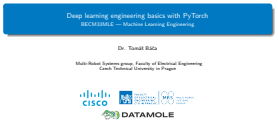
Deep learning engineering basics with PyTorch

BECM33MLE — Machine Learning Engineering

Dr. Tomáš Báča

Multi-Robot Systems group, Faculty of Electrical Engineering
Czech Technical University in Prague



- always recommend to work in a **Virtual Environment**
- it stores copies of the particular dependencies locally
- one solution to a **dependency hell**

Dependencies (requirements.txt)

```
numpy==2.3.3
matplotlib==3.10.6
scikit-learn==1.7.2
pandas==2.3.2
jupyter
jupyterlab-vim
wandb
transformers # huggingface
drawdata # drawing datasets
```

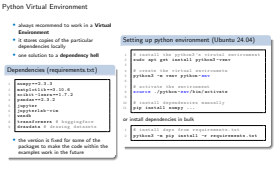
- the version is fixed for some of the packages to make the code within the examples work in the future

Setting up python environment (Ubuntu 24.04)

```
# install the python3's virutal environment
sudo apt get install python3-venv

# create the virtual environmetn
python3 -m venv python-env

# activate the environment
source ./python-env/bin/activate

# install dependencies manually
pip install numpy ...
```

or install dependencies in bulk

```
# install deps from requirements.txt
python3 -m pip install -r requirements.txt
```

- there are other options to make sure you dependencies are met and are not colliding with other projects:
  - Conda package manager,
  - Docker,
  - Singularity / Apptainer container system,
  - Nix package manager.

- web-interface with an editor
- allows execution of portions of your code
- remembers the state of variables
- remembers where you left it
- embedded visualization
- well-suited for learning and prototyping
- (bonus) vim-like key bindings
- **Jupyter Notebook**
  - single document, simple
- **Jupyter Lab**
  - multiple documents at once
  - similar to an IDE
  - better file management

Running Jupyter Notebook

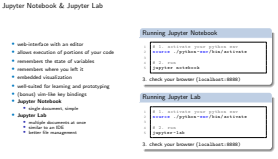
```
# 1. activate your python env
source ./python-env/bin/activate

# 2. run
jupyter notebook
```

3. check your browser (`localhost:8888`)

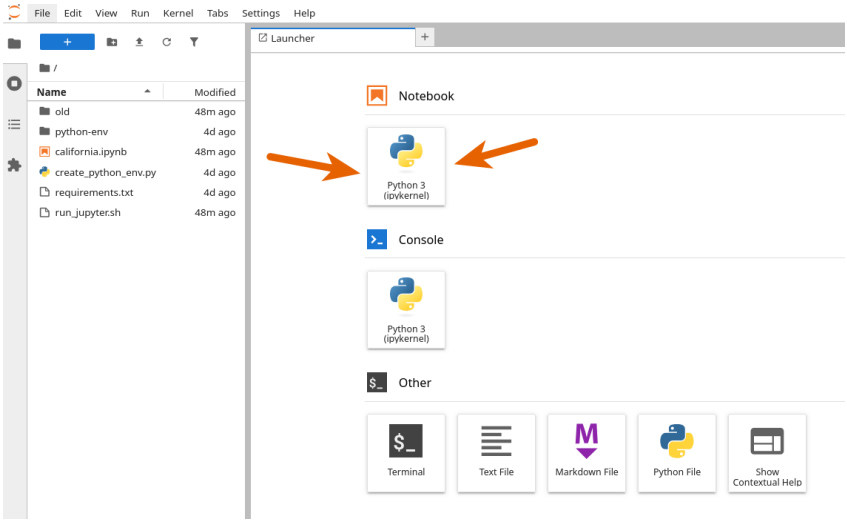
Running Jupyter Lab

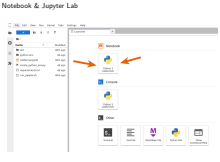
```
# 1. activate your python env
source ./python-env/bin/activate

# 2. run
jupyter-lab
```

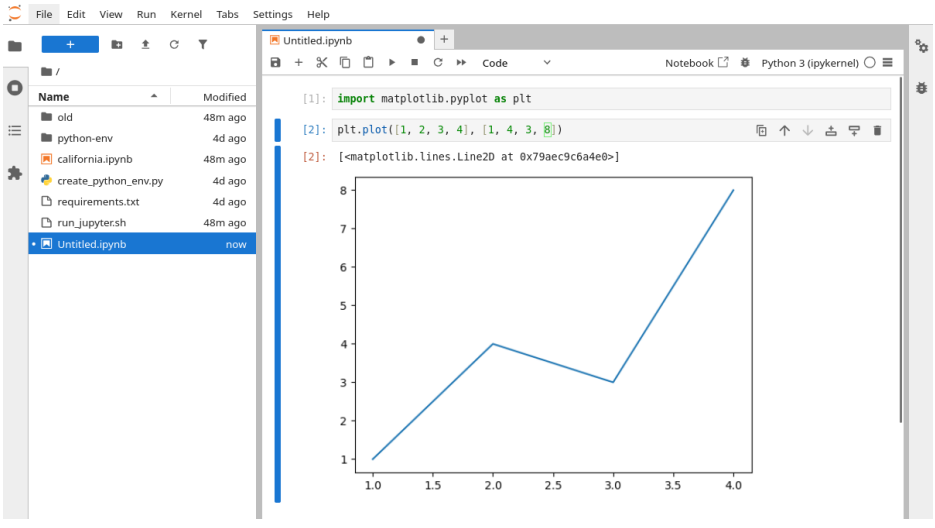
3. check your browser (`localhost:8888`)

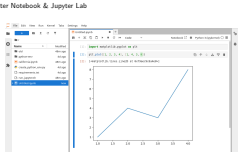


- You can, of course, run your python scripts manually. However, this makes development and tweaking of the algorithms tedious.
- In practice, you can design the first part of the architecture in Jupyter and then move to more traditional program architecture.









