AI Intro Basics #1

Neural Networks, Work Environment and a First Example

Roland Potthast, Stefanie Hollborn and Jan Keller

EUMETNET

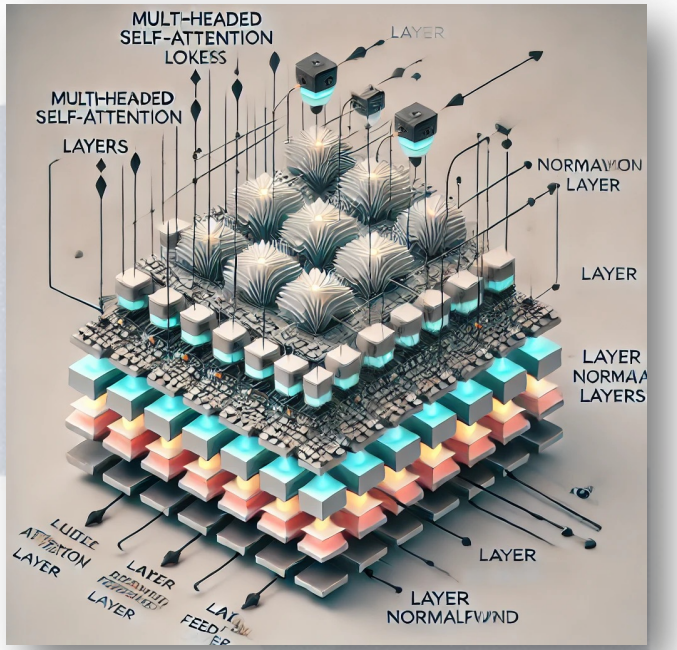
E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

1.1

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Downscaling via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]



AI generated Images,
© Roland Potthast 2024

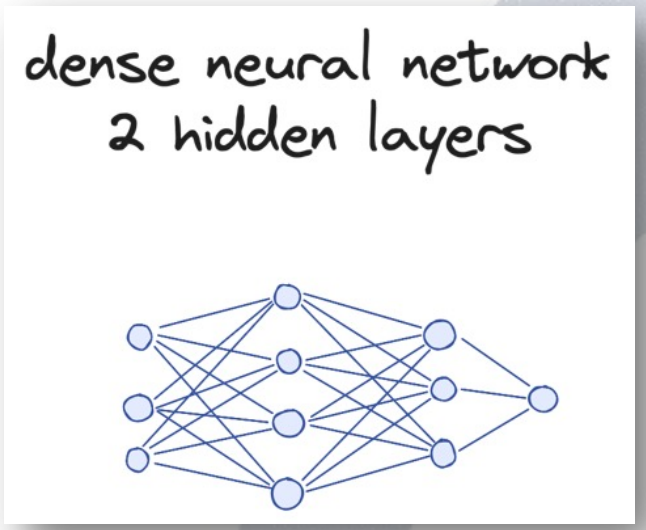
EUMETNET

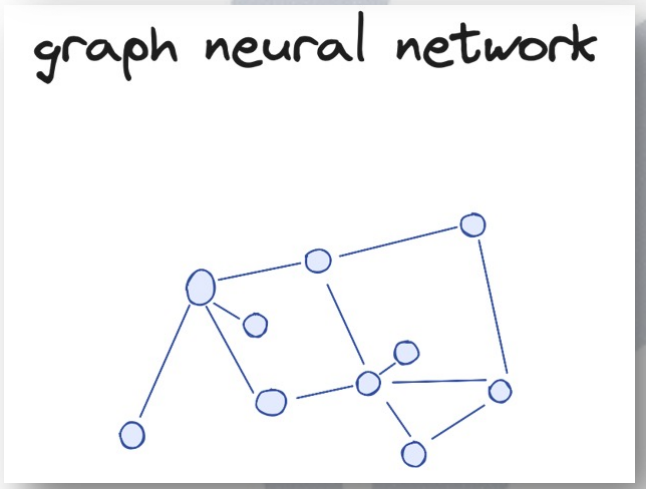
Neurons, Networks and Artificial Intelligence

Neurons are specialized cells in the brain that transmit information via electrical signals, forming neural networks that process tasks like learning and memory.

These networks inspire artificial neural networks in AI, enabling machines to recognize patterns, learn from data, and make decisions, simulating aspects of human intelligence.

Tools like PyTorch are widely used to build and train these neural networks, making it easier for developers to create sophisticated AI models.

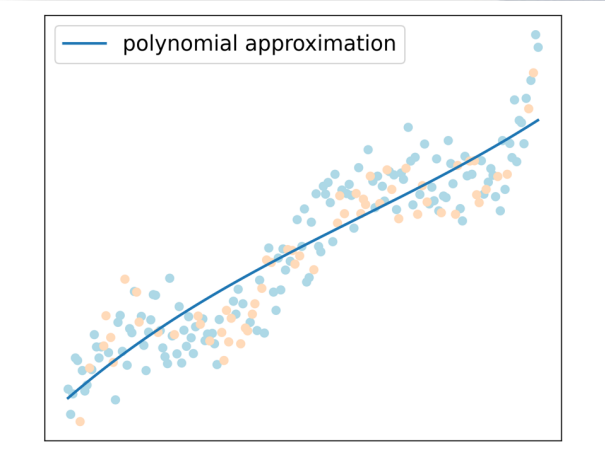




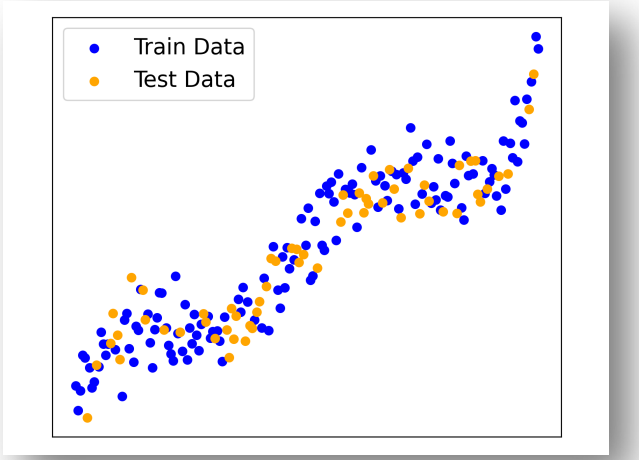
In a neural network, a signal is passed through layers of interconnected neurons, starting from the input layer, where data is provided. Each neuron processes the signal by applying weights and an activation function, then passes the output to the next layer, until it reaches the final output layer, where a prediction or decision is made.

NN Approximating Complex Mappings

Data and processes can be modeled using **mathematical functions**. To do this, the functions are trained on a set of **training data** and evaluated on a separate set of **test data**. During the training process, the parameters of the functions are adjusted to best fit and describe the data.

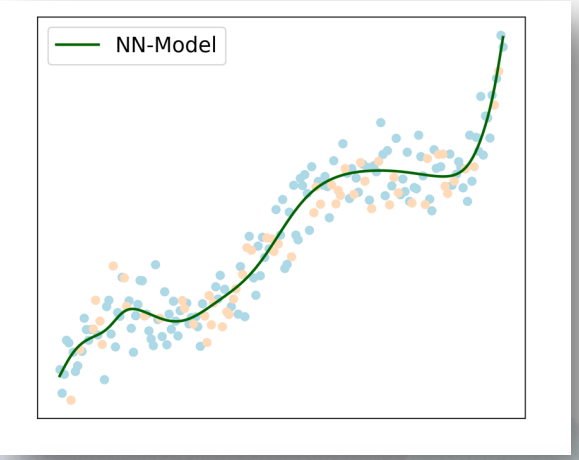


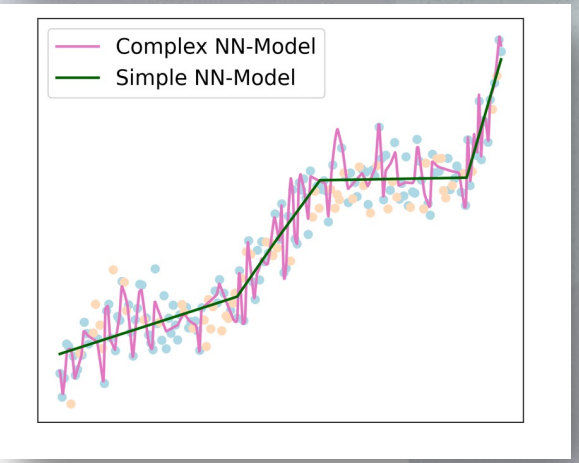
The quality of the approximation depends on the quality of the **training data**, the chosen **architecture**, and the **training process**.



Data can be approximated using various types of functions and methods, such as polynomials or trigonometric functions.

Neural networks, however, offer a highly flexible and powerful class of ansatz functions for tackling complex **approximation tasks**.





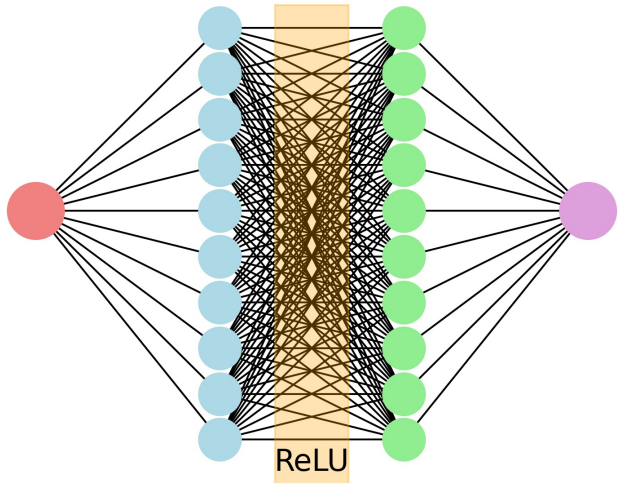
EUMETNET

Feed Forward Networks

How it works

In broad terms, neural network architecture refers to how neurons (the basic units) are arranged and connected in layers to process data. At its core, a neural network consists of three main types of layers.

Simple NN-Model with input, two hidden layers, and output

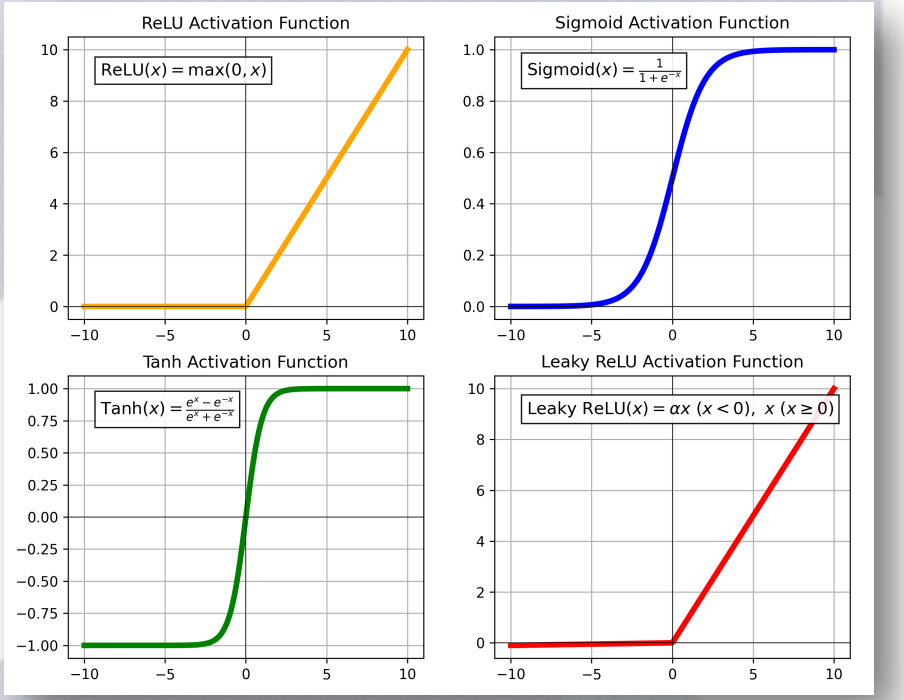


Input Layer:

This is where data is fed into the network. Each node in the input layer represents a feature or dimension of the input data (e.g., pixel values for an image or words in a sentence).

Hidden Layers:

These layers sit between the input and output layers and are responsible for learning patterns in the data. Each neuron in a hidden layer receives inputs from neurons in the previous layer, processes this information by applying weights and biases, and then passes the results through an **activation function** introducing nonlinearity. Networks can have multiple hidden layers, which allow them to learn more complex patterns.



Output Layer:

This is the final layer where predictions or classifications are made. The number of neurons in this layer typically corresponds to the number of classes in a classification task or the size of the predicted output in a regression task.

EUMETNET

How do Networks look like?

Tensors are the fundamental data structures used in machine learning frameworks like PyTorch and TensorFlow, serving as multi-dimensional arrays that can represent data of various dimensions.
They are essentially generalized matrices that can hold numbers in any number of dimensions, enabling the representation of everything from simple scalars (0D), vectors (1D), and matrices (2D) to more complex, high-dimensional structures (e.g., 3D or 4D tensors).

