A hands-on introduction to the machine learning framework

PyTorch

Studierendentage 2025
7-11 April
Fachbereich Physik
Ruprecht-Karls-Universität Heidelberg

Slides credits: Sebastian Dittmeier

About me

Giulia Fazzino, PhD

Experimental
Particle Physics

- PhD Focused on Graph Neural Networks (since November 2024)
- Context: Online Track Reconstruction with FPGAs for the ATLAS experiment @ CERN

- Background
    - Master in Particle Physics in 2023 (Clermont Ferrand, Dortmund & Bologna)
    - Master thesis on the use of Deep Neural Networks for Pile-up Suppression for the ATLAS experiment

Slides credits: Sebastian Dittmeier

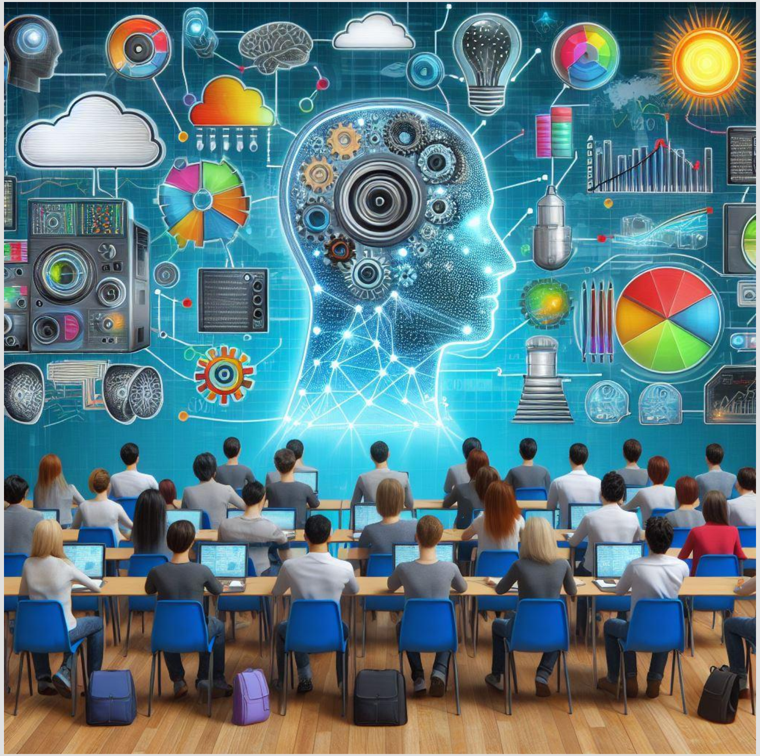
What can you expect in the coming days?

Today, 07.04.2025	The Basics *MNIST, Linear Regression*
Tuesday, 08.04.2025	A deeper dive *CNNs @ MNIST, RNNs @ names*
Wednesday, 09.04.2025	The Problem *Tracking, TrackML kNN search*
Thursday, 10.04.2025	The Solution *Graph Neural Networks, PyTorch Geometric, TrackML GNN*
Friday, 11.04.2025	The Add-On *Fun & Games*

How does it work?

- Course in English
- Slides are linked on the website of this course
- Links to corresponding notebooks are given on the slides, and also available via the website of this course
- You can run the notebooks via
    - https://colab.research.google.com (possibly with GPU)
    - https://jupyter.kip.uni-heidelberg.de

Quick Intro
Machine Learning



created with https://designer.microsoft.com/image-creator

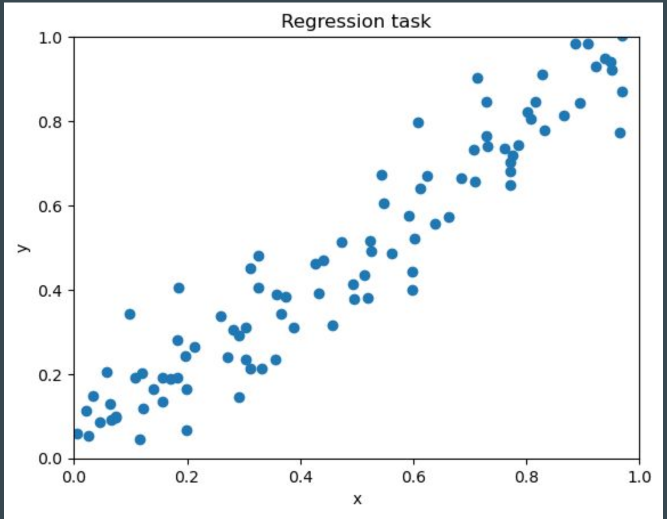
What is Machine Learning?

- Arthur Samuel (1959): Field of study that gives computers the ability to learn without being explicitly programmed
- Tom Mitchell (1998): It is a computer program that learns from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with E.

On February 24, 1956, Arthur Samuel's Checkers program, which was developed for play on the IBM 701, was demonstrated on public television. [source]

