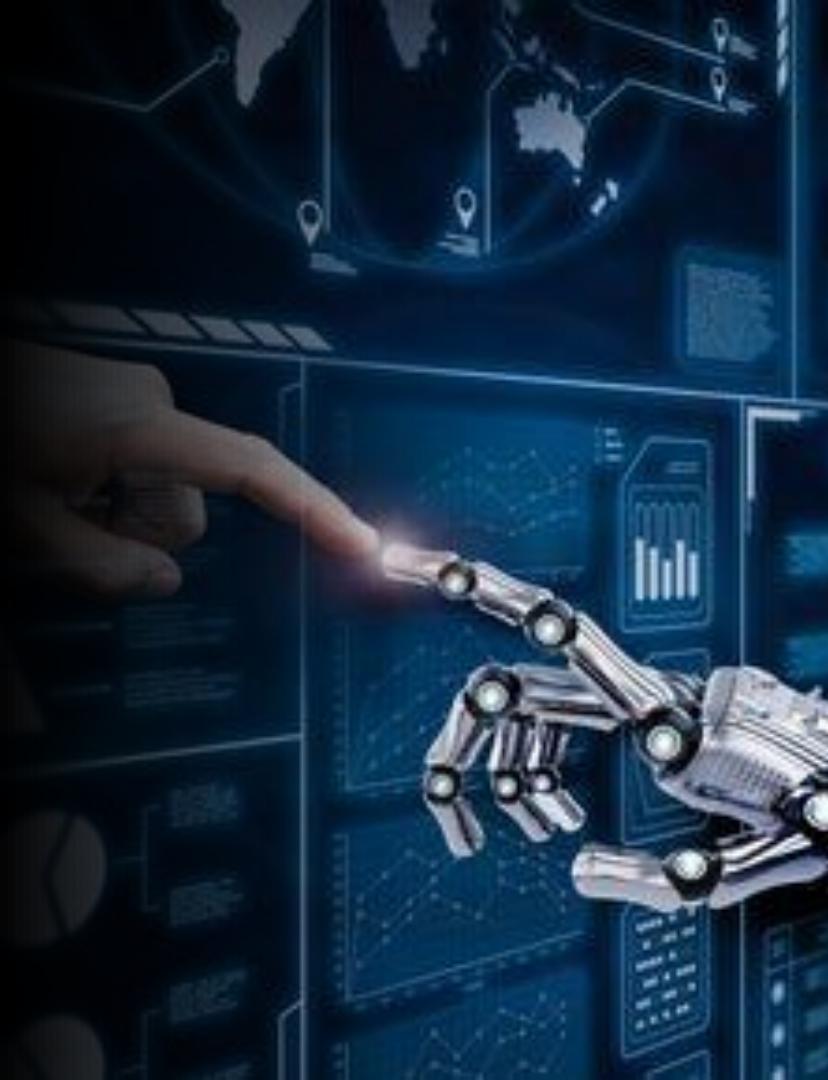


**Classification:** Assigning categorical labels



**Regression:** Predicting continuous outcomes

# Deep Neural Networks



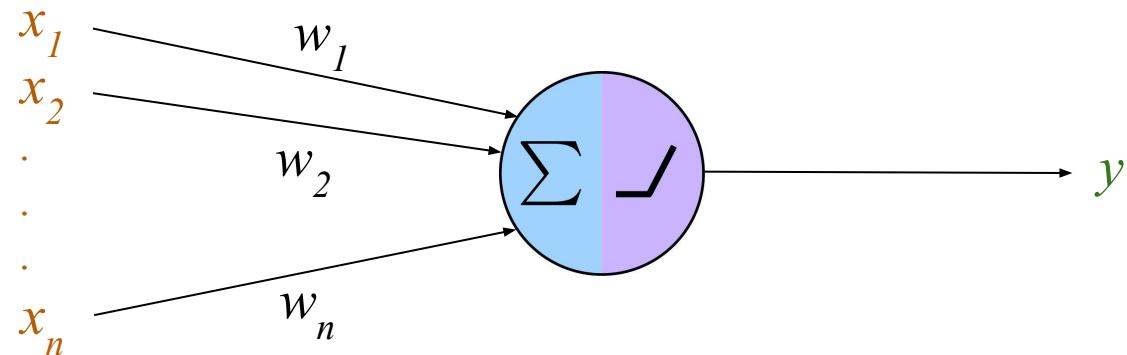
# Perceptron



*Input (features)*

*Perceptron (neuron)*

*Output (label)*



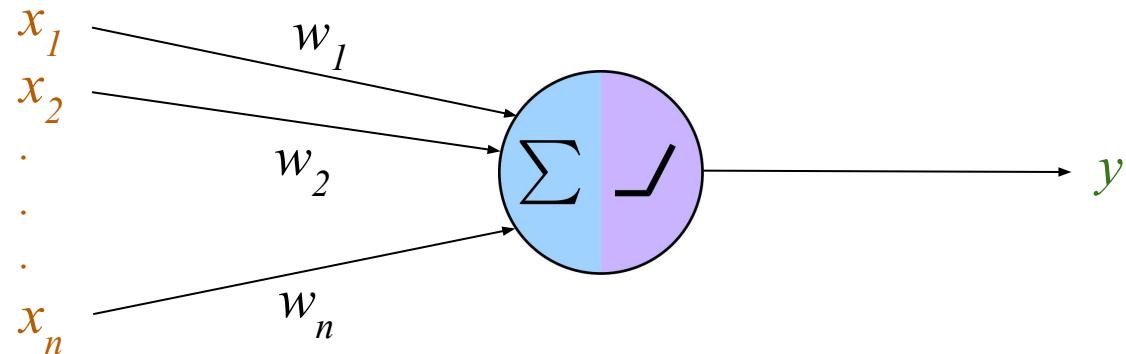
# Perceptron



*Input (features)*

*Perceptron (neuron)*

*Output (label)*



Pre-activation with weights ( $w$ ) and bias ( $b$ )

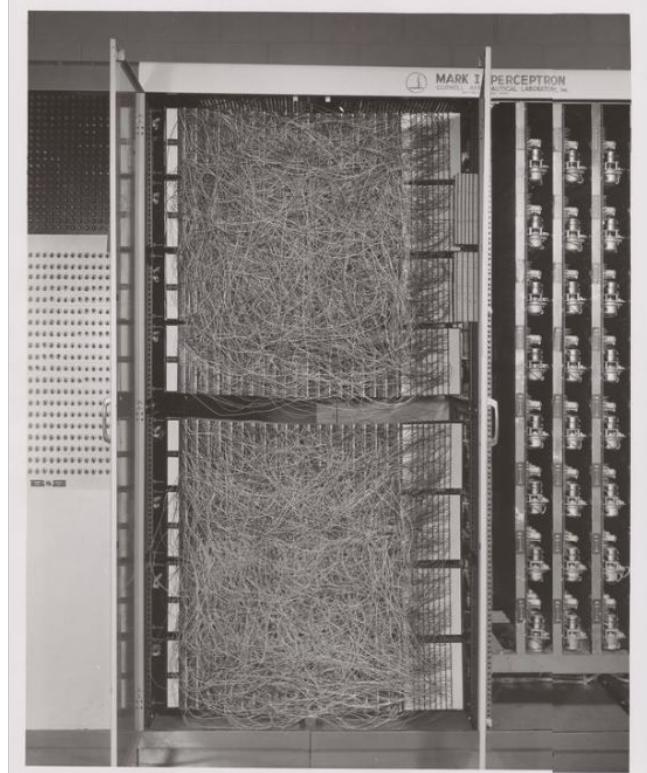
$$a = \sum_{i=1}^n w_i x_i + b$$

# Mark I Perceptron



## Why “weights” and “biases”?

In electrical engineering, you view a function as an electrical circuit with wires and biases. That’s where “ $w$ ” and “ $b$ ” stem from.