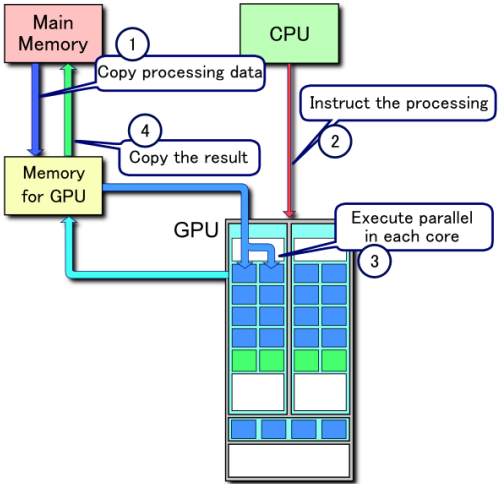
- use <tab> for code completion
- use `shift+<tab>` for explanation of objects and their properties
- use `ctrl+enter` to evaluate current cell

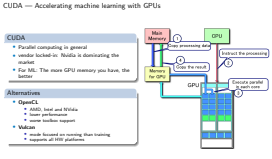
CUDA

- Parallel computing in general
- vendor locked-in: Nvidia is dominating the market
- For ML: The more GPU memory you have, the better

Alternatives

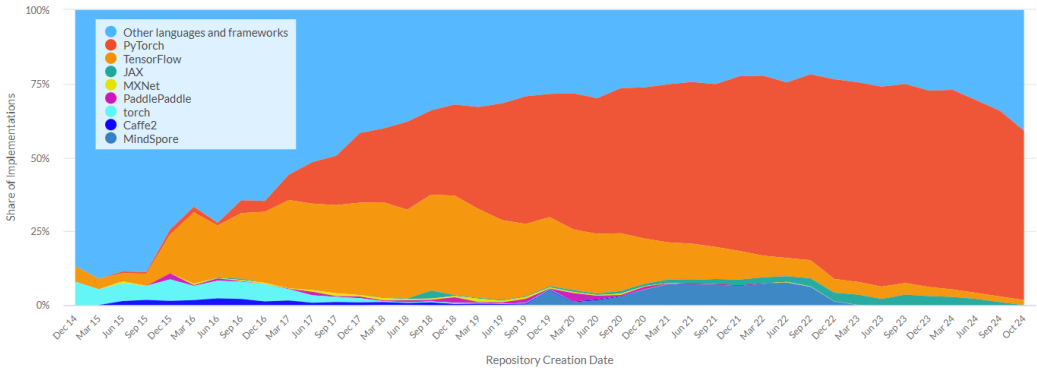
- **OpenCL**
  - AMD, Intel and NVidia
  - lower performance
  - worse toolbox support
- **Vulcan**
  - mode focused on running than training
  - supports all HW platforms





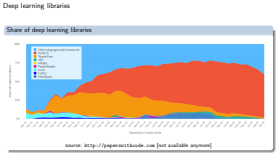
- CUDA is useful for parallel processing in general:
  - signal processing (Fourier transform)
  - image / video encoding
  - compression
  - cryptography
  - physics simulations (particle simulations)
- Installing CUDA is platform specific, always follow steps dedicated to your hardware and operating system

Share of deep learning libraries



source: `http://paperswithcode.com` [not available anymore]

Lecture 3: Deep learning engineering basics with PyTorch
└Deep learning libraries
└Deep learning libraries
2025-10-11



- In recent years: PyTorch is dominating the publicly released models and code.

PyTorch

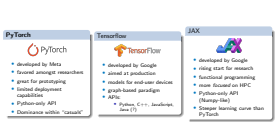
- developed by Meta
- favored amongst researchers
- great for prototyping
- limited deployment capabilities
- Python-only API
- Dominance within “casuals”

Tensorflow

- developed by Google
- aimed at production
- models for end-user devices
- graph-based paradigm
- APIs:
  - Python, C++, JavaScript, Java (?)

JAX

- developed by Google
- rising start for research
- functional programming
- more *focused* on HPC
- Python-only API (Numpy-like)
- Steeper learning curve than PyTorch



Installing PyTorch

- generate installation command at `https://pytorch.org/get-started/locally/`

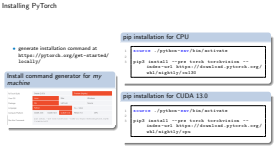
Install command generator for *my machine*

pip installation for CPU

```
source ./python-env/bin/activate

pip3 install --pre torch torchvision --
    index-url https://download.pytorch.org/
    whl/nightly/cu130
```

pip installation for CUDA 13.0

```
source ./python-env/bin/activate

pip3 install --pre torch torchvision --
    index-url https://download.pytorch.org/
    whl/nightly/cpu
```

- there are other options to make sure you dependencies are met and are not colliding with other projects:
  - Conda package manager,
  - Docker,
  - Singularity / Apptainer container system,
  - Nix package manager.

Representing multi-dimensional data

- tensors are generalization of n-dimensional matrices (scalars, vectors, matrices, n-dimensional tensor, ...)
- propagation of information in deep learning (DL) linear layers is modeled by tensor operations:

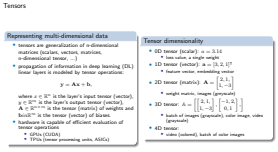
$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b},$$

where $x \in \mathbb{R}^n$ is the layer's input tensor (vector), $y \in \mathbb{R}^m$ is the layer's output tensor (vector), $\mathbf{A} \in \mathbb{R}^{n \times m}$ is the tensor (matrix) of weights and $\mathbf{b} in \mathbb{R}^m$ is the tensor (vector) of biases.

- hardware is capable of efficient evaluation of tensor operations
  - GPUs (CUDA)
  - TPUs (tensor processing units, ASICs)

Tensor dimensionality

- 0D tensor (scalar): $a = 3.14$
  - loss value, a single weight
- 1D tensor (vector): $\mathbf{a} = [3, 2, 1]^\intercal$
  - feature vector, embedding vector
- 2D tensor (matrix): $\mathbf{A} = \begin{bmatrix} 2, 1, \\ 1, -3 \end{bmatrix}$
  - weight matrix, images (greyscale)
- 3D tensor: $\mathbb{A} = \left[\begin{bmatrix} 2, 1, \\ 1, -3 \end{bmatrix}, \begin{bmatrix} -1, 2, \\ 0, 1 \end{bmatrix}, \right]$
  - batch of images (grayscale), color image, video (grayscale)
- 4D tensor:
  - video (colored), batch of color images