Basic Tensor Structure:

1. **Scalars (0D)**: A single number, like a basic data point (e.g., 3 or 5.5).
2. **Vectors (1D)**: A one-dimensional array of numbers (e.g., a list like [1,2,3][1, 2, 3][1,2,3]).
3. **Matrices (2D)**: A two-dimensional grid of numbers (e.g., a table or grid of values, like a spreadsheet).
4. **Higher-Dimensional Tensors (3D and above)**: Tensors can have more than two dimensions, representing more complex structures like sequences of matrices or batches of images (e.g., 4D tensors can represent a batch of colored images, where dimensions might be: batch size, height, width, and color channels).

```
# Vector (1-dimensional tensor)
vector = torch.tensor([1.0, 2.0, 3.0])
print(vector)
```

```
tensor([1., 2., 3.])
```

```
# 3D Tensor (tensor of matrices)
tensor_3d = torch.ones((2, 3, 4))  # Two 3x4 matrices
print(tensor_3d)
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

```
# Create a tensor with requires_grad=True to track gradients
x = torch.tensor(2.0, requires_grad=True)

# Perform some operations
y = x**2 + 3*x + 1

# Backpropagate to compute gradients
y.backward()

# Print the gradient (dy/dx)
print(x.grad)  # Output will be dy/dx = 2*x + 3
```

```
tensor(7.)
```

Tools at your Fingertips

In **PyTorch**, tensors are the core structure used for building and training neural networks. PyTorch's flexibility with tensors allows developers to easily handle different types of data. Tensors in PyTorch are optimized to run efficiently on both CPUs and GPUs, enabling fast computation of neural networks.

Automatic Differentiation:

PyTorch supports automatic differentiation using its autograd feature. This means that during neural network training, PyTorch computes the gradients of tensors automatically with respect to a loss function, which helps in optimizing the model.

Broadcasting:

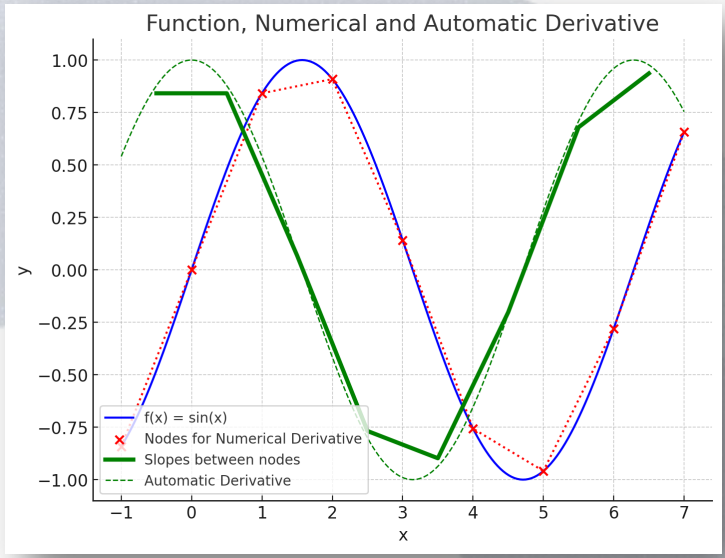
PyTorch supports broadcasting, which allows it to handle tensors of different shapes in operations like addition or multiplication by automatically expanding them to a compatible shape.

Multi-Dimensional Tensors:

In more advanced scenarios, tensors in PyTorch can represent complex data types like batches of images (4D tensors with dimensions: batch size, number of channels, height, and width) or even sequences of data in time (e.g., 3D tensors for recurrent networks with dimensions representing batch size, sequence length, and feature size).

Efficient GPU Utilization:

PyTorch provides seamless support for moving tensors to GPUs for faster computation. Tensors can be moved between the CPU and GPU using .to(device) methods, making large-scale computations and deep learning models more efficient.

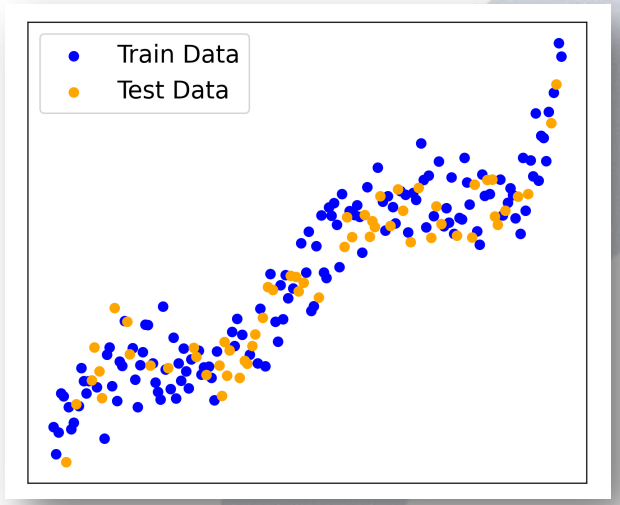


EUMETNET

Training to fit – do not overfit!

Training is the process of teaching a model to make accurate predictions by adjusting its parameters. Data is processed in smaller groups called **batches**, with the number of samples in each batch determined by the **batch size**. A complete pass through the whole training dataset is called an **epoch**.

During a **forward pass**, the model processes input data through its layers to generate and output. The difference between the predicted outcome and the actual result is measured using a **loss function**, which quantifies the error. The goal is to minimize this error over time.

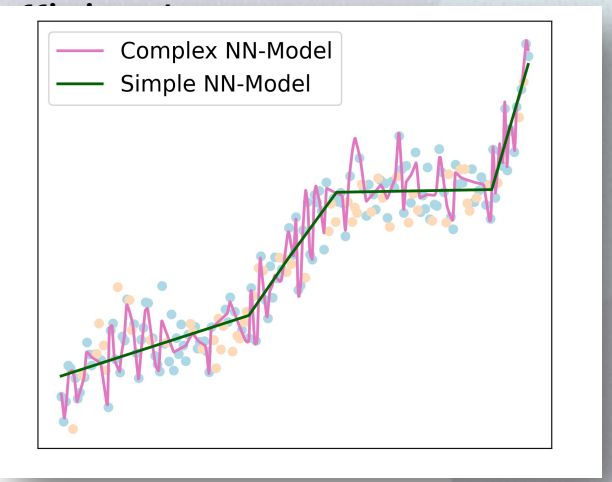


The reduce the error, the model uses **backpropagation**, where the loss is propagated backward through the network to update the model's weights. This process is repeated typically after each batch and repeated over many epochs.

The fully trained model can perform **inference**, making predictions on new, unseen data.

Training data need to be diverse, representative and sufficiently large to capture the underlying patterns of the task.

During training, **overfitting** can occur if the model becomes to specialized in the training data, learning even the noise and the irrelevant patterns. **Non-convergence** in neural network training occurs when the model fails to learn from the data, meaning that the loss function does not decrease

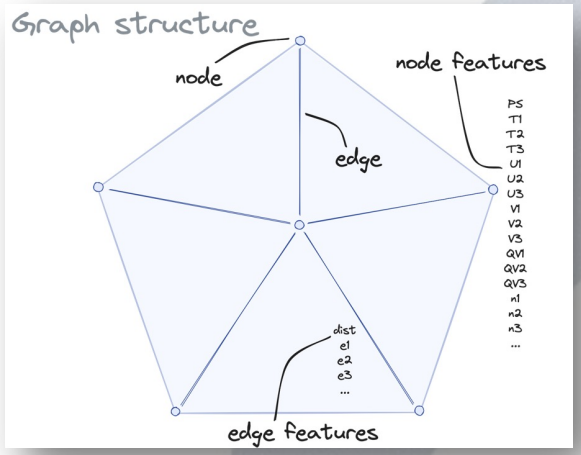


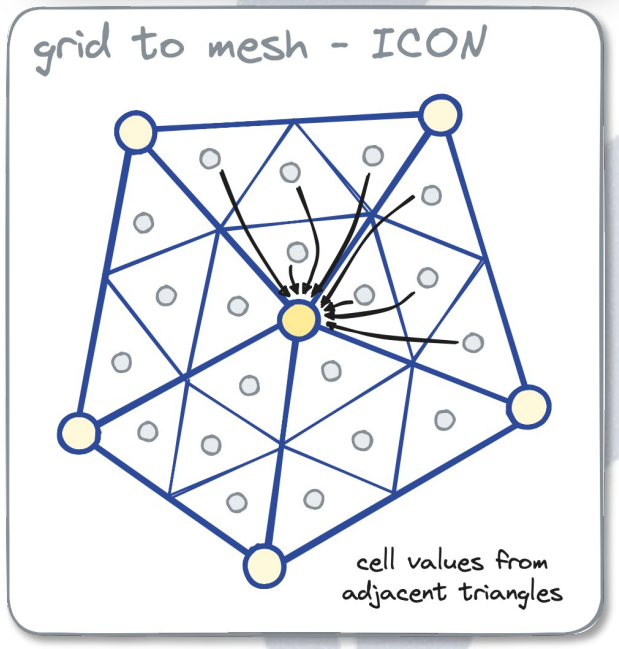
Graph Neural Networks – Basic Idea 1/2

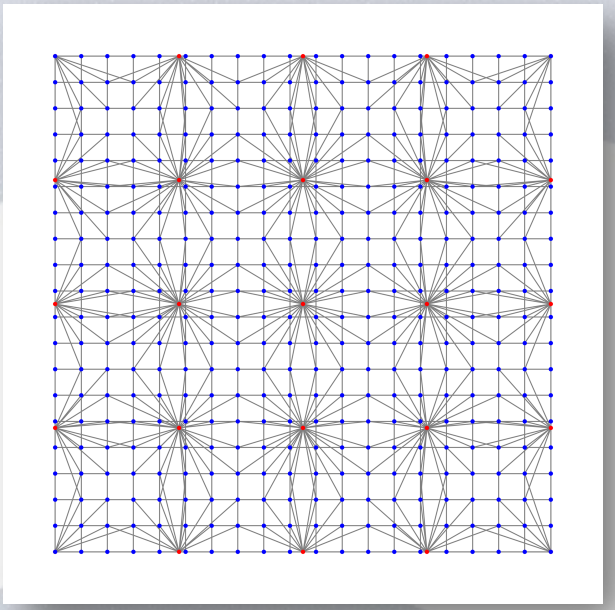
Graph Neural Networks (GNNs) are a type of neural network designed to work on graph-structured data, where entities (nodes) are connected by relationships (edges). Unlike traditional neural networks, GNNs use graph structures to capture both the features of individual nodes and their connectivity within the graph.

By passing information between nodes through edges, GNNs allow for the modelling of complex relationships, making them effective for tasks like social network analysis, molecular modelling, and recommendation systems.

GNNs leverage techniques like message passing to aggregate and update node representations based on their neighbours.







To construct a graph, start by defining its **nodes (vertices)**, which represent individual entities in your data. Then, establish **edges** between these nodes, representing the relationships or connections between the entities, forming the structure of the graph.

Graph Neural Networks – Basic Idea 2/2

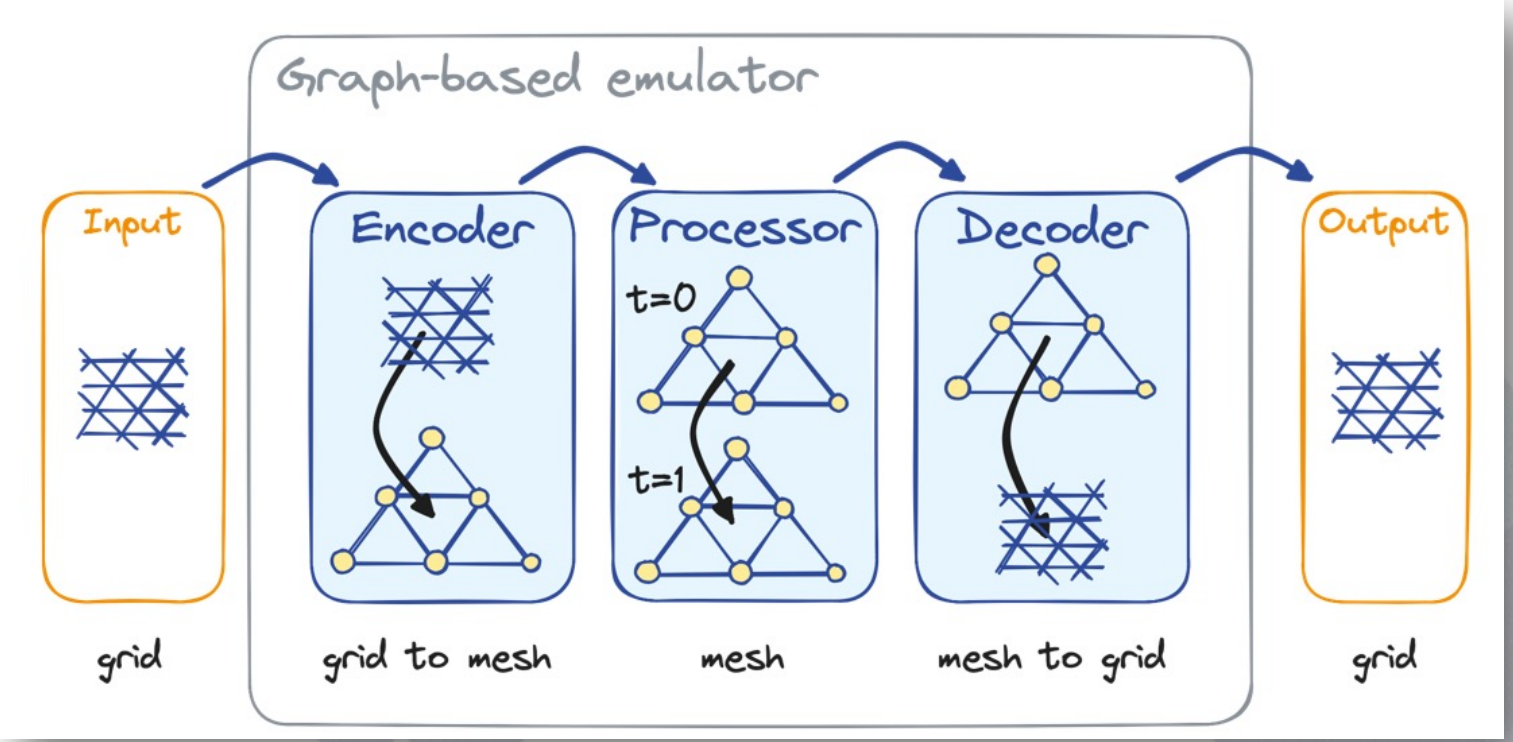
In graph-based models using an encoder, processor, and decoder, the **encoder** transforms raw graph data, such as node features and edge connections, into an initial set of embeddings that represent the graph's structure and features.

The **processor** (often a Graph Neural Network) then iteratively updates these embeddings by passing messages between connected nodes, refining their representations based on neighbouring nodes and edges.

After processing, the **decoder** takes the updated embeddings and generates a final output, which could be a prediction about node classification, edge prediction, or a graph-level task. This framework is widely used in applications like molecular property prediction, knowledge graphs, and social network analysis.

To learn more about Graph Neural Networks (GNNs), it's helpful to start by understanding how **message passing** works between nodes, where information is shared across connected nodes in the graph.

You should also explore **basic GNN applications**, like node classification and link prediction, which are common tasks in social networks and recommendation systems.



Transformer Networks 1/2

Transformer neural networks are a type of deep learning model that perform extremely well at processing sequential data, such as language, by using **self-attention mechanisms**. Thereby, the model decides how much **focus** to place on different words in a sentence when making predictions or understanding the context.

Unlike traditional models, transformers can handle input data in parallel rather than sequentially, making them more efficient and scalable and improving performance in tasks like natural language processing (**NLP**), machine translation, and text summarization.

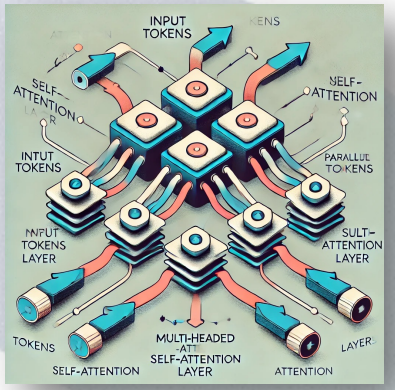
Transformers have become the foundation for many state-of-the-art models, including **BERT** and **GPT**, which are widely used in modern AI applications.

Here is a simple example sentence with token representations for a Transformer neural network:

- **Sentence**: "The cat sat on the mat."
- **Token representation**:
  - Token 1: "The"
  - Token 2: "cat"
  - Token 3: "sat"
  - Token 4: "on"
  - Token 5: "the"
  - Token 6: "mat"

Each of these **tokens** will be transformed into **embeddings** (the word's numeric representations) that will interact with each other through the Query (Q), Key (K), and Value (V) mechanisms during the self-attention process to determine relationships between words, like how "cat" and "sat" are closely related in this sentence.

•**Q (Query)**: Represents the current token (word or part of the input) that is being processed.
•**K (Key)**: Vector used to compare against all other tokens to find relevance to the current token.
•**V (Value)**: Contains information from the tokens that are deemed important or relevant.



Each input token interacts with the others based on these Q, K, and V values

EUMETNET

Recommendations Basics 1/3

Understand the Basics:

Begin by studying the fundamental concepts such as how neurons function, how layers are structured, and what activation functions do.

Grasping these core ideas will make more complex topics easier to understand later.

Take Interactive Courses:

Online learning platforms offer great tutorials that include both theory and practical coding exercises.

Look for courses that offer a mix of hands-on practice with programming and theoretical lessons on how neural networks operate.

Practice with Frameworks:

Using libraries like PyTorch or TensorFlow allows you to build and train neural networks.

Working through coding examples will help solidify your understanding of network architectures and training processes.

EUMETNET

Recommendations Basics 2/3

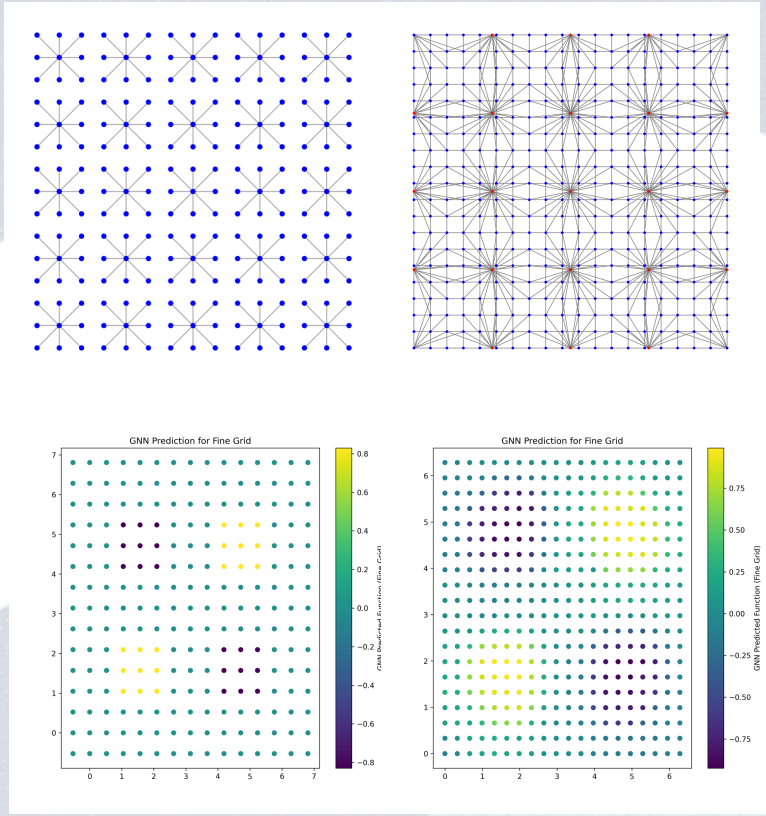
Work on Projects:

To implement neural networks for real-world tasks first break it down to generic examples.

Pilot Projects provide a deeper understanding of how to apply neural networks to solve problems, which is essential for mastering the subject.

Explore Advanced Architectures:

Once comfortable with the basics, study advanced models like Graph Neural Networks (GNNs) for working with graph-structured data and Transformers for handling sequential data and tasks such as language translation and text generation.



Recommendations Basics 3/3

Frameworks

In machine learning and artificial intelligence, frameworks such as

- TensorFlow,
- **PyTorch**, and
- Scikit-learn

provide essential tools for building, training, and deploying neural networks.

These frameworks streamline tasks like defining models, handling data, and running optimization algorithms, making the development process more efficient and accessible to researchers and developers.

Tools for machine learning extend beyond just building models; they also include platforms for

- **managing data** (e.g., Pandas or Xarray),
- **visualization** (e.g., Matplotlib, Cartopy), and
- **hyperparameter** tuning (e.g., Mlflow Tracking, Optuna, Hyperopt).

These tools help improve the accuracy and efficiency of model training and evaluation.

Environment

In a production environment, MLOps (Machine Learning Operations) is crucial for integrating machine learning into broader software and business workflows.

MLOps tools like Kubernetes, MLflow, and Kubeflow help manage the entire lifecycle of machine learning models, from development to deployment and monitoring.

They ensure that models can be scaled, updated, and maintained efficiently while tracking performance and preventing issues like model drift.

E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

1.2

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Downscaling via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

Portable AI Work Environment

Windows Computer

GPU Nodes

EWC

Linux Workstation

HPC Cluster

GPU Computer

Virtual Server (Cloud)

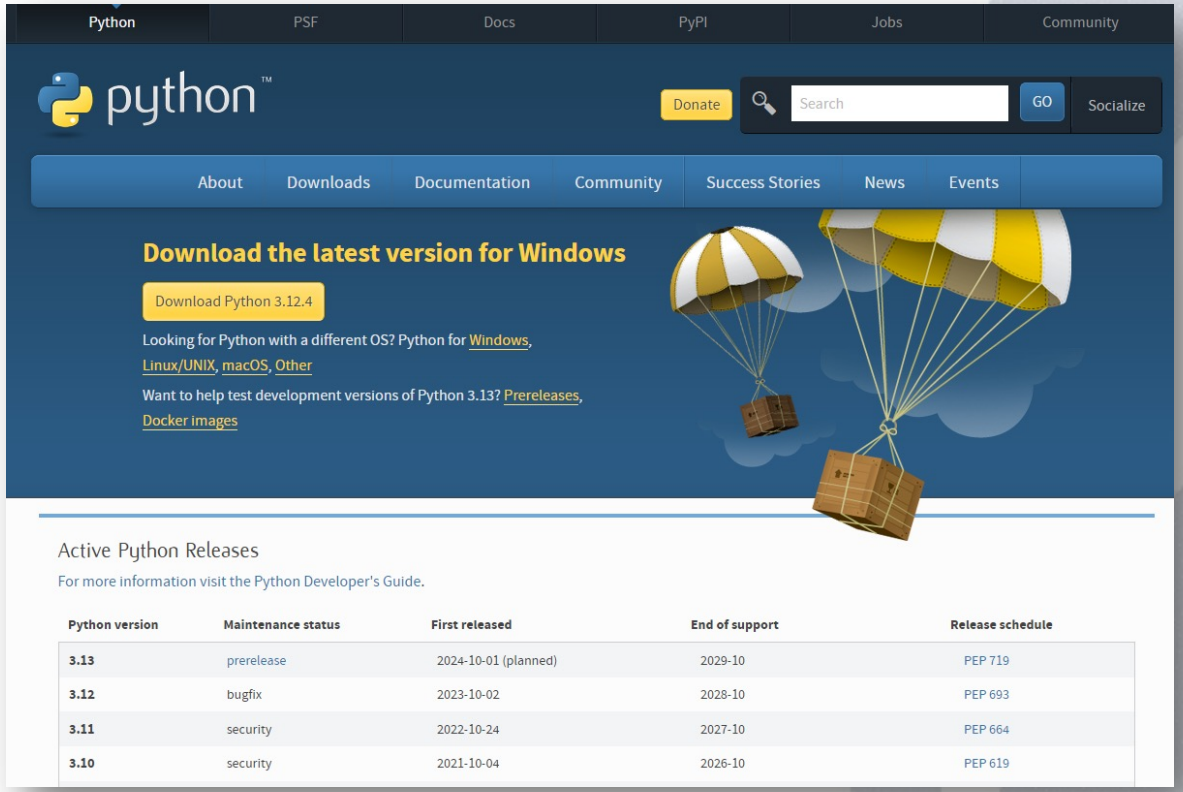
Google Colab

Recommendation: Carry out portable design, do not limit yourself to one working environment only!

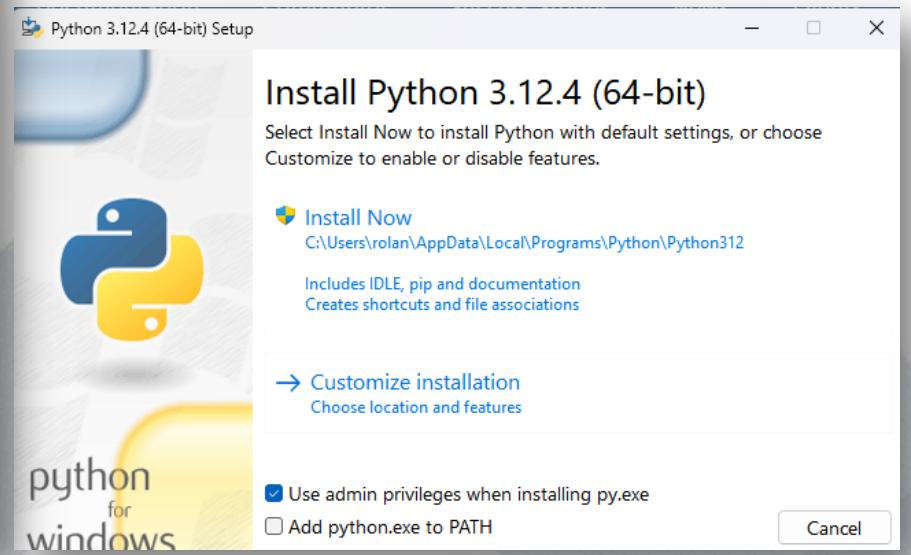
EUMETNET

Python for Windows

- Download Python from https://www.python.org/downloads/



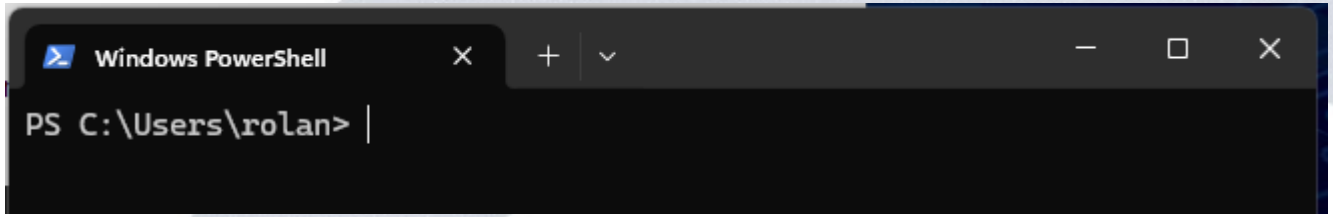
- Install Python,
  Klick Add to PATH!!



EUMETNET

Windows Powershell

- Open Windows Powershell



- To check if you installed a python version you might use

  > $env:Path -split ';' | Select-String -Pattern "python"

