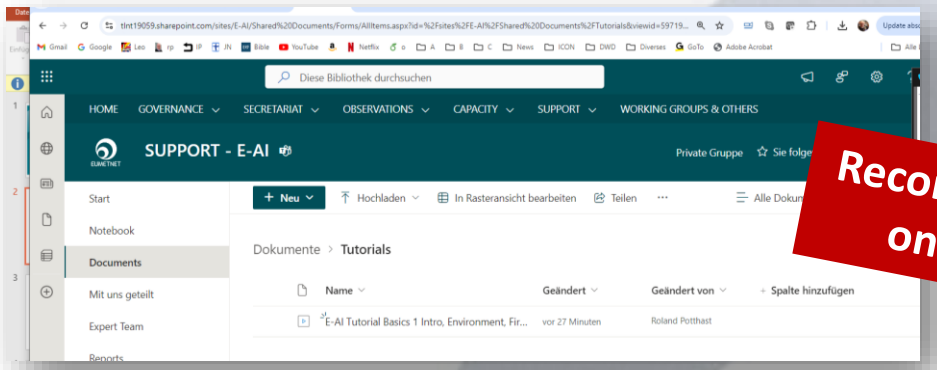
AI Intro Basics #2

Dynamics, Data Recovery, Data Assimilation Examples

Roland Potthast, Stefanie Hollborn and Jan Keller

EUMETNET

Material / Tutorials



Recordings available on Sharepoint

- EUMETNET
  E-AI Sharepoint

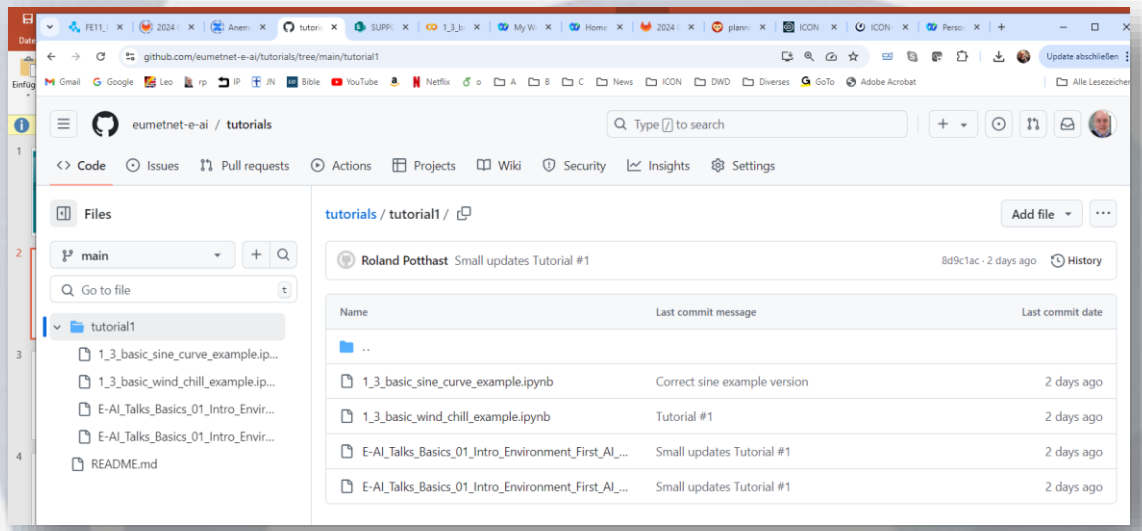
https://tlnt19059.sharepoint.com/sites/E-AI/SitePages/CollabHome.aspx

To get access to EUMETNET **SharePoint** write an email to marek.Jacob@dwd.de

- Github eumetnet-e-ai Repo

https://github.com/eumetnet-e-ai/tutorials/

Codes and Slides available on Github



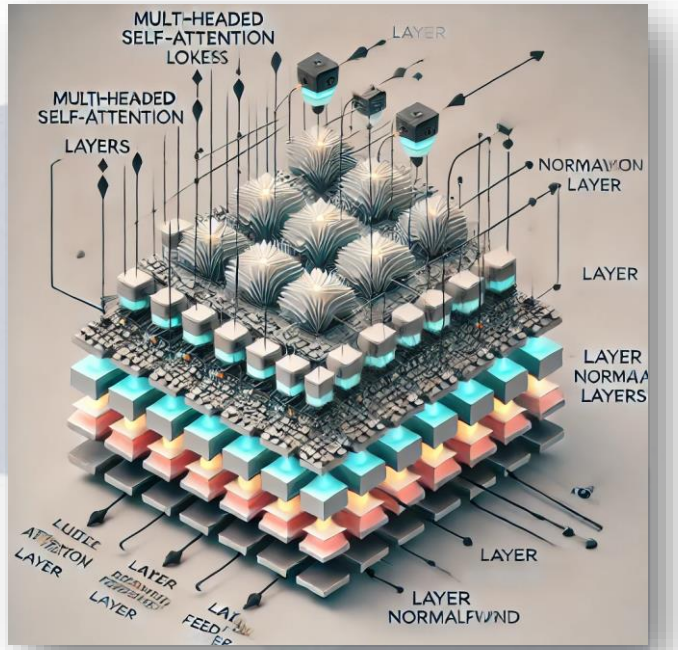
E-AI Basic Tutorials

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 Data Recovery/Denoising 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]

2.1



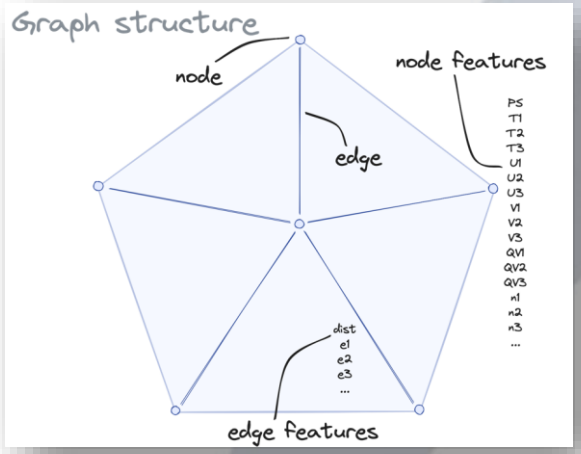
AI generated Images,
© Roland Potthast 2024

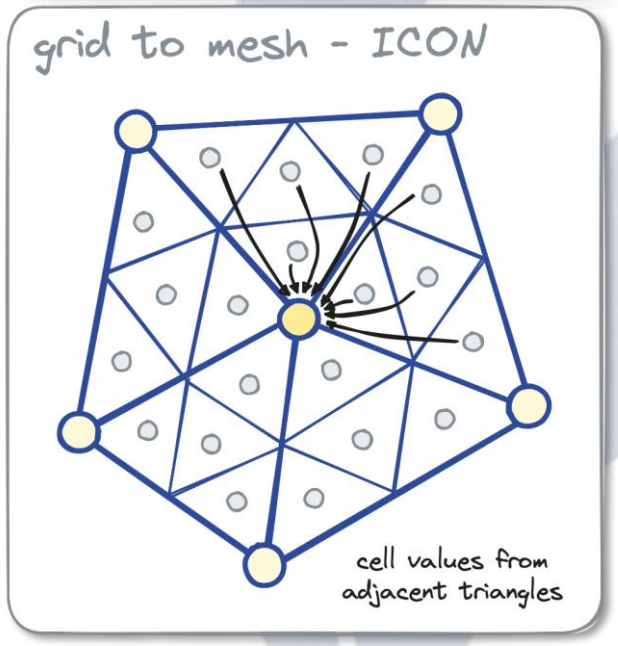
Graph Neural Networks – Basic Idea 1/2

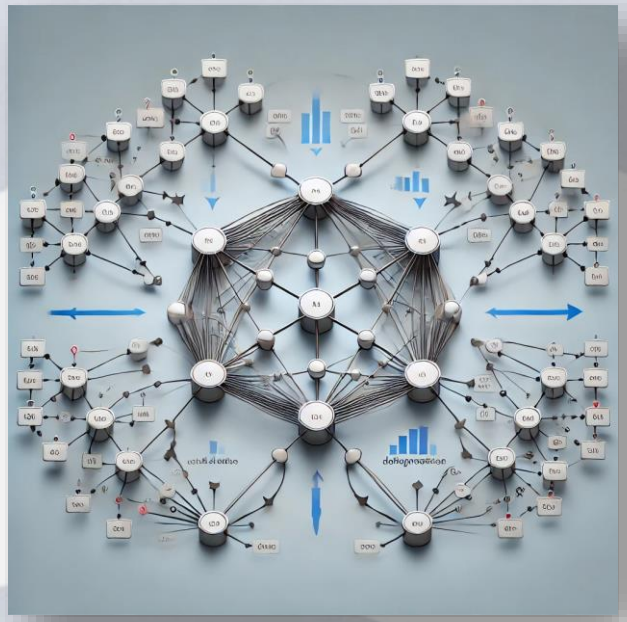
Graph Neural Networks (GNNs) are a type of neural network designed to work on graph-structured data, where entities (nodes) are connected by relationships (edges). Unlike traditional neural networks, GNNs use graph structures to capture both the features of individual nodes and their connectivity within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

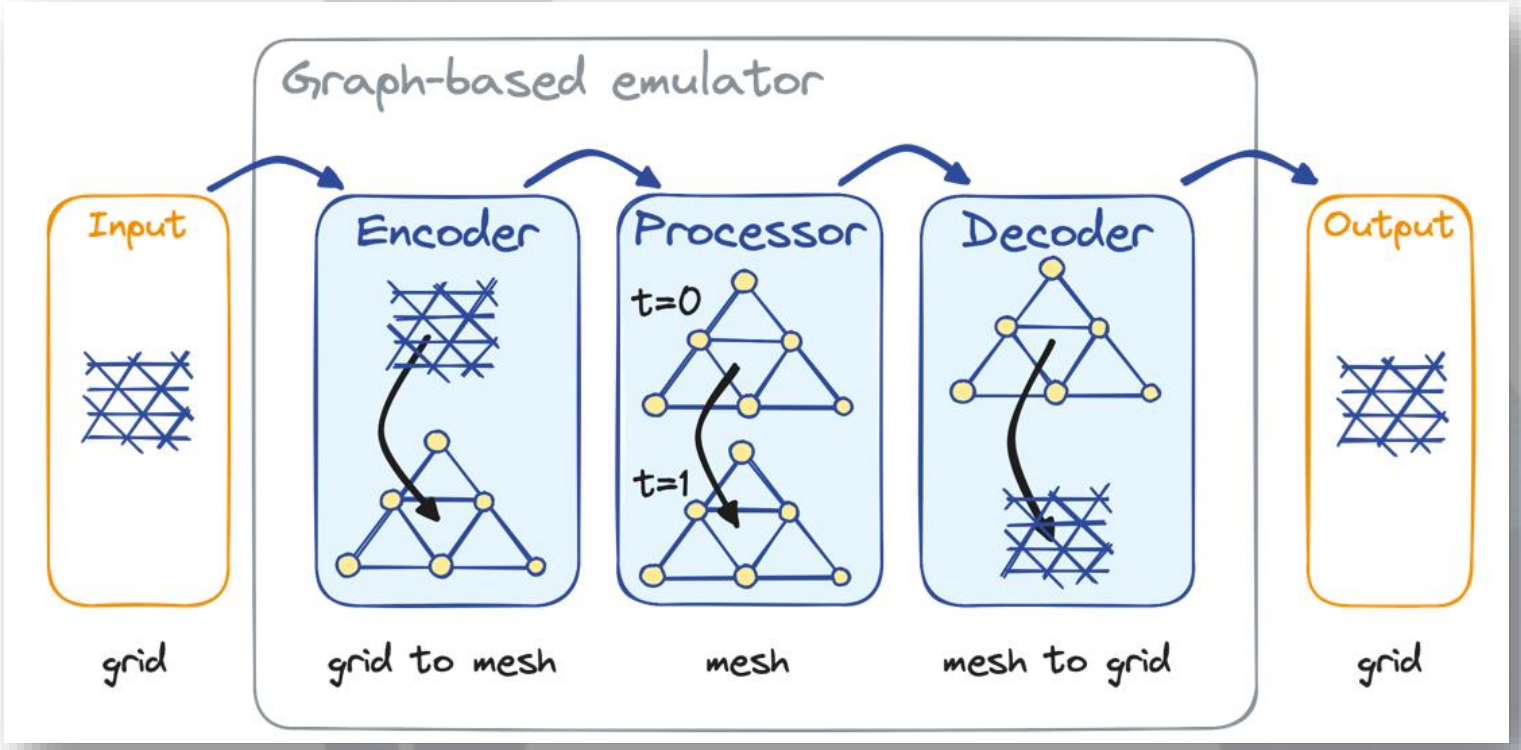
In graph-based models using an encoder, processor, and decoder, the **encoder** transforms raw graph data, such as node features and edge connections, into an initial set of embeddings that represent the graph's structure and features.