*Cf. Cornell University Library*

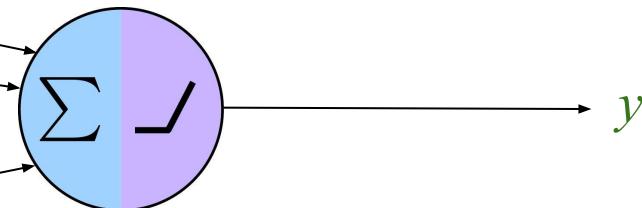
# Perceptron



*Input (features)*

$$\begin{matrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{matrix} \quad \begin{matrix} w_1 \\ w_2 \\ w_3 \\ \cdot \\ \cdot \\ w_n \end{matrix}$$

*Perceptron (neuron)*



*Output (label)*

Pre-activation with weights ( $w$ ) and bias ( $b$ )

$$a = \sum_{i=1}^n w_i x_i + b$$

Post-activation, activation function ( $f$ )

$$y = f(a)$$

# Activation functions



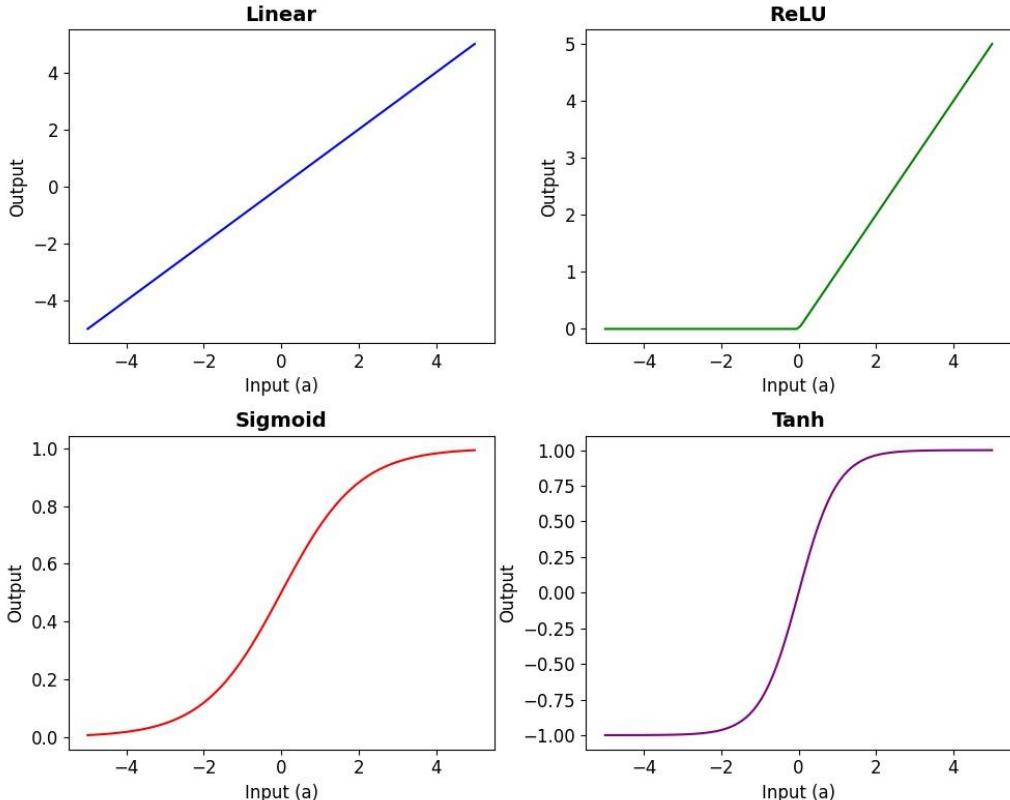
Activation functions may introduce non-linearity and determine neuron outputs.

**Linear:** Returns input.

**ReLU (Rectified Linear Unit):** Returns zero for negative inputs and the input value for positive inputs.

**Sigmoid and Tanh (Hyperbolic Tangent):** Smoothen outputs. Suffer from vanishing gradients for extreme inputs.

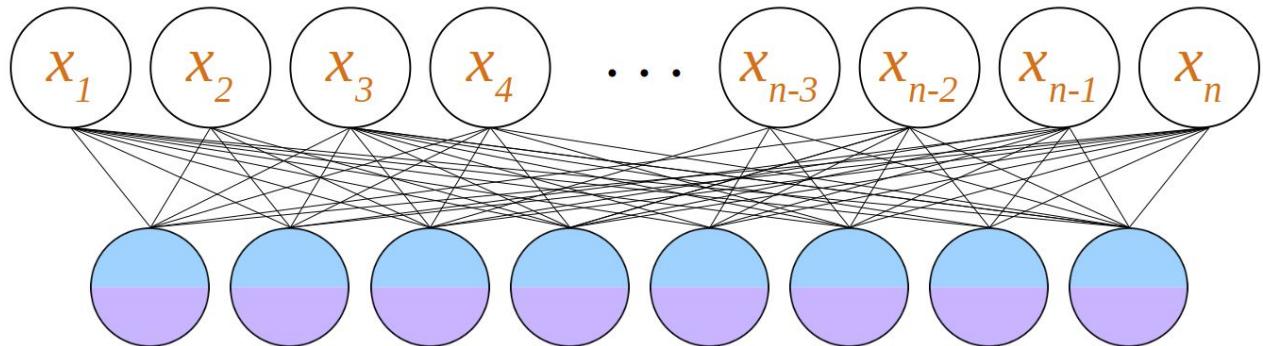
**Softmax:** Converts raw scores into probabilities (used for classification output).