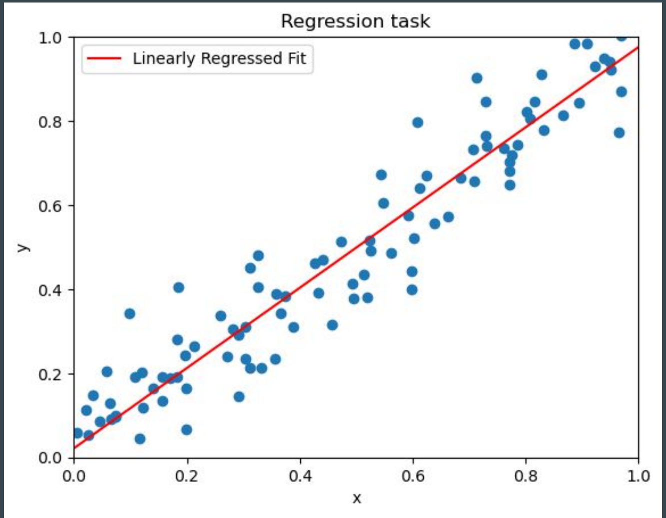
Supervised Learning

- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y



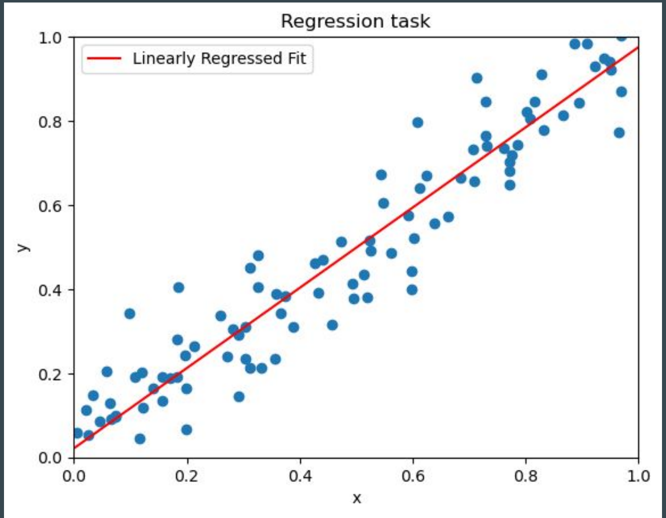
Supervised Learning

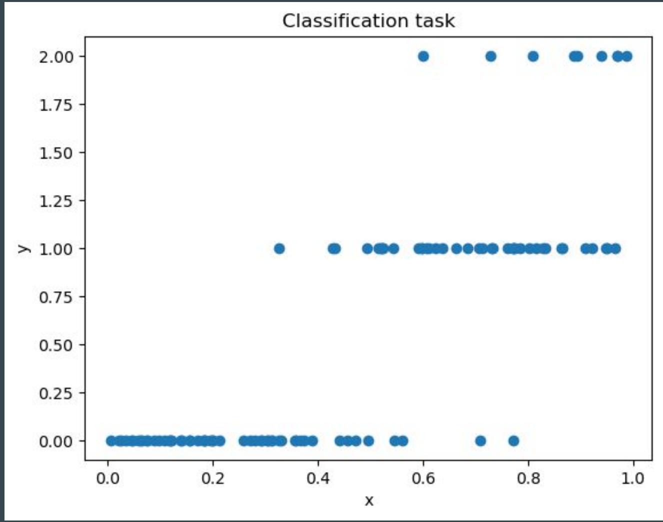
- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y



Supervised Learning

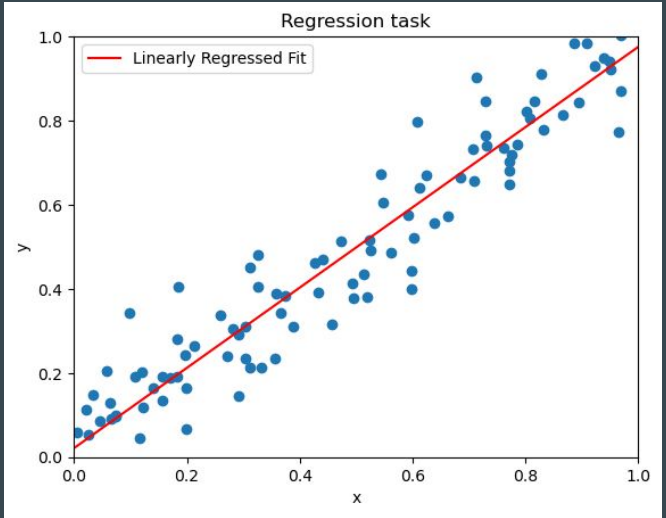
- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y
- **Classification** : discrete y

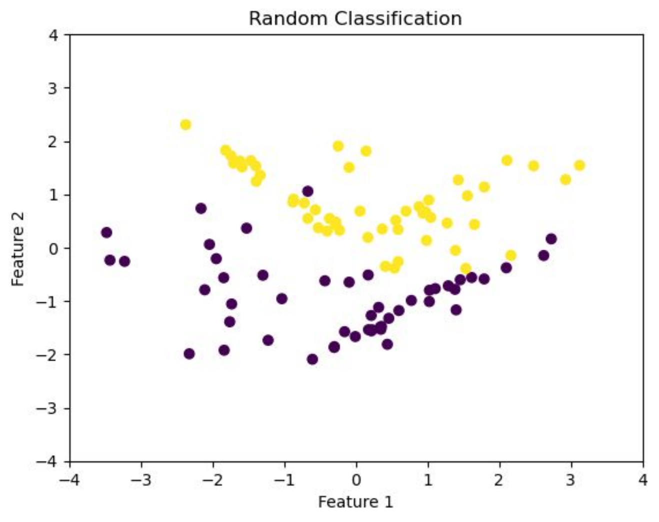




Supervised Learning

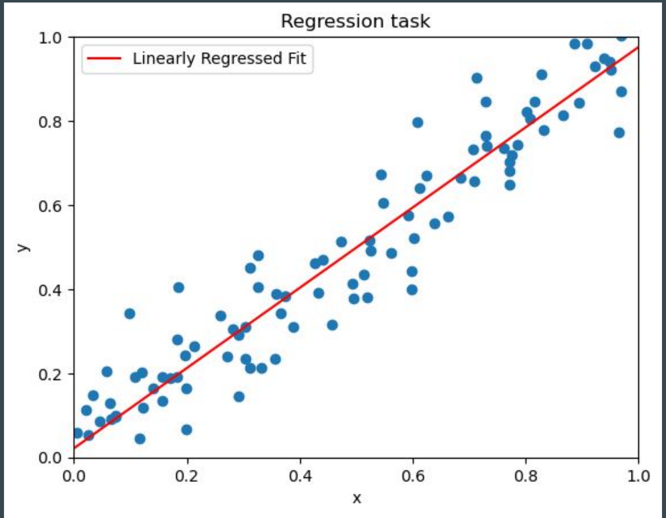
- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y
- **Classification** : discrete y
- In general:
  multidimensional x and y

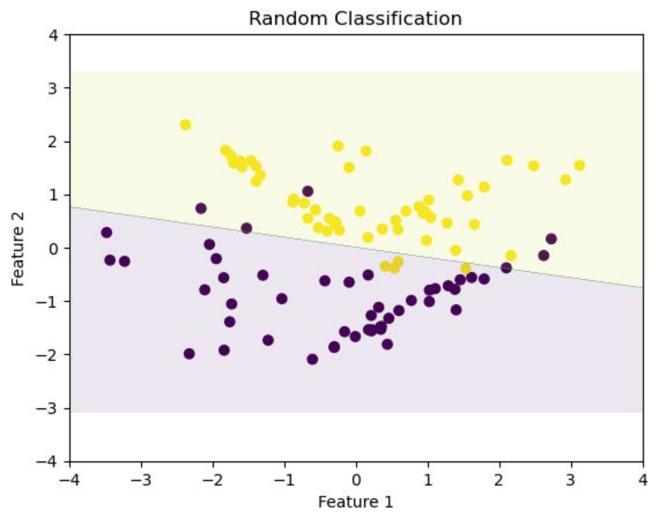




Supervised Learning

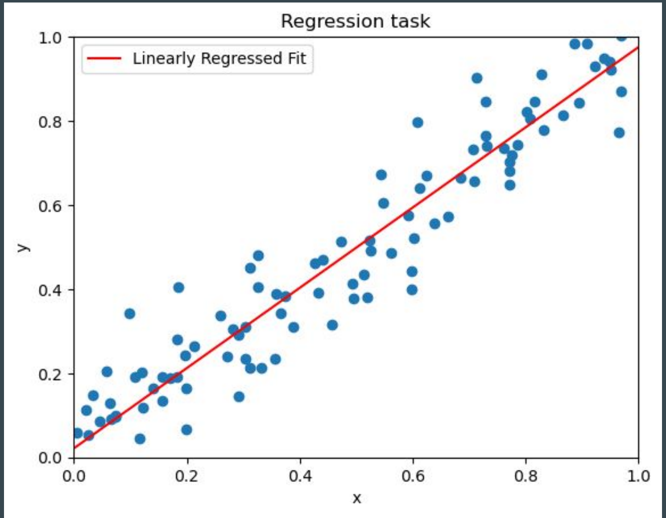
- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y
- **Classification** : discrete y
- In general:
  multidimensional x and y