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.



Import the necessary Libraries

- Import ML Libraries needed for your project.
- You might need to install some packages as shown here.
- We rely on PyTorch mainly for our tutorial.

PyTorch was primarily developed by *Facebook's AI Research* (FAIR) group, but made an open source project and is widely used.

- To achieve reproducibility we set the seed.

```
%pip install torch_geometric
```

```
import numpy as np  # Numerical operations
import matplotlib.pyplot as plt  # Plotting
from matplotlib.animation import FuncAnimation  # For animations
plt.rcParams['figure.figsize'] = [16, 4]  # Set default figure size

import torch  # PyTorch for deep learning
import torch.nn as nn  # Neural network module
import torch.nn.functional as F  # Functional API

import torch_geometric.data as geom_data  # Graph-based data handling
import torch_geometric.nn as geom_nn  # Graph neural network layers
```

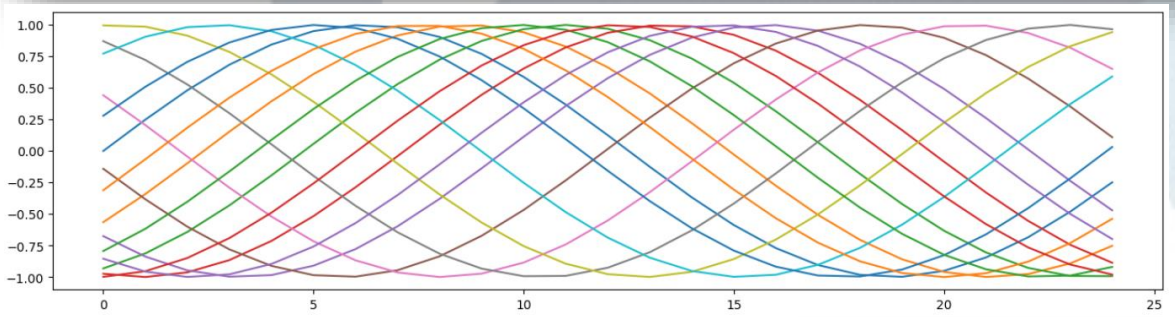
```
torch.manual_seed(0)
```

Setup simple example

- We prepare a simple example of moving sinusoidal functions.
- Setup points in one dimension: x_grid
- Define sine functions moving with speed v.
- Visualize all functions by matplotlib.

```
xa = 10       # Spatial domain length
nx = 25       # Number of spatial grid points
nt = 15       # Number of time steps
v = 0.6       # Wave speed

# Create spatial grid excluding last point
x_grid = np.linspace(0, xa, nx+1)[:-1]
z = np.zeros([nt, nx]) # Initialize solution array

# Fill array with sine wave values over time steps
for j in range(nt):
    z[j,:] = np.sin((2*np.pi/xa) * x_grid - v * j)
```

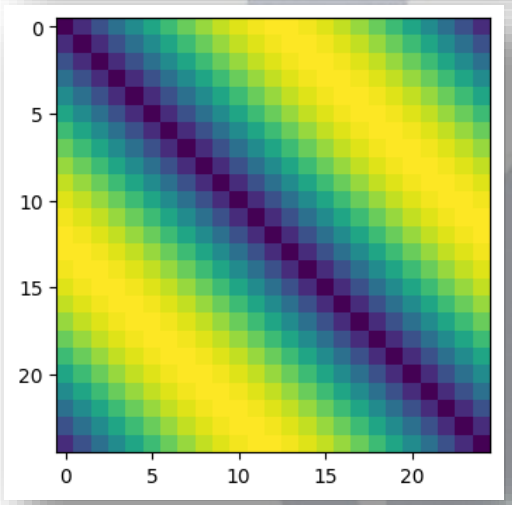
```
plt.plot(z.T);  # Plot the transposed matrix 'z'
```



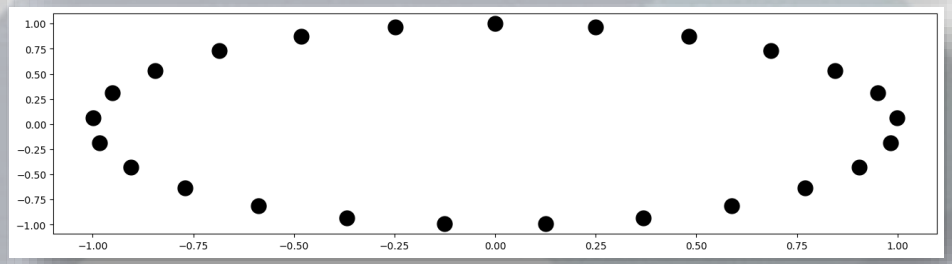
To construct a local graph, we define distances between grid points

- We map our points from a line segment to a circle (p1,p2)
- then calculate the difference of the first and second coordinate
- And use it to calculate the Euclidean distance.
- Visualize by matplotlib

```
p1 = np.sin(2 * np.pi * x_grid / xa)  # Sine over grid 'x_grid'
p2 = np.cos(2 * np.pi * x_grid / xa)  # Cosine over grid 'x_grid'

p1m = np.tile(p1, (nx, 1)).T  # Repeat 'p1', transpose
p2m = np.tile(p2, (nx, 1)).T  # Repeat 'p2', transpose

diff1 = p1m - p1m.T  # Difference between 'p1m' and transpose
diff2 = p2m - p2m.T  # Difference between 'p2m' and transpose

diff = np.sqrt(diff1**2 + diff2**2)  # Euclidean distance

plt.imshow(diff);  # Plot the difference matrix
```




```
plt.plot(p1,p2,'.',color='black',markersize=30)
```

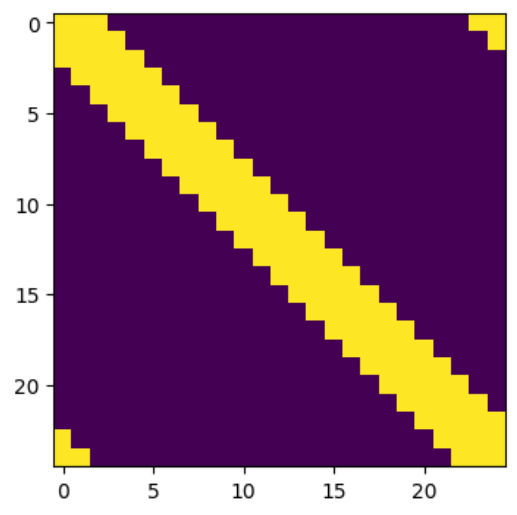


Now generate Adjacency Matrix and Edge Index

An adjacency matrix is a square matrix used to represent a graph, where each element indicates whether a pair of vertices (nodes) are adjacent (connected by an edge) in the graph.

An edge index is a list or matrix that represents the connections (edges) in a graph by storing pairs of node indices, where each pair corresponds to an edge between two nodes.

```
adjm = (diff < 0.5).astype(int)  # Adjacency matrix (threshold < 0.5)
adjm_t = torch.tensor(adjm, dtype=torch.float)  # Convert to tensor
edge_index = adjm_t.nonzero().t()  # Non-zero indices as edges
edge_index = edge_index.contiguous()  # Ensure continuous memory

plt.imshow(adjm)  # Plot adjacency matrix
```




```
edge_index

tensor([[ 0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,
          3,  3,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  7,
          7,  7,  7,  7,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9, 10, 10, 10, 10,
         10, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 14, 14,
         14, 14, 14, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17,
         18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 21, 21, 21,
         21, 21, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24],
        [ 0,  1,  2, 23, 24,  0,  1,  2,  3, 24,  0,  1,  2,  3,  4,  1,  2,  3,
          4,  5,  2,  3,  4,  5,  6,  3,  4,  5,  6,  7,  4,  5,  6,  7,  8,  5,
          6,  7,  8,  9,  6,  7,  8,  9, 10,  7,  8,  9, 10, 11,  8,  9, 10, 11,
         12,  9, 10, 11, 12, 13, 10, 11, 12, 13, 14, 11, 12, 13, 14, 15, 12, 13,
         14, 15, 16, 13, 14, 15, 16, 17, 14, 15, 16, 17, 18, 15, 16, 17, 18, 19,
         16, 17, 18, 19, 20, 17, 18, 19, 20, 21, 18, 19, 20, 21, 22, 19, 20, 21,
         22, 23, 20, 21, 22, 23, 24,  0, 21, 22, 23, 24,  0,  1, 22, 23, 24]])
```

Prepare Training Data and Test Data

- In this code, ntr1 and ntr2 define the start and end indices for selecting the training data, while nte1 and nte2 define the start and end indices for the testing data.
- The variables X_train and Y_train represent the input and target data for training, where X_train is a slice from z, and Y_train is the next time step's data from z.
- Similarly, X_test and Y_test are selected as the input and target data for testing, using slices from z for evaluation.

```
ntr1 = 0                # Start index for training
ntr2 = round(nt*0.8)    # End index for training
nte1 = ntr2+1           # Start index for testing
nte2 = nt - 1           # End index for testing

X_train = z[ntr1:ntr2, :]  # Training input data
Y_train = z[ntr1+1:ntr2+1, :]  # Training target data
X_test = z[nte1:nte2, :]  # Testing input data
Y_test = z[nte1+1:nte2+1, :]  # Testing target data
```

For the beginning one could choose training data = test data as a first simple step (to work with a very limited sample)

```
ntr1 = 0       # Start index for training
ntr2 = nt - 1  # End index for training
nte1 = 0       # Start index for testing
nte2 = nt - 1  # End index for testing

X_train = z[ntr1:ntr2, :]  # Training input data
Y_train = z[ntr1+1:ntr2+1, :]  # Training target data
X_test = z[nte1:nte2, :]  # Testing input data
Y_test = z[nte1+1:nte2+1, :]  # Testing target data
```

Setup the Feature Tensors and Label Tensors

- We define a feature tensor features_val to represent **locations** (here just the indices of the points)
- We define a feature tensor features_k given by the **values** at the points
- The labels are the **target** values, i.e. the desired output of our neural network
- In PyTorch, a **DataLoader** is an essential utility that simplifies the process of feeding data into a model. It combines a dataset with a sampler, handling
  - batching,
  - shuffling, and
  - parallel processing of data.

```
# Lists to store training and testing data
train_list = []
test_list = []

# Create a normalized tensor (features_val) to represent locations, reshaped to (nx, 1)
features_loc = torch.tensor(np.arange(1, nx+1) / nx, dtype=torch.float).unsqueeze(1)

# Iterate over each training instance
for k in range(X_train.shape[0]):
    # Convert the k-th row of X_train to a tensor, reshaped to (nx, 1)
    features_val = torch.tensor(X_train[k, :], dtype=torch.float).unsqueeze(1)

    # Concatenate location and value features along dimension 1
    features_k = torch.cat((features_val, features_loc), dim=1)

    # Convert the k-th row of Y_train to a tensor and reshape to (nx, 1)
    labels_k = torch.tensor(Y_train[k, :], dtype=torch.float).unsqueeze(1)

    # Create a data object with features, labels, and edge_index for training
    data = geom_data.Data(x=features_k, y=labels_k, edge_index=edge_index)
    train_list.append(data)
```

```
# Create DataLoaders for training and testing with specified batch size
train_loader = geom_data.DataLoader(train_list, batch_size=bs, shuffle=True)
test_loader = geom_data.DataLoader(test_list, batch_size=bs, shuffle=False)
```

Define the Model

- Initialization of Layers

```python
class GNNModel(nn.Module):
    def __init__(self, num_features, hidden_channels, num_feats_y):
        super(GNNModel, self).__init__()

        # Convolutional Message Passing Layers
        self.conv1 = geom_nn.GCNConv(num_features, hidden_channels[0])
        self.conv2 = geom_nn.GCNConv(hidden_channels[0], hidden_channels[1])
        self.conv3 = geom_nn.GCNConv(hidden_channels[1], hidden_channels[2])
        self.conv4 = geom_nn.GCNConv(hidden_channels[2], hidden_channels[3])

        # Dense layer for regression
        self.fc1 = nn.Linear(hidden_channels[3], hidden_channels[2])
        self.dropout1 = nn.Dropout(p=0.1)
        self.fc2 = nn.Linear(hidden_channels[2], hidden_channels[0])
        self.dropout2 = nn.Dropout(p=0.1)
        self.fc3 = nn.Linear(hidden_channels[0], num_feats_y)
```

```python
def forward(self, x, edge_index):

    # Message Passing Layers (GCNConv)
    x = self.conv1(x, edge_index)
    x = F.leaky_relu(x)
    x = self.conv2(x, edge_index)
    x = F.leaky_relu(x)
    x = self.conv3(x, edge_index)
    x = F.leaky_relu(x)
    x = self.conv4(x, edge_index)
    x = F.leaky_relu(x)

    x = self.fc1(x)
    x = F.leaky_relu(x)
    x = self.fc2(x)
    x = F.leaky_relu(x)
    x = self.fc3(x)

    return x
```

- Forward Propagation:
  The GCNConv layer, short for Graph Convolutional Network Convolution layer, is a key component of Graph Neural Networks (GNNs). It is designed to perform convolution operations on graph-structured data. GCNConv works on graphs, where nodes and edges represent relationships between entities.