# Shallow neural networks



Input layer

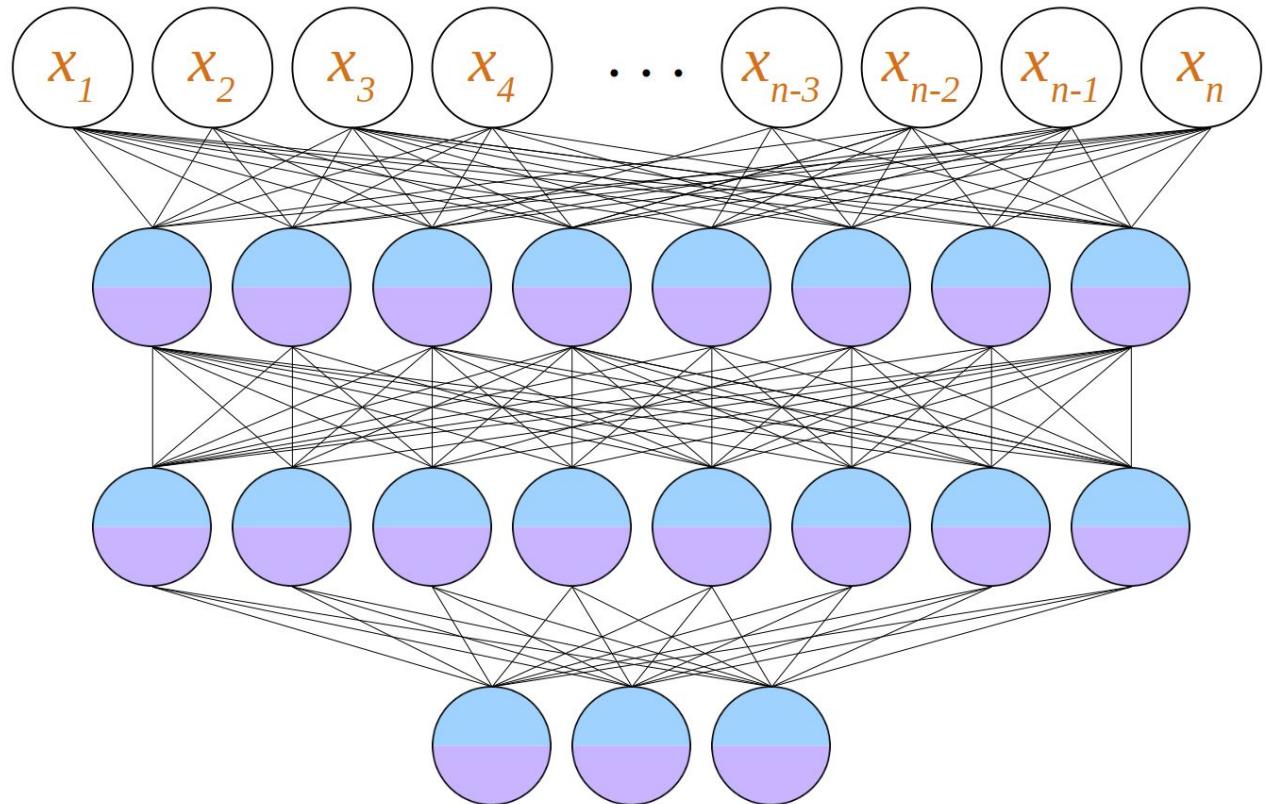


Output layer

# Deep neural networks



Input layer



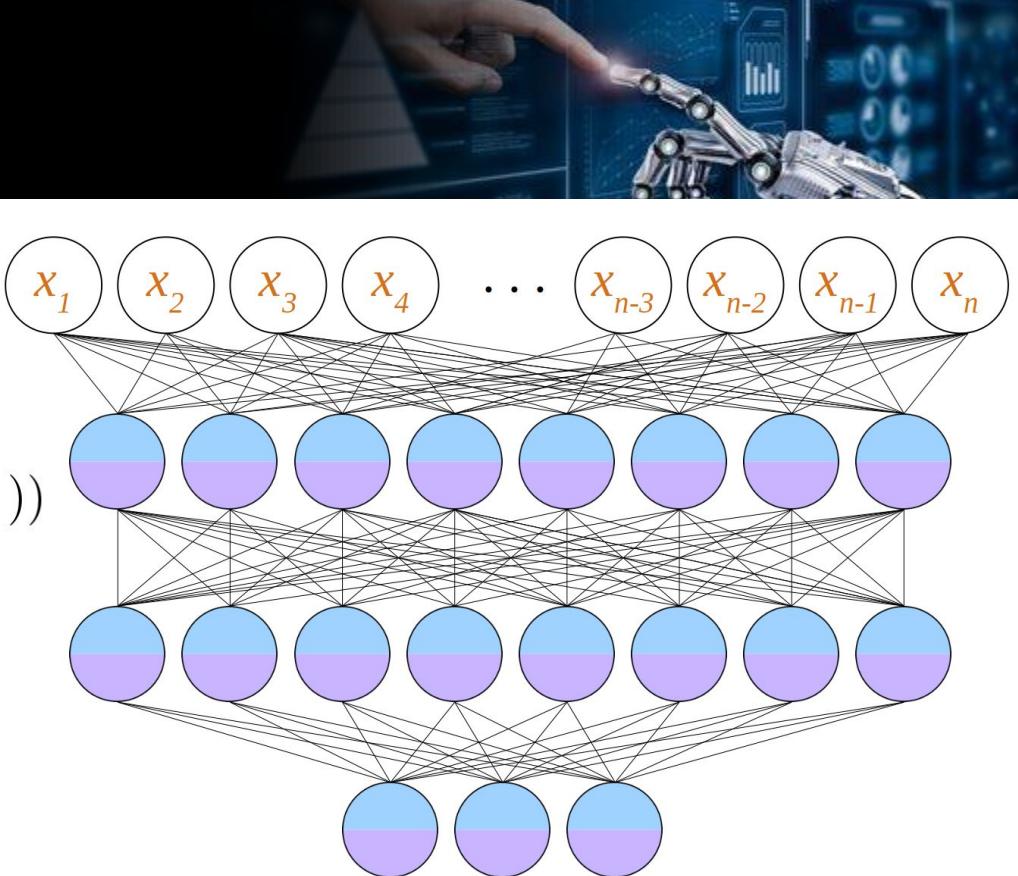
Hidden layers

Output layer

# Architecture

A deep neural network transforms  $\mathbf{x}$  stepwise, layer by layer:

$$\Phi(\mathbf{x}) = \phi_K(\phi_{K-1}(\dots \phi_2(\phi_1(\mathbf{x})) \dots))$$

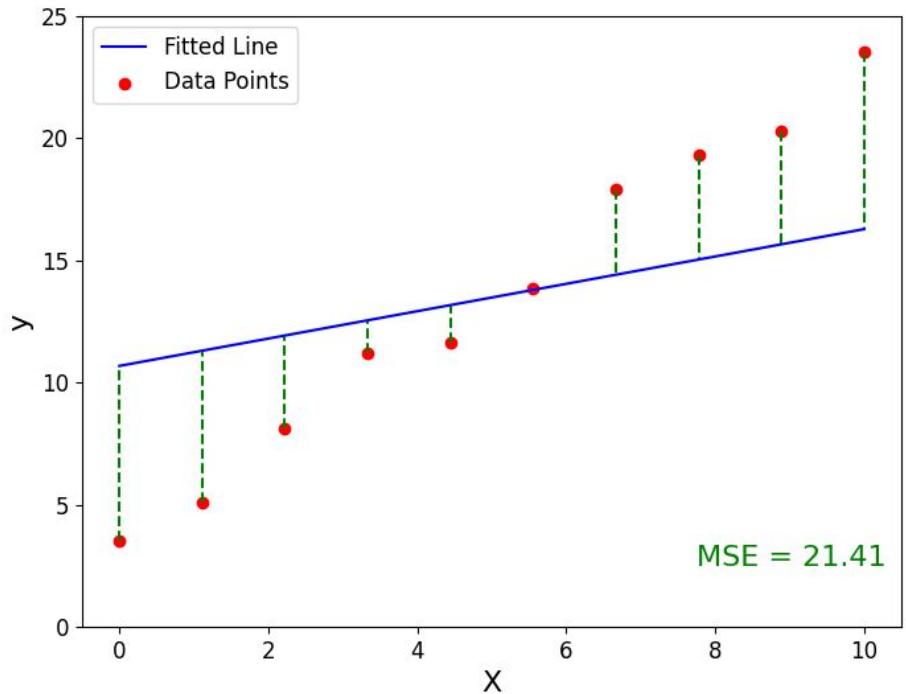


# Training



How does one choose the weights and biases?