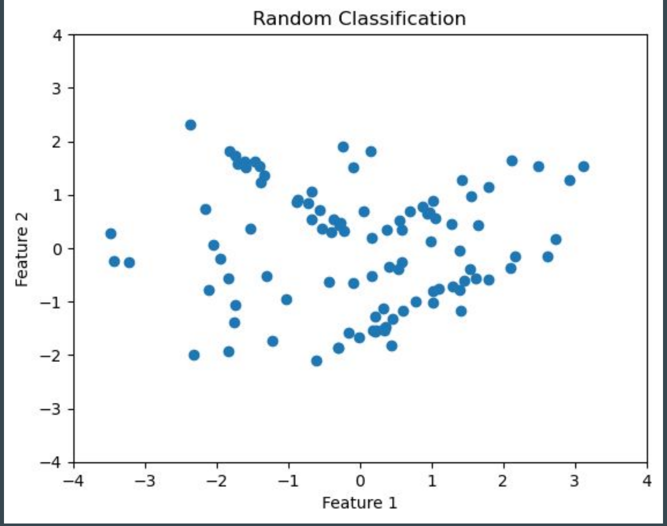




Supervised Learning

- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y
- **Classification** : discrete y
- In general:
  multidimensional x and y
- Unsupervised learning
  - No labels
  - Find interesting structure in data



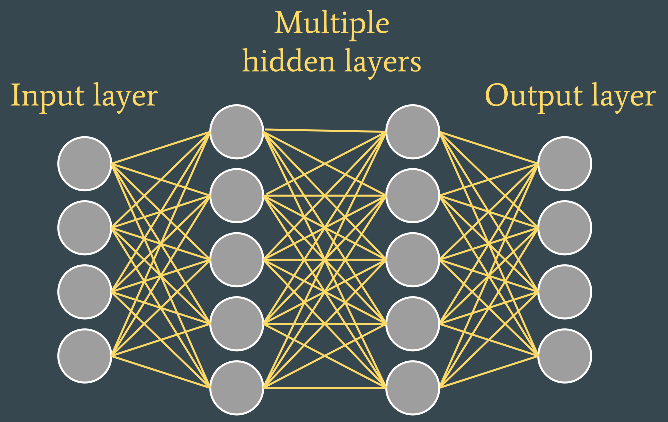


Supervised Learning

- Data set
  (inputs x, labels y)
  learn mapping $x \rightarrow y$
- **Regression** : continuous y
- **Classification** : discrete y
- In general:
  multidimensional x and y
- Unsupervised learning
  - No labels
  - Find interesting structure in data
- Reinforcement learning

Deep Learning

- Subset of machine learning based on artificial neural networks with representation learning
- Multiple layers of interconnected neurons
- Many different architectures
  - Deep Neural Networks (or Multi-Layer Perceptrons)
  - Convolutional Neural Networks
  - Recurrent Neural Networks
  - Transformers
  - Graph Neural Networks
  - ...



PyTorch

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

https://pytorch.org/

PyTorch

The BASICS

Tensors, Datasets & Models

→ PyTorch Cheat Sheet

created with https://designer.microsoft.com/image-creator

Tensors

- Specialized data structure, very similar to arrays and matrices
- Similar to NumPy's `ndarrays`
  → but can run on GPU
  - Share same underlying memory!
- Optimized for automatic differentiation

[→ Link to Notebook]()

[→ Link to documentation]()

```
import torch
import numpy as np

# initialize a tensor from data
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

# initialize a tensor from a numpy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)

# initialize a tensor with random or constant values
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

# initialize a tensor from another tensor
x_ones = torch.ones_like(x_data)
# retains the properties of x_data
x_rand = torch.rand_like(x_data, dtype=torch.float)
# overrides the datatype of x_data
```

Tensors

- Specialized data structure, very similar to arrays and matrices
- Similar to NumPy's `ndarrays`
  → but can run on GPU
  - Share same underlying memory!
- Optimized for automatic differentiation

→ Link to Notebook

→ Link to documentation

```python
# Attributes of a Tensor
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Tensors

- Specialized data structure, very similar to arrays and matrices
- Similar to NumPy's `ndarrays`
  → but can run on GPU
  - Share same underlying memory!
- Optimized for automatic differentiation

[→ Link to Notebook]

[→ Link to documentation]

```
# Operations on Tensors
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")

# Standard numpy-like indexing and slicing
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)

# Joining tensors
t1 = torch.cat([tensor, tensor], dim=0)
# along existing dimension
print(t1)
t2 = torch.stack([tensor, tensor], dim=0)
# creates a new dimension
print(t2)
```

Tensors

- Specialized data structure, very similar to arrays and matrices
- Similar to NumPy's `ndarrays`
  → but can run on GPU
  - Share same underlying memory!
- Optimized for automatic differentiation

→ Link to Notebook

→ Link to documentation

```
# Arithmetic operations
# This computes the matrix multiplication between two
tensors. y1, y2, y3 will have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)
y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)
print(f"y1 = {y1} \ny2 = {y2} \ny3 = {y3}")

# This computes the element-wise product. z1, z2, z3 will
have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
print(f"z1 = {z1} \nz2 = {z2} \nz3 = {z3}")
```

Tensors

- Specialized data structure, very similar to arrays and matrices
- Similar to NumPy's `ndarrays`
  → but can run on GPU
  - Share same underlying memory!
- Optimized for automatic differentiation

[→ Link to Notebook]

[→ Link to documentation]

```python
# Bridge with NumPy
# Tensor to NumPy array
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")

# A change in the tensor reflects in the NumPy array
t.add_(1)    # in-place addition
print(f"t: {t}")
print(f"n: {n}")

# NumPy array to Tensor
n = np.ones(5)
t = torch.from_numpy(n)

# Changes in the NumPy array reflects in the tensor
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

Datasets & DataLoaders

- Decouple code
  dataset ↔ model training
- `torch.utils.data.Dataset`
  stores samples and labels
- `torch.utils.data.DataLoader`
  wraps iterable around it