Training Loop

Training loops in PyTorch always follow a particular basic shape.

- They typically start by setting the model to training mode using model.train(), followed by iterating through the data using a DataLoader.
- For each batch, the gradients are zeroed (optimizer.zero_grad()), predictions are made using the model, and the loss is computed with a loss function (e.g., nn.MSELoss).
- The loss.backward() call computes the gradients, which are then used by the optimizer (e.g., optimizer.step()) to update the model's weights.
- This process repeats for multiple epochs, and at the end of each epoch, performance on the validation or test set is usually evaluated with model.eval() to track progress.

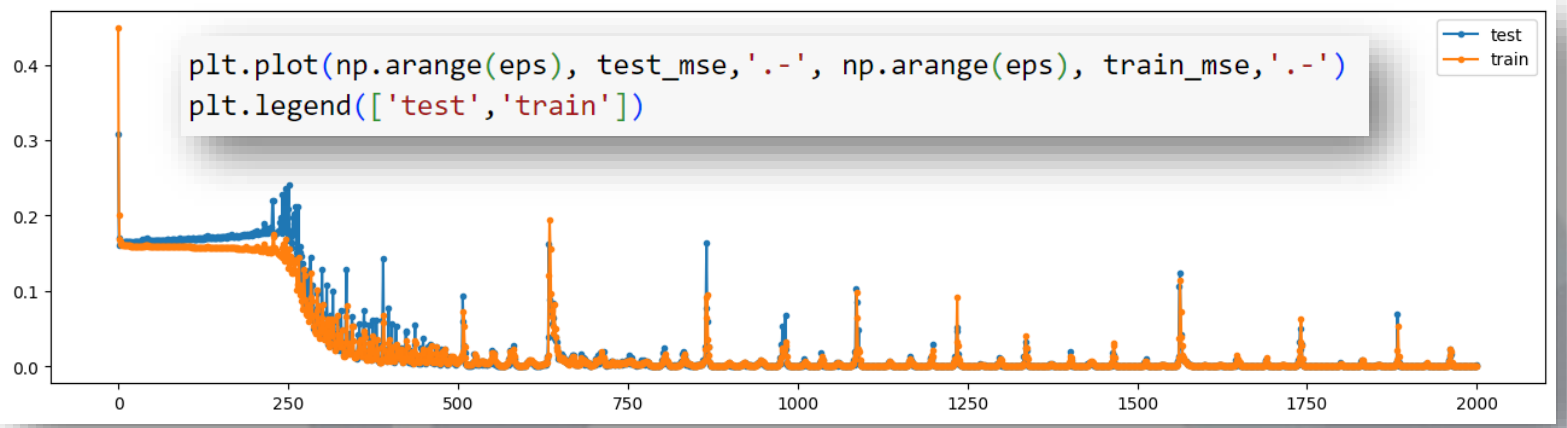
```
eps=2001
hidden_channels=[2*nt, 2*nt, 2*nt, 2*nt]
n_features_x = 2
n_features_y = 1

# Initialize and train the model
model = GNNModel(n_features_x, hidden_channels, n_features_y)

optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=0)
criterion = nn.MSELoss()

train_mse=[]
test_mse=[]
```

```
for epoch in range(eps):  # Loop for training epochs
    total_loss = 0.0
    model.train()  # Set the model to training mode

    train_mse_tmp = []
    for batch in train_loader:
        optimizer.zero_grad()
        output = model(batch.x, batch.edge_index)

        loss = criterion(output, batch.y)
        train_mse_tmp.append(loss.detach().numpy())
        if epoch%300==0: visualize(batch)

        loss.backward()
        total_loss += loss.item()
        optimizer.step()
        optimizer.zero_grad()

    train_mse.append(np.mean(train_mse_tmp))
    if epoch%100==0: print(f'Epoch {epoch + 1}: {train_mse[epoch]}')
```

Development of the Loss Function

- We evaluate the model performance on the evaluation test set, which is has not seen before.
- The red and blue curves show the current model performance on the training data set and the test data set.

```
test_mse_tmp=[]
model.eval()  # Set the model to evaluation mode

for batch in test_loader:
    y_pred = model(batch.x, batch.edge_index)
    y_test = batch.y
    test_loss = criterion(y_pred,y_test)
    test_mse_tmp.append(test_loss.detach().numpy())

test_mse.append(np.mean(test_mse_tmp))

if epoch%100==0:
   print(f'RMSE on Test Set: {test_mse[epoch]}')
```

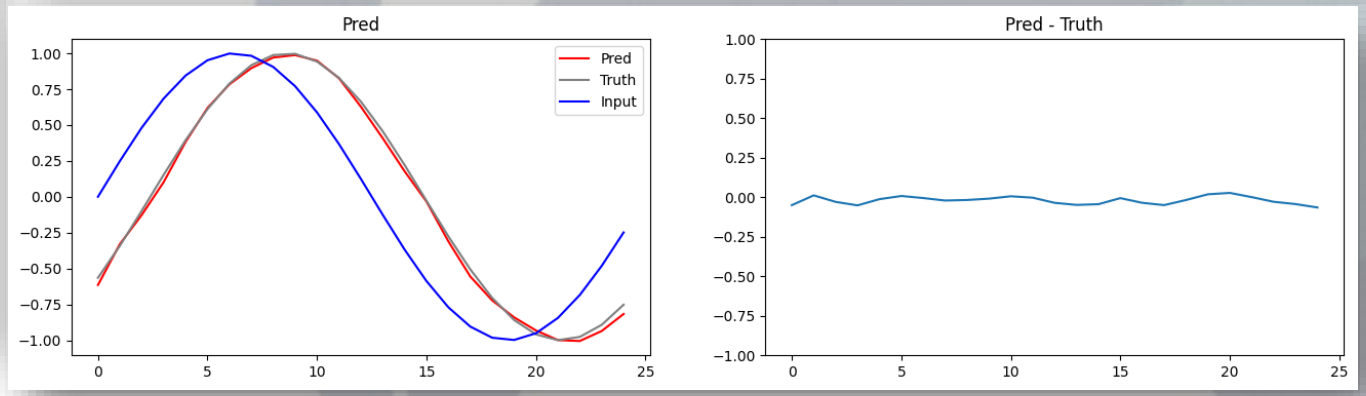


Testing the Learning Results, Visualization

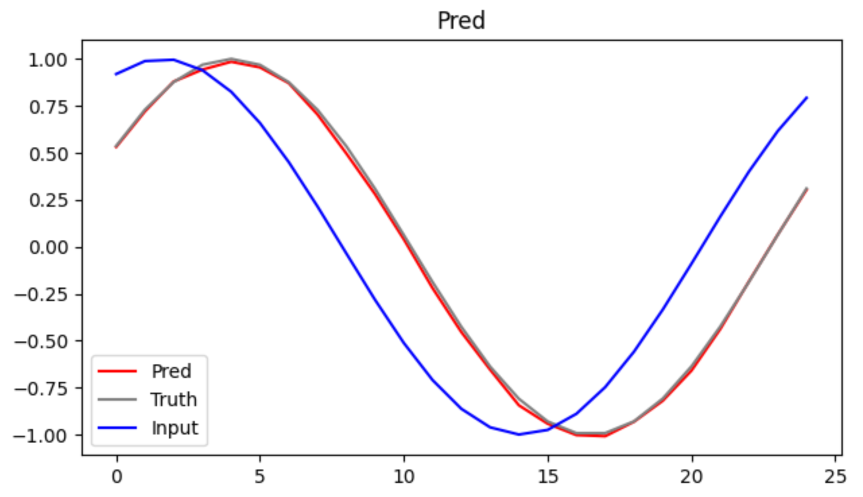
- For testing the outcome, we display some concrete examples.

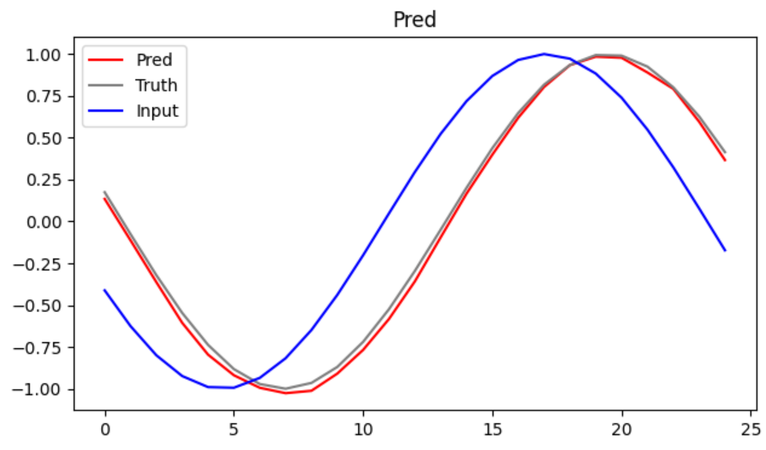
```
features_loc = torch.tensor(np.arange(1,nx+1)/nx, dtype=torch.float).unsqueeze(1)

for t in range(nt-1):
    features_val = torch.tensor(X_test[t,:], dtype=torch.float).unsqueeze(1)
    features_k = torch.cat((features_val, features_loc), dim=1)
    Y_pred = model(features_k, edge_index)

    if t % 1 == 0:
        print(t)
        X_test_tmp = features_k.detach().numpy()
        Y_test_tmp = Y_test[t,:].reshape(25, 1)
        Y_pred_tmp = Y_pred.detach().numpy()

        # Calculate the difference between true state and predicted state
        diff = Y_pred_tmp - X_test_tmp[:,0:1]
        diff2 = Y_pred_tmp - Y_test_tmp
```

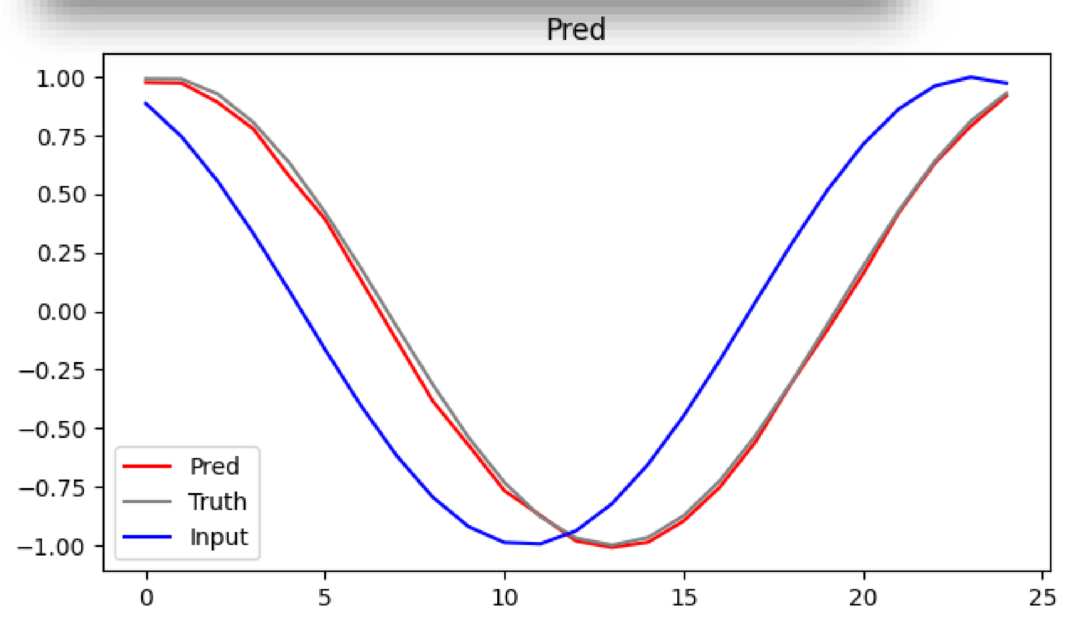
```
# Create a figure with 3 subplots side by side
fig, axs = plt.subplots(1, 3)

axs[0].plot(Y_pred_tmp,color='red')
axs[0].plot(Y_test_tmp,color='gray')
axs[0].plot(X_test_tmp[:,0],color='blue')
axs[0].legend(['Pred','Truth','Input'])
axs[0].set_title('Pred')

axs[1].plot(diff2)
axs[1].set_ylim(bottom=-1,top=1)
axs[1].set_title('Pred - Truth')

# Show the plots
plt.show()
```

