# PyTorch Workshop

**João Oliveira Parracho**

18 December 2024

# Table of Contents


- What is PyTorch?
- Tensors
- Loading Datasets
- Building a model
- Training loop
- Inference

# What is PyTorch?

# PyTorch

- **Open-source** machine learning framework developed by **Meta AI** (Facebook).
- Widely used for **signal processing tasks** such as **image denoising, audio enhancement, and compression**.
- User friendly for researchers and developers to build deep-learning models
- Strong **GPU support**
- **Pytorch website**: https://pytorch.org/

PyTorch

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA STUDENT BRANCH CHAPTER

# PyTorch installation

| | | | | | |
|---|---|---|---|---|---|
| PyTorch Build | Stable (2.5.1) | | Preview (Nightly) | | |
| Your OS | Linux | Mac | Windows | | |
| Package | Conda | Pip | LibTorch | Source | |
| Language | Python | | C++ / Java | | |
| Compute Platform | CUDA 11.8 | CUDA 12.1 | CUDA 12.4 | ROCm 6.2 | CPU |

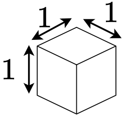
Run this Command:

```
pip3 install torch torchvision torchaudio --index-url https://download.pyt
orch.org/whl/cu118
```

Previous versions of PyTorch >

# What are tensors?

# Tensors: Scalar




**Numpy:**

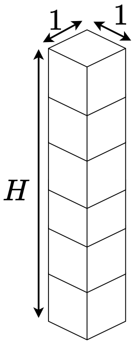
```python
import numpy as np

x = 1

```

**Torch:**

```python
import torch

x = torch.Tensor(1)

```

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA STUDENT BRANCH CHAPTER

# Tensors: 1D Array



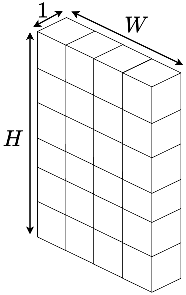

**Numpy:**

```python
import numpy as np

x = np.array([1.0,2.0,3.0])

```

**Torch:**

```python
import torch

x = torch.Tensor([1.0,2.0,3.0])

```

# Tensors: 2D Array




## Numpy:

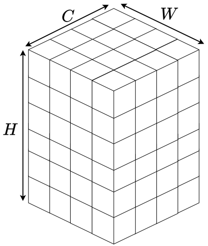
```python
import numpy as np

x = np.array([[1.0,2.0,3.0],
              [4.0,5.0,6.0]])
```

## Torch:

```python
import torch

x = torch.Tensor([[1.0,2.0,3.0],
                  [4.0,5.0,6.0]])
```

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# Tensors: 3D Array




## Numpy:

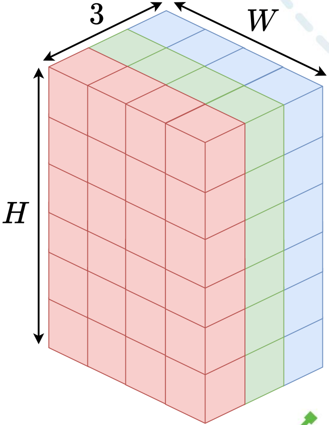
```python
import numpy as np

x = np.array([[[1.0,2.0],[3.0,4.0]],
              [[5.0,6.0],[7.0,8.0]]])

```

## Torch:

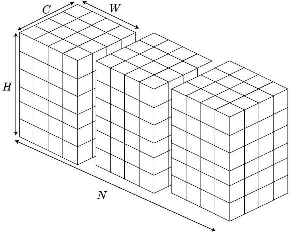
```python
import torch

x = torch.Tensor([[[1.0,2.0],[3.0,4.0]],
                  [[5.0,6.0],[7.0,8.0]]])

```

POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# Tensors: 3D Array



# Tensors: 4D Array




## Numpy:

```python
import numpy as np

x = np.array([[[[1,2],[3,4]],[[5,6],[7,8]]],
              [[[1,2],[3,4]],[[5,6],[7,8]]]])
```

## Torch:

```python
import torch

x=torch.Tensor([[[[1,2],[3,4]],[[5,6],[7,8]]],
               [[[1,2],[3,4]],[[5,6],[7,8]]]])
```