[→ Link to Notebook]

[→ Link to documentation]

```python
# Loading a dataset: MNIST
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

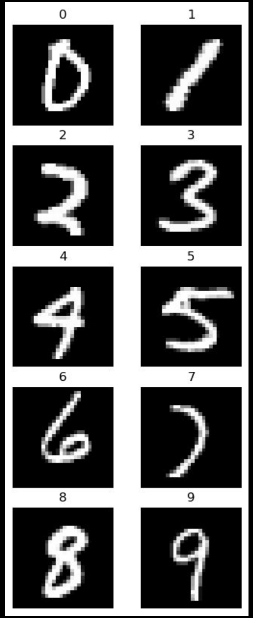
Datasets & DataLoaders

- Decouple code
  dataset ↔ model training
- `torch.utils.data.Dataset`
  stores samples and labels
- `torch.utils.data.DataLoader`
  wraps iterable around it

→ Link to Notebook

→ Link to documentation

```
# Iterating and Visualizing the Dataset
figure = plt.figure(figsize=(4, 10))
cols, rows = 2, 5
label = -1
for i in range(1, cols * rows + 1):
    while (label != (i-1)):
        sample_idx = torch.randint(len(training_data),
size=(1,)).item()
        img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



Datasets & DataLoaders

- Decouple code
  dataset ↔ model training
- `torch.utils.data.Dataset`
  stores samples and labels
- `torch.utils.data.DataLoader`
  wraps iterable around it

[→ Link to Notebook]

[→ Link to documentation]

```
# Creating a custom dataset
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset (Dataset):
    # The __init__ method is run once when instantiating the Dataset object.
    # img_dir is the directory where the images are stored
    # annotations_file could be a CSV file with image file names and labels
    # example: img1.jpg, 0
    def __init__(self, annotations_file , img_dir, transform=None,
target_transform =None):
        self.img_labels = pd.read_csv(annotations_file )
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    # The __len__ method returns the number of samples in our dataset.
    def __len__(self):
        return len(self.img_labels)

    # The __getitem__ method loads and returns a sample from the dataset at the
given index idx.
    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels .iloc[idx, 0])
        image = read_image( img_path)
        label = self.img_labels .iloc[idx, 1]
        if self.transform :
            image = self.transform (image)
        if self.target_transform :
            label = self.target_transform (label)
        return image, label
```

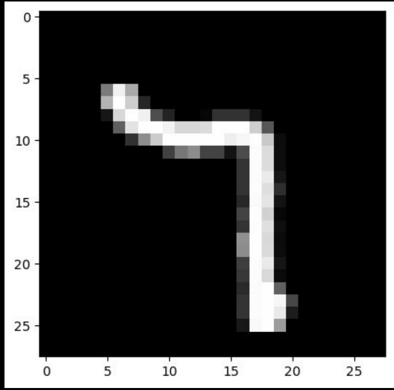
Datasets & DataLoaders

- Decouple code
  dataset ↔ model training
- `torch.utils.data.Dataset`
  stores samples and labels
- `torch.utils.data.DataLoader`
  wraps iterable around it

[→ Link to Notebook]

[→ Link to documentation]

```
# Preparing the data for training with DataLoaders
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64,
shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64,
shuffle=True)

# Iterate through the DataLoader
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
# squeeze removes all dimensions of size 1
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```



Transforms

- Data does not always come in form required for machine learning
- `torchvision.transforms` modify features and labels

```
from torchvision.transforms import Lambda

ds = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10,
dtype=torch.float).scatter_(0, torch.tensor(y), value=1))
)
# Lambda transforms apply any user-defined lambda function.
Here, we define a function to turn the integer into a one-hot
encoded tensor. It first creates a zero tensor of size 10
(the number of labels in MNIST), and calls scatter_ which
assigns a value=1 on the index as given by the label y.

ds_dl = DataLoader(ds, batch_size=64, shuffle=True)
train_features, train_labels = next(iter(ds_dl))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

→ Link to Notebook

→ Link to documentation

Build the Neural Network

- Neural Networks comprise of layers/modules
- `torch.nn`
  provides all building blocks
- Nested structure allows building complex structures

[→ Link to Notebook]

[→ Link to documentation]

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

Build the Neural Network

- Neural Networks comprise of layers/modules
- `torch.nn`
  provides all building blocks
- Nested structure allows building complex structures

[→ Link to Notebook]()

[→ Link to documentation]()

```
# Neural Network definition for processing MNIST dataset
# Initialization and definition of forward pass
# The forward pass is the sequence of computations
# that are applied to the input data to generate the output.
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
# 2D image flattened to 1D tensor
        self.linear_relu_stack = nn.Sequential(
# Sequential container
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Build the Neural Network

- Neural Networks comprise of layers/modules
- `torch.nn`
  provides all building blocks
- Nested structure allows building complex structures

[→ Link to Notebook]

[→ Link to documentation]

```
# Create an instance of the NeuralNetwork class
model = NeuralNetwork().to(device)
print(model)

# pass input data through the model (with background
operations)
# don't call model.forward() directly!
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
# Applies the Softmax function to an n-dimensional input
Tensor
# rescaling them so that the elements of the n-dimensional
output
# Tensor lie in the range [0,1] and sum to 1.
```

Build the Neural Network

- Neural Networks comprise of layers/modules
- `torch.nn`
  provides all building blocks
- Nested structure allows building complex structures

→ Link to Notebook

→ Link to documentation

```
# Model Layers
input_image = torch.rand(3, 28, 28)
print(input_image.size())

flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())

layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())

print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3, 28, 28)
logits = seq_modules(input_image)

print(f"logits: {logits}")
softmax_fn = nn.Softmax(dim=1)
pred_probab = softmax_fn(logits)
print(f"pred_probab: {pred_probab}")
```

Build the Neural Network

- Neural Networks comprise of layers/modules
- `torch.nn`

  provides all building blocks
- Nested structure allows building complex structures

→ Link to Notebook

→ Link to documentation

```
#Model Parameters

print("Model structure: ", model, "\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} |
Values : {param[:2]} \n")
```

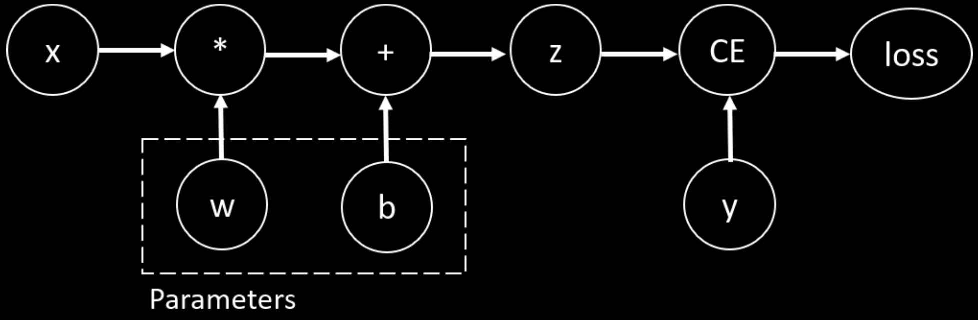
Automatic Differentiation & Optimization

created with https://designer.microsoft.com/image-creator

Automatic Differentiation

- Training neural networks
  → back propagation
- Parameters are adjusted according to the gradient of the loss function wrt parameters
- `torch.autograd`
  supports automatic computation of gradient for any computational graph