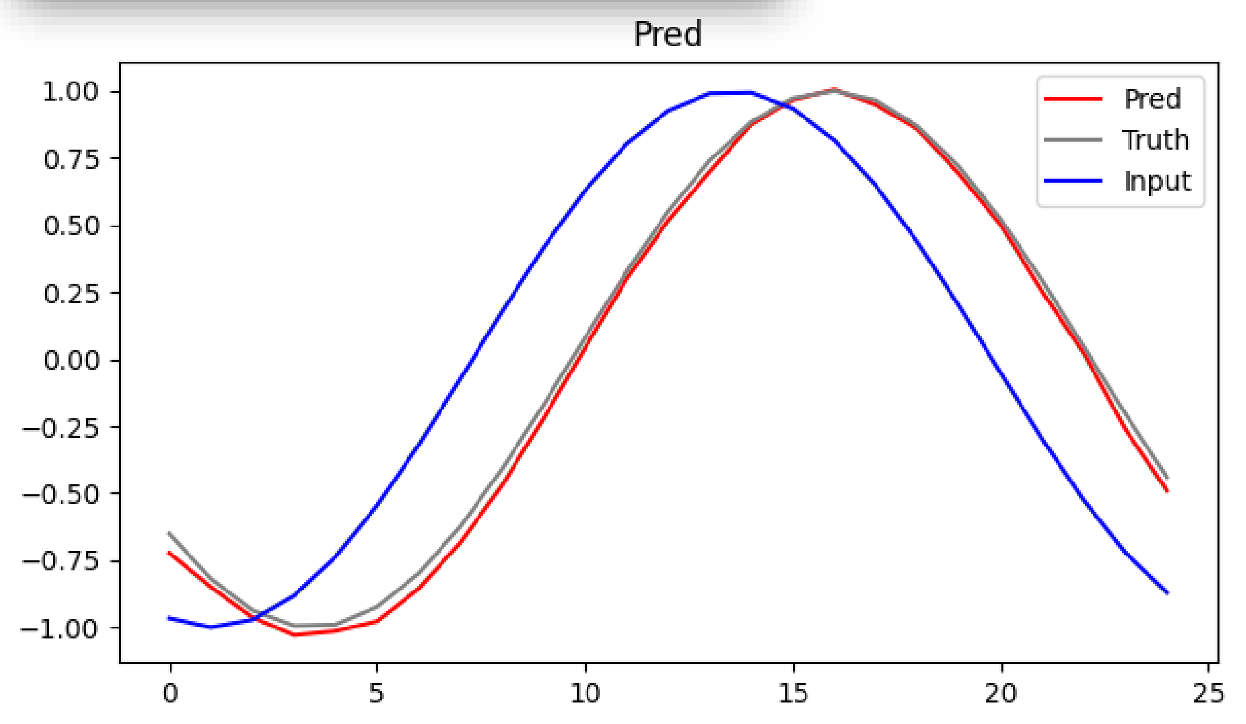


Examples from the Full Data Set









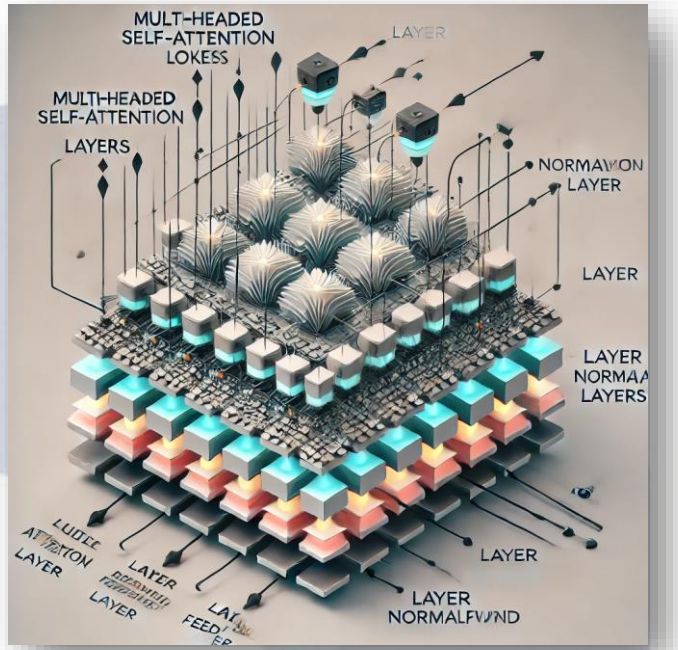
E-AI Basic Tutorials

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 Data Recovery/Denoising 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]

2.2



AI generated Images,
© Roland Potthast 2024

The idea of Encoder-Decoder Neural Networks

The encoder-decoder architecture can be used to **compress** an image into a smaller, more meaningful representation (like finding the important features) and to **reconstruct** it back into an image. This has two parts:

- **Encoder**: This part compresses (encodes) the input (image) into a smaller, abstract representation.
- **Decoder**: This part takes the compressed representation and reconstructs (decodes) it back into the original form (image).

The **encoder** takes in an image and processes it through several steps to shrink it down into a more compact form. It consists of:

- **Input Image**: A 2D image, typically represented as a matrix of pixel values (e.g., 28x28 pixels for grayscale images or 64x64 for color images).
- **Convolutional Layers**: These layers apply filters that scan over the image and extract important features, like edges, textures, or patterns. Each filter creates a "feature map" that represents the important features detected in the image.
- **Pooling Layers**: These layers downsample the feature maps, reducing the image's size while keeping important information. Think of it as shrinking the image while retaining the key patterns.
- **Bottleneck**: After several convolution and pooling steps, you reach the bottleneck layer (in orange), which is the smallest and most compressed version of the image, also called the **latent space** or **feature representation**. This captures the most essential information about the input image.

The idea of Encoder-Decoder Neural Networks

Why Use Encoder-Decoders?

- **Compression:** The encoder compresses the input image into a much smaller representation, which is useful for things like removing noise or finding only the essential information in an image.
- **Reconstruction:** The decoder allows you to reconstruct the image from the compressed version. This is useful for tasks like denoising, super-resolution (making low-res images clearer), or image segmentation (breaking down an image into different regions).

The decoder reverses the process. It takes the compressed representation (the bottleneck) and reconstructs the original image. It consists of:

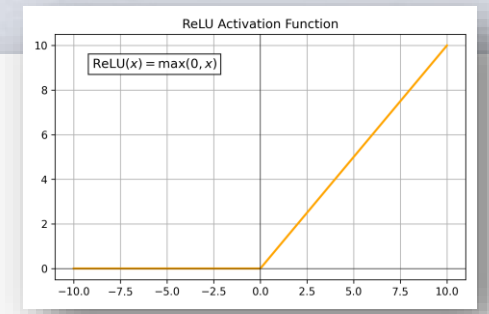
- **Upsampling Layers:** These layers gradually increase the size of the compressed representation, typically by reversing the downsampling process used by the encoder (like using ConvTranspose2d layers in PyTorch).
- **Deconvolutional Layers** (Transposed Convolutions): These layers work similarly to convolutional layers but in reverse. Instead of reducing the size, they expand the smaller feature maps back to the original image size.
- **Output Image:** The final layer reconstructs the image, trying to match the original input as closely as possible.

Define the Encoder

Convolutional Layers:

nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding):

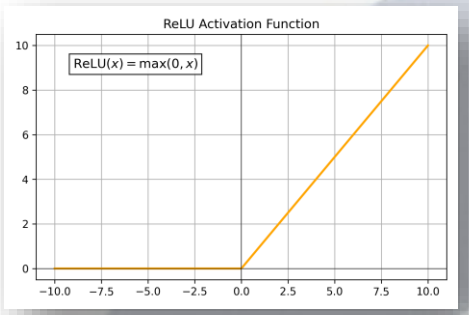
- in_channels: The number of channels in the input image. For example, 1 for grayscale images and 3 for RGB images.
- out_channels: The number of output channels (or feature maps). These represent the number of filters that are applied to the input image. In this case, the encoder will progressively learn more complex features with more filters.
- kernel_size=3: The size of the filters used in the convolution. Here, the filter size is 3x3 pixels.
- stride=2: The stride controls how much the filter moves over the input. A stride of 2 reduces the spatial dimensions of the output.
- padding=1: Padding adds extra pixels around the input to maintain the spatial dimensions of the image. In this case, the padding is set to 1 to keep the size reduction controlled.

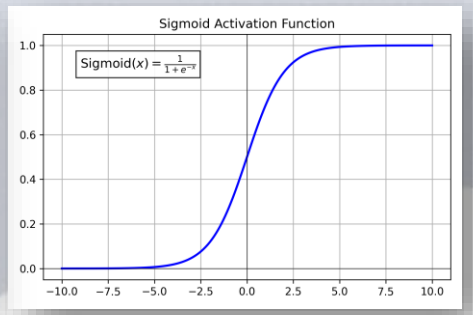



```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from random import choice

# Define the Encoder
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        return x
```

The Decoder and the Autoencoder

- kernel_size=3: The size of the convolution filter (3x3 in this case), which affects how the up sampling is performed.
- An **autoencoder** is a type of neural network used to learn efficient, compressed representations of data, often for dimensionality reduction or denoising. It consists of two main parts: an
  - encoder that compresses the input into a lower-dimensional form and
  - a decoder that reconstructs the original data from the compressed representation.

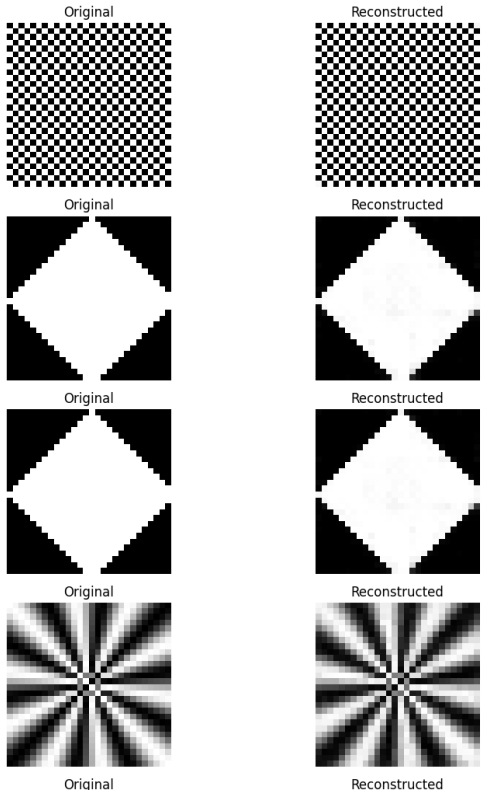





```python
# Define the Decoder
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.conv_trans1 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=0)
        self.conv_trans2 = nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2, padding=1, output_padding=1)
        self.conv_trans3 = nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2, padding=1, output_padding=1)

    def forward(self, x):
        x = torch.relu(self.conv_trans1(x))
        x = torch.relu(self.conv_trans2(x))
        x = torch.sigmoid(self.conv_trans3(x))  # Use sigmoid for pixel intensity between [0, 1]
        return x

# Create Autoencoder (Encoder + Decoder)
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Generate Examples

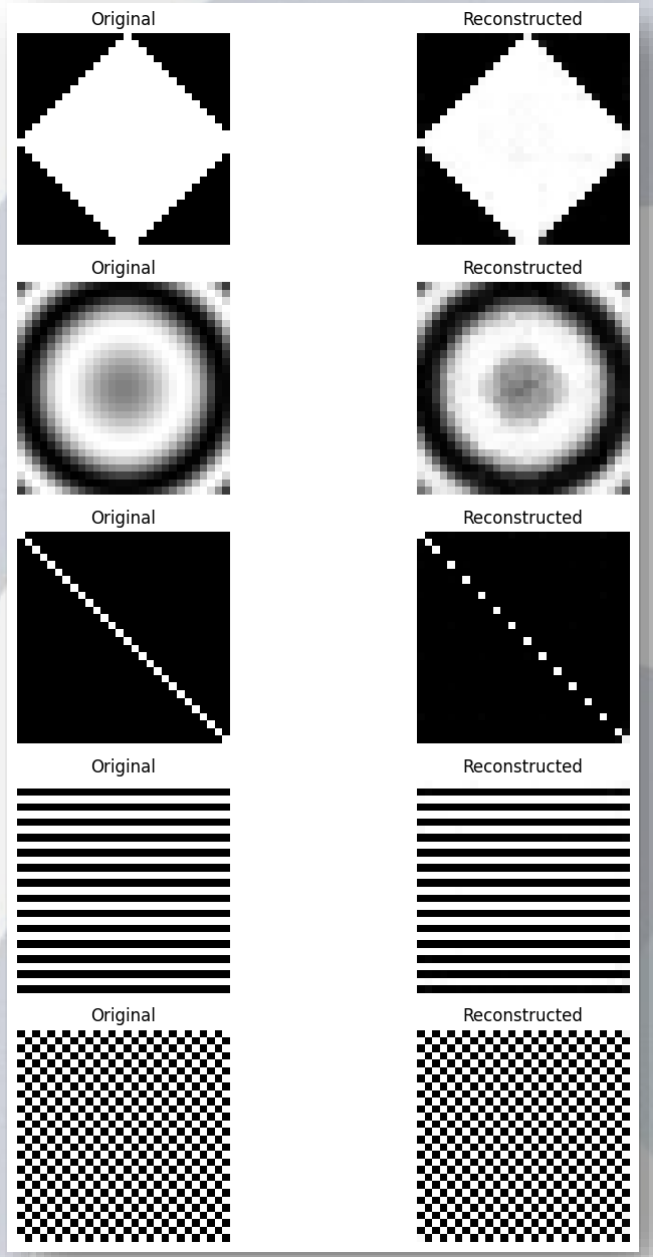
We construct some patterns to work on it with our encoder decoder architecture.