Example definition of tensors

```python
import torch

# 1D tensor (row)
x = torch.tensor([1, 2])

# 1D tensor (column)
x = torch.tensor([1, 2]).view(-1, 1)

# 2D tensor (per row)
x = torch.tensor([[1, 2],
                  [3, 4]])
```

Moving tensor to the GPU device

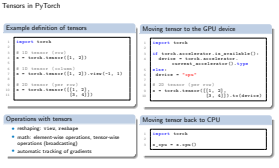
```python
import torch

if torch.accelerator.is_available():
  device = torch.accelerator.
      current_accelerator().type
else:
 device = "cpu"

# 2D tensor (per row)
x = torch.tensor([[1, 2],
                  [3, 4]]).to(device)
```

Operations with tensors

- reshaping: `view`, `reshape`
- math: element-wise operations, tensor-wise operations (broadcasting)
- automatic tracking of gradients

Moving tensor back to CPU

```python
import torch

x_cpu = x.cpu()
```

XOR classification task

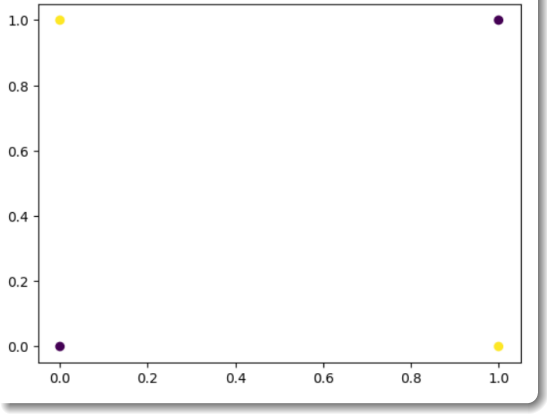
- linearly non-separable classes
- can not be solve by purely linear classification (SVM, perceptron)
- solutions within classical ML:
  - lifting to higher dimensions ($+ x_1 x_2$)
  - decision tree
  - kernel SVM

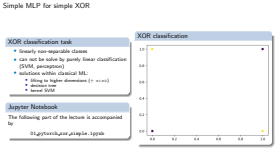
Jupyter Notebook

The following part of the lecture is accompanied by

`01_pytorch_xor_simple.ipynb`

XOR classification

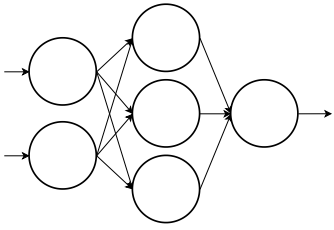




1. Imports

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import torch
from torch import nn
from torch import optim
from torch.autograd import Variable
```

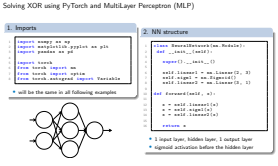
- will be the same in all following examples



2. NN structure

```python
class NeuralNetwork(nn.Module):
 def __init__(self):

    super().__init__()

    self.linear1 = nn.Linear(2, 3)
    self.sigm1 = nn.Sigmoid()
    self.linear2 = nn.Linear(3, 1)

def forward(self, x):

    x = self.linear1(x)
    x = self.sigm1(x)
    x = self.linear2(x)

    return x
```

- 1 input layer, hidden layer, 1 output layer
- sigmoid activation before the hidden layer



3. prepare the model

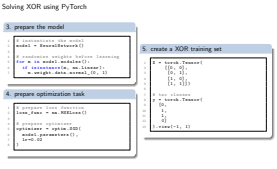
```
# instantiate the model
model = NeuralNetwork()

# randomize weights before learning
for m in model.modules():
  if isinstance(m, nn.Linear):
    m.weight.data.normal_(0, 1)
```

4. prepare optimization task

```
# prepare loss function
loss_func = nn.MSELoss()

# prepare optimizer
optimizer = optim.SGD(
  model.parameters(),
  lr=0.02
)
```

5. create a XOR training set

```
X = torch.Tensor(
    [[0, 0],
     [0, 1],
     [1, 0],
     [1, 1]])

# two classes
y = torch.Tensor(
  [0,
   1,
   1,
   0]
).view(-1, 1)
```

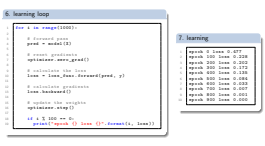
- The **loss function** defines how the "output error" of the neural net is computer.
  - different loss functions are suitable for different tasks
  - **MSELoss** stands for *Mean Square Error Loss*, is suitable for regression (and simple classification)
- the `SGD` optimizer performs *Stochastic Gradient Descend* to find a local optimum of a function
- the `lr` argument defines the step size of the gradient descend-based optimization methods

6. learning loop

```
for i in range(1000):

    # forward pass
    pred = model(X)

    # reset gradients
    optimizer.zero_grad()

    # calculate the loss
    loss = loss_func.forward(pred, y)

    # calculate gradients
    loss.backward()

    # update the weights
    optimizer.step()

    if i % 100 == 0:
      print("epoch {} loss {}".format(i, loss))
```

7. learning

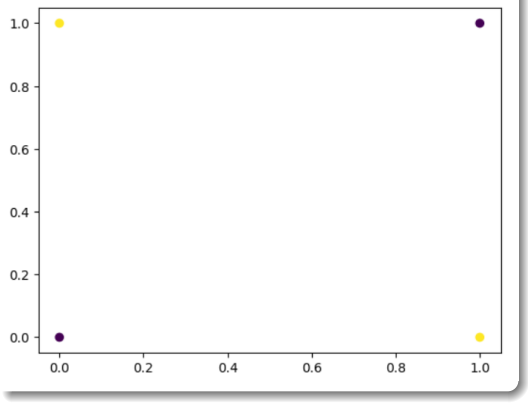
```
epoch 0 loss 0.477
epoch 100 loss 0.228
epoch 200 loss 0.202
epoch 300 loss 0.172
epoch 400 loss 0.135
epoch 500 loss 0.084
epoch 600 loss 0.033
epoch 700 loss 0.007
epoch 800 loss 0.001
epoch 900 loss 0.000
```

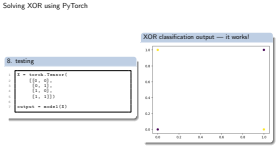



8. testing