```
PS C:\Users\rolan> $env:Path -split ';' | Select-String -Pattern "python"

C:\Users\rolan\AppData\Local\Programs\Python\Python310\
C:\Users\rolan\AppData\Local\Programs\Python\Python310\Scripts
C:\Users\rolan\AppData\Local\Programs\Python\Python312\Scripts\
C:\Users\rolan\AppData\Local\Programs\Python\Python312\
```

- You can now already use python,
  by calling python in the form, leave by typing exit()

```
PS C:\Users\rolan\all> C:\Users\rolan\AppData\Local\Programs\Python\Python312\python
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

EUMETNET

Virtual Environments: Fast Installation

- We recommend to work in a virtual environment
- **This is the same on Windows, Linux, locally or on a virtual server in the cloud.**
- A Virtual Environment will be installed in a particular folder, e.g. ropy. This folder should be in a useful place, it does not need to be in your home directory!
- You install a virtual python environment based on a particular python version (please use python versions higher than 3.8 at least!!!) by the command:

```
PS C:\Users\rolan\all> C:\Users\rolan\AppData\Local\Programs\Python\Python312\python -m venv ropy312
```

- This will use python 3.12 and install it into the folder all\ropy312

EUMETNET

Virtual Environments: Check

- The virtual environment is now located in this folder:

```
PS C:\Users\rolan\all\ropy312\Scripts> ls


    Verzeichnis: C:\Users\rolan\all\ropy312\Scripts


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         27/07/2024     12:23           2050 activate
-a----         27/07/2024     12:23           1001 activate.bat
-a----         27/07/2024     12:23          26199 Activate.ps1
-a----         27/07/2024     12:23            393 deactivate.bat
-a----         27/07/2024     12:23         108404 pip.exe
-a----         27/07/2024     12:23         108404 pip3.12.exe
-a----         27/07/2024     12:23         108404 pip3.exe
-a----         27/07/2024     12:23         270104 python.exe
-a----         27/07/2024     12:23         258840 pythonw.exe
```

```
PS C:\Users\rolan\all\ropy312> ls


    Verzeichnis: C:\Users\rolan\all\ropy312


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         27/07/2024     12:23                Include
d-----         27/07/2024     12:23                Lib
d-----         27/07/2024     12:23                Scripts
-a----         27/07/2024     12:23            311 pyvenv.cfg


PS C:\Users\rolan\all\ropy312>
```

In the Scripts/ folder – which might be called bin/ under Linux, there is a programme called activate. When you call it from anywhere, it will activate this environment in the folder where you have called it.