```python
def generate_checkerboard(size=28):
    z = np.indices((size, size)).sum(axis=0) % 2
    return z

def generate_star_pattern(size=28):
    x = np.linspace(-1, 1, size)
    y = np.linspace(-1, 1, size)
    x, y = np.meshgrid(x, y)
    z = np.sin(10 * np.arctan2(y, x))
    return (z - z.min()) / (z.max() - z.min())

def generate_spiral(size=28):
    x = np.linspace(-1, 1, size)
    y = np.linspace(-1, 1, size)
    x, y = np.meshgrid(x, y)
    z = np.sin(np.sqrt(x**2 + y**2) * 6)
    return (z - z.min()) / (z.max() - z.min())

def generate_noise_with_circle(size=28):
    z = np.random.rand(size, size)
    circle_mask = (np.sqrt(np.linspace(-1, 1, size)[:, None]**2
                           + np.linspace(-1, 1, size)[None, :]**2) <= 0.5)
    z[circle_mask] = 1  # Set circle region to 1
    return z
```

Construct Examples

```python
# Pattern Generators (same as before)
def generate_concentric_circles(size=28):
    x = np.linspace(-1, 1, size)
    y = np.linspace(-1, 1, size)
    x, y = np.meshgrid(x, y)
    z = np.sin(5 * (x**2 + y**2))
    return (z - z.min()) / (z.max() - z.min())

def generate_vertical_stripes(size=28):
    z = np.zeros((size, size))
    z[:, ::2] = 1  # Every second column is set to 1
    return z

def generate_horizontal_stripes(size=28):
    z = np.zeros((size, size))
    z[::2, :] = 1  # Every second row is set to 1
    return z

def generate_diagonal_lines(size=28):
    z = np.zeros((size, size))
    for i in range(size):
        z[i, i] = 1  # Create diagonal line
    return z
```



Generate a random pattern

- We define a list of all our patterns to generate.
- Then, for training we write a function which chooses from the list and generates a pattern.
- It appends the pattern to the patterns list.

```python
# Create a dictionary of patterns
pattern_generators = {
    "Concentric Circles": generate_concentric_circles,
    "Vertical Stripes": generate_vertical_stripes,
    "Horizontal Stripes": generate_horizontal_stripes,
    "Diagonal Lines": generate_diagonal_lines,
    "Checkerboard": generate_checkerboard,
    "Star Pattern": generate_star_pattern,
    "Spiral": generate_spiral,
    "Noise with Circle": generate_noise_with_circle,
    "Diamond": generate_diamond,
    "Grid of Dots": generate_grid_of_dots
}

# Training Loop
def generate_random_pattern_batch(batch_size):
    patterns = []
    for _ in range(batch_size):
        generator = choice(list(pattern_generators.values()))
        pattern = generator(28)
        patterns.append(pattern)
    return torch.tensor(np.array(patterns), dtype=torch.float32).unsqueeze(1)
```

Training Loop

- We now do the training based on standard PyTorch syntax
- For all epochs
- Run 100 steps with batches of size 16, i.e. processing a list of 16 images
- Calculate loss and
- update coefficients

```
# Instantiate the model
autoencoder = Autoencoder()
```

```
# Loss function and optimizer
criterion = nn.MSELoss()  # Reconstruction error
optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)

epochs = 20
batch_size = 16
for epoch in range(epochs):
    autoencoder.train()
    for _ in range(100):  # Simulate batches
        # Generate random batch of patterns
        batch = generate_random_pattern_batch(batch_size)

        # Forward pass
        outputs = autoencoder(batch)
        loss = criterion(outputs, batch)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

Visualization of Results

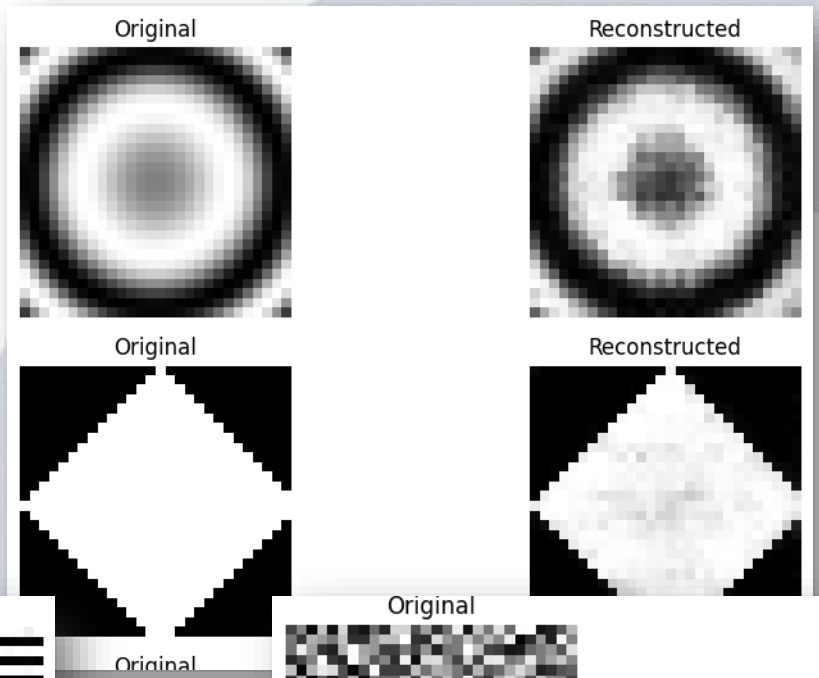
- We define a visualization routine
- Generating random input images
- Then applying the autoencoder step
- Showing input image and output image

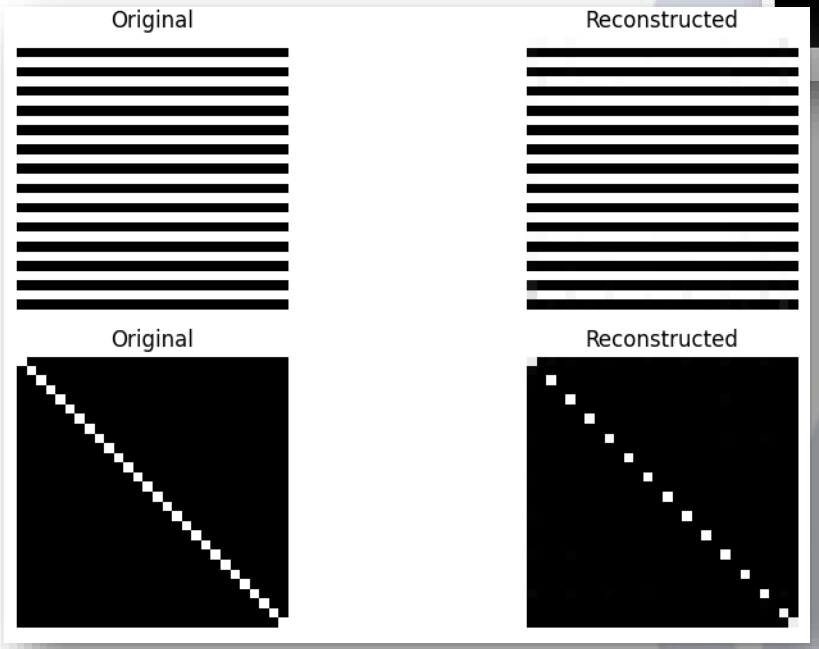
```python
# Visualize 10 random examples of input and reconstructed images
def visualize_reconstructions(autoencoder, pattern_generators):
    autoencoder.eval()
    with torch.no_grad():
        fig, axes = plt.subplots(10, 2, figsize=(10, 25))
        for i in range(10):
            generator = choice(list(pattern_generators.values()))
            pattern_image = generator(28)
            input_tensor = torch.tensor(pattern_image,
                        dtype=torch.float32).unsqueeze(0).unsqueeze(0)

            # Get reconstruction
            reconstructed = autoencoder(input_tensor)

            # Display input and reconstructed images
            axes[i, 0].imshow(pattern_image, cmap='gray')
            axes[i, 0].set_title("Original")
            axes[i, 0].axis('off')

            axes[i, 1].imshow(reconstructed[0, 0].numpy(), cmap='gray')
            axes[i, 1].set_title("Reconstructed")
            axes[i, 1].axis('off')

    plt.tight_layout()
    plt.show()
```

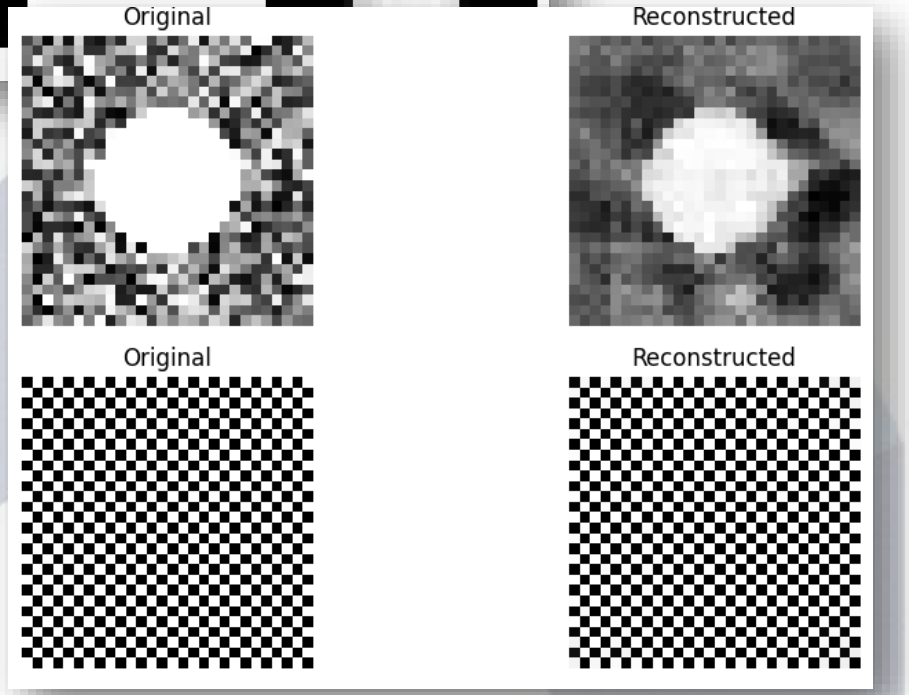
Original and Autoencoder Reconstruction

The Autoencoder applied to the original image should reconstruct the original image.

But we now have a tool where we can modify the imput, and then see if it can reconstruct the original input.







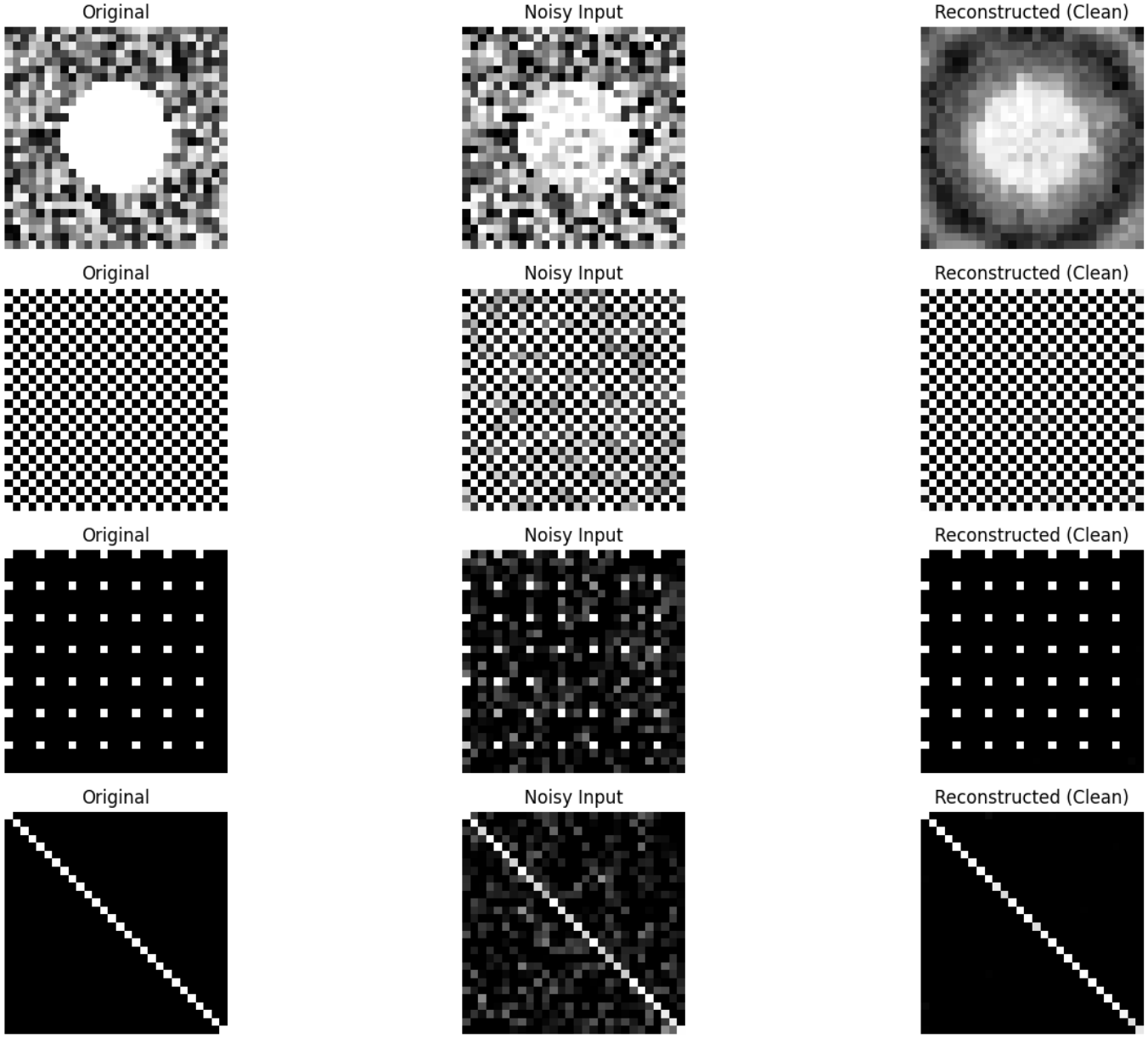
Define specific Loss Functions (e.g. Kullback-Leibler)