# Tensors: Initialisation

```python
import numpy as np
import torch

# Initialising with zeros
x = np.zeros((2,2))
y = torch.zeros((2,2))

# Initialising with ones
x = np.ones((2,2))
y = torch.ones((2,2))

# Initialising randomly
x = np.random.rand(2,2)
y = torch.rand((2,2))

```

# Tensors: Attributes

```python
import torch

x = torch.rand((4,3,20,20))

print(x.shape) # torch.Size([4, 3, 20, 20])
print(x.type()) # torch.FloatTensor
print(x.device) # cpu
```

# Tensors: From numpy to PyTorch, and CUDA

- Convert numpy array to Pytorch tensor:
  - **torch.from_numpy()**
- Convert Pytorch tensor to numpy array:
  - **y.numpy()**
- Change Pytorch tensor device:
  - **y.to()**

```python
import torch
import numpy as np

x = np.random.rand(4,3,20,20)
# Convert numpy array to PyTorch tensor
y = torch.from_numpy(x) # Returns a cpu tensor
# Convert PyTorch tensor to numpy array
z = y.numpy()

if torch.cuda.is_available():
    device = "cuda"
else:
    device ="cpu"
# Load the tensor to the cpu or cuda (gpu)
y = y.to(device)
```

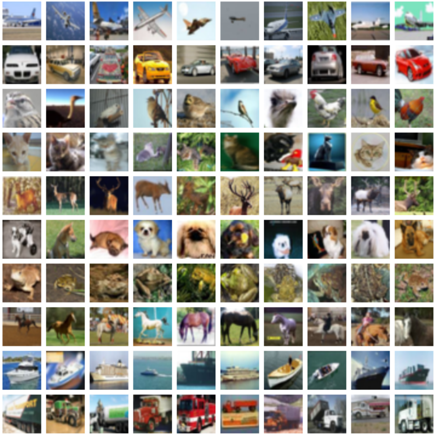
# Tensors: Operations

```
import torch

x = torch.rand((4,3,20,20))

# Indexing
x_1 = x[:,:,0:10,:] # torch.Size ([4, 3, 10, 20])
x_2 = x[:,:,10:,:] # torch.Size ([4, 3, 10, 20])
x_3 = x[:,:,:,0:10] # torch.Size ([4, 3, 20, 10])
x_4 = x[:,:,:,10:] # torch.Size ([4, 3, 20, 10])

# Concatenate tensors
y_1 = torch.cat((x_1,x_2),dim=2) # torch.Size ([4, 3, 20, 20])
y_2 = torch.cat((x_3,x_4),dim=3) # torch.Size ([4, 3, 20, 20])

# Swap tensors dimension
z = x.permute(3,0,2,1) # # torch.Size ([20, 4, 20, 3])
```

# CIFAR-10 Dataset

- 60000 $32 \times 32$ RGB images
- It has **10 classes**, with 6000 images per class.
- **Training images:** 50000
- **Test images:** 10000

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck



# Dataset: training, validation, and test set

- **Data Division**: Split data into three sets — **training, validation, and test**
- **Training Phase**: Train the model using the training set to adjust the initial parameters.
- **Validation Phase**: Evaluate the model in the validation set and **fine-tune the initial parameters** based on these results.
  - Usually validation set is around 20%,25% of the total training set images
- **Inference**: Evaluate the trained model on the test set to verify its real-world performance.

# Create a custom Dataset class

- The custom dataset class must **inherit** the abstract **torch.utils.data.Dataset** class
- **It must implement 3 methods**:
  - __init__
    - A constructor method used to initialise the network's parameters.
  - __len__
    - Returns size of the dataset
  - __getitem__
    - Enables indexing so that dataset[i] retrieves the $i^{th}$ sample.

```python
from torch.utils.data import Dataset
from torchvision import transforms
from torchvision import datasets

class Custom_Dataset(Dataset):
    def __init__(self, split="train"):
        is_train = True if split == "train" or split == "val" else False

        transform = transforms.ToTensor()
        self.dataset = datasets.CIFAR10('data', train=is_train,
                                        download=True,
                                        transform=transform)

        self.indices = np.arange(0, len(self.dataset))
        # lets use 20% of the trainning dataset to validation
        if split == "train":
            self.indices = self.indices[0:int(len(self.dataset)*0.8)]
        elif split == "val":
            self.indices = self.indices[int(len(self.dataset)*0.8):]
        elif split == "test":
            # self.dataset.targets: returns the image classes (0,1,..,9)
            self.indices = np.array(self.dataset.targets)