```
# simple one-layer neural network
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True) # weights
b = torch.randn(3, requires_grad=True)  # bias
z = torch.matmul(x, w)+b
loss =
torch.nn.functional.binary_cross_entropy_with_logits(z, y)

print(f"Gradient function for z ={z.grad_fn}")
print(f"Gradient function for loss ={loss.grad_fn}")
```



→ Link to Notebook

→ Link to documentation

Automatic Differentiation

- Training neural networks
  → back propagation
- Parameters are adjusted according to the gradient of the loss function wrt parameters
- `torch.autograd`
  supports automatic computation of gradient for any computational graph

→ Link to Notebook

→ Link to documentation

```
# Computing Gradients
loss.backward()
print(w.grad)
print(b.grad)

# Disabling Gradient Tracking
z = torch.matmul(x, w)+b
print(z.requires_grad)

with torch.no_grad():
    z = torch.matmul(x, w)+b
print(z.requires_grad)

z = torch.matmul(x, w)+b
z_det = z.detach()
print(z_det.requires_grad)
```

Training & Optimization

- Train, validate and test the model
- Training: iterative process
  - Guess the output
  - Calculate the loss
  - Collect derivatives
  - Optimize using gradient descent
- Choice of optimizer depends on the task, data, resources, ...

[→ Link to Notebook]()

[→ Link to documentation]()

```
# Re-using the code from previous notebooks
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

Training & Optimization

- Train, validate and test the model
- Training: iterative process
  - Guess the output
  - Calculate the loss
  - Collect derivatives
  - Optimize using gradient descent
- Choice of optimizer depends on the task, data, resources, …

→ Link to Notebook

→ Link to documentation

```
# set hyperparameters
learning_rate = 1e-3
batch_size = 64
epochs = 10

# Initialize the loss function
# In this case, we use CrossEntropyLoss for classification
# Regression problems would use MSELoss
loss_fn = nn.CrossEntropyLoss()

# Initialize the optimizer, here: Stochastic Gradient
Descent
# other options: Adam, RMSprop, etc.
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
```

Training & Optimization

- Train, validate and test the model
- Training: iterative process
  - Guess the output
  - Calculate the loss
  - Collect derivatives
  - Optimize using gradient descent
- Choice of optimizer depends on the task, data, resources, ...

[→ Link to Notebook]()

[→ Link to documentation]()

```
# loops over our optimization code
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch
normalization and dropout layers
    # Unnecessary in this situation but added for best
practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size
+ len(X)
            print(f"loss: {loss:>7f}
[{current:>5d}/{size:>5d}]")
```

Training & Optimization

- Train, validate and test the model
- Training: iterative process
  - Guess the output
  - Calculate the loss
  - Collect derivatives
  - Optimize using gradient descent
- Choice of optimizer depends on the task, data, resources, …

[→ Link to Notebook]()

[→ Link to documentation]()

```
# evaluate the model's performance against the test dataset
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch
normalization and dropout layers
    # Unnecessary in this situation but added for best
practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that
no gradients are computed during test mode
    # also serves to reduce unnecessary gradient
computations and memory usage for tensors with
requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%,
Avg loss: {test_loss:>8f} \n")
```

Training & Optimization

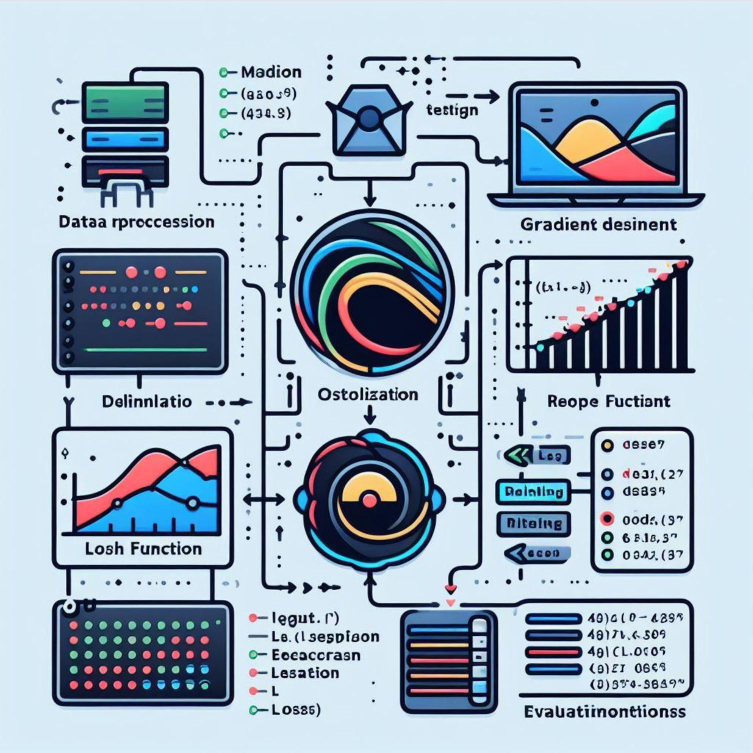
- Train, validate and test the model
- Training: iterative process
  - Guess the output
  - Calculate the loss
  - Collect derivatives
  - Optimize using gradient descent
- Choice of optimizer depends on the task, data, resources, ...

[→ Link to Notebook]()

[→ Link to documentation]()

```python
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")

# saving the model
torch.save(model, 'model.pth')
# lading it again from disk
model = torch.load('model.pth')
```

Coding Time
Linear Regression



created with https://designer.microsoft.com/image-creator

Task Description: $y = \sum a_i x_i$

Get the Data

Download the Data
Trainset, Testset

Visualize the Data $y(x_i)$

Create a custom Dataset

Instantiate DataLoaders

Build the Model

Define the neural network

Define loss function
and optimizer

Define train and test loops

Find the Results

Train the model

Visualize train loss
per epoch

Retrieve linear coefficients

Submit your results here

Data Description

10 input features x_i , 1 output feature y

in the csv:

x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, y

```
-0.475951,-0.682632,-0.443747,-0.081366,-0.357999,0.036786,-0.476114,0.952171,0.465629,-0.769452,0.040309
-0.227450,0.257002,-0.749884,0.967097,-0.113550,0.579117,0.588237,-0.277477,-0.167792,0.168516,-0.113973
0.520344,-0.624383,-0.423666,0.340438,-0.000703,-0.642863,-0.173717,-0.601610,0.063399,0.664741,0.866751
```

y is a linear combination of x_i's (+ Gaussian noise)
Find the coefficients a_i that fulfill
$y = \sum a_i x_i$

If you have spare time...