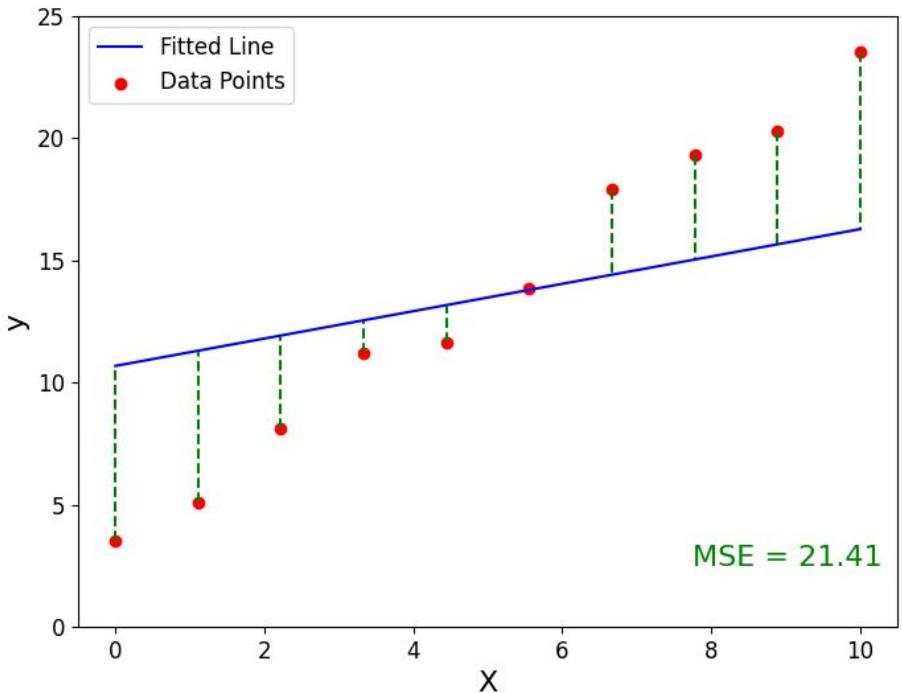
# Loss functions



For regression, we use the mean squared error (MSE)

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

# Loss functions



For regression, we use the mean squared error (MSE)

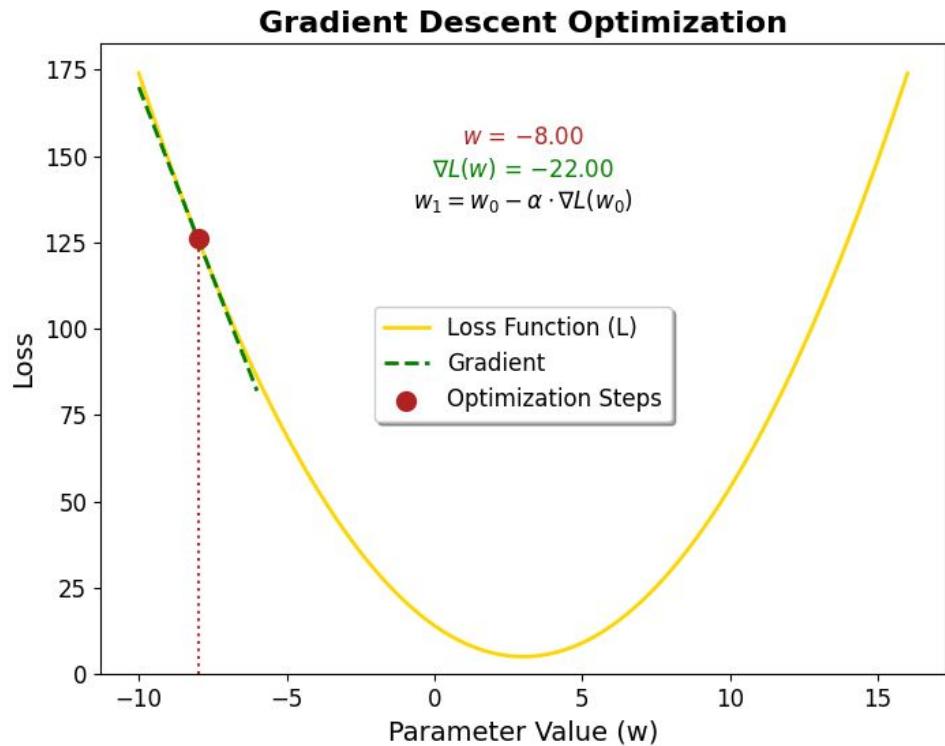
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

For classification, we use cross entropy, quantifying the dissimilarity between the distributions over predicted labels ( $q$ ) and true labels ( $p$ )

$$H(p, q) = - \sum_{i=1}^n p_i \log q_i$$

# Optimisation

We want to **minimise the loss**, i.e.,  
match observations as closely as  
possible.

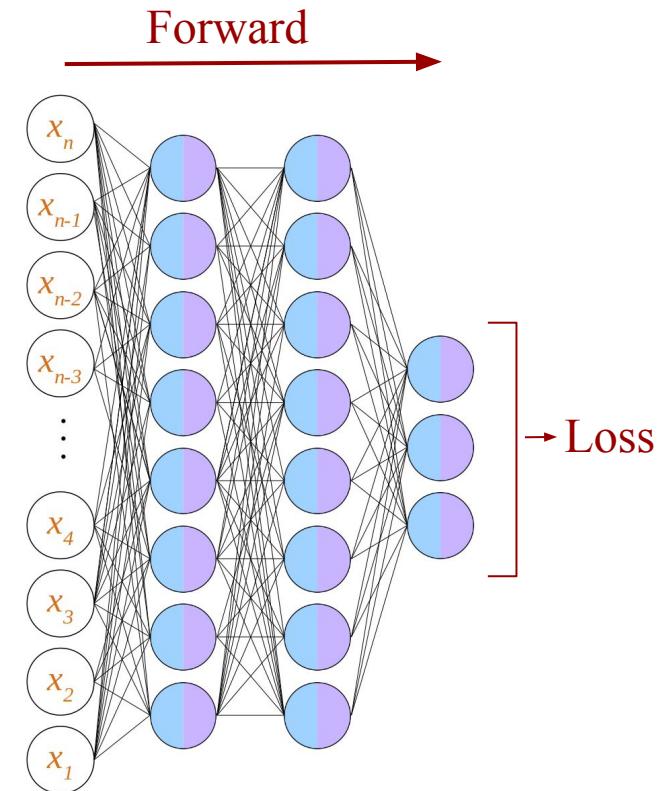


# Backpropagation

We want to compute the gradient of the loss to optimise the parameters of the neural network.

$$\frac{\partial L_i^k}{\partial w_{pq}^k}$$

The loss is a function final output of the neural network and propagate the errors in the loss “backwards” through the network.



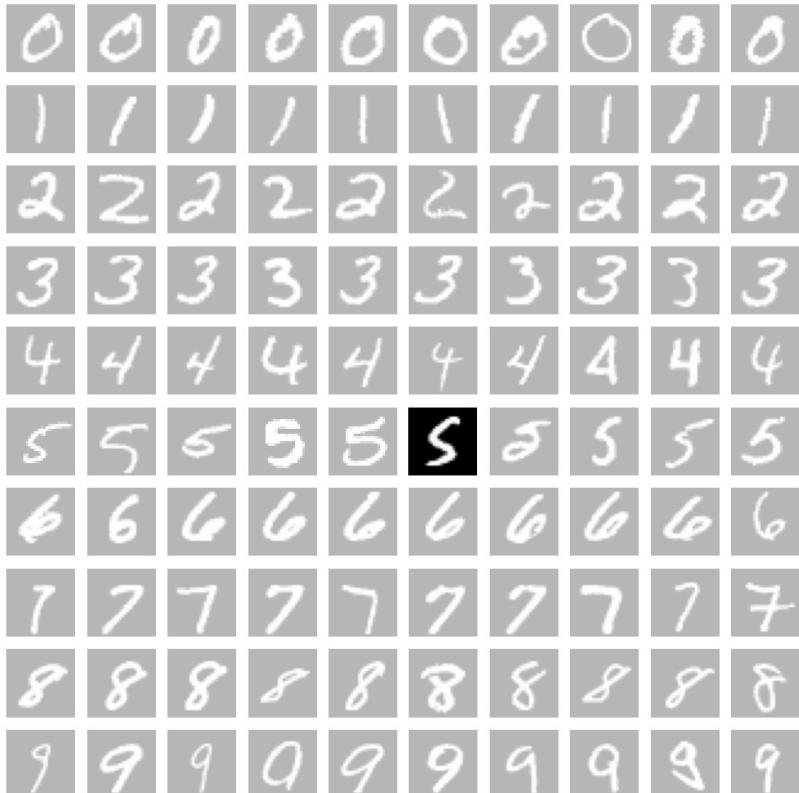
# Batches and SGD



0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9

We could compute gradients based on the whole data set. But that's computationally expensive.