```
X = torch.Tensor(
    [[0, 0],
     [0, 1],
     [1, 0],
     [1, 1]])

output = model(X)
```

XOR classification output — it works!

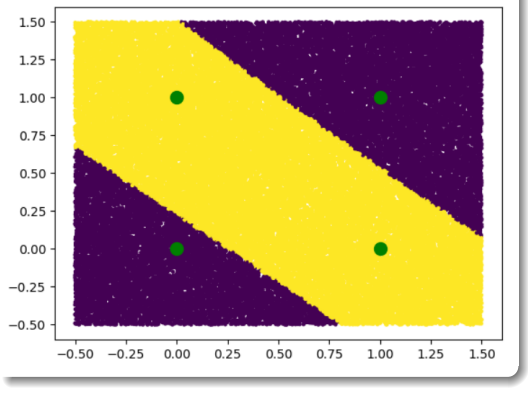


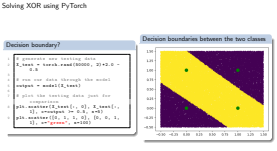


Decision boundary?

```
# generate new testing data
X_test = torch.rand(50000, 2)*2.0 - 0.5

# run our data through the model
output = model(X_test)

# plot the testing data just for comparison
plt.scatter(X_test[:, 0], X_test[:, 1], c=output >= 0.5, s=5)
plt.scatter([0, 1, 1, 0], [0, 0, 1, 1], c="green", s=100)
```

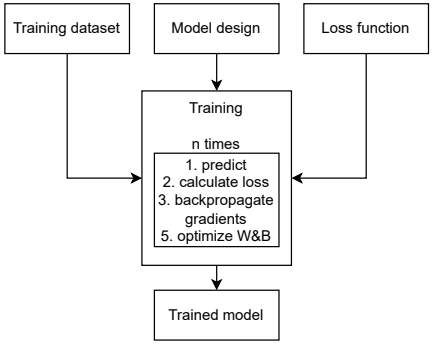
Decision boundaries between the two classes

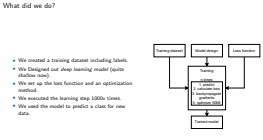




What did we do?

- We created a training dataset including labels.
- We Designed out *deep learning model* (quite shallow now).
- We set up the loss function and an optimization method.
- We executed the learning step 1000x times.
- We used the model to predict a class for new data.





More complex XOR task

deeper XOR classification task

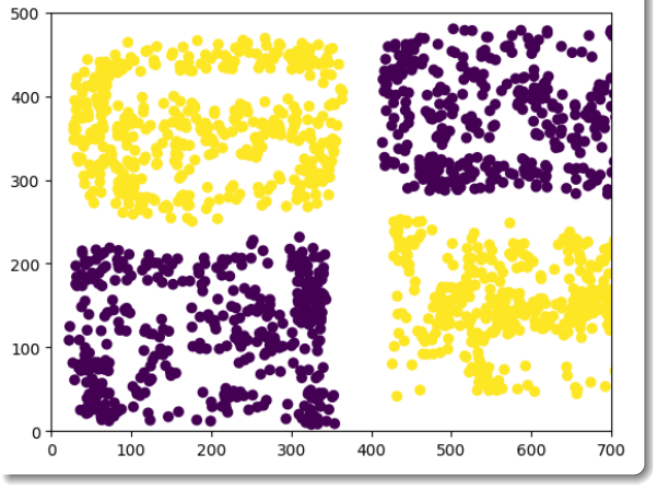
- non-singular classes
- dataset needs scaling
- decision boundaries not linear
- more training data
  - might need splitting to batches
- might benefit from running using CUDA

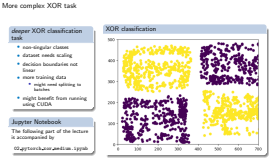
Jupyter Notebook

The following part of the lecture is accompanied by

`02_pytorch_xor_medium.ipynb`

XOR classification





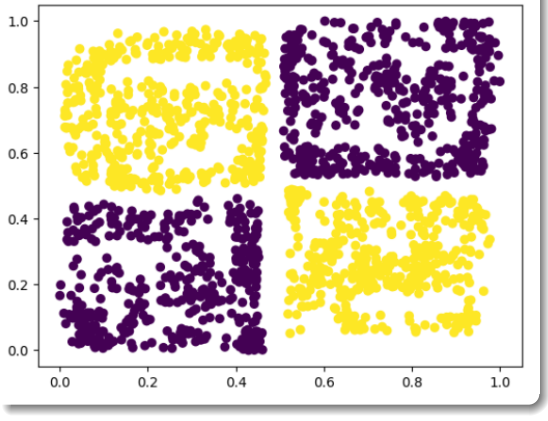
Dataset preprocessing

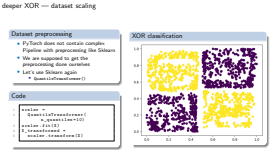
- PyTorch does not contain complex Pipeline with preprocessing like Sklearn
- We are supposed to get the preprocessing done ourselves
- Let's use Sklearn again
  - `QuantileTransformer()`

Code

```
scaler =
  QuantileTransformer(
      n_quantiles=10)
scaler.fit(X)
X_transformed =
  scaler.transform(X)
```

XOR classification





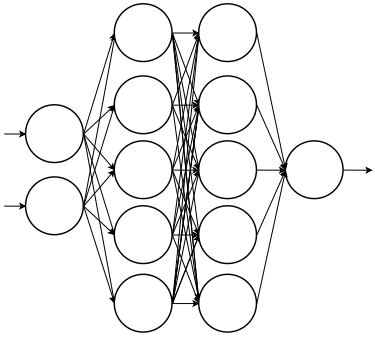
deeper XOR — learning batches

Batches

- proper batch size is important:
- too large:
  - data might not fit into memory
  - the model might generalize poorly
- too small:
  - many learning steps with an epoch

Code

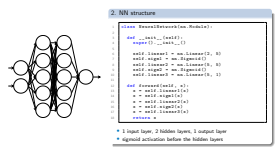
```python
# convert the transformed dataset to tensors
X_tens = torch.tensor(X_transformed).to(torch.float32)
y_tens = torch.tensor(y).to(torch.float32).view(-1, 1)