The KL (Kullback-Leibler) divergence is a measure of how one probability distribution differs from a second, reference probability distribution. It's commonly used in various machine learning applications, especially in the context of probabilistic models.

```python
# Variational Autoencoder (VAE) Model
class VAE(nn.Module):
    def __init__(self, latent_dim=2):
        super(VAE, self).__init__()
        self.encoder = Encoder(latent_dim)
        self.decoder = Decoder(latent_dim)

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = reparameterize(mu, logvar)
        reconstructed = self.decoder(z)
        return reconstructed, mu, logvar

# Loss function: Reconstruction + KL divergence
def vae_loss_function(reconstructed, original, mu, logvar):
    recon_loss = nn.functional.mse_loss(reconstructed, original, reduction='sum')
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + kl_loss
```

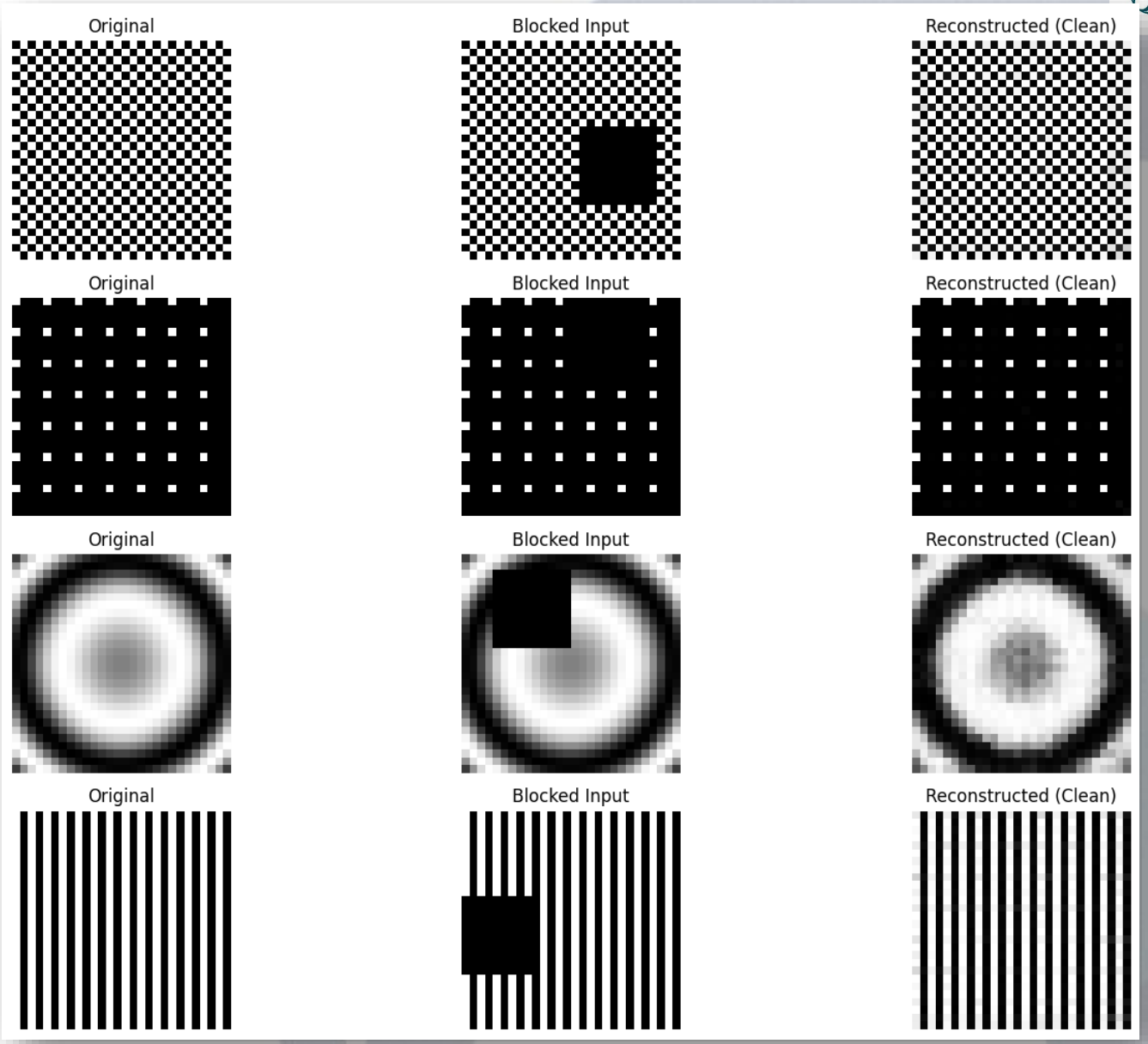
Reconstruct Noisy Images

- We can add noise to the image before reconstruction.
- This can also be taken into account during training.
- The reconstructed image can remove the noise.



Reconstruct Missing Data

- We can remove some parts of the patterns
- Let the autoencoder reconstruct the missing parts.
- This can also be added to the training loop to prepare the system.



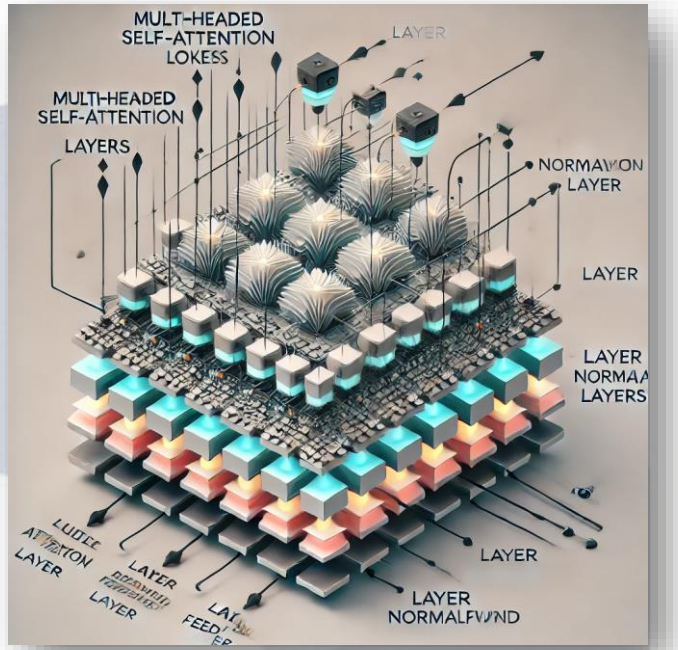
E-AI Basic Tutorials

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 Data Recovery/Denoising 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]

2.3



AI generated Images,
© Roland Potthast 2024

Data assimilation example

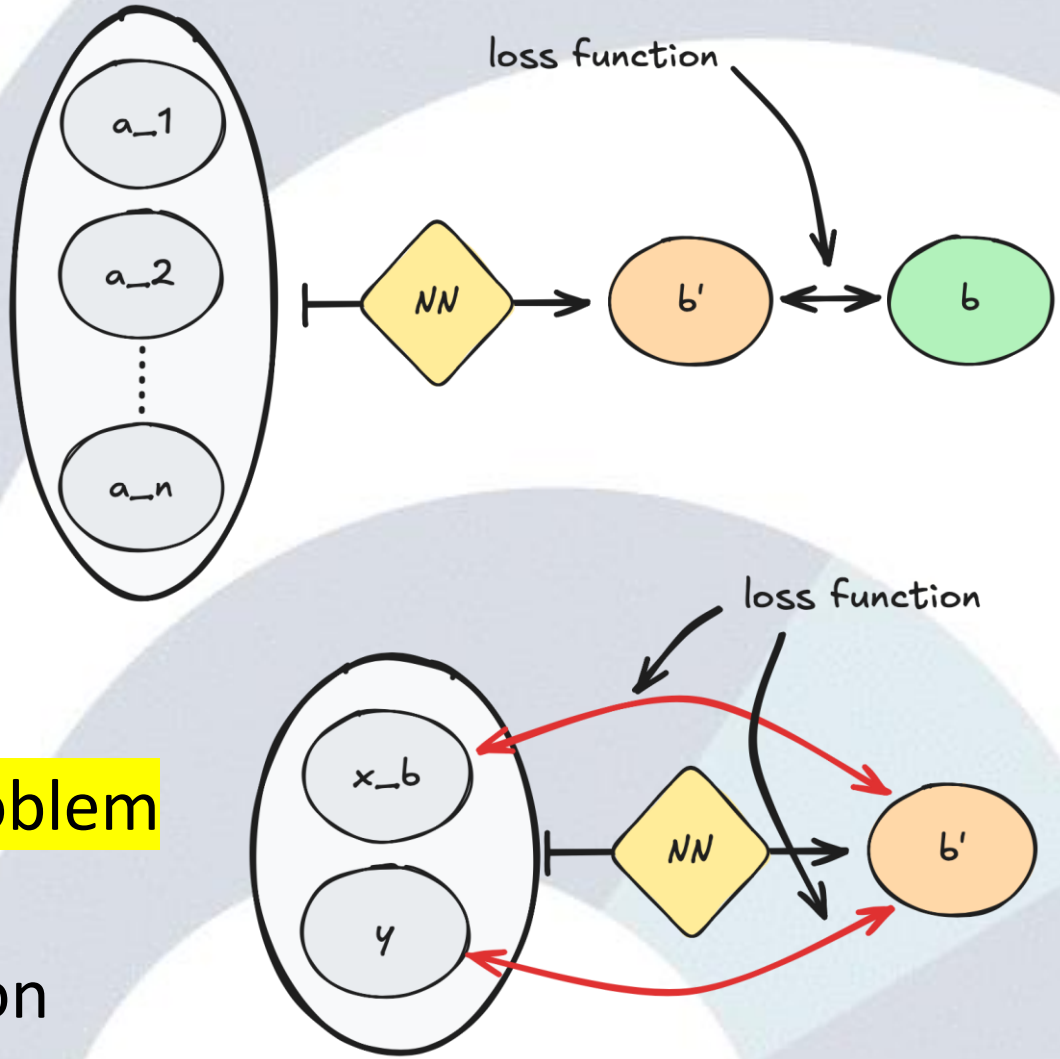
- Example with custom loss function

- Normally, NNs are trained like this:

  - b is known, i.e., the training data is sampled from $\{a_1, a_2, \ldots, a_n, b\}$

- We take a different approach as data assimilation is also an optimization problem