```

```python
    def __getitem__(self, idx):
        image = self.dataset[self.indices[idx]][0]
        label = self.dataset[self.indices[idx]][1]
        return image, label

    def __len__(self):
        return len(self.indices)
```

# How to load images from the custom dataset?

- **torch.utils.data.DataLoader** : Provides an iterable over the given dataset.
- **dataset**: Dataset from which to load the data
- **batch_size**: How many images loaded per batch
- **shuffle**: Enable data reshuffled at every epoch
- **num_workers**: Number subprocesses to use for data loading.

```
from torch.utils.data import DataLoader

train_set = Custom_Dataset(split="train")


trainloader=DataLoader(train_set,
                       batch_size=4,
                       shuffle=False,
                       num_workers=2)


```

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# How to load images from the custom dataset?

```
from torch.utils.data import DataLoader

train_set = Custom_Dataset(split="train")
val_set = Custom_Dataset(split="val")
test_set =  Custom_Dataset(split="test")

trainloader = DataLoader(train_set,batch_size=4, shuffle=False,num_workers=2)
valloader = DataLoader(val_set, batch_size=4, shuffle=False, num_workers=2)
testloader = DataLoader(test_set, batch_size=4, shuffle=False, num_workers=2)

for i, data in enumerate(trainloader):
    inputs, labels = data

for i, data in enumerate(valloader):
    inputs, labels = data

for i, data in enumerate(testloader):
    inputs, labels = data
```

POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# Create Custom Balanced dataset

- The CIFAR-10 has the same number of images per class
- In the Custom_dataset(), **it was not ensured that the validation and training set have the same number of images per class**

**How can this problem be solved?**

```python
class Custom_Dataset_Balanced_Split(Dataset):
    def __init__(self, split="train"):
        is_train = True if split == "train" or split == "val" else False

        transform = transforms.ToTensor()
        self.dataset = datasets.CIFAR10('data', train=is_train,
                                        download=True, transform=transform)

        # self.dataset.targets: returns the image classes (0,1,..,9)
        # The classes are not ordered -> self.dataset.targets : [6,2,5,9,6,1,...,3]
        x = np.array(self.dataset.targets)
        # np.argsort(x) : return the indeces ordered such that is possible to
        # retrive the classes like : [0,0,0,...,1,1,1,...,2,2,2,...,9,9,9]
        sorted_indices = np.argsort(x)
        num_elem_class = int(np.sum(x==0))
        s = np.zeros((10,num_elem_class),int)
        # For each class, store the corresponding indices in an array.
        for i in range(10):
            s[i] = sorted_indices[i*num_elem_class:(i+1)*num_elem_class]
```

Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

```python
    indicez_zip = zip(s[0], s[1], s[2],s[3],s[4], s[5],s[6],s[7],s[8],s[9])

    self.sorted_indices = [item for indices in indicez_zip for item in indices]

    # for idx in range(len()):
    #    print(self.dataset[self.sorted_indices[idx]][1])
    # Output: 0,1,2,3,4,5,6,7,8,9,0,1,2,..,9

    # lets use 20% of the trainning dataset to validation
    if split == "train":
        self.sorted_indices = self.sorted_indices[0:int(len(self.sorted_indices)*0.8)]
    elif split == "val":
        self.sorted_indices = self.sorted_indices[int(len(self.sorted_indices)*0.8):]

    def __getitem__(self, idx):
        image = self.dataset[self.sorted_indices[idx]][0]
        label = self.dataset[self.sorted_indices[idx]][1]
        return image, label

    def __len__(self):
        return len(self.sorted_indices)
```

# 2D Convolution

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 2 & 5 & 4 & 5 & 2 \\ 2 & 3 & 5 & 4 & 4 \\ 2 & 3 & 4 & 5 & 2 \\ 2 & 4 & 4 & 2 & 2 \\ 4 & 4 & 2 & 2 & 1 \end{bmatrix}$$

$$\mathbf{I} \qquad \mathbf{K} \qquad b \qquad \mathbf{I} * \mathbf{K} + b$$

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size $(N, C_{\text{in}}, H, W)$ and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

- Input: $(N, C_{in}, H_{in}, W_{in})$ or $(C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

# 2D Convolution: Parameters

- **in_channels**: Number of channels in the input
- **out_channels**: Number of output channels
- **kernel_size**: Size of the convolving kernel
- **bias**: If True, adds a learnable bias to the output
- **dilation**: Spacing between kernel elements
- **padding**: Padding added to all four sides of the input.