Check the effect of different noise levels in the data,
if you train on one and infer on the other

The default data set linked before has Gaussian noise with $\sigma = 0.1$

Here is data with $\sigma = 0$ (Trainset, Testset)
Here is data with $\sigma = 10$ (Trainset, Testset)

Happy Coding!

created with https://designer.microsoft.com/image-creator

Let’s see the results!

created with https://designer.microsoft.com/image-creator

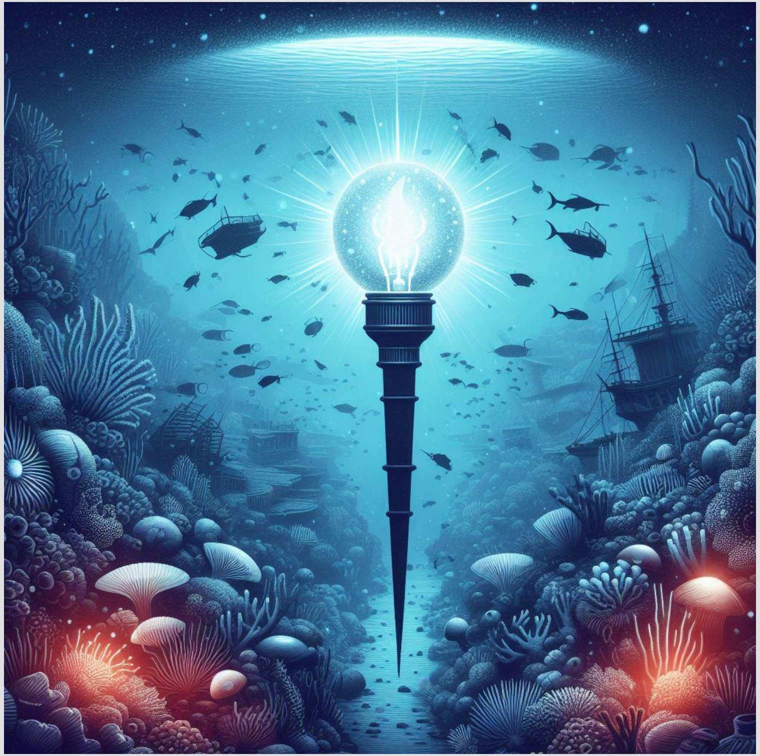
What can you expect in the coming days?

Monday, 07.04.2025	The Basics *MNIST, Linear Regression*
Today, 08.04.2025	*A deeper dive* *CNNs @ MNIST, RNNs @ names*
Wednesday, 09.04.2025	The Problem *Tracking, TrackML kNN search*
Thursday, 10.04.2025	The Solution *Graph Neural Networks, PyTorch Geometric, TrackML GNN*
Friday, 11.04.2025	The Add-On *Fun & Games*

PyTorch

A deeper dive

Optimizers, Losses, Activations, Normalization, Regularization



created with https://designer.microsoft.com/image-creator

Optimizers in PyTorch – torch.optim

- Optimizer
  - Takes the parameters and learning rate
  - Performs update through `step()` method
- Variety of algorithms, e.g
  - SGD: Stochastic Gradient Descent
  - AdaGrad: "adaptive gradient", penalizes the learning rate for parameters that are frequently updated
  - RMSprop: Divide the gradient by a running average of its recent magnitude
  - Adam: "adaptive moment estimation", aimed at large datasets and/or high-dim parameter spaces.
    Running averages with exponential forgetting of gradients and second moments of gradients
  - AdamW: Adam with decoupled weight decay, to improve regularization in Adam
  - and more...

Learning Rate

- Often useful to reduce the learning rate as training progresses
- Common schedules: Time based decay, step decay, exponential decay
- Adjusting the learning rate – torch.optim.lr_scheduler
- Several methods
  - LambdaLR: initial lr × λ (function)
  - StepLR: decays lr by γ every step_size epochs
  - ConstantLR: decays lr by a small constant factor until epochs reach total_iters
  - LinearLR: decays lr by a linearly changing small multiplicative factor until epochs reach total_iters
  - ExponentialLR: decays lr by γ every epoch
  - CosineAnnealingLR: rapidly decreasing large initial lr to a minimum, then rapidly increase again → "warm restart"
  - and many more

How to use adaptive learning rate scheduling?

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

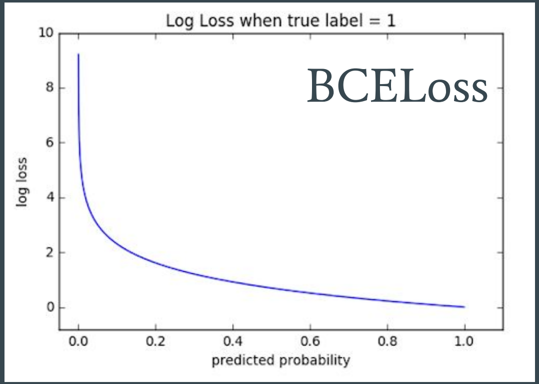
Loss Functions in PyTorch – torch.nn

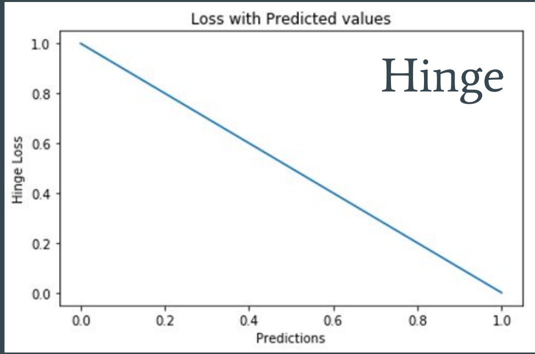
- Evalautes how well ML algorithm models featured data set
- Optimizer: minimize to improve model performance
- Several functions available for classification
    - BCELoss: Binary Cross-Entropy Loss, most commonly used
    - HingeEmbeddingLoss: Hinge Loss, primarily developed for support vector machine, penalizes wrong and right, not confident, predictions
- And for regression
    - MSELoss: Mean Square Error
    - L1Loss: Mean Absolute Error (MAE)
    - HuberLoss: Combination of MSE and MAE