# split dataset into batches
from torch.utils.data import DataLoader, TensorDataset

dataset = TensorDataset(X_tens, y_tens)
dataloader = DataLoader(
                dataset,
                batch_size=32,
                shuffle=True
               )
```

2. NN structure

```
1  class NeuralNetwork(nn.Module):
2
3    def __init__(self):
4      super().__init__()
5
6      self.linear1 = nn.Linear(2, 5)
7      self.sigm1 = nn.Sigmoid()
8      self.linear2 = nn.Linear(5, 5)
9      self.sigm2 = nn.Sigmoid()
10     self.linear3 = nn.Linear(5, 1)
11
12   def forward(self, x):
13     x = self.linear1(x)
14     x = self.sigm1(x)
15     x = self.linear2(x)
16     x = self.sigm2(x)
17     x = self.linear3(x)
18     return x
```

- 1 input layer, 2 hidden layers, 1 output layer
- sigmoid activation before the hidden layers

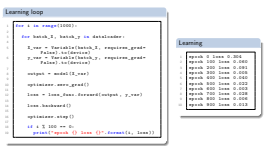


Learning loop

```
for i in range(1000):

  for batch_X, batch_y in dataloader:

    X_var = Variable(batch_X, requires_grad=
        False).to(device)
    y_var = Variable(batch_y, requires_grad=
        False).to(device)

    output = model(X_var)

    optimizer.zero_grad()

    loss = loss_func.forward(output, y_var)

    loss.backward()

    optimizer.step()

    if i % 100 == 0:
      print("epoch {} loss {}".format(i, loss))
```

Learning

```
epoch 0 loss 0.304
epoch 100 loss 0.060
epoch 200 loss 0.091
epoch 300 loss 0.005
epoch 400 loss 0.040
epoch 500 loss 0.022
epoch 600 loss 0.003
epoch 700 loss 0.028
epoch 800 loss 0.006
epoch 900 loss 0.013
```

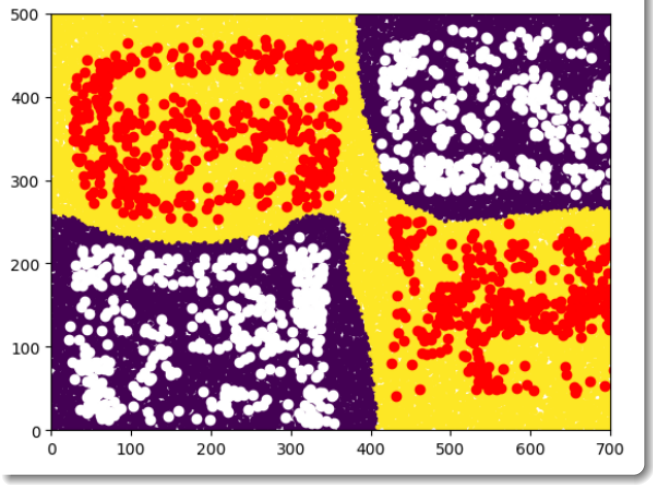



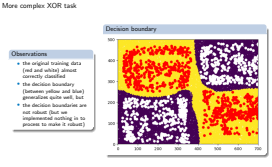
More complex XOR task

Observations

- the original training data (red and white) almost correctly classified
- the decision boundary (between yellow and blue) generalizes quite well, but
- the decision boundaries are not robust (but we implemented nothing in to process to make it robust)

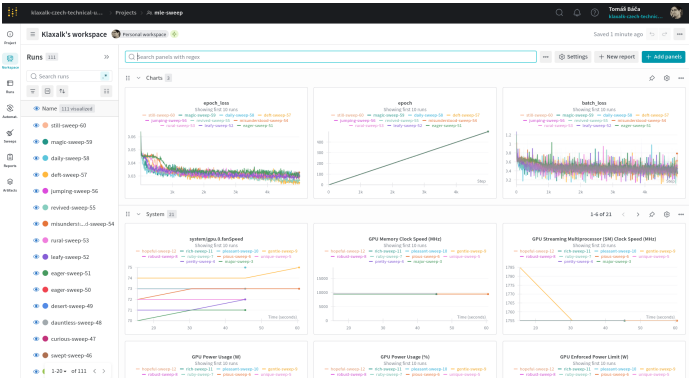
Decision boundary





Weights & Biases

- `http://wandb.ai`
- Online system for Monitoring
- Python API that integrates with your learning process
- Can gather arbitrary learning data, metrics
- Can gather resulting models and their weights (and biases)
- Facilitates parameter sweeps



Jupyter Notebook

The following part of the lecture is accompanied by `03_pytorch_separability_wandb.ipynb`

Installation

```
pip3 install wandb
```

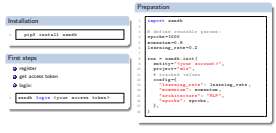
First steps

1. register
2. get access token
3. login:

```
wandb login <your access token>
```

Preparation

```
import wandb

# define reusable params:
epochs=1000
momentum=0.9
learning_rate=0.2

run = wandb.init(
  entity="<your account>",
  project="mle",
  # tracked values
  config={
    "learning_rate": learning_rate,
    "momentum": momentum,
    "architecture": "MLP",
    "epochs": epochs,
  },
)
```

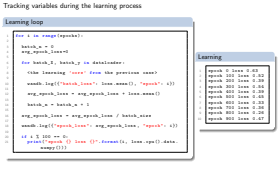



Learning loop

```python
for i in range(epochs):

  batch_n = 0
  avg_epoch_loss=0

  for batch_X, batch_y in dataloader:

    <the learning 'core' from the previous case>

    wandb.log({"batch_loss": loss.mean(), "epoch": i})

    avg_epoch_loss = avg_epoch_loss + loss.mean()

    batch_n = batch_n + 1

  avg_epoch_loss = avg_epoch_loss / batch_size

  wandb.log({"epoch_loss": avg_epoch_loss, "epoch": i})

  if i % 100 == 0:
    print("epoch {} loss {}".format(i, loss.cpu().data.
        numpy()))
```

Learning