  → We use the data assimilation cost function as loss function

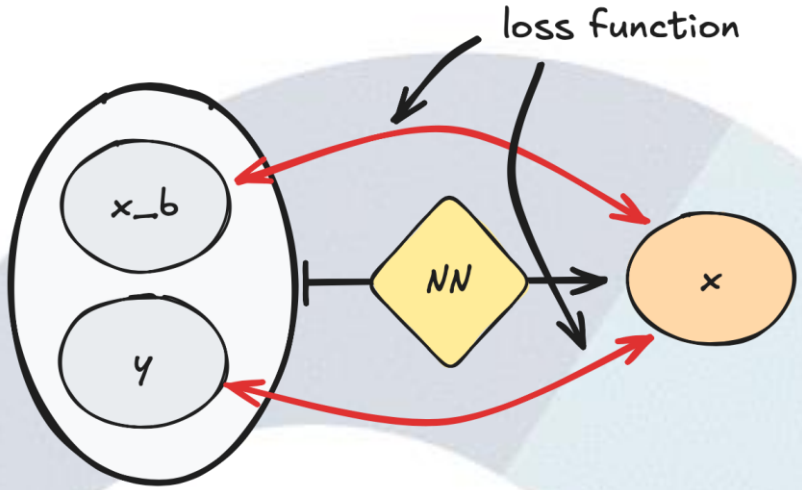


Defining the loss function

- The data assimilation cost function

$$J(x) = \underbrace{\frac{1}{2}(x - x_b)^{\mathrm{T}} B^{-1}(x - x_b)}_{\text{background}} + \underbrace{\frac{1}{2}\left(H(x) - y_o\right)^{\mathrm{T}} R^{-1}\left(H(x) - y_o\right)}_{\text{observations}}$$

- The neural network is trained like this:

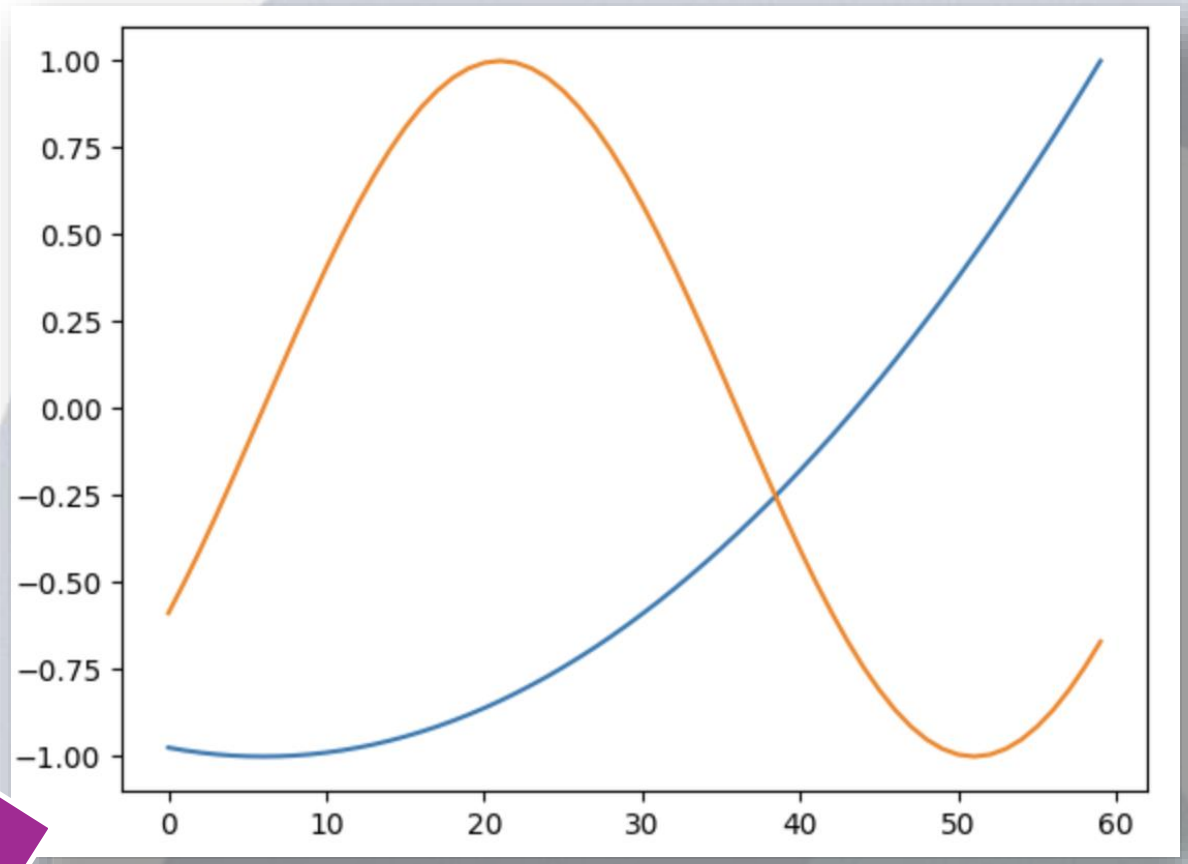


Loading packages and preparing the data

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader,Dataset
```

```
# Define grid parameters
nx         = 60
dx         = 2*np.pi/nx

# Generate spatial grid
x = np.arange(0, 2*np.pi, dx)
x  = np.round(x, 3)
```

```
# Generate a set of curves for training
n_samples = 1000
ht        = 2*np.pi/(n_samples)
vector    = ht*np.arange(0,n_samples)

# Generate two different curves
zt = np.zeros([2, n_samples, nx])
for i in range(n_samples):
    xtmp = 1*(x-np.ones(len(x))*vector[i])
    zt[0,i, :] = xtmp**2/max(xtmp**2)
    zt[1,i, :] = np.sin(x-np.ones(len(x))*vector[i])
zt[0,:, :] = zt[0,:, :]*2-1
```



Preparing the data (cont'd)

```
# Build training dataset by randomly selecting from zt
vrand = np.random.choice([0, 1], size=n_samples)
z     = np.array([zt[vrand[t],t,:] for t in np.arange(n_samples)])
z     = z[np.random.permutation(np.arange(n_samples)),:]
```

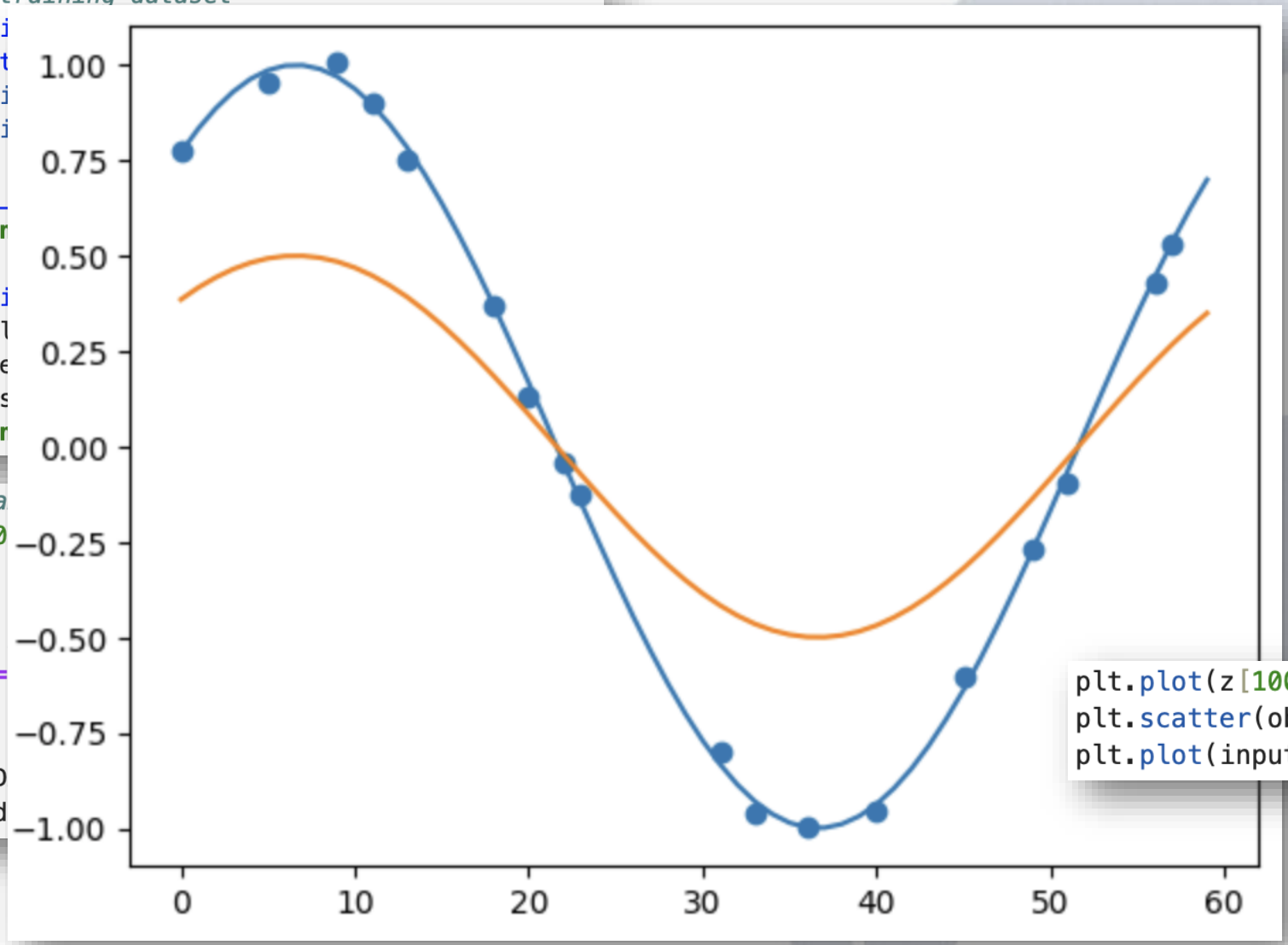
```
# Define the model equivalent calculator
def sample_obs(d, idx, tensor=False,perturb=0):
    if tensor:
        n1  = d.size()[0]
        obs = d[torch.arange(d.size(0)).unsqueeze(-1), torch.from_numpy(np.array(idx))]
    else:
        obs = d[idx]
        if perturb>0:
            obs += np.random.normal(0,perturb/2.,len(obs))
    return obs
```

```
# Define parameters
n_obs   = 18
obs_err = 0.05

# Get obs locations
random  = [np.random.choice(np.arange(10),size=3,replace=False) for i in np.arange(6)]
obs_idx = (np.array(random)+np.arange(0,60,10)[:,np.newaxis]).flatten()
obs_idx = np.sort(obs_idx)
obs_idx = np.tile(obs_idx,n_samples).reshape((n_samples,n_obs))

# Define the input for the observation values and locations
obs       = np.array([sample_obs(z[it, :], obs_idx[it,:], perturb=obs_err) for it in np.arange(n_samples)])
input_obs = torch.tensor(np.stack((obs,obs_idx/nx)))nput
```

Preparing the data (cont'd)

```
# Define the training dataset
class DATraini
    def __init
        self.i
        self.i

    def __len_
        return

    def __geti
        fg=sel
        obs=se
        oidx=s
        return
```

```
# Define para
batch_size=10
fac=0.5

# Define the
input_model =

# Initialize
train_set = D
training_load
```




```
plt.plot(z[100,:])
plt.scatter(obs_idx[100,:],obs[100,:])
plt.plot(input_model.detach().numpy()[100,:])
```

Preparing the data (cont'd)

```
# Setup the covariance matrix
def calculate_covariance_matrix(x, sigma=1):
    points = np.vstack([x.ravel()]).T
    dist_sq = np.sum((points[:, np.newaxis, :] - points[np.newaxis, :, :]) ** 2, axis=-1)
    covariance_matrix = np.exp(-dist_sq / (2 * sigma ** 2))
    return covariance_matrix

# Calculate the background covariance matrix with regularization
sigma=1.5
b_cov = torch.tensor(calculate_covariance_matrix(x, sigma=sigma))*0.5
lambda_identity = 0.001 * torch.eye(b_cov.shape[0])
regularized_b_cov = b_cov + lambda_identity
```

Network and Loss function

```python
# Define the model
class DataAssimilationNN(nn.Module):
    def __init__(self,nx,n_obs):
        super(DataAssimilationNN,self).__init__()
        self.fc1=nn.Linear(nx+2*n_obs,(nx+n_obs)**2) # First hidden layer
        self.fc2=nn.Linear((nx+n_obs)**2,nx)  # Second hidden layer
        self.fc3=nn.Linear(nx,nx) # Output layer

    def forward(self,xin):
        x = torch.cat((xin["fg"].float(), xin["obs"].float(), xin["idx"].float()),dim=1)
        x = torch.tanh(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x.view(-1,nx).double()
```

Network and Loss function

```python
# Define the loss function
class DataAssimilationLoss(nn.Module):
    def __init__(self, b_matrix=None,nx=None,obs_err=None):
        super(DataAssimilationLoss,self).__init__()
        self.b_matrix=b_matrix
        self.nx=nx
        self.obs_err=obs_err

    def forward(self,ana_inc,inputs):
        # Calculate distance from background
        inv_b_matrix=torch.inverse(self.b_matrix)
        bg_mismatch=torch.einsum('bi,ij,bj->b',ana_inc,inv_b_matrix,ana_inc).mean()/len(ana_inc)
        # Calculate distance from observations
        o_idx = (inputs["idx"]*nx).numpy().astype(int)
        y_hat=sample_obs(ana_inc.double(), o_idx, tensor=True)
        obs_mismatch=torch.mean((y_hat-inputs["obs"])**2)/((self.obs_err**2)*inputs["obs"].shape[1])
        # Calculate total loss
        total_loss=bg_mismatch+obs_mismatch
        return total_loss,bg_mismatch,obs_mismatch
```