# Batches and SGD



~~We could compute gradients based on the whole data set. But that's computationally expensive.~~

We could use a single, randomly chosen (stochastic) sample to compute the gradients. But that will be a not be representative.

# Batches and SGD



5 | 1 0 9 3 5 7 5 0 6

8 9 5 2 4 4 8 3 2 0

1 6 2 5 1 4 8 7 5 3

We could compute gradients based on the whole data set. But that's computationally expensive.

We could use a single, randomly chosen (stochastic) sample to compute the gradients. But that will be not be representative.

**Mini-batches:** We could randomly shuffle the data and compute the gradients based on a small number (batch) of samples. That's fast and representative!

# Training loop



- 1) Initialise weights ( $\mathbf{W}$ ) and biases ( $\mathbf{b}$ )
- 2) In each epoch, loop over all mini-batches. For each mini-batch,
  - a) Feedforward: Use current  $\mathbf{W}$  and  $\mathbf{b}$  to make model predictions.
  - b) Compute the loss,  $L$ .
  - c) Backpropagation: Compute the gradient of the loss w.r.t.  $\mathbf{W}$  and  $\mathbf{b}$  layer by layer.
  - d) Update  $\mathbf{W}$  and biases  $\mathbf{b}$  for each layer.

# Data augmentation

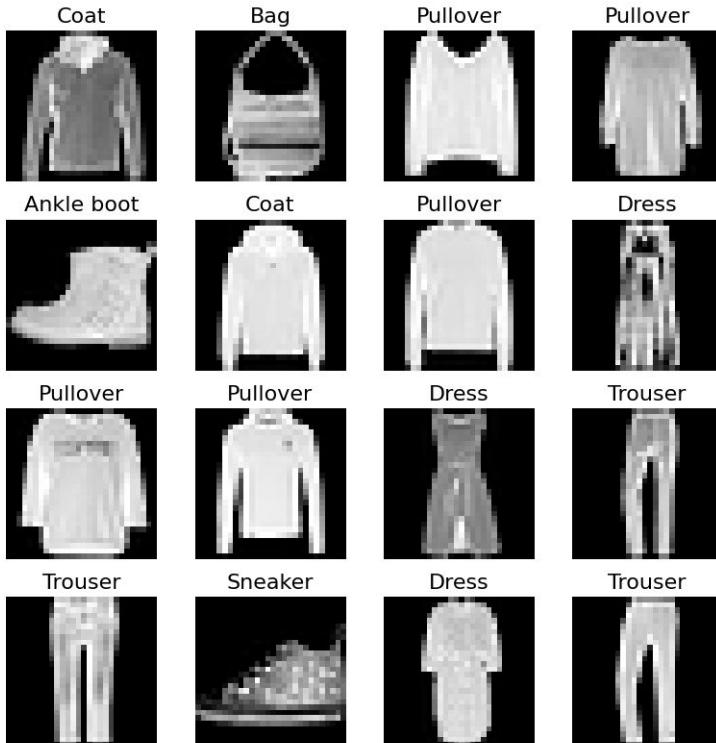
Data augmentation: Techniques to artificially increase dataset size. Mitigates overfitting, improves model generalization.

Common data augmentation techniques:

- Horizontal/Vertical Flipping
- Random Rotation
- Random Crop
- Random Resizing and Padding
- Color Jittering
- Gaussian Blur
- Random Noise Addition



# Testing



How well does your model capture your training data?

Confront your trained network with ***unseen*** data with known labels. How does the model perform? Can it **generalise**?

You need to be careful when choosing the metrics!

Only if you pass these tests, apply to unlabelled data.



# Step by step guide



- 1) Split test and training data. Perform data augmentation.
- 2) Taylor the neural network architecture (layers, activation functions...)
- 3) Choose a meaningful loss function.
- 4) Choose the optimisation algorithm, batch size, learning rate ...
- 5) Train your network
- 6) Test performance

# PyTorch



# Basics

```
import torch # PyTorch  
  
scalar = torch.tensor(1) # Define scalar  
  
vector = torch.tensor([1, 2]) # Define vector
```

PyTorch has its own objects (tensors) with their own methods.



# Basics

```
import torch # PyTorch  
  
scalar = torch.tensor(1) # Define scalar  
  
vector = torch.tensor([1, 2]) # Define vector  
  
vector*vector # Example of operation  
  
vector.argmax() # Example of method
```

PyTorch has its own objects (tensors) with their own methods.



# Basics

```
import torch # PyTorch

scalar = torch.tensor(1) # Define scalar

vector = torch.tensor([1, 2]) # Define vector

vector*vector # Example of operation

vector.argmax() # Example of method

torch.unsqueeze(vector, dim=0) # Add dimensions
```

PyTorch has its own objects (tensors) with their own methods.



# Basics

```
import torch # PyTorch

scalar = torch.tensor(1) # Define scalar

vector = torch.tensor([1, 2]) # Define vector

vector*vector # Example of operation

vector.argmax() # Example of method

torch.unsqueeze(vector, dim=0) # Add dimensions

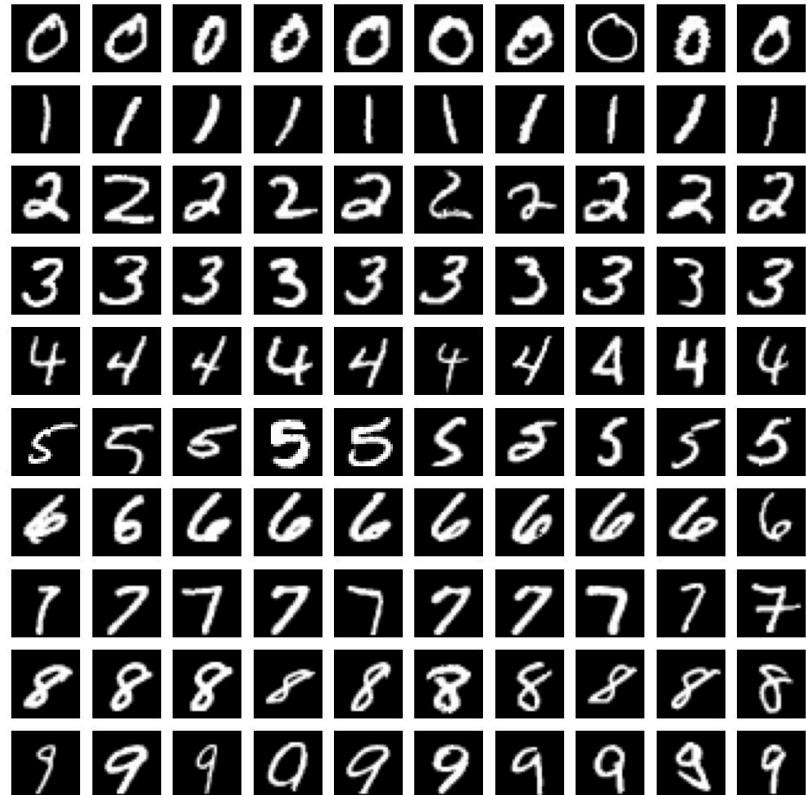
# Dealing with numpy output
array = np.arange(1,3)
torch.from_numpy(array)
```

PyTorch has its own objects (tensors) with their own methods.