```
epoch 0 loss 0.63
epoch 100 loss 0.52
epoch 200 loss 0.39
epoch 300 loss 0.54
epoch 400 loss 0.39
epoch 500 loss 0.45
epoch 600 loss 0.33
epoch 700 loss 0.36
epoch 800 loss 0.26
epoch 900 loss 0.47
```

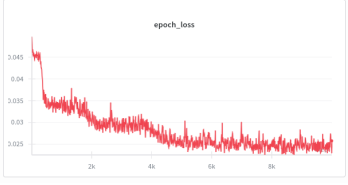



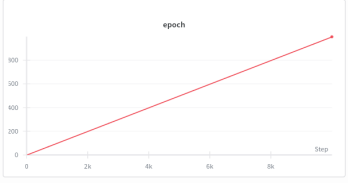
Visualizing the variables in WandB

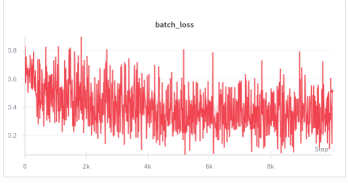
What do you get?

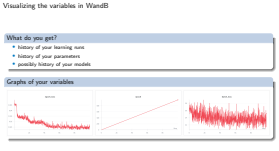
- history of your learning runs
- history of your parameters
- possibly history of your models

Graphs of your variables





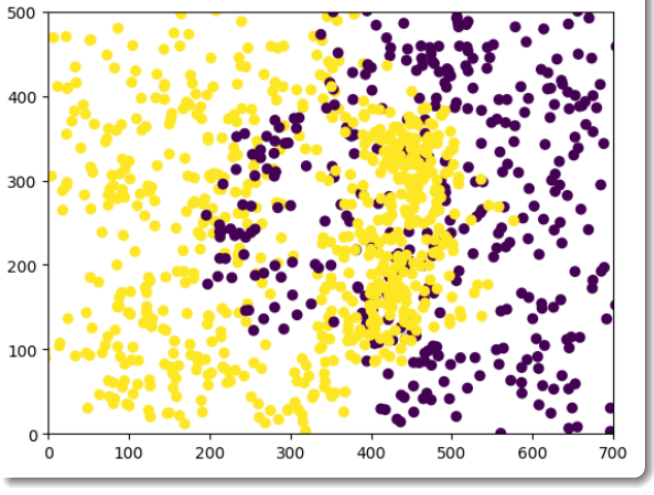


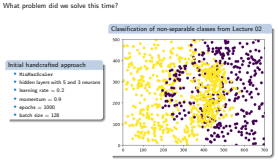


Initial handcrafted approach

- `MinMaxScaler`
- hidden layers with 5 and 3 neurons
- learning rate = 0.2
- momentum = 0.9
- epochs = 1000
- batch size = 128

Classification of non-separable classes from Lecture 02

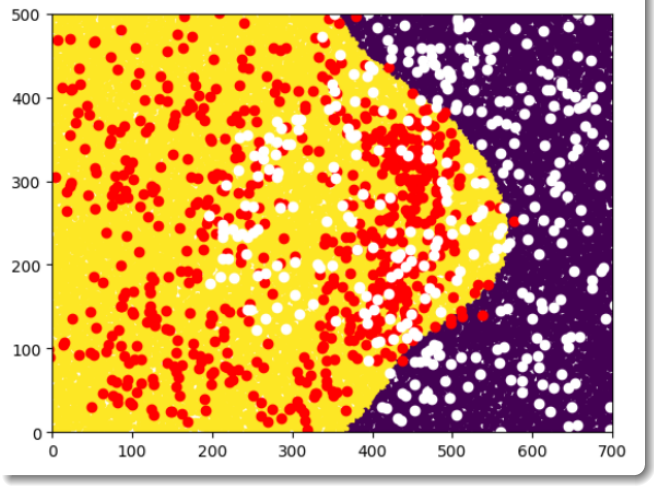


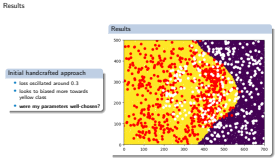


Initial handcrafted approach

- loss oscillated around 0.3
- looks to biased more towards yellow class
- **were my parameters well-chosen?**

Results





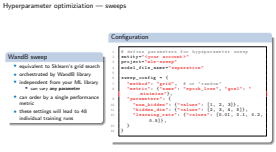
- The parameters were chosen arbitrarily with more-or-less (less) decent expert knowledge. For most real-world settings, the parameters can not be chosen just like this, but they need to be selected by an automated routine, similarly to the Sklearn's *grid search*.

WandB sweep

- equivalent to Sklearn's grid search
- orchestrated by WandB library
- independent from your ML library
  - can vary **any parameter**
- can order by a single performance metric
- these settings will lead to 48 individual training runs

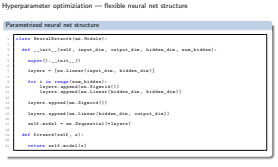
Configuration

```
# define parameters for hypeparameter sweep
entity="<your account>"
project="mle-sweep"
model_file_name="separation"

sweep_config = {
  "method": "grid",  # or "random"
  "metric": {"name": "epoch_loss", "goal": "
      minimize"},
  "parameters": {
    "num_hidden": {"values": [1, 2, 3]},
    "hidden_dim": {"values": [2, 3, 4, 5]},
    "learning_rate": {"values": [0.01, 0.1, 0.2,
        0.5]},
  }
}
```

Parametrized neural net structure

```python
class NeuralNetwork(nn.Module):

  def __init__(self, input_dim, output_dim, hidden_dim, num_hidden):

    super().__init__()

    layers = [nn.Linear(input_dim, hidden_dim)]

    for i in range(num_hidden):
        layers.append(nn.Sigmoid())
        layers.append(nn.Linear(hidden_dim, hidden_dim))

    layers.append(nn.Sigmoid())

    layers.append(nn.Linear(hidden_dim, output_dim))

    self.model = nn.Sequential(*layers)

  def forward(self, x):

    return self.model(x)
```

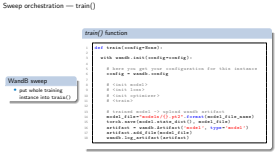



WandB sweep

- put whole training instance into `train()`

train() function

```python
def train(config=None):

  with wandb.init(config=config):