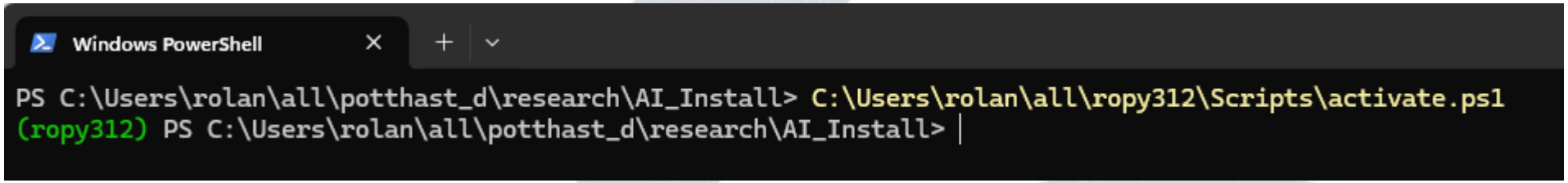
Check the python version by calling .\python –V in the Scripts/ or bin/ folder:

```
PS C:\Users\rolan\all\ropy312\Scripts> .\python -V
Python 3.12.4
```

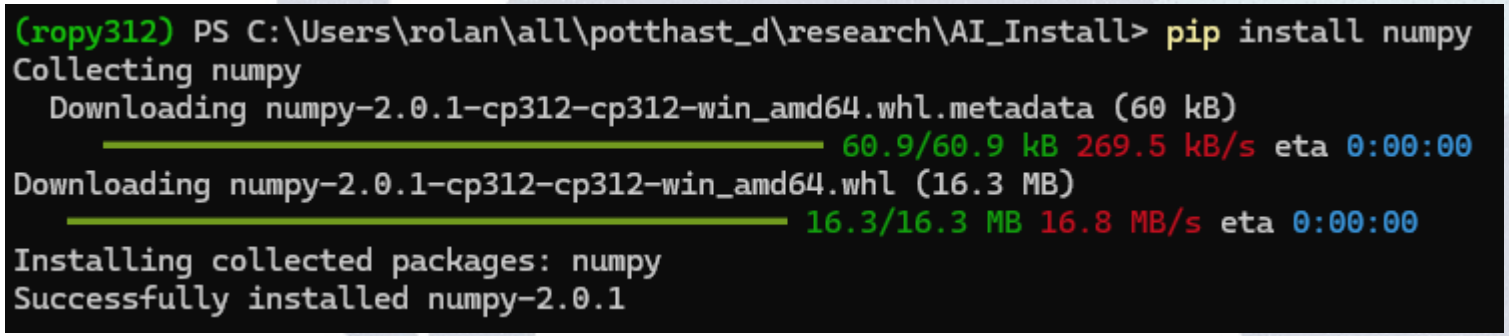
EUMETNET

Virtual Environments: Activation

- You activate the virtual environment in a folder by calling it. Under PowerShell you need to call the activate.ps1 version, under Linux or with relative path you can call just activate



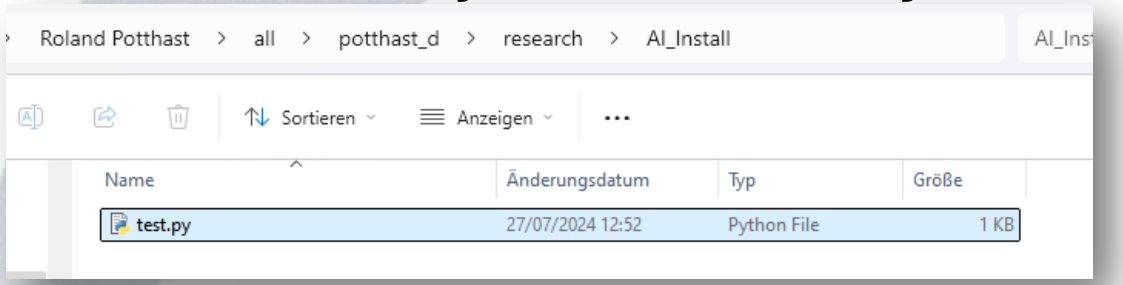
- You are now in a framework where you can install all packages you need based on the standard “pip” package installer of python. Start by installing “numpy”, the basic numerical package of python.



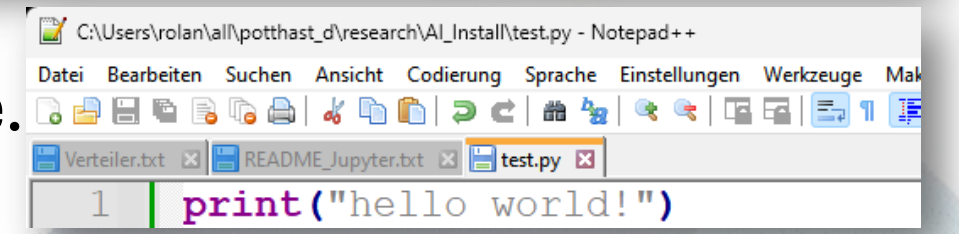
EUMETNET

Virtual Environment: Creating and Running Python Programmes

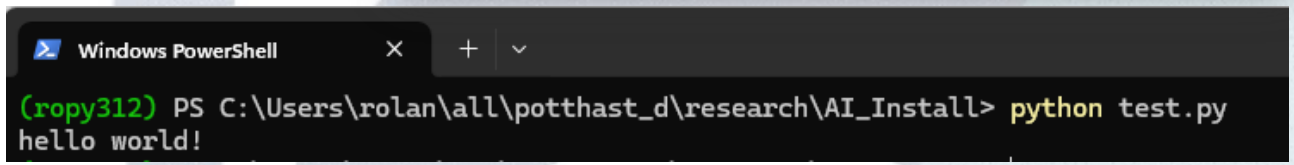
- You can now create a text file “test.py” under windows or linux in the current directory where you are and where you activated your virtual environment.



- Edit it with your favourite editor and write print(“hello world!”) into the file.



- Execute the python programme by calling
  > python test.py
  under your virtual environment.



EUMETNET

Jupyter Notebooks and Jupyter Lab

- We start by installing the package Jupyter, which is a very popular and common web-interface to your python framework. Type

> pip install jupyter

```
(ropy312) PS C:\Users\rolan\all\potthast_d\research\AI_Install> pip install jupyter
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl.metadata (995 bytes)
Collecting notebook (from jupyter)
  Downloading notebook-7.2.1-py3-none-any.whl.metadata (10 kB)
Collecting qtconsole (from jupyter)
```

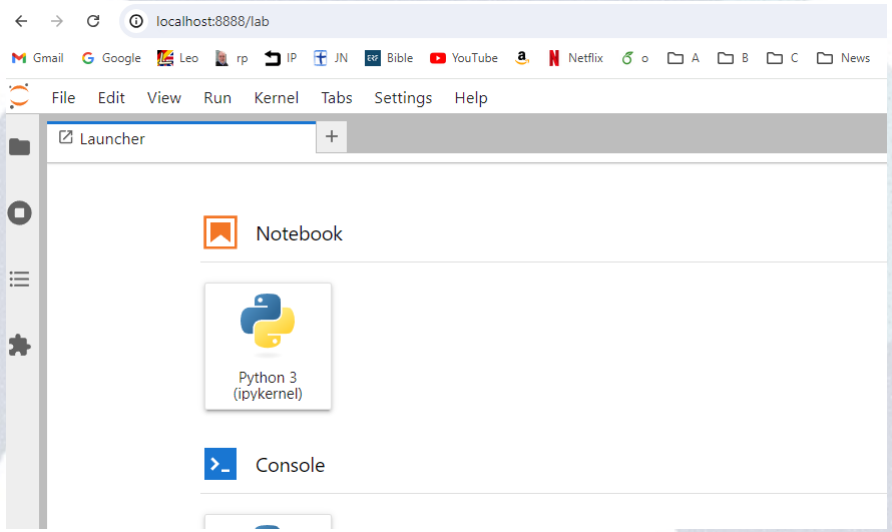
Then lauch jupyter lab by typing

> jupyter lab

```
(ropy312) PS C:\Users\rolan\all\potthast_d\research\AI_Install> jupyter lab
```

within your virtual environment.

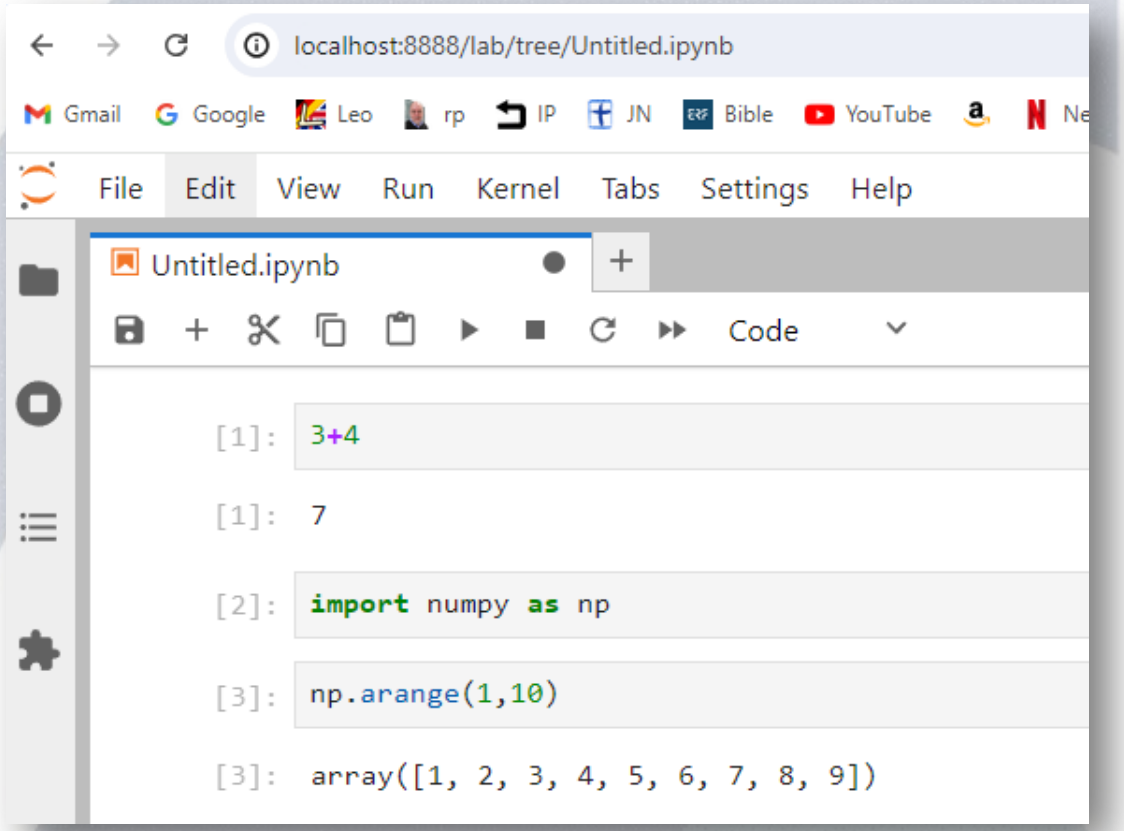
- It will open a browser window based on a local webserver which will be an user interface with your virtual python environment.