# Computer vision

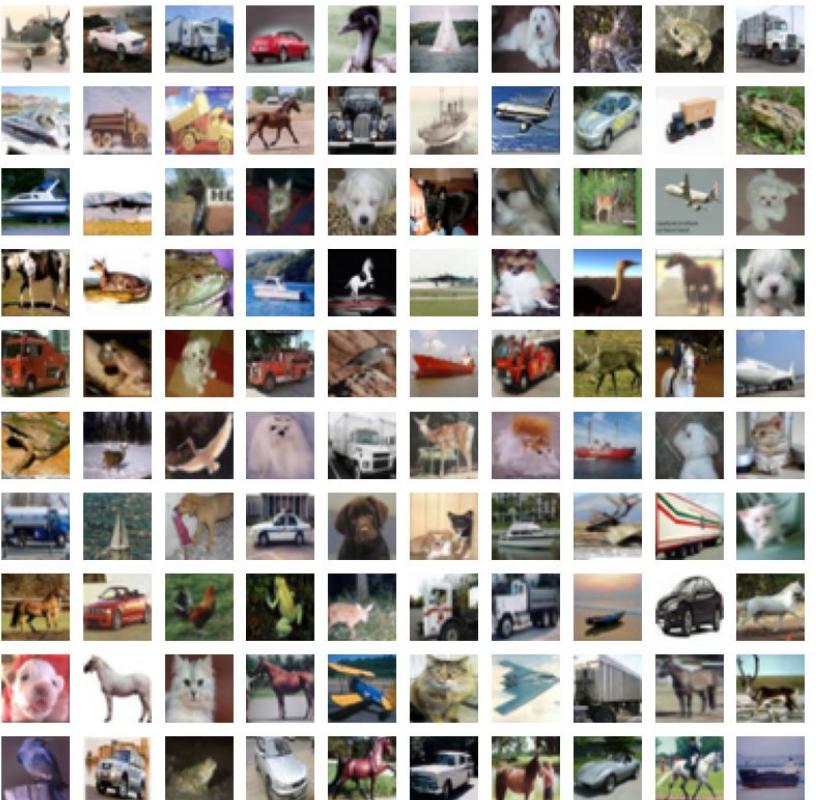
```
from torchvision import datasets  
  
train = datasets.MNIST(root='./data', train=True, download=True)
```



# Computer vision

```
from torchvision import datasets
```

```
train = datasets.CIFAR10(root='./', train=True, download=True)
```



# Transforms



```
import torchvision.transforms as transforms
```

```
means = (0.5,0.5,0.5)
```

```
stds = (0.5,0.5,0.5)
```

*# Define the data transform including normalization*

```
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
```

```
    transforms.Normalize(means, stds),
```

```
])
```

Transforms preprocess data for resizing images, augmentation, and normalization.

# Transforms



```
import torchvision.transforms as transforms
```

```
means = (0.5,0.5,0.5)
```

```
stds = (0.5,0.5,0.5)
```

```
# Define the data transform including normalization
```

```
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
```

```
    transforms.Normalize(means, stds),
```

```
])
```

# Converts the image to a PyTorch tensor

# Normalize for three channels

# Data loaders

```
from torch.utils.data import DataLoader  
  
Train_loader = DataLoader(dataset=train, batch_size=32,  
shuffle=True)
```



PyTorch DataLoader: Batches data, shuffles, and prepares data for efficient training.

# Data loaders

```
from torch.utils.data import DataLoader  
  
Train_loader = DataLoader(dataset=train, batch_size=32,  
shuffle=True)  
  
Test_loader = DataLoader(dataset=test, batch_size=32,  
shuffle=False)
```



PyTorch DataLoader: Batches data, shuffles, and prepares data for efficient training.

# Step by step



- 1) Load your training and test data

```
from torchvision.datasets import ImageFolder
```

- 2) Define the required transforms and transform the data

```
trainset = ImageFolder(root='XYZ/train', transform=train_transform)
```

- 3) Pass the dataset to the dataloader

```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
```

# Building Neural Networks



# A new class

```
import torch
import torch.nn as nn

# Define a neural network
class MyNet(nn.Module):
    def __init__(self):
        super().__init__()

    # Define layers and activations functions

    def forward(self, x):

        # Transform x stepwise, layer by layer:

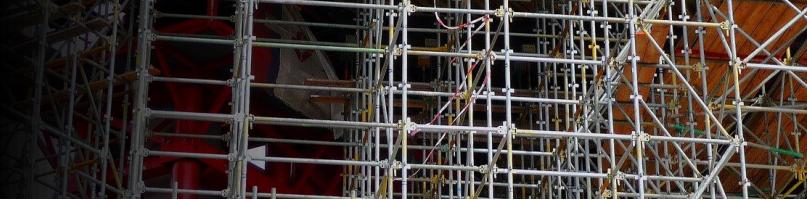
        return x
```



We define a PyTorch neural network class inheriting from `nn.Module`, with initialization for layers and a forward pass method.

$$\Phi(\mathbf{x}) = \phi_K(\phi_{K-1}(\dots \phi_2(\phi_1(\mathbf{x})) \dots))$$

# Linear regression



```
import torch
import torch.nn as nn

# Define a neural network
class MyNet(nn.Module):
    def __init__(self):
        super().__init__()

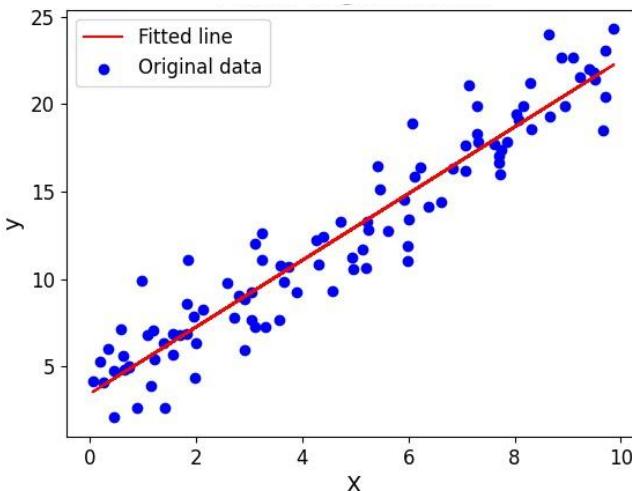
        # Define layers and activations functions

    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

Linear regression can be viewed as a single-layer fully connected neural network, predicting a single numerical value ( $y$ ), i.e. with one output neuron.



# Linear regression



```
import torch
import torch.nn as nn

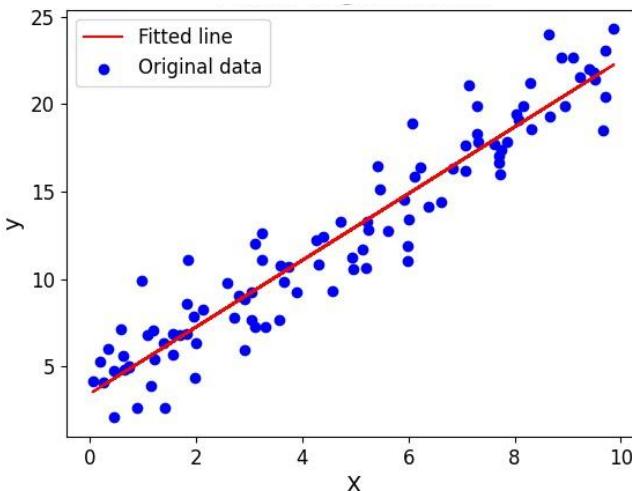
# Define a neural network
class MyNet(nn.Module):
    def __init__(self):
        super().__init__()