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$



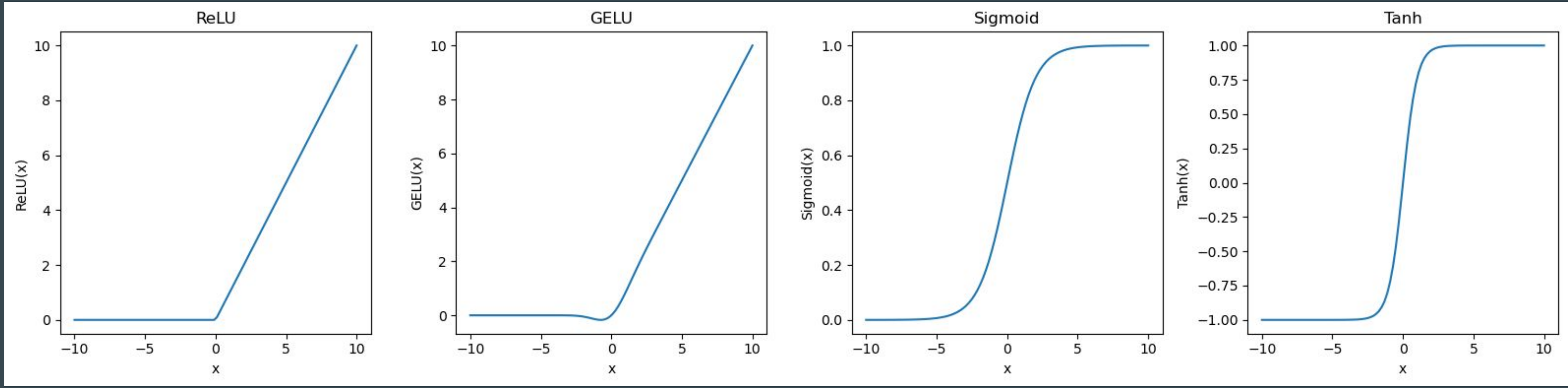


https://builtin.com/machine-learning/common-loss-functions

Activation Functions in PyTorch – torch.nn

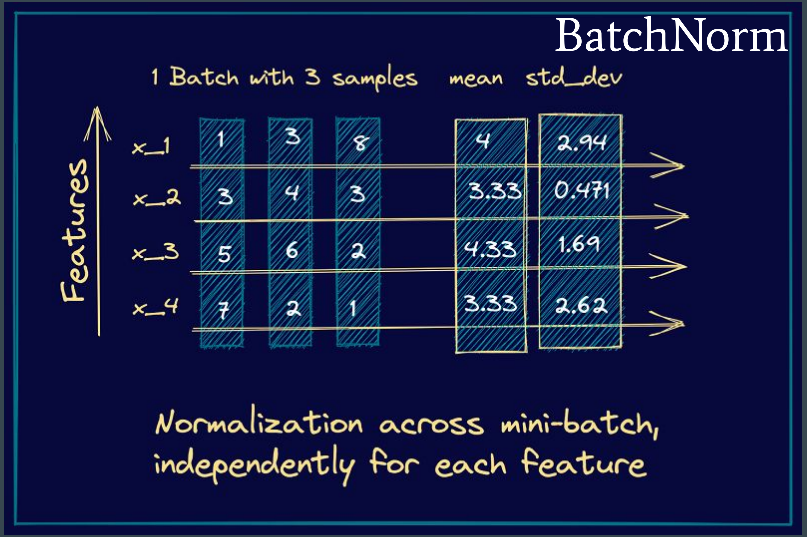
- Adds non-linearity, helps the network to learn complex patterns in the data
- Vanishing gradients can be problem (Sigmoid, Tanh)
- Lots of functions available
  - ReLU, GELU, Sigmoid, Tanh, ...
  - Softmax: rescales tensor to lie in [0,1], and sum = 1

$$\mathrm{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$



Normalization Layers in PyTorch – torch.nn

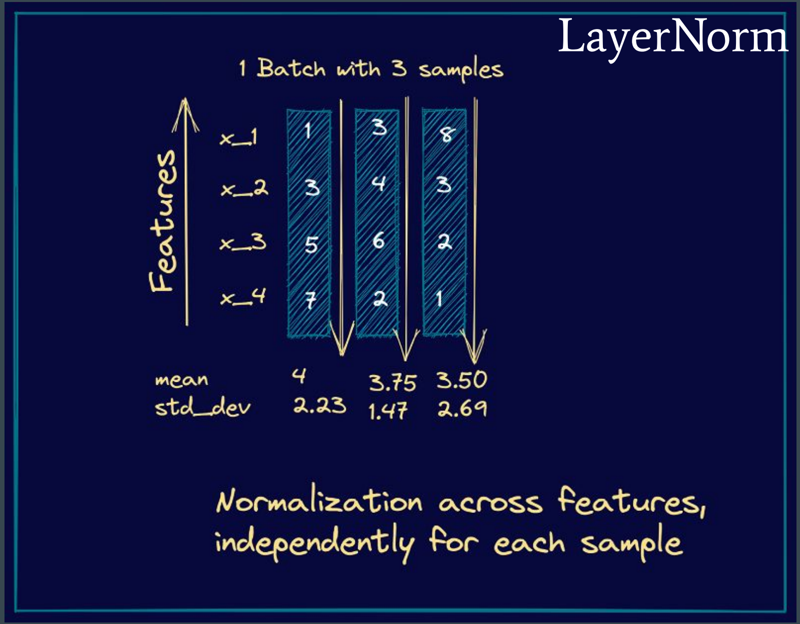
- Feature scaling – transform the range of features to a standard scale
- Improves performance and training stability
- Several methods:
    - BatchNorm*X*d: normalization wrt batch statistics
    - LayerNorm: normalization across all features
      better for RNNs, transformers
    - InstanceNorm*X*d: normalization across batch
      and channel; helps generative models
    - and more



https://www.pinecone.io/learn/batch-layer-normalization/

Normalization Layers in PyTorch – torch.nn

- Feature scaling – transform the range of features to a standard scale
- Improves performance and training stability
- Several methods:
  - BatchNorm*X*d: normalization wrt batch statistics
  - LayerNorm: normalization across all features better for RNNs, transformers
  - InstanceNorm*X*d: normalization across batch and channel; helps generative models
  - and more



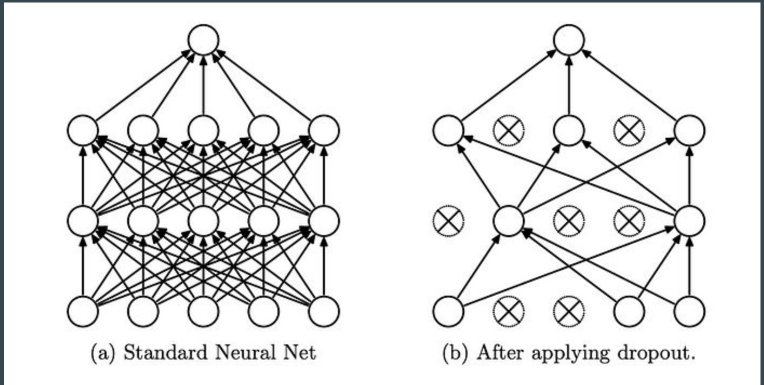
https://www.pinecone.io/learn/batch-layer-normalization/