EUMETNET

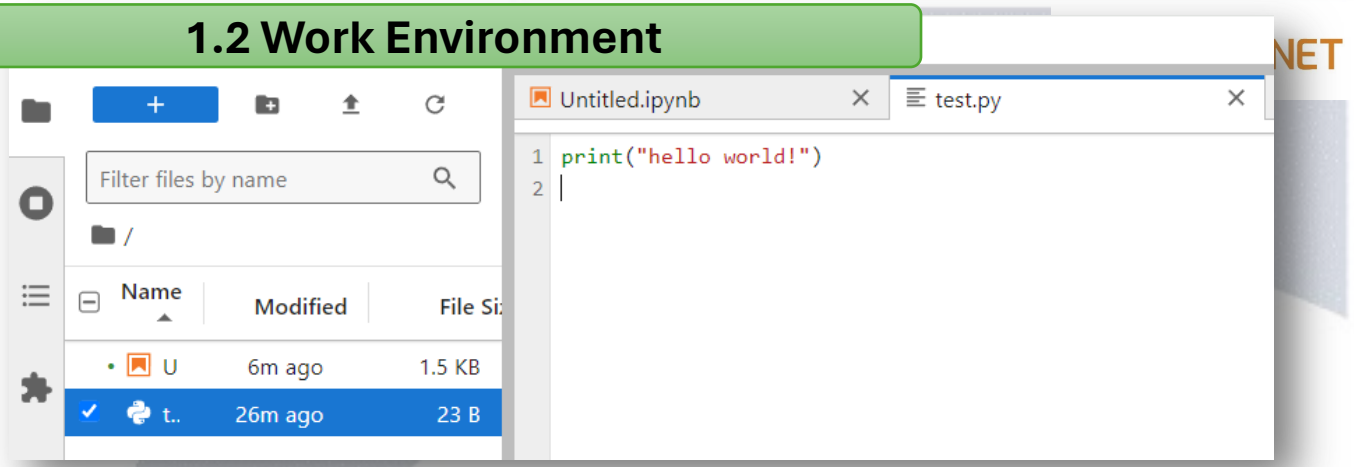
Virtual Environment: Using a Jupyter Notebook

- You can use python interactively typing into the cells.
- Packages are loaded and addressed by import ... as ..., for example the numpy package by > import numpy as np, a typical standard command.
- Cells are executed by shift-enter, when your curser is placed there.



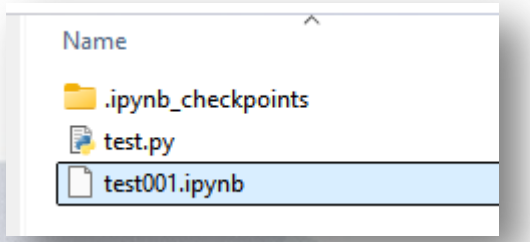
- You can now develop applications in python, usually you will consoult either consult online tutorials, websites or your preferred AI assistant.

Virtual Environment: Jupyter Lab

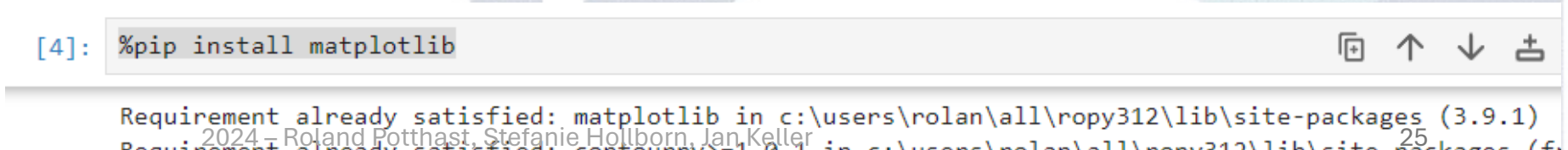


- Jupyter Lab also provides an Editor for your python codes
- Jupyter notebooks are saves as .ipynb files.
- Here, you can choose the pull-down menu to save the notebook as test001.ipynb
- Go back to your virtual environment and install the plotting library matplotlib by typing

> pip install matplotlib



In your jupyter notebook you can carry out such commands by typing
%pip install matplotlib
into a cell and executing that cell.

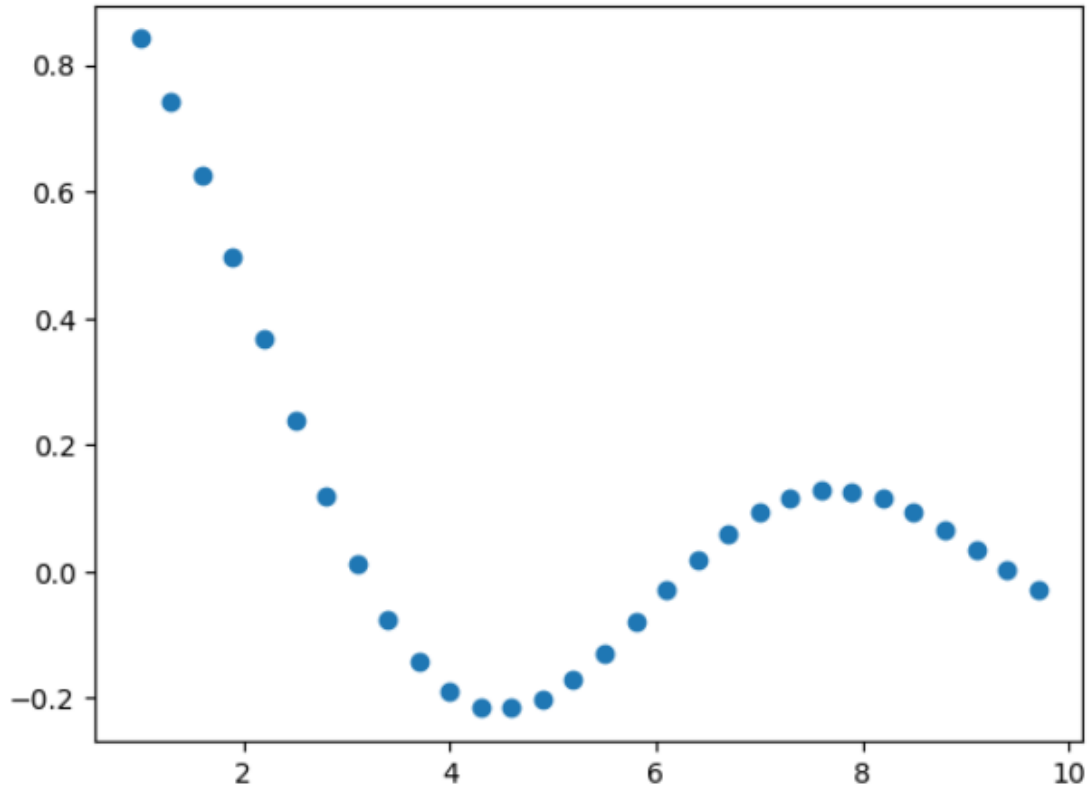


EUMETNET

Virtual Environment: Creating Visualizations

- Before going to AI applications, you need to be literate in standard python applications.
- Visualization based on matplotlib, e.g. the scatter() library function, and further plots, will be the basis for developing, checking and testing AI applications.
- There is an extensive amount of material on the Web and provided by AI assistants!

```
[2]: import numpy as np
     import matplotlib.pyplot as plt

[3]: x = np.arange(1,10,0.3)
     plt.scatter(x,1/x*np.sin(x))

[3]: <matplotlib.collections.PathCollection at 0x22e7f50af00>
```



EUMETNET

Python on Linux

- On linux, you approach might depend on your rights. Let us assume that you are a normal user and that there is python3.10 installed. Then, you can test this by typing > python –V
- You install your virtual environment the same way as described above.
- In your virtual environment you can install your packages.

1. numpy
2. matplotlib
3. jupyter
4. cartopy
5. eccodes

1. netcdf4
2. scikit-learn
3. torch
4. pyyaml

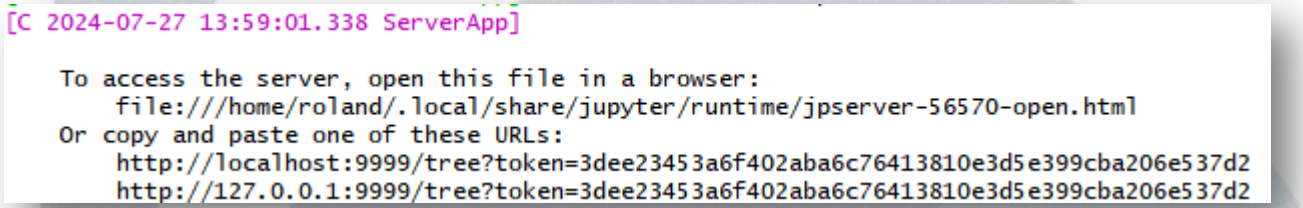
1. Tensorflow
2. keras

EUMETNET

Jupyter on a remote Linux computer accessed with Windows Browser

- You can run your python environment on some remote Linux computer, but still carry out all interaction with your local Windows browser.
- To this end install some shell on windows, for example the git shell, which has all you need: https://git-scm.com/download/win
- Make sure you can login to the remote Linux computer, for example using its IP address, e.g. 192.168.178.79. You can find this address by typing ifconfig (Linux) or ipconfig (Windows) on the computer.
- To this end you start your jupyter notebook on the Linux computer without browser on some port by
  > jupyter notebook --no-browser --port=9999 &

The output will also give you a token necessary to establish the connection.



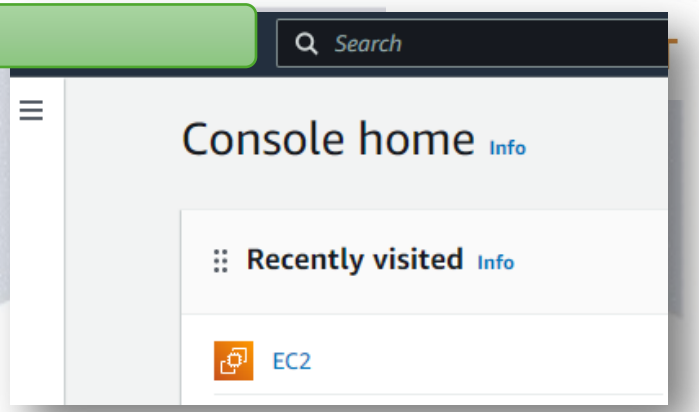
Then setup a connection from a local shell on the Windows computer by port forwarding

> ssh -N -f -L localhost:9997:localhost:9999 roland@192.178.168.79

- And in your Windows browser use the local port 9997 and the token:

http://localhost:9997/tree?token=3dee23453a6f402aba6c76413810e3d5e399cba206e537d2

Python on a Virtual Server in the Cloud



- Here we describe how to setup your python environment on an AWS EC2 Linux Server.
- You need an AWS account. Then create an instance of a server.
- It will create or use a .pem key for you to access the server. Download it, store e.g. in ~/all/ropy312/keys/testAWS.pem.
- AWS will also tell you the public ip of the server, e.g. 52.59.241.188
- Then you can access the server by

> ssh -i ~/all/ropy312/keys/testAWS.pem ec2-user@52.59.241.188

- Now, you can carry out all steps as for the above windows or linux servers, i.e. python3 –V
- You can setup python, torch and access your environment by jupyter lab through port forwarding.

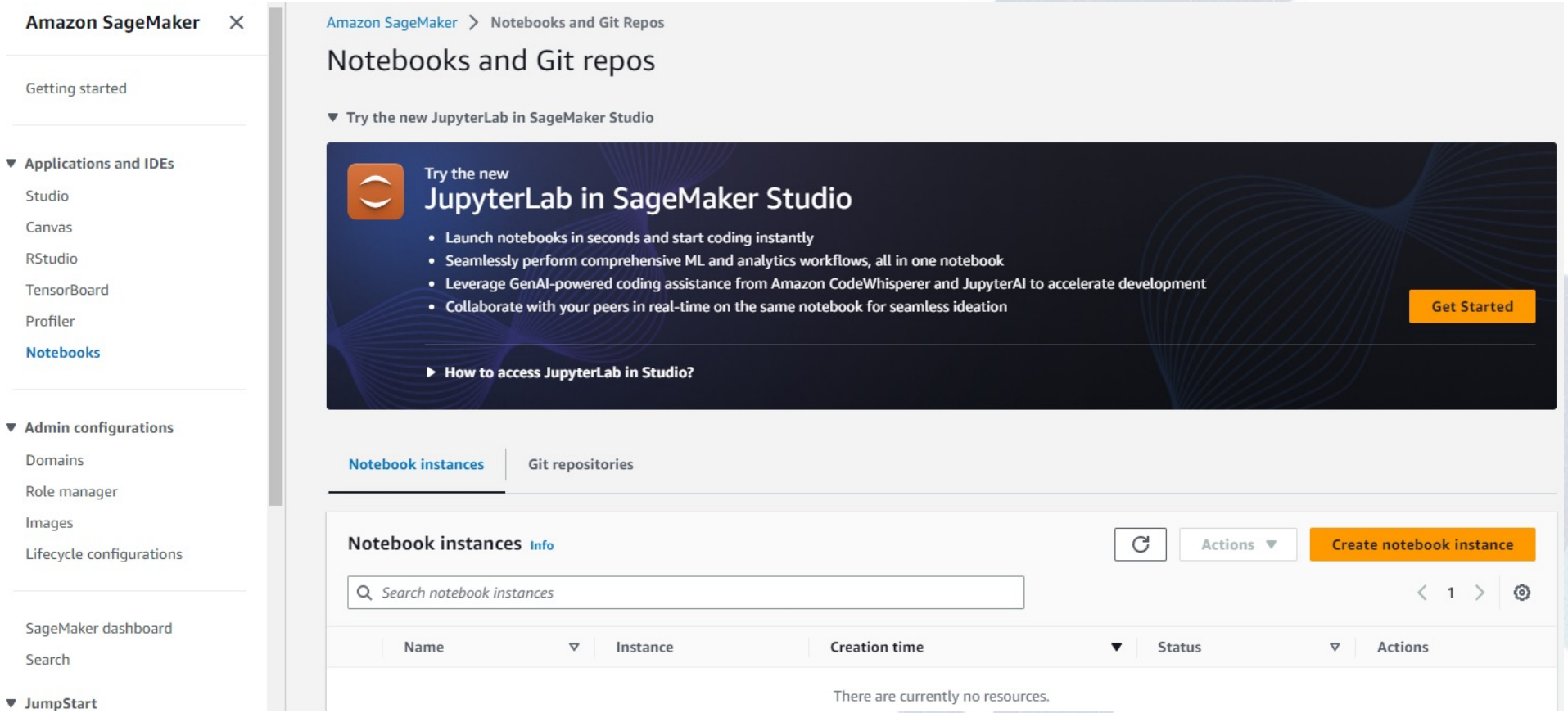
```
rolan@White-WIN MINGW64 ~/all/potthast_d/research/AI_Install (master)
$ ssh -i ~/all/ropy312/keys/testAWS.pem ec2-user@52.59.241.188
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
Last login: Sat Jul 27 14:32:00 2024 from 91.13.156.227
[ec2-user@ip-172-31-35-45 ~]$
```