        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        x = self.linear(x)

    return x
```

Linear regression can be viewed as a single-layer fully connected neural network, predicting a single numerical value ( $y$ ), i.e. with one output neuron.



# Initialisation

```
# Call instance of your class
```

```
model = MyNet()
```

```
# Define your loss
```

```
criterion = nn.MSELoss()
```

```
# Define your optimisation alogorithm
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

# Initialisation

```
# Call instance of your class
```

```
model = MyNet()
```

```
# Define your loss
```

```
criterion = nn.MSELoss()
```

```
# Define your optimisation alogorithm
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
# Synthetic data
```

```
X = 2 * torch.rand(100, 1)
```

```
y = 3 * X + 2 + 0.5 * torch.randn(100, 1)
```

# Training

In each epoch (we have a single batch),

- 1) Feedforward: Make model predictions.
- 2) Compute the loss.
- 3) Backpropagation: Compute the gradient of the loss.
- 4) Update weights and biases.

# Training

In each epoch (we have a single batch),

- 1) Feedforward: Make model predictions.
- 2) Compute the loss.
- 3) Backpropagation: Compute the gradient of the loss.
- 4) Update weights and biases.



```
num_epochs = 100

for epoch in range(num_epochs):

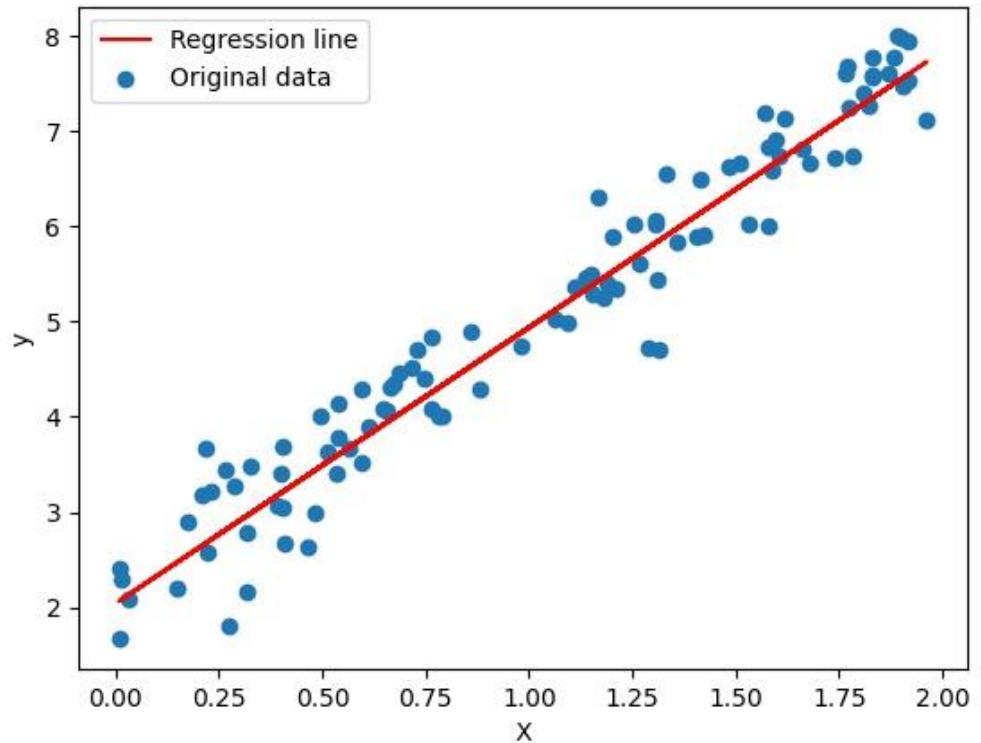
    # 1) Feedforward
    predictions = model(X)

    # 2) Compute the loss
    loss = criterion(predictions, y)

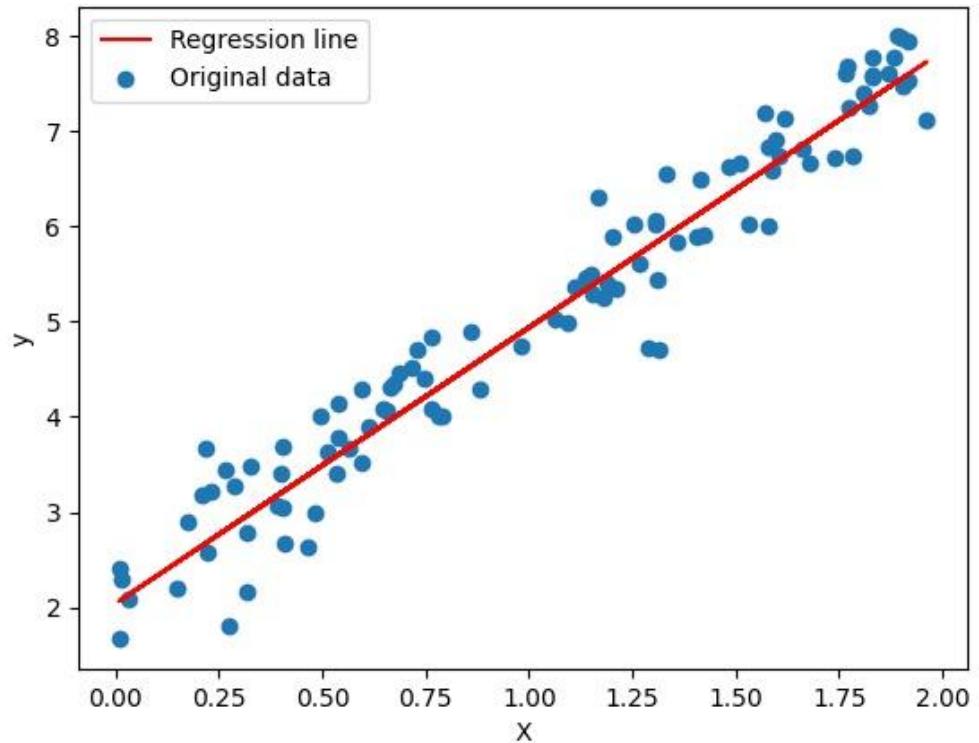
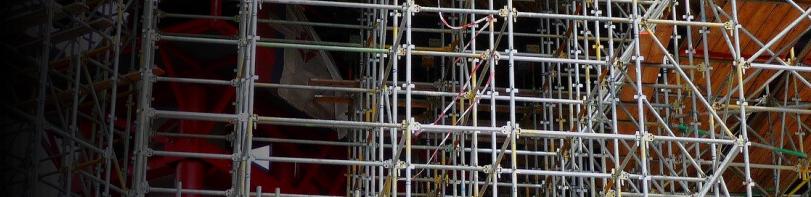
    # 3) Backpropagation of errors
    optimizer.zero_grad()
    loss.backward()

    # 4) Optimization, update W and b
    optimizer.step()
```