# Linear layer

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies an affine linear transformation to the incoming data: $\boxed{y = xA^T + b.}$

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

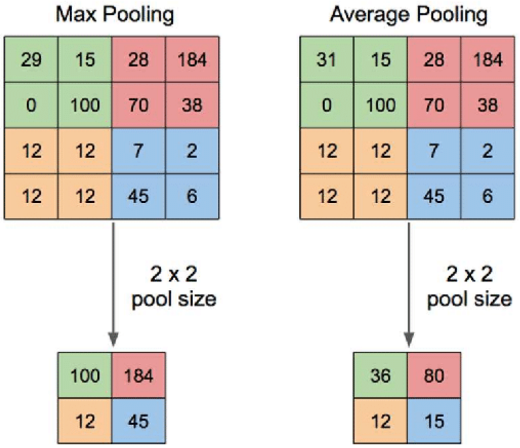
**Parameters**

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

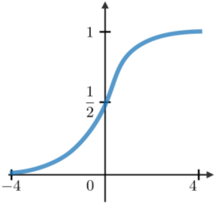
# Pooling layer




IEEE

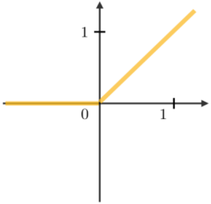
IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA STUDENT BRANCH CHAPTER

# Activate functions



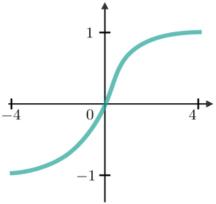

**Sigmoid**

$$g(z) = \frac{1}{1+e^{-z}}$$



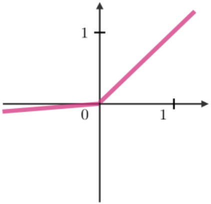

**ReLU**

$$g(z) = \max(0, z)$$




**Tanh**

$$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

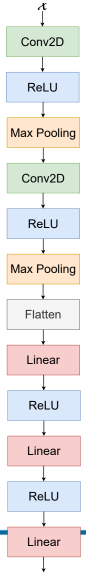



**Leaky ReLU**

$$g(z) = \max(\epsilon z, z)$$

with $\epsilon \ll 1$

# Model




```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 5*5 : Resolution on the input in the first
        # linear layer.
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        # flatten all dimensions except batch
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

# Model: What if I want to add more layers to the Model?

- Create a new model that inherits the Net() class
- In the __init __() define the new additional layers
- Change the forward()

```python
class Net_2(Net):
    def __init__(self):
        super(Net_2, self).__init__()
        self.fc3 = nn.Linear(84, 42)
        self.fc4 = nn.Linear(42, 10)

    def forward(self, x):
        x = self.pool((F.relu(self.conv1(x))))
        x = self.pool((F.relu(self.conv2(x))))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

# Let’s learn how to train a classifier

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# What does the code for train a model looks like?

- Create dataset ✓

```
train_set = Custom_Dataset_Balanced_Split(split="train")
val_set = Custom_Dataset_Balanced_Split(split="val")
test_set =  Custom_Dataset_Balanced_Split(split="test")

trainloader = DataLoader(train_set,batch_size=10,
                          shuffle=False,num_workers=2)

valloader = DataLoader(val_set, batch_size=10,
                        shuffle=False, num_workers=2)

testloader = DataLoader(test_set, batch_size=10,
                         shuffle=False, num_workers=2)
```

# What does the code for train a model looks like?

- Create dataset ✓
- Create a model ✓

```
device = "cuda" if torch.cuda.is_available() else "cpu"

net = Net()

net.to(device)
```

# What does the code for train a model looks like?

- Create dataset ✓
- Create a model ✓
- Create optimiser

# AutoGrad

- PyTorch module that performs automatic differentiation on tensors.
- Keeps track of the operations performed on tensors and builds a computation graph
- It uses the the computation graph to calculate the gradient with respect to its inputs, allowing for backpropagation to be performed in deep learning models.