EUMETNET

Python AI by AI Ready Tools in the Cloud

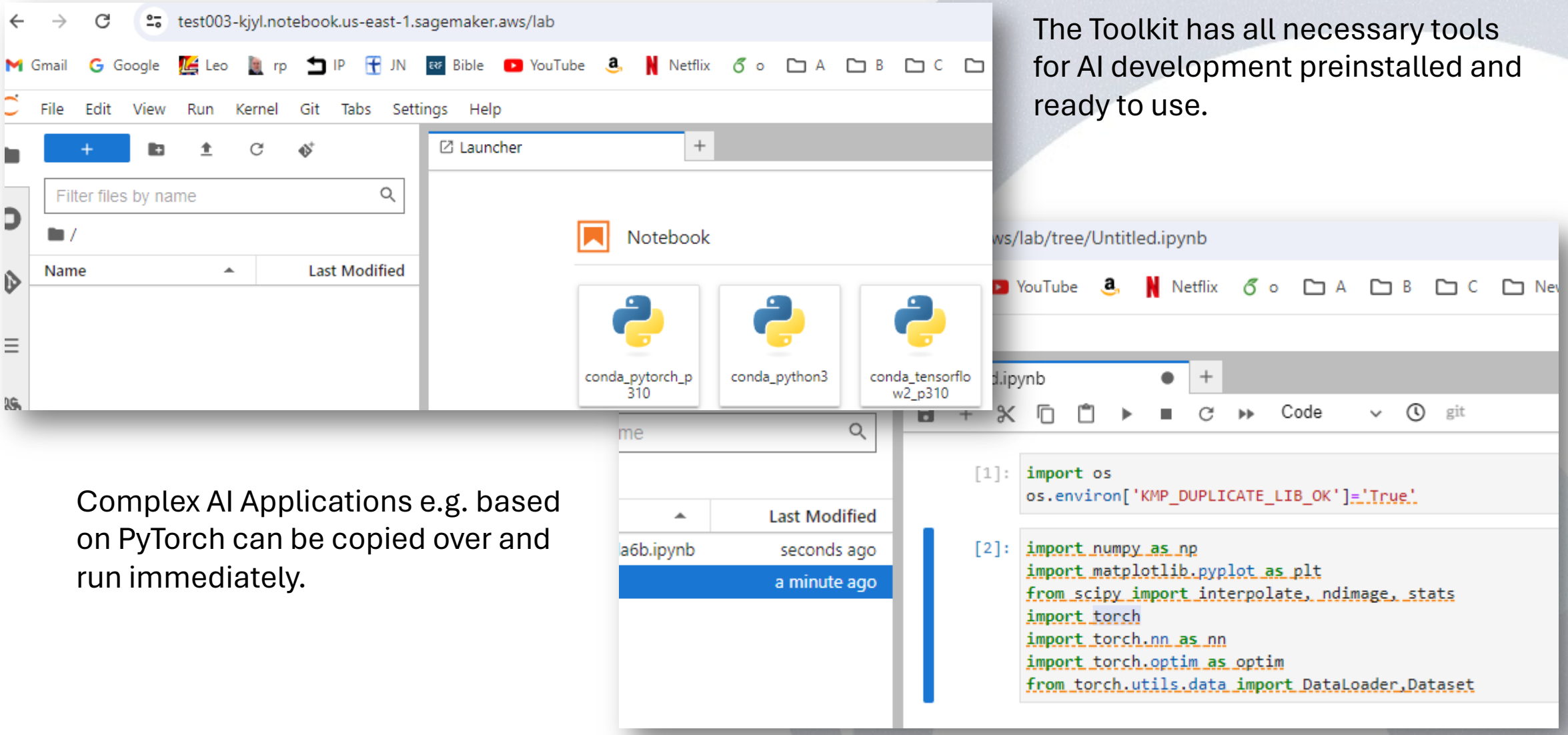
AI Ready Jupyter Notebook in 3 Klicks

AWS provides specialized AI enabled packages, where you can directly access jupyter lab or jupyter notebooks with installed torch packages



EUMETNET

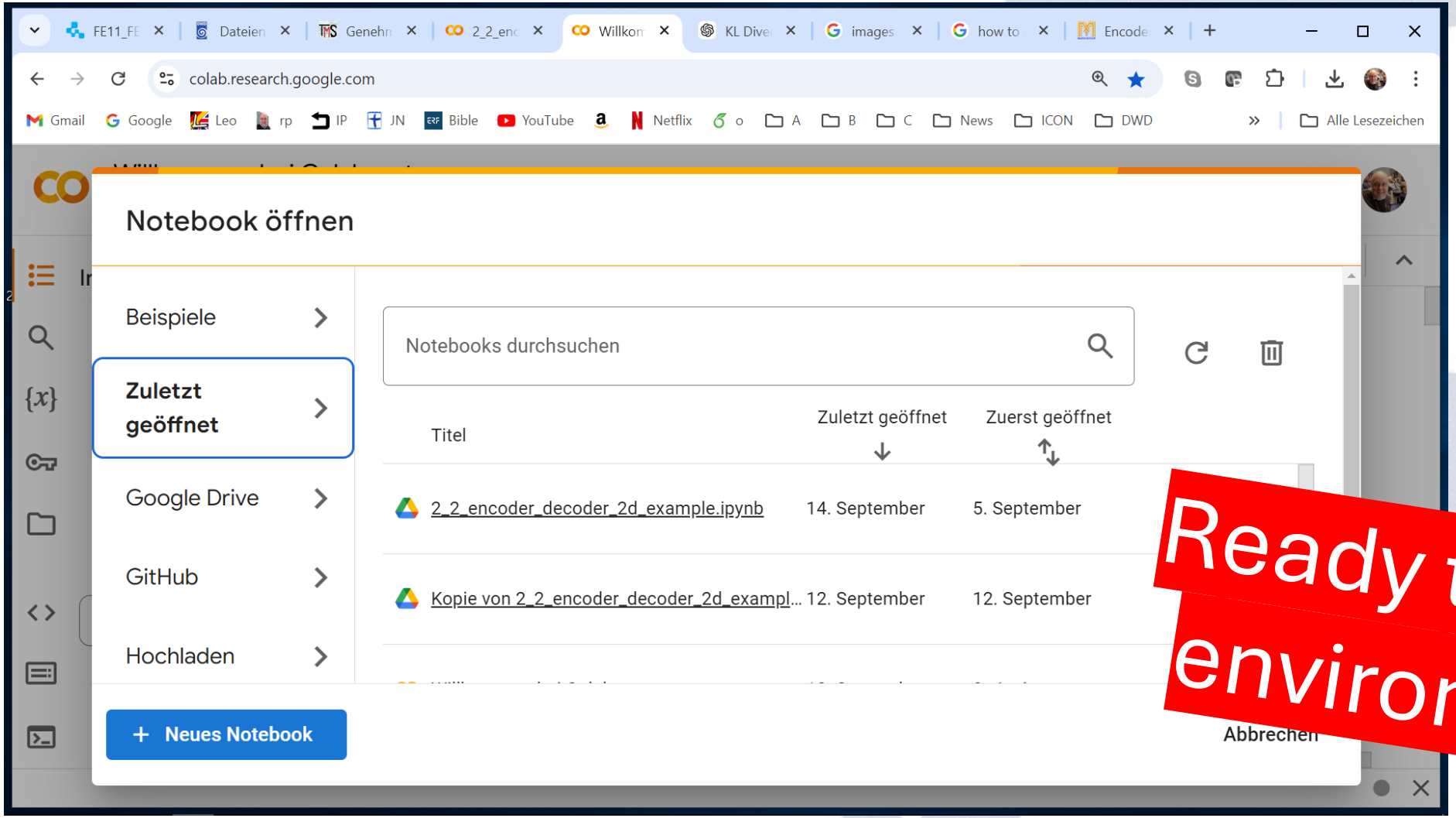
Jupyter Notebook AI Ready



The Toolkit has all necessary tools for AI development preinstalled and ready to use.

Complex AI Applications e.g. based on PyTorch can be copied over and run immediately.

Using Google Colab enables AI programming, integrated with Gemini



E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

1.3

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Downscaling via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

EUMETNET

Simple PyTorch example

- A Simple AI Example which runs on all these platforms

- Loading necessary packages

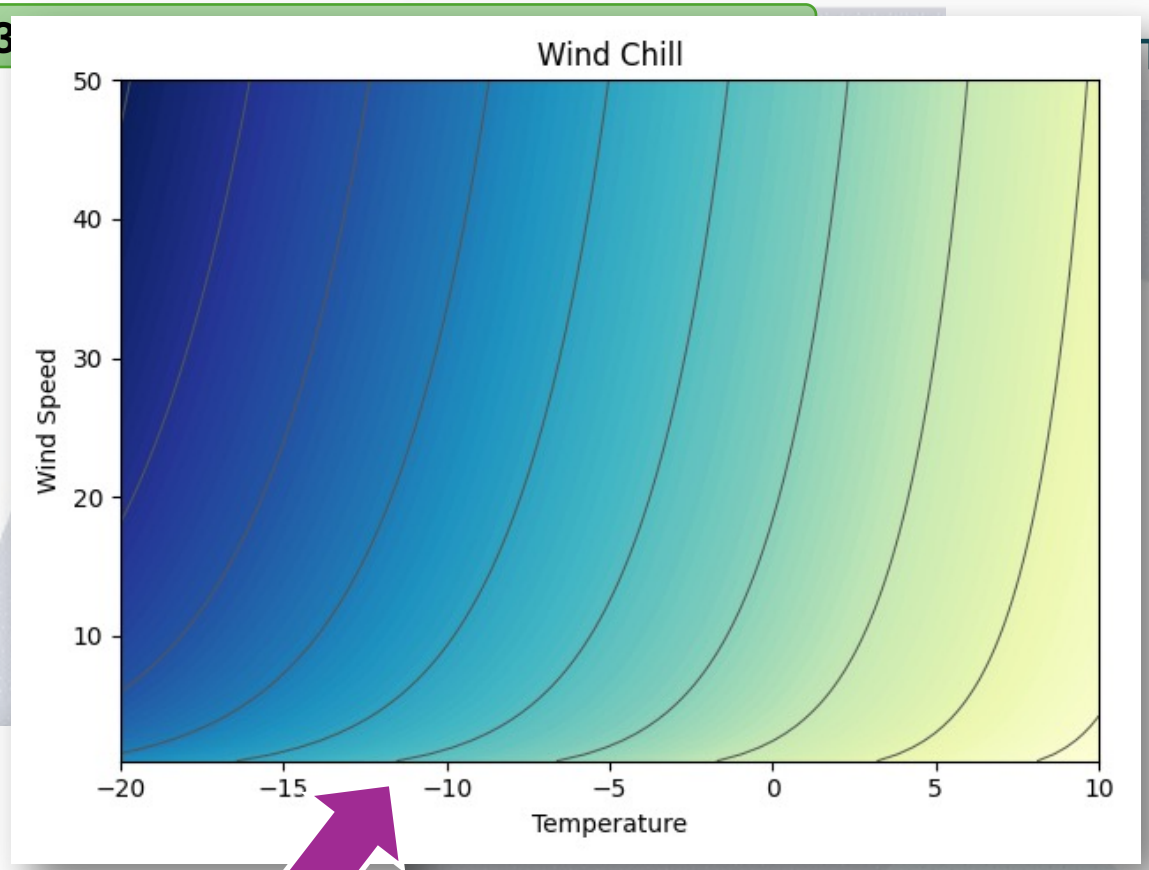
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

- Preparing the data
- Setting up the neural network architecture
- Running the training loop

- Looking at the results

Data

- Example: we try to learn the calculation of wind chill



- Prepare the data:

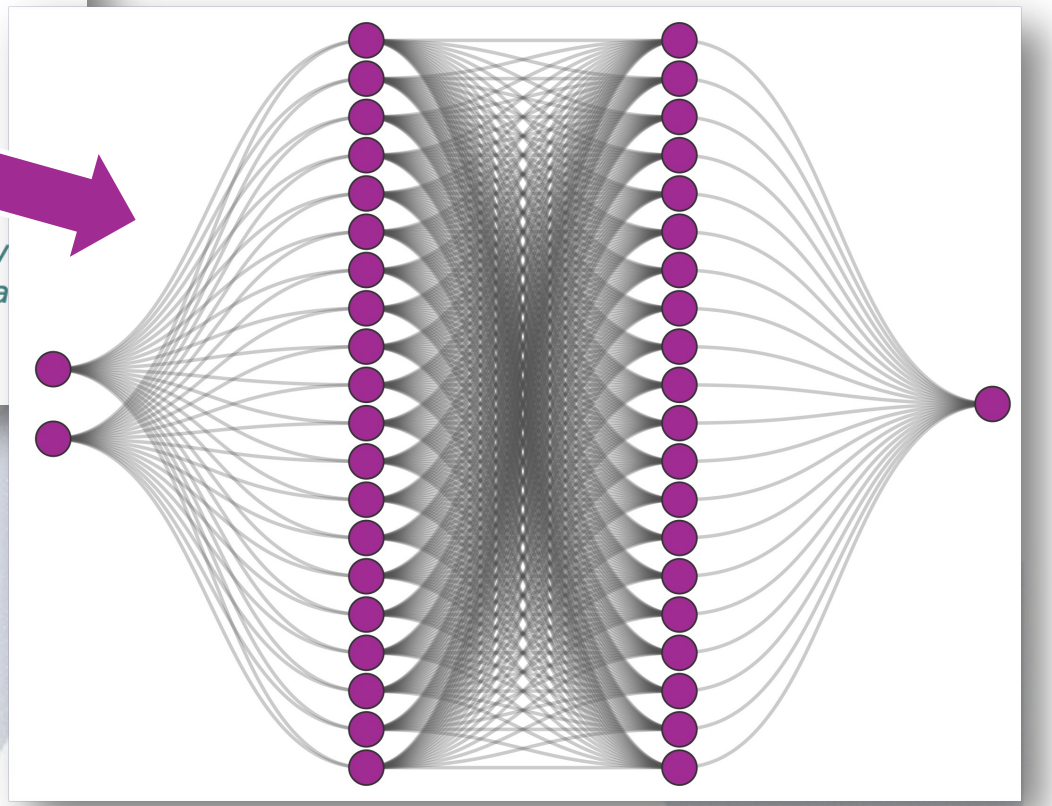
```
# Generate data
n_samples = 2000

tt = np.random.uniform(-20, 10, n_samples)  # Temperature in Celsius
ff = np.random.uniform(0, 50, n_samples)  # Wind speed in km/h

# Wind Chill Formula
wc = 13.12 + 0.6215 * tt - 11.37 * (ff ** 0.16) + 0.3965 * tt * (ff ** 0.16)

# Convert to PyTorch tensors
x_train = torch.tensor(np.column_stack((tt, ff)), dtype=torch.float32)
y_train = torch.tensor(wc, dtype=torch.float32).view(-1, 1)
```