    # here you get your configuration for this instance
    config = wandb.config

    # <init model>
    # <init loss>
    # <init optimizer>
    # <train>

    # trained model -> upload wandb artifact
    model_file="models/{}.pt2".format(model_file_name)
    torch.save(model.state_dict(), model_file)
    artifact = wandb.Artifact('model', type='model')
    artifact.add_file(model_file)
    wandb.log_artifact(artifact)
```

WandB sweep

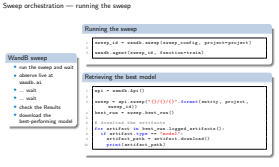
- run the sweep and wait
- observe live at `wandb.ai`
- ... wait
- ... wait
- check the Results
- download the best-performing model

Running the sweep

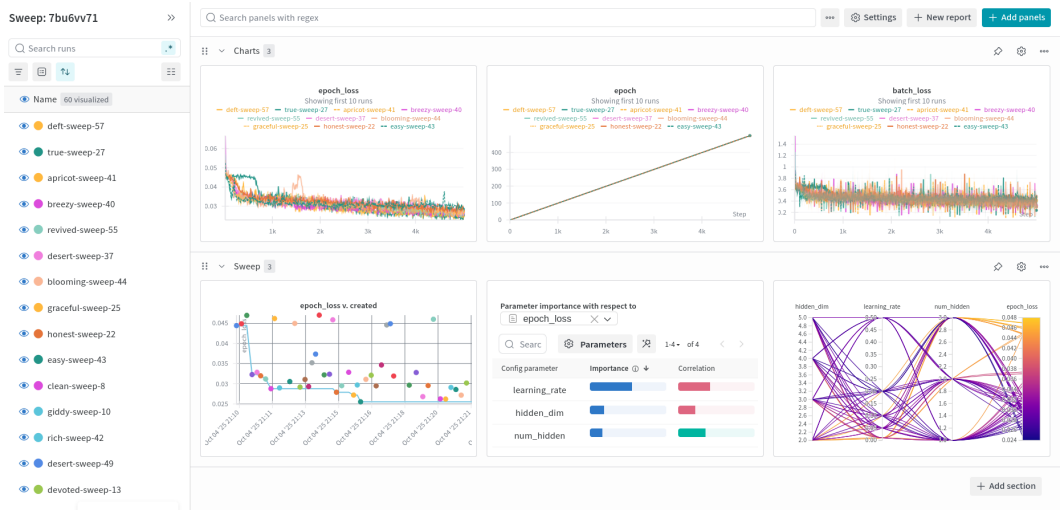
```
sweep_id = wandb.sweep(sweep_config, project=project)

wandb.agent(sweep_id, function=train)
```

Retrieving the best model

```
api = wandb.Api()

sweep = api.sweep("{}/{}/{}".format(entity, project,
     sweep_id))
best_run = sweep.best_run()

# download the artifacts
for artifact in best_run.logged_artifacts():
  if artifact.type == "model":
    artifact_path = artifact.download()
    print(artifact_path)
```

WandB sweep monitoring

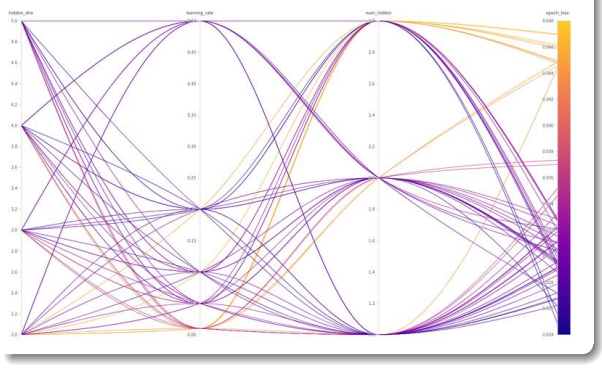


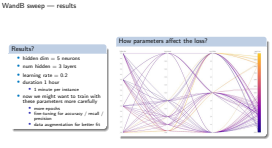
WandB sweep — results

Results?

- hidden dim = 5 neurons
- num hidden = 3 layers
- learning rate = 0.2
- duration 1 hour
  - 1 minute per instance
- now we might want to train with these parameters more carefully
  - more epochs
  - fine-tuning for accuracy / recall / precision
  - data augmentation for better fit

How parameters affect the loss?





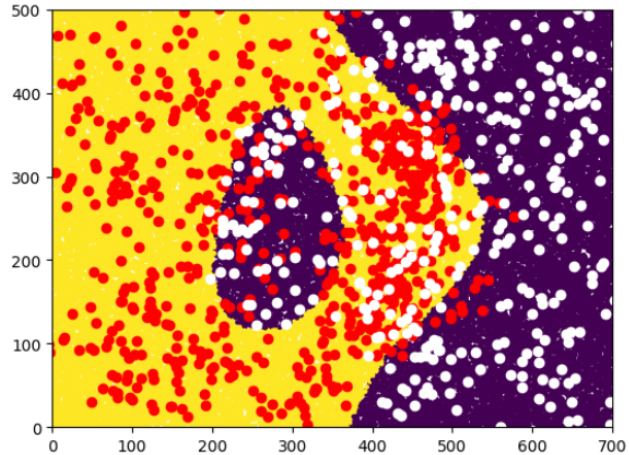
After automated sweep

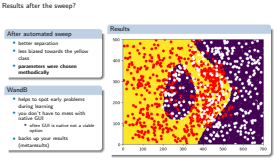
- better separation
- less biased towards the yellow class
- **parameters were chosen methodically**

WandB

- helps to spot early problems during learning
- you don't have to mess with native GUI
  - often GUI is native not a viable option
- backs up your results (metaresults)

Results





PyTorch tools?

Common Layers

- Linear (a.k.a. fully-connected)
- Convolutional (signal processing, image processing)
- Pooling (reduce dimensionality)
- Activation (Sigmoid, ReLU, Tanh)

Optimizers

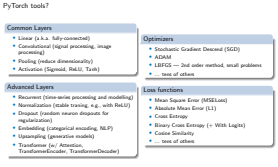
- Stochastic Gradient Descend (SGD)
- ADAM
- LBFGS — 2nd order method, small problems
- ... tens of others

Advanced Layers

- Recurrent (time-series processing and modelling)
- Normalization (stable traning, e.g., with ReLU)
- Dropout (random neuron dropouts for regularization)
- Embedding (categorical encoding, NLP)
- Upsampling (generative models)
- Transformer (w/ Attention, TransformerEncoder, TransformerDecoder)

Loss functions

- Mean Square Error (MSELoss)
- Absolute Mean Error (L1)
- Cross Entropy
- Binary Cross Entropy (+ With Logits)
- Cosine Similarity
- ... tens of others



Models

- models for variety all the common frameworks
- models from notable creators (Google, Meta, Microsoft, OpenAI)
- examples for running the models
- many distributed using Huggingface's own python library
- tracking of finetunned models
- notable models:
  - OpenAI's GPT
  - Google's OWL
  - Meta's Llama

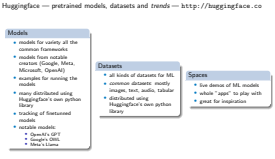
Datasets

- all kinds of datasets for ML
- *common datasets*: mostly images, text, audio, tabular
- distributed using Huggingface's own python library

Spaces

- live demos of ML models
- whole "apps" to play with
- great for inspiration

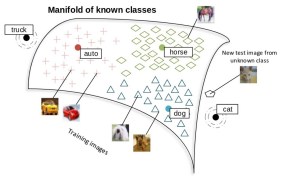
Lecture 3: Deep learning engineering basics with PyTorch
Hugging-face
Huggingface — pretrained models, datasets and *trends* — `http://huggingface.co`
2025-10-11



Try using a model via Huggingface

google/owlv2-large-patch14

- zero-shot open vocabulary detector
  - can recognize objects it was not trained on
- will fit in *normal-ish* GPU
- similar mechanism as noted in **lecture 01**



Jupyter Notebook

The following part of the lecture is accompanied by

06_huggingface_owl.ipynb