# What does the code for train a model looks like?

- Create dataset ✓
- Create a model ✓
- Create optimiser ✓
- Define the loss ✓

```
optimizer = optim.Adam(net.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()
```

IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# What does the code for train a model looks like?

- Create dataset ✓
- Create a model ✓
- Create optimiser ✓
- Define the loss ✓
- Training Loop



IEEE Signal Processing Society
POLYTECHNIC OF LEIRIA
STUDENT BRANCH CHAPTER

# Training Loop

```python
for epoch in range(num_epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    correct = 0
    total = 0
    for i, data in enumerate(trainloader):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        # Transfer the inputs to the device in use
        inputs=inputs.to(device)
        labels=labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad() # DO NOT FORGET!!
```

# Training Loop

```python

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step() # update model learnable weights

        total += labels.size(0)
        correct += accuracy(outputs, labels)
        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

    print(f'Accuracy of the network on training set: {100 * correct / total} %')
```

# What does the code for train a model looks like?

- Create dataset ✓
- Create a model ✓
- Create optimiser ✓
- Define the loss ✓
- Training Loop ✓
- Validation Loop ✓

# Validation Loop

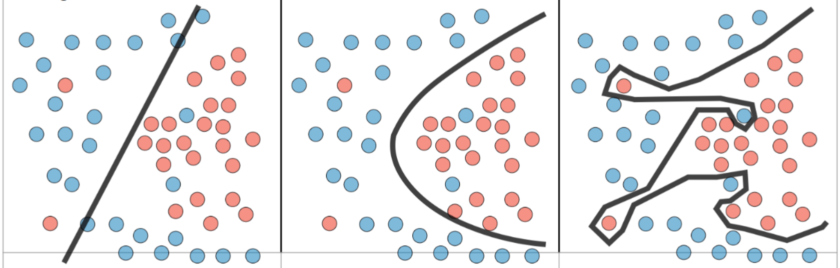
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our
outputs
with torch.no_grad():
    for data in valloader:

        images,labels = data
        images=images.to(device)
        labels=labels.to(device)
        # calculate outputs by running images through the network
        outputs = net(images)

        total += labels.size(0)
        correct += accuracy(outputs, labels)

accuracy_val = correct / total
```

# Validation Loop: Save checkpoint

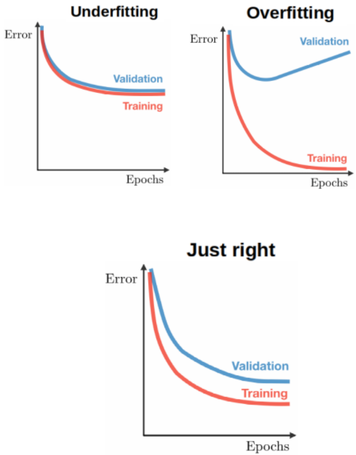
```python
# use the validation
if accuracy_val > accuracy_val_max:
    accuracy_val_max = accuracy_val
    # Save the model weights
    # The best checkpoint will be later
    # used for inference
    torch.save(net.state_dict(), checkpoint_path)
    best_epoch = epoch

print(f'Accuracy of the network on validation set: {np.round(100 * accuracy_val,3)}
 %')
```

# Model fitting: Classification of red vs blue dots

| Underfitting | Just right | Overfitting |
|---|---|---|
| • High training error<br>• Training error close to validaton error<br>• High bias | • Training error slightly lower than validation error | • Very low training error<br>• Training error much lower than validation error |



# Model fitting (what we generally see)



# What does the code for the inference look like?

- Create dataset ✓
- Create a model ✓
- Load checkpoint

```
net.load_state_dict(torch.load(checkpoint_path))
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

# What does the code for the inference look like?

- Create dataset ✓
- Create a model ✓
- Load checkpoint ✓
- Inference loop

```python
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images=images.to(device)
        labels=labels.to(device)
        # calculate outputs
        outputs = net(images)
        # Class with the highest energy is what we
        # choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total} %')
```

# What does the code for the inference look like?

- Create dataset ✓
- Create a model ✓
- Load checkpoint ✓
- Inference loop ✓
- Performance per class

```python
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images=images.to(device)
        labels=labels.to(device)
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
```

# What does the code for the inference look like?

- Create dataset ✓
- Create a model ✓
- Load checkpoint ✓
- Inference loop ✓
- Performance per class ✓

```python
# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

# Thank You