EUMETNET

Neural Network Architecture

- Set up the architecture

```
# Step 2: Build a Neural Network Model with Hidden Layers
class wind_chill_model(nn.Module):
    def __init__(self):
        super(wind_chill_model, self).__init__()
        self.fc1 = nn.Linear(2, 20)  # First hidden layer
        self.fc2 = nn.Linear(20, 20)  # Second hidden layer
        self.fc3 = nn.Linear(20, 1)  # Output layer
        self.relu = nn.ReLU()  # Activation function

    def forward(self, x):
        x = self.relu(self.fc1(x))  # Apply ReLU after the first hidden lay
        x = self.relu(self.fc2(x))  # Apply ReLU after the second hidden la
        x = self.fc3(x)  # Output layer (no activation for regression)
        return x
```



- Initialize the model

```
model = wind_chill_model()

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

EUMETNET

Training loop

```
# Training loop
train_loss = []  # Initialize loss list
n_epoch = 10000  # Set number of epochs

for epoch in range(n_epoch):
    model.train()  # Set model to train mode
    optimizer.zero_grad()  # Clear gradients
    y_pred = model(x_train)  # Forward pass
    loss = criterion(y_pred, y_train)  # Compute loss
    loss.backward()  # Backpropagate error
    optimizer.step()  # Update weights

    # Print loss every 500 epochs
    if (epoch + 1) % 500 == 0:
        print(f'Epoch [{epoch + 1}/{n_epoch}], Loss: {loss.item():.4f}')

    train_loss.append(loss.item())  # Save loss
```

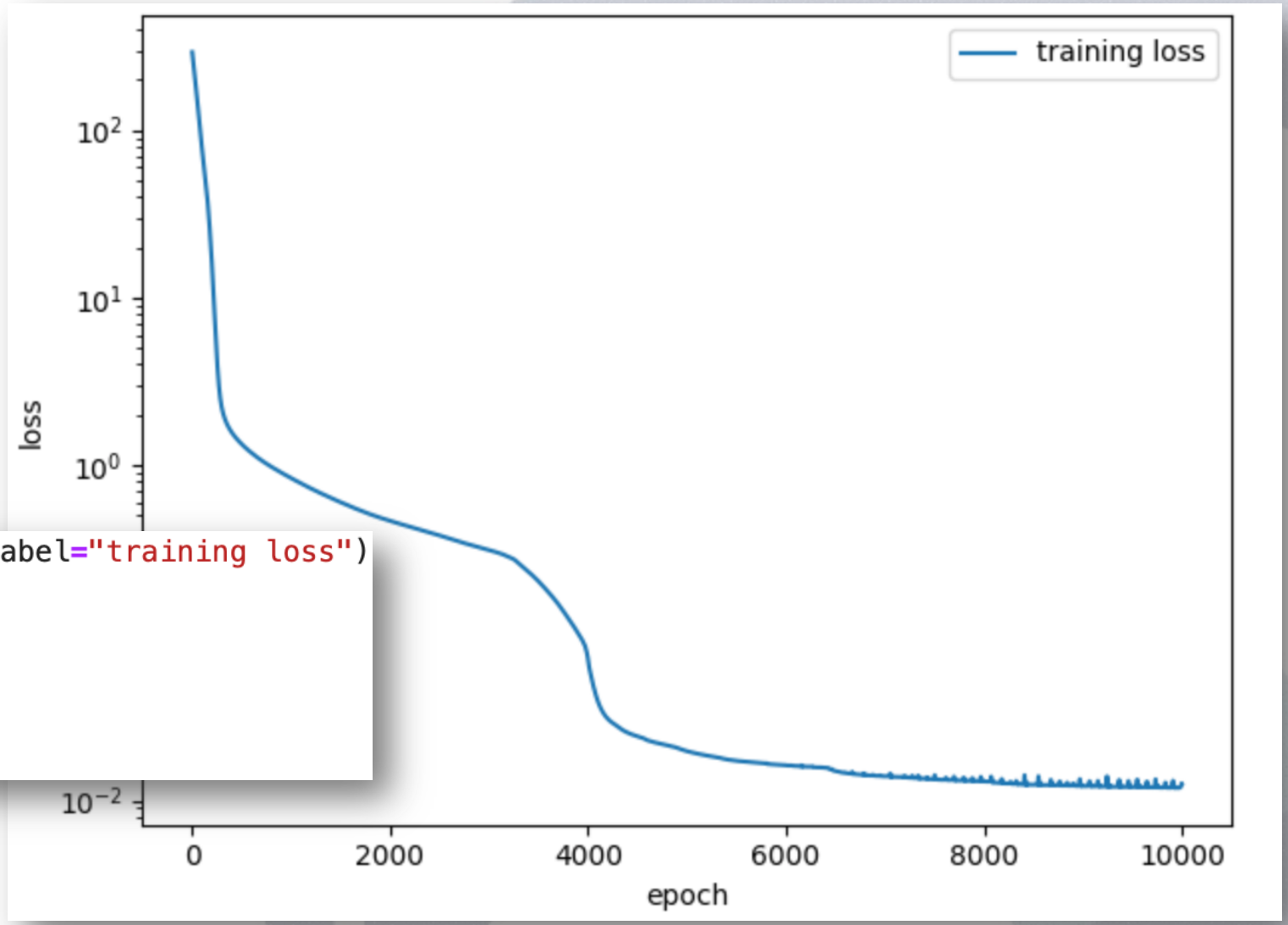
```
Epoch [500/10000], Loss: 1.3382
Epoch [1000/10000], Loss: 0.8385
Epoch [1500/10000], Loss: 0.6018
Epoch [2000/10000], Loss: 0.4658
Epoch [2500/10000], Loss: 0.3813
Epoch [3000/10000], Loss: 0.3119
Epoch [3500/10000], Loss: 0.2030
Epoch [4000/10000], Loss: 0.0687
Epoch [4500/10000], Loss: 0.0241
Epoch [5000/10000], Loss: 0.0197
Epoch [5500/10000], Loss: 0.0172
Epoch [6000/10000], Loss: 0.0162
Epoch [6500/10000], Loss: 0.0150
Epoch [7000/10000], Loss: 0.0139
Epoch [7500/10000], Loss: 0.0139
Epoch [8000/10000], Loss: 0.0130
Epoch [8500/10000], Loss: 0.0124
Epoch [9000/10000], Loss: 0.0122
Epoch [9500/10000], Loss: 0.0120
Epoch [10000/10000], Loss: 0.0126
```

- epoch == full data set iteration

EUMETNET

Look at the results

- “loss curve”




```
plt.plot(np.arange(n_epoch),train_loss,label="training loss")
plt.yscale('log')
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.tight_layout()
```

EUMETNET

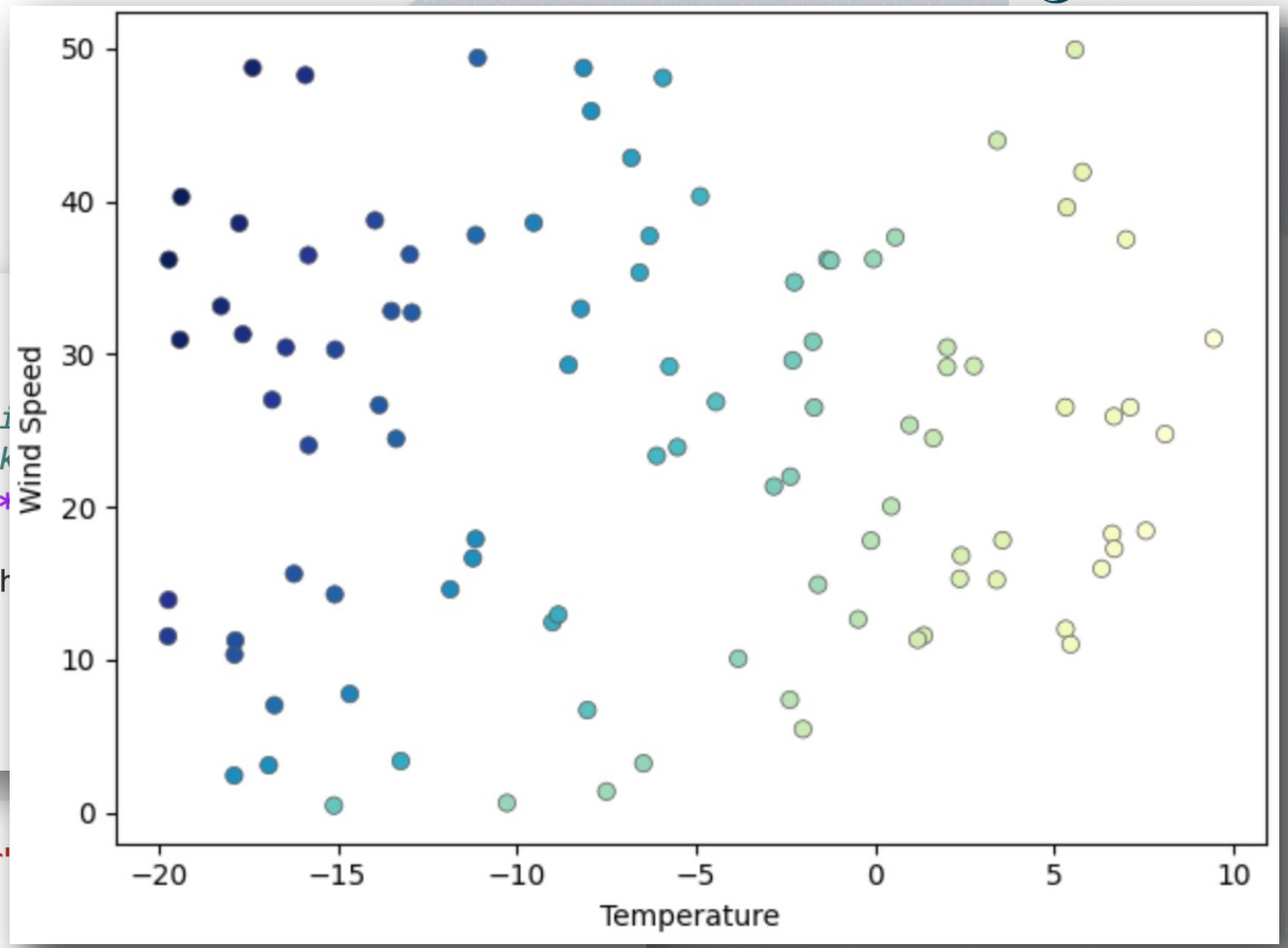
Look at the results

- Validation data set

```
# Create a validation data set
n_vsamples=100

vtt = np.random.uniform(-20, 10, n_vsamples)  # Temperature i
vff = np.random.uniform(0, 50, n_vsamples)  # Wind speed in k
vwc = 13.12 + 0.6215 * vtt - 11.37 * (vff ** 0.16) + 0.3965 *

x_val = torch.tensor(np.colum , dtype=torch
y_val = torch.tensor(vwc, d iew(-1, 1)

# Make the predict s with the model
y_pred=model(x_val)
```

Inference

```
# Create a scatter plot for the wind chill estimates
plt.scatter(vtt,vff,c=y_pred.detach().numpy(),cmap="YlGnBu_r"
plt.xlabel("Temperature")
plt.ylabel("Wind Speed")
plt.tight_layout()
```



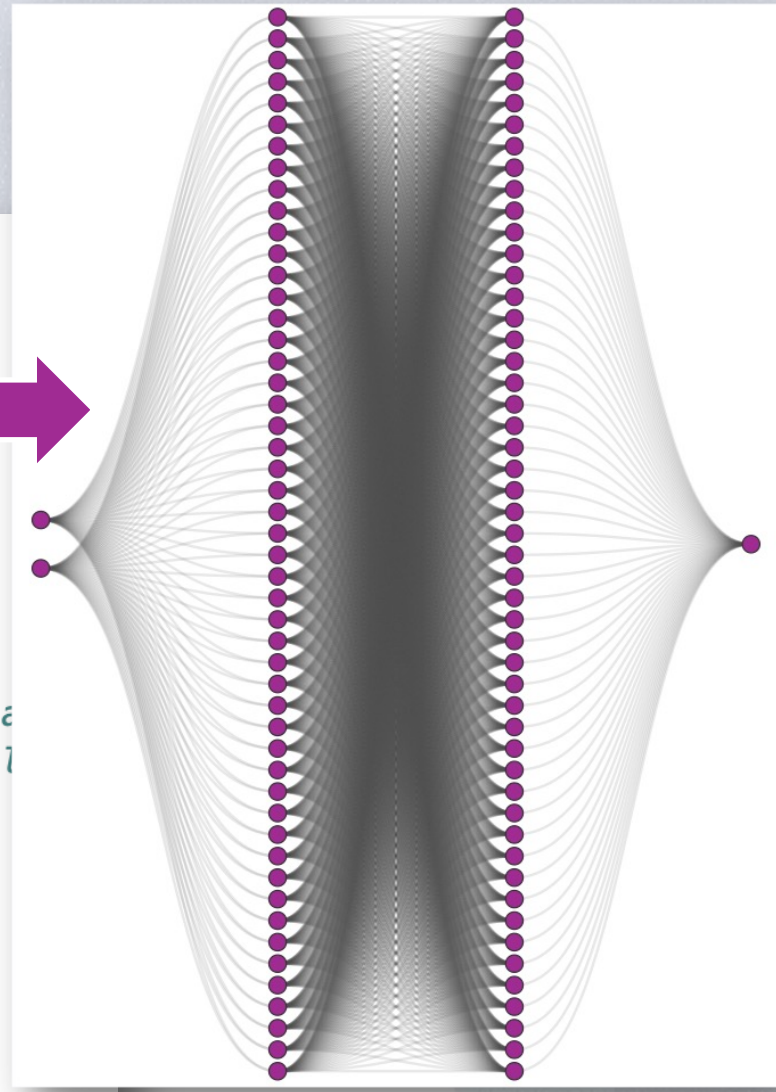