```python
import requests
from PIL import Image
import torch

from transformers import Owlv2Processor, Owlv2ForObjectDetection

processor = Owlv2Processor.from_pretrained("google/owlv2-large-patch14")
model = Owlv2ForObjectDetection.from_pretrained("google/owlv2-large-patch14")

url = "http://images.cocodataset.org/val2017/000000039769.jpg"
image = Image.open(requests.get(url, stream=True).raw)
texts = [["a photo of a cat", "a photo of a dog"]]
inputs = processor(text=texts, images=image, return_tensors="pt")

with torch.no_grad():
  outputs = model(**inputs)

# Target image sizes (height, width) to rescale box predictions [batch_size, 2]
target_sizes = torch.Tensor([image.size[::-1]])
# Convert outputs (bounding boxes and class logits) to Pascal VOC Format (xmin, 
results = processor.post_process_object_detection(outputs=outputs, target_sizes=t
i = 0  # Retrieve predictions for the first image for the corresponding text que
text = texts[i]
boxes, scores, labels = results[i]["boxes"], results[i]["scores"], results[i]["la
for box, score, label in zip(boxes, scores, labels):
    box = [round(i, 2) for i in box.tolist()]
    print(f"Detected {text[label]} with confidence {round(score.item(), 3)} at lo
```

https://huggingface.co/google/owlv2-large-patch14#use-with-transformers

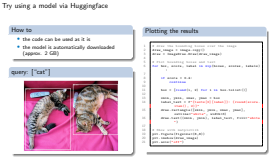
How to

- the code can be used as it is
- the model is automatically downloaded (approx. 2 GB)

query: ["cat"]

Plotting the results

```python
# draw the bounding boxes over the image
draw_image = image.copy()
draw = ImageDraw.Draw(draw_image)

# Plot bounding boxes and text
for box, score, label in zip(boxes, scores, labels):

    if score < 0.4:
        continue

    box = [round(i, 2) for i in box.tolist()]

    xmin, ymin, xmax, ymax = box
    label_text = f"{texts[0][label]}: {round(score.item(), 2)}"
    draw.rectangle([xmin, ymin, xmax, ymax], outline="white", width=3)
    draw.text((xmin, ymin), label_text, fill="white")

# Show with matplotlib
plt.figure(figsize=(8,8))
plt.imshow(draw_image)
plt.axis("off")
```

Try using a model via Huggingface

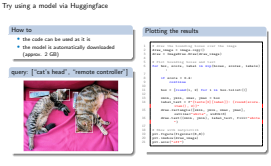
How to

- the code can be used as it is
- the model is automatically downloaded (approx. 2 GB)

query: ["cat's head", "remote controller"]

Plotting the results

```
1  # draw the bounding boxes over the image
2  draw_image = image.copy()
3  draw = ImageDraw.Draw(draw_image)
4
5  # Plot bounding boxes and text
6  for box, score, label in zip(boxes, scores, labels)
       :
7
8      if score < 0.4:
9          continue
10
11     box = [round(i, 2) for i in box.tolist()]
12
13     xmin, ymin, xmax, ymax = box
14     label_text = f"{texts[0][label]}: {round(score.
           item(), 2)}"
15     draw.rectangle([xmin, ymin, xmax, ymax],
           outline="white", width=3)
16     draw.text((xmin, ymin), label_text, fill="white
           ")
17
18 # Show with matplotlib
19 plt.figure(figsize=(8,8))
20 plt.imshow(draw_image)
21 plt.axis("off")
```

Try using a model via Huggingface

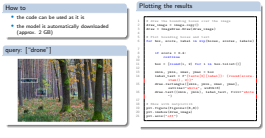
How to

- the code can be used as it is
- the model is automatically downloaded (approx. 2 GB)

query: [“drone”]

Plotting the results

```
# draw the bounding boxes over the image
draw_image = image.copy()
draw = ImageDraw.Draw(draw_image)

# Plot bounding boxes and text
for box, score, label in zip(boxes, scores, labels)
      :

    if score < 0.4:
        continue

    box = [round(i, 2) for i in box.tolist()]

    xmin, ymin, xmax, ymax = box
    label_text = f"{texts[0][label]}: {round(score.
          item(), 2)}"
    draw.rectangle([xmin, ymin, xmax, ymax],
          outline="white", width=3)
    draw.text((xmin, ymin), label_text, fill="white
          ")

# Show with matplotlib
plt.figure(figsize=(8,8))
plt.imshow(draw_image)
plt.axis("off")
```

Thanks for your attention

Lecture 3: Deep learning engineering basics with PyTorch
Hugging-face
Conclusion
2025-10-11
Conclusion
Thanks for your attention