# Final fit



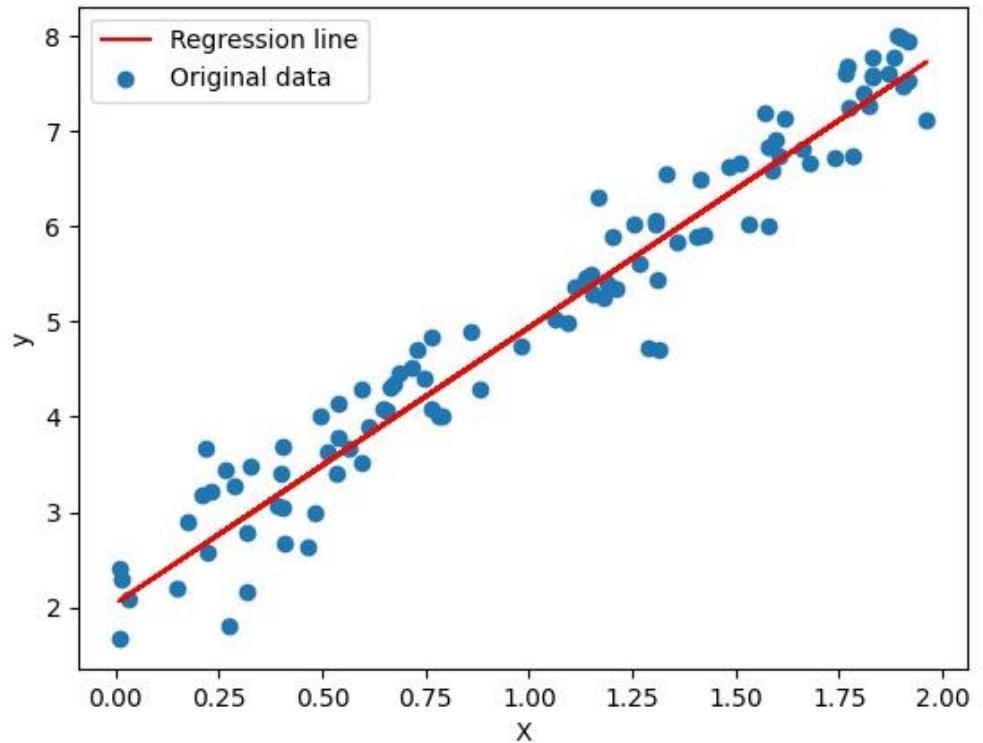
# Final fit



```
print(model.linear.weight.data, model.linear.bias.data)
```

```
tensor([[2.9014]]) tensor([2.0385])
```

# Final fit

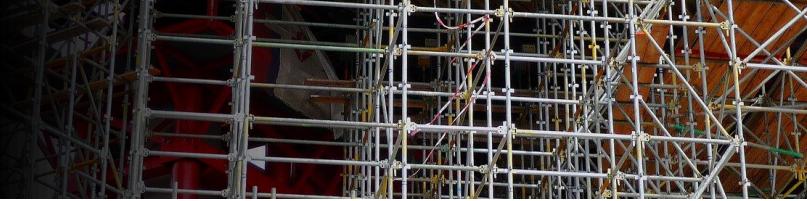


```
print(model.linear.weight.data, model.linear.bias.data)
```

```
tensor([[2.9014]]) tensor([2.0385])
```

```
y = 3 * X + 2 + 0.5 * torch.randn(100, 1)
```

# Deep neural network



```
import torch
import torch.nn as nn

# Define a neural network
class MyDeepNet(nn.Module):
    def __init__(self):
        super().__init__()

    # Define layers and activations functions

    def forward(self, x):

        # Transform x stepwise, layer by layer:

        return x
```

# Deep neural network



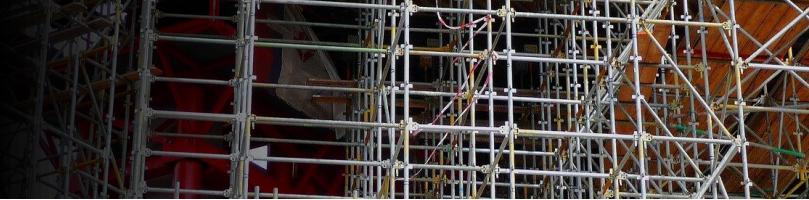
```
import torch
import torch.nn as nn

# Define a neural network
class MyDeepNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

$$\Phi(\boldsymbol{x}) = \phi_K(\phi_{K-1}(\dots \phi_2(\phi_1(\boldsymbol{x})) \dots))$$

# Deep neural network



```
import torch
import torch.nn as nn

# Define a neural network
class MyDeepNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```
# Call instance of your class
model = MyDeepNet()

# Define your loss
criterion = nn.CrossEntropyLoss()

# Define your optimisation alogorithm
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

$$\Phi(\mathbf{x}) = \phi_K(\phi_{K-1}(\dots \phi_2(\phi_1(\mathbf{x})) \dots))$$

# Training

```
# Set the model to training mode
model.train()

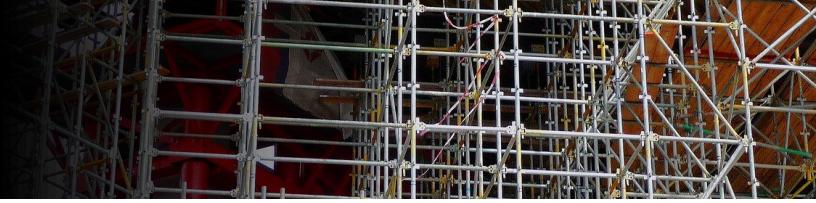
for epoch in range(num_epochs):
    for images, labels in train_loader:
        # 1) Feedforward
        outputs = model(images)

        # 2) Compute the loss
        loss = criterion(outputs, labels)

        # 3) Backpropagation of errors
        optimizer.zero_grad() # clear gradients
        loss.backward()

        # 4) Optimization, update W and b
        optimizer.step() # update weights and biases
```

The **bare bone training loop looks the same**: There are 4 steps. We now also loop over mini-batches.



# Testing

```
# Set the model to evaluation mode
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        # Determine likeliest label
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        # Is it right or wrong?
        correct += (predicted == labels).sum().item()

accuracy = correct / total
```

# Testing

```
# Set the model to evaluation mode
model.eval()
correct = 0
total = 0

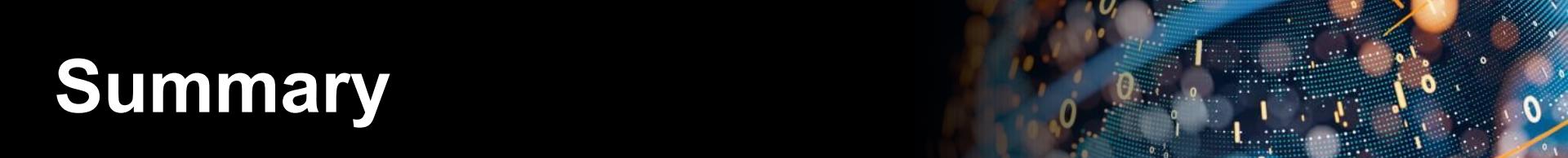
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        # Determine likeliest label
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        # Is it right or wrong?
        correct += (predicted == labels).sum().item()

accuracy = correct / total
```

There are often better choices than accuracy ...



# Summary



Based on this lecture, you should better be able to

- 1) Explain basic concepts in Deep Learning, including activation and loss functions, gradient descent, backpropagation, batching, data augmentation, and neural network architecture.
- 2) Outline the basic steps in training and testing neural networks.
- 3) Understand and point to essential commands in PyTorch (including torchvision).
- 4) Translate the concepts of Deep Learning into PyTorch.