```
plt.scatter(vtt,vff,c=(y_pred-y_val).detach().numpy(),vmin=-0.5,vmax=0.5,cmap="RdBu_r")
plt.xlabel("Temperature")
plt.ylabel("Wind Speed")
plt.tight_layout()
```

EUMETNET

Tune the model

- Changing architecture and learning rate

```
# Step 2: Build a Neural Network Model with Hidden Layers
class wind_chill_model(nn.Module):
    def __init__(self):
        super(wind_chill_model, self).__init__()
        self.fc1 = nn.Linear(2, 50)  # First hidden layer
        self.fc2 = nn.Linear(50, 50)  # Second hidden layer
        self.fc3 = nn.Linear(50, 1)  # Output layer
        self.relu = nn.ReLU()  # Activation function

    def forward(self, x):
        x = self.relu(self.fc1(x))  # Apply ReLU after the first hidden la
        x = self.relu(self.fc2(x))  # Apply ReLU after the second hidden l
        x = self.fc3(x)  # Output layer (no activation for regression)
        return x

model_50 = wind_chill_model()

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model_50.parameters(), lr=0.00025)
```



Tune the model

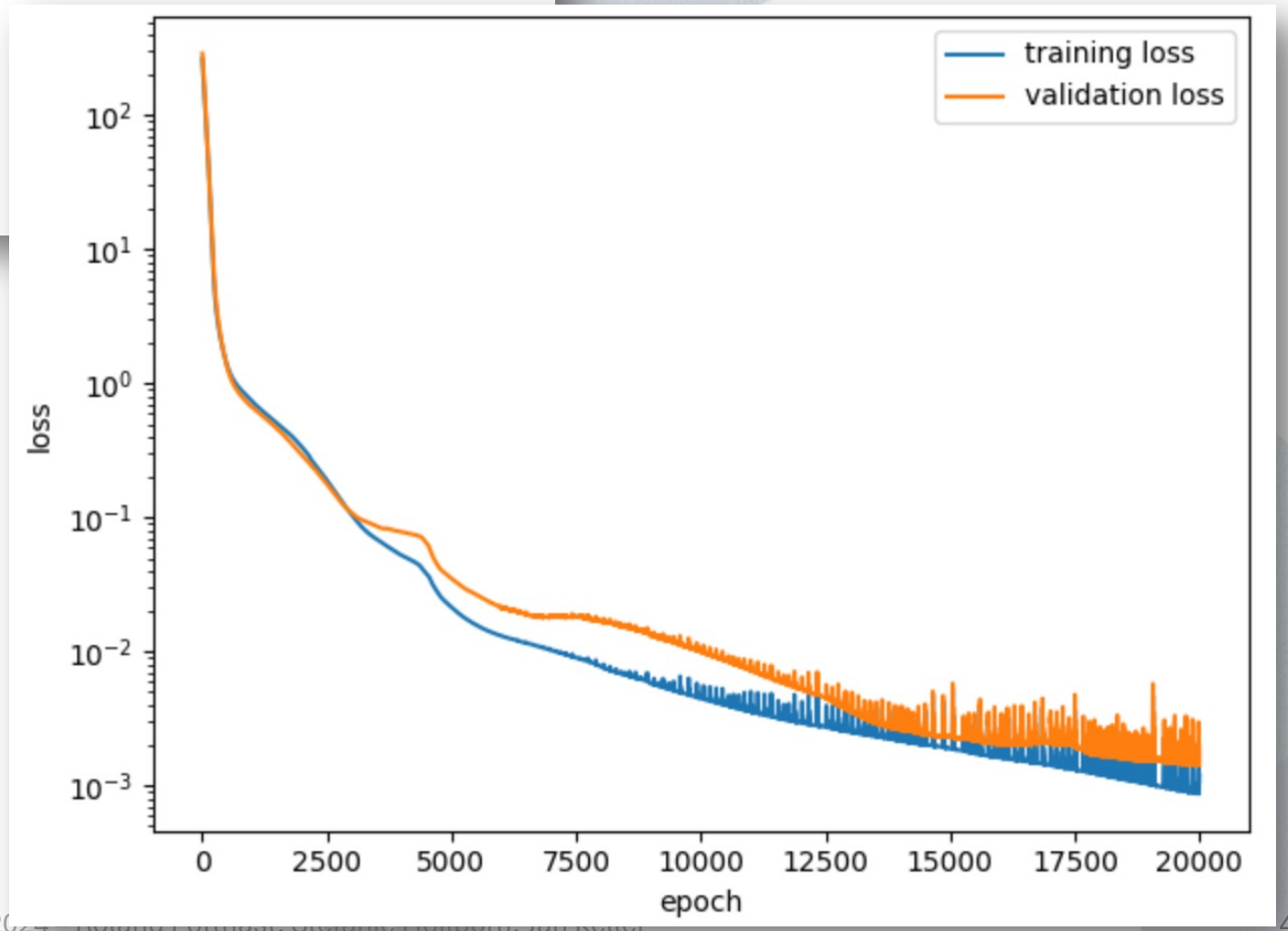
```
train_loss=[]
validation_loss=[]

# Training loop
n_epoch=20000
for epoch in range(n_epoch):
    model_50.train()
    optimizer.zero_grad()
    y_pred=model_50(x_train)
    loss=criterion(y_pred,y_train)
    loss.backward()
    optimizer.step()

    if (epoch+1)%500==0:
        print(f'Epoch [{epoch+1}/{n_epoch}], Loss: {loss.item():.4f}')
    train_loss.append(loss.item())
    y_pred=model_50(x_val)
    vloss=criterion(y_pred,y_val)
    validation_loss.append(vloss.item())
```

```
Epoch [500/20000], Loss: 1.3586
Epoch [1000/20000], Loss: 0.7397
Epoch [1500/20000], Loss: 0.5076
Epoch [2000/20000], Loss: 0.3357
Epoch [2500/20000], Loss: 0.1899
Epoch [3000/20000], Loss: 0.1051
Epoch [3500/20000], Loss: 0.0691
Epoch [4000/20000], Loss: 0.0524
Epoch [4500/20000], Loss: 0.0381
Epoch [5000/20000], Loss: 0.0215
Epoch [5500/20000], Loss: 0.0158
Epoch [6000/20000], Loss: 0.0131
Epoch [6500/20000], Loss: 0.0116
Epoch [7000/20000], Loss: 0.0103
Epoch [7500/20000], Loss: 0.0090
Epoch [8000/20000], Loss: 0.0078
Epoch [8500/20000], Loss: 0.0067
Epoch [9000/20000], Loss: 0.0058
Epoch [9500/20000], Loss: 0.0051
Epoch [10000/20000], Loss: 0.0045
Epoch [10500/20000], Loss: 0.0040
Epoch [11000/20000], Loss: 0.0045
Epoch [11500/20000], Loss: 0.0032
Epoch [12000/20000], Loss: 0.0030
Epoch [12500/20000], Loss: 0.0028
Epoch [13000/20000], Loss: 0.0025
Epoch [13500/20000], Loss: 0.0024
Epoch [14000/20000], Loss: 0.0022
Epoch [14500/20000], Loss: 0.0035
Epoch [15000/20000], Loss: 0.0019
Epoch [15500/20000], Loss: 0.0020
Epoch [16000/20000], Loss: 0.0016
Epoch [16500/20000], Loss: 0.0015
Epoch [17000/20000], Loss: 0.0014
Epoch [17500/20000], Loss: 0.0016
Epoch [18000/20000], Loss: 0.0016
Epoch [18500/20000], Loss: 0.0012
Epoch [19000/20000], Loss: 0.0011
Epoch [19500/20000], Loss: 0.0010
Epoch [20000/20000], Loss: 0.0011
```

EUMETNET

Tune the model

```
# Loss curve
plt.plot(np.arange(n_epoch),train_loss,label="training loss")
plt.plot(np.arange(n_epoch),validation_loss,label="validation loss")
plt.yscale('log')
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.tight_layout()
```



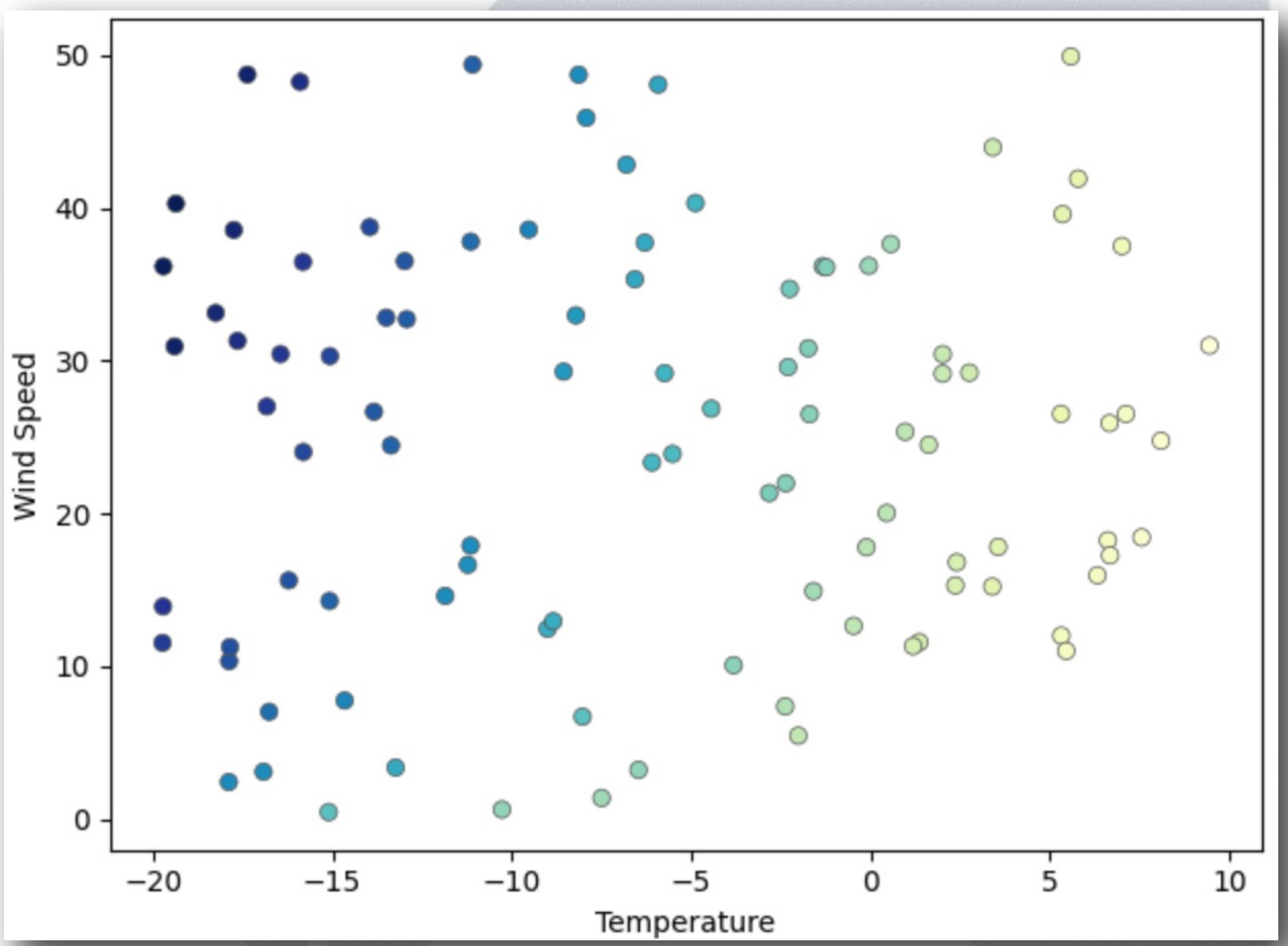
- “loss curve”

EUMETNET

Tune the model

- Validation results

```
# Make the predictions with the model
y_pred=model_50(x_val)
```



EUMETNET

AI on GPU Nodes

- You need to adapt your code to send torch tensors to the gpu, see example

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

```
# Convert to PyTorch tensors
x_train = torch.tensor(x, dtype=torch.float32).unsqueeze(1).to(device)
y_train = torch.tensor(y, dtype=torch.float32).unsqueeze(1).to(device)
```

- Send tensors back to cpu to generate plots

```
with torch.no_grad():
    predicted = model(x_test).to("cpu")

pred2 = predicted.detach().numpy()
# Plot the results
plt.plot(x, y, label='Original curve')
plt.plot(x_test.to("cpu"), pred2, label='Fitted curve')
plt.legend()
plt.show()
plt.savefig("img.png")
```

E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Downscaling via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

Thank You!