```python
# Setup for Training
model = DataAssimilationNN(nx, n_obs) # Set up the model
criterion = DataAssimilationLoss(b_matrix=regularized_b_cov, nx=nx, obs_err=obs_err) # Loss function
optimizer = optim.Adam(model.parameters(), lr=0.0025) # Initialize optimizer

# Define the number of training epochs.
num_epochs = 1000
train_loss = []
```

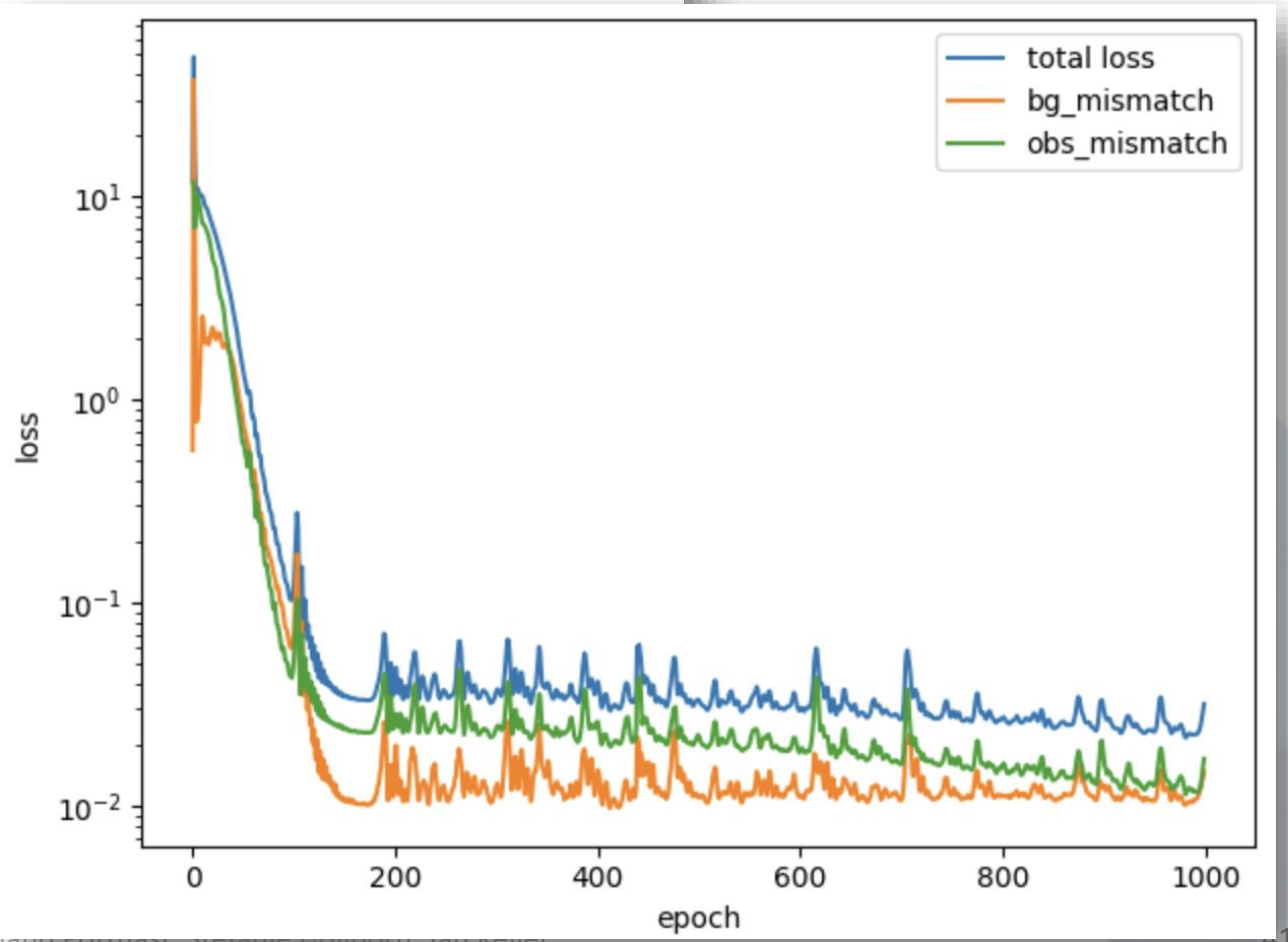
Training loop

```
# Training loop
for epoch in range(num_epochs):
    for i_batch, train_data in enumerate(training_loader): # Iterate ov
        optimizer.zero_grad()  # Clear gradients
        pred = model(train_data) # Forward pass
        loss, fg_loss, obs_loss = criterion(pred, train_data) # Compute
        loss.backward() # Backward pass
        optimizer.step() # Optimization step (parameter update).
    train_loss.append([loss.item(), fg_loss.item(), obs_loss.item()],
    if (epoch+1) % 20 == 0:
        print('Epoch [{:04d}/{:04d}], '.format(epoch+1,num_epochs)+
              'Loss:      {:12.6f} '.format(train_loss[-1][0])+
                         '{:12.6f} '.format(train_loss[-1][1])+
                         '{:12.6f} '.format(train_loss[-1][2]))
```

```
Epoch [0020/1000], Loss:      7.418088     2.250303     5.167785
Epoch [0040/1000], Loss:      2.945354     1.570370     1.374985
Epoch [0060/1000], Loss:      0.814474     0.441097     0.373376
Epoch [0080/1000], Loss:      0.243199     0.146268     0.096931
Epoch [0100/1000], Loss:      0.109769     0.063325     0.046444
Epoch [0120/1000], Loss:      0.052472     0.020258     0.032214
Epoch [0140/1000], Loss:      0.036799     0.011749     0.025049
Epoch [0160/1000], Loss:      0.033744     0.010363     0.023381
Epoch [0180/1000], Loss:      0.034473     0.010763     0.023710
Epoch [0200/1000], Loss:      0.039673     0.014411     0.025261
Epoch [0220/1000], Loss:      0.057310     0.017873     0.039437
Epoch [0240/1000], Loss:      0.044590     0.016141     0.028449
Epoch [0260/1000], Loss:      0.038121     0.013651     0.024470
Epoch [0280/1000], Loss:      0.039294     0.013798     0.025496
Epoch [0300/1000], Loss:      0.035736     0.012314     0.023422
Epoch [0320/1000], Loss:      0.047304     0.017475     0.029829
Epoch [0340/1000], Loss:      0.039073     0.014817     0.024257
Epoch [0360/1000], Loss:      0.037360     0.012978     0.024382
Epoch [0380/1000], Loss:      0.034245     0.013123     0.021123
Epoch [0400/1000], Loss:      0.039233     0.013401     0.025832
Epoch [0420/1000], Loss:      0.030800     0.009936     0.020863
Epoch [0440/1000], Loss:      0.060790     0.021876     0.038914
Epoch [0460/1000], Loss:      0.033874     0.012375     0.021499
Epoch [0480/1000], Loss:      0.035382     0.013769     0.021614
Epoch [0500/1000], Loss:      0.033482     0.011092     0.022390
Epoch [0520/1000], Loss:      0.031248     0.011197     0.020051
Epoch [0540/1000], Loss:      0.032537     0.012823     0.019714
Epoch [0560/1000], Loss:      0.034860     0.013581     0.021279
Epoch [0580/1000], Loss:      0.030087     0.011104     0.018983
Epoch [0600/1000], Loss:      0.029402     0.011070     0.018332
Epoch [0620/1000], Loss:      0.042756     0.015781     0.026975
Epoch [0640/1000], Loss:      0.030691     0.012069     0.018622
Epoch [0660/1000], Loss:      0.028503     0.011042     0.017461
Epoch [0680/1000], Loss:      0.030532     0.011165     0.019366
Epoch [0700/1000], Loss:      0.026641     0.010612     0.016029
Epoch [0720/1000], Loss:      0.029542     0.011799     0.017743
Epoch [0740/1000], Loss:      0.027402     0.011324     0.016078
Epoch [0760/1000], Loss:      0.026576     0.010764     0.015812
Epoch [0780/1000], Loss:      0.028498     0.012358     0.016139
Epoch [0800/1000], Loss:      0.026903     0.011328     0.015575
```

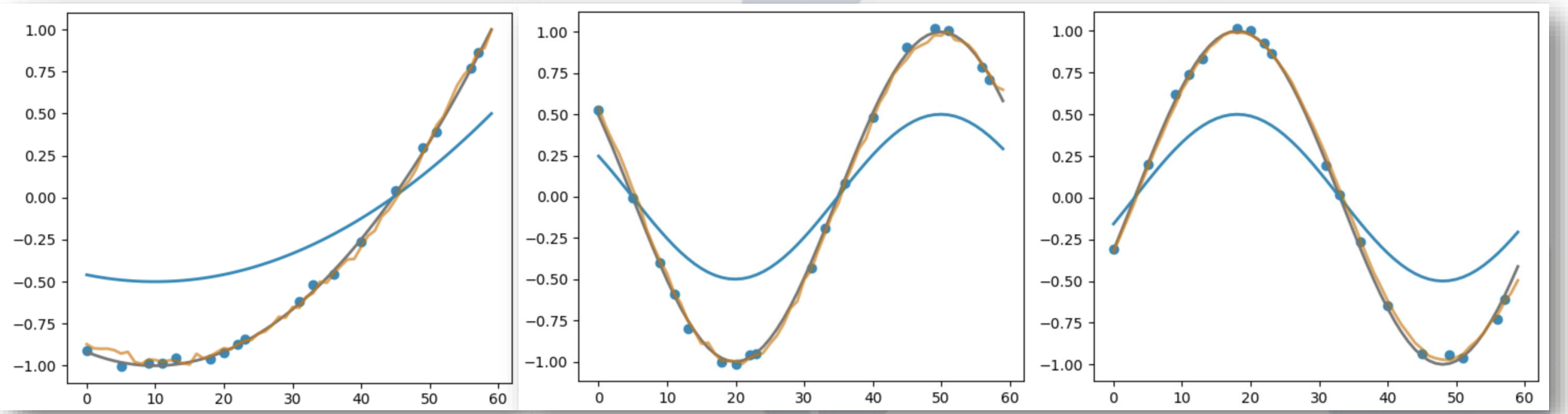
Training loop

```
# Loss curve
plt.plot(np.arange(num_epochs),np.array(train_loss)[:,0],label="total loss")
plt.plot(np.arange(num_epochs),np.array(train_loss)[:,1],label="bg_mismatch")
plt.plot(np.arange(num_epochs),np.array(train_loss)[:,2],label="obs_mismatch")
plt.yscale('log')
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.tight_layout()
```



Results

```
output=model(train_data)
fig,axs=plt.subplots(2, 2, figsize=(10, 8))
for i in np.arange(4):
    ax=axs[int(i/2),i%2]
    ax.plot(z[train_data["no"].detach().numpy()[i*100],:],color="#777777",linewidth=2)
    ax.plot(train_data["fg"].detach().numpy()[i*100,:],color="#0088bb",linewidth=2)
    ax.scatter(train_data["idx"].detach().numpy()[i*100,:]*nx,train_data["obs"].detach().numpy()[i*100,:],color="#0088bb")
    ax.plot(output.detach().numpy()[i*100,:],color="#dd7700aa",linewidth=2)
fig.tight_layout()
```

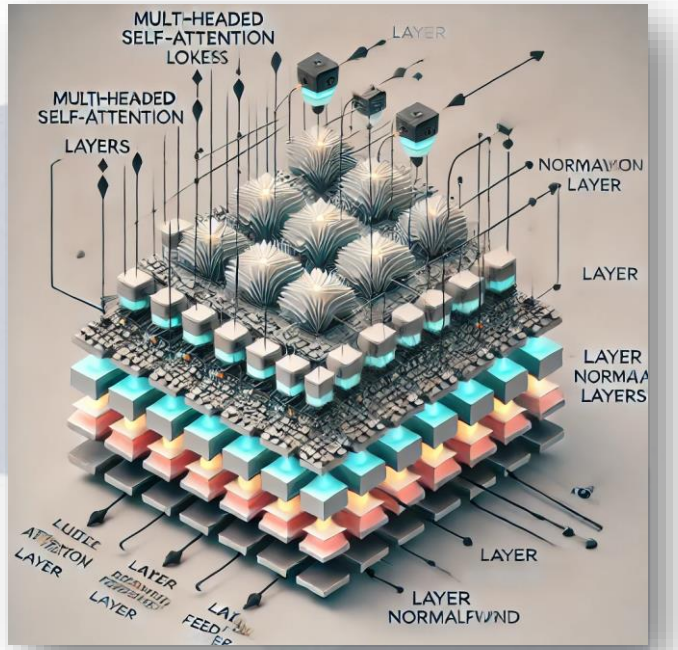


E-AI Basic Tutorials

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 Data Recovery/Denoising 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