Regularization in PyTorch

- Regularization is used to prevent models from overfitting
- Dropout Layers: During training, randomly zeroes some of the elements of the input tensor with probability *p*.
- More general techniques:
- L1/L2 Regularization: penalty for large weights

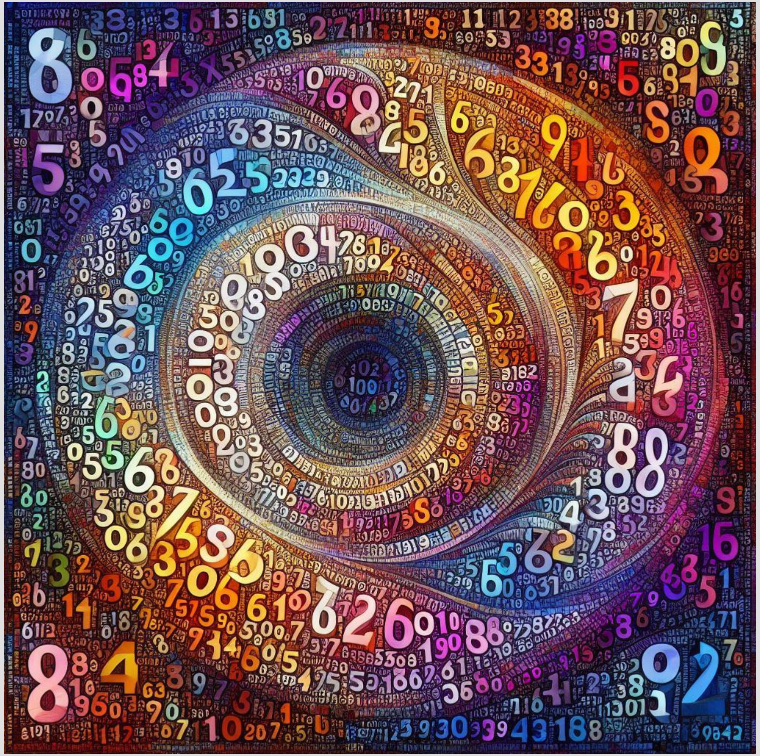
$$L_{training} = L_{loss} + L_{1/2}, \quad L_{1/2} = \lambda \sum |w_i|^{1/2},$$

- Data Augmentation: Transformations, noise injections



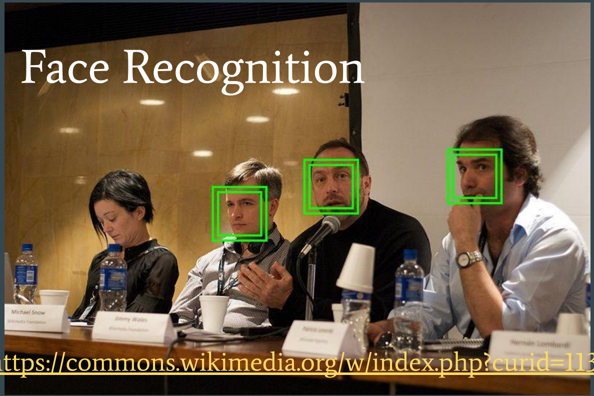
https://medium.com/analytics-vidhya/understanding-regularization-with-pytorch-26a838d94058

MNIST Experiment
with CNNs



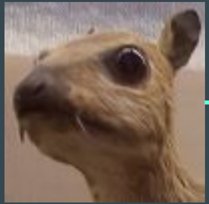
created with https://designer.microsoft.com/image-creator

Convolutional Neural Network



https://commons.wikimedia.org/w/index.php?curid=11309460

- Feed-Forward Neural Network
- Applications in Computer Vision
  - Image and video recognition
  - Image & document analysis
  - Image classification
- Learns feature engineering via filter (= kernel) optimization



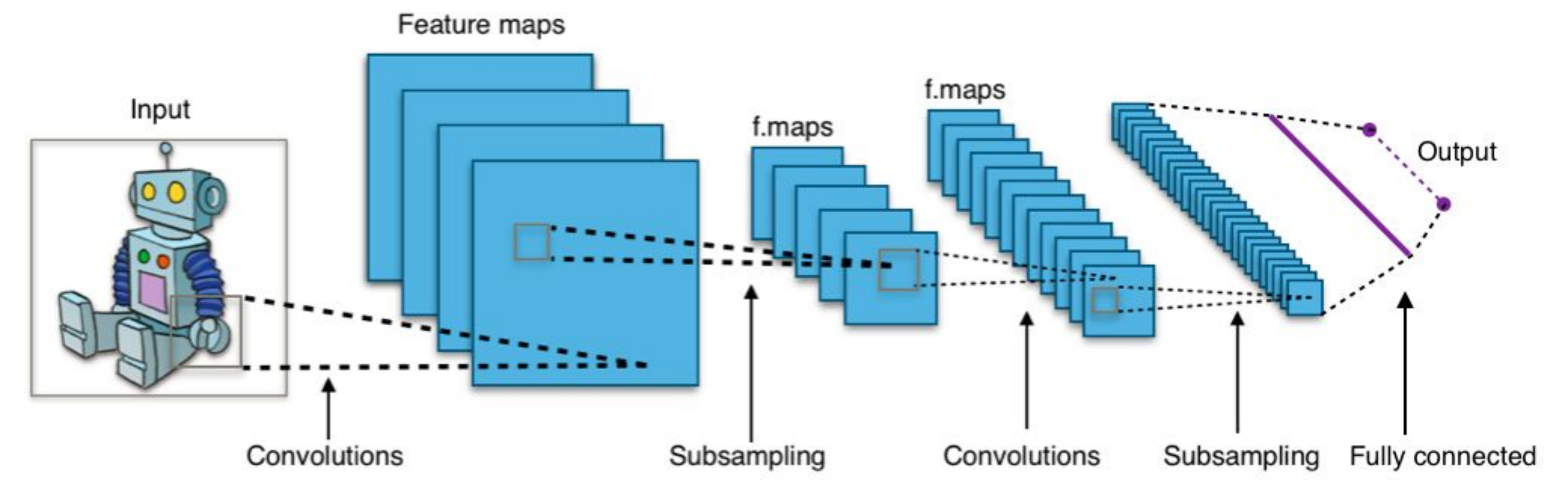
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Classical image processing: sharpen

https://en.wikipedia.org/wiki/Kernel_(image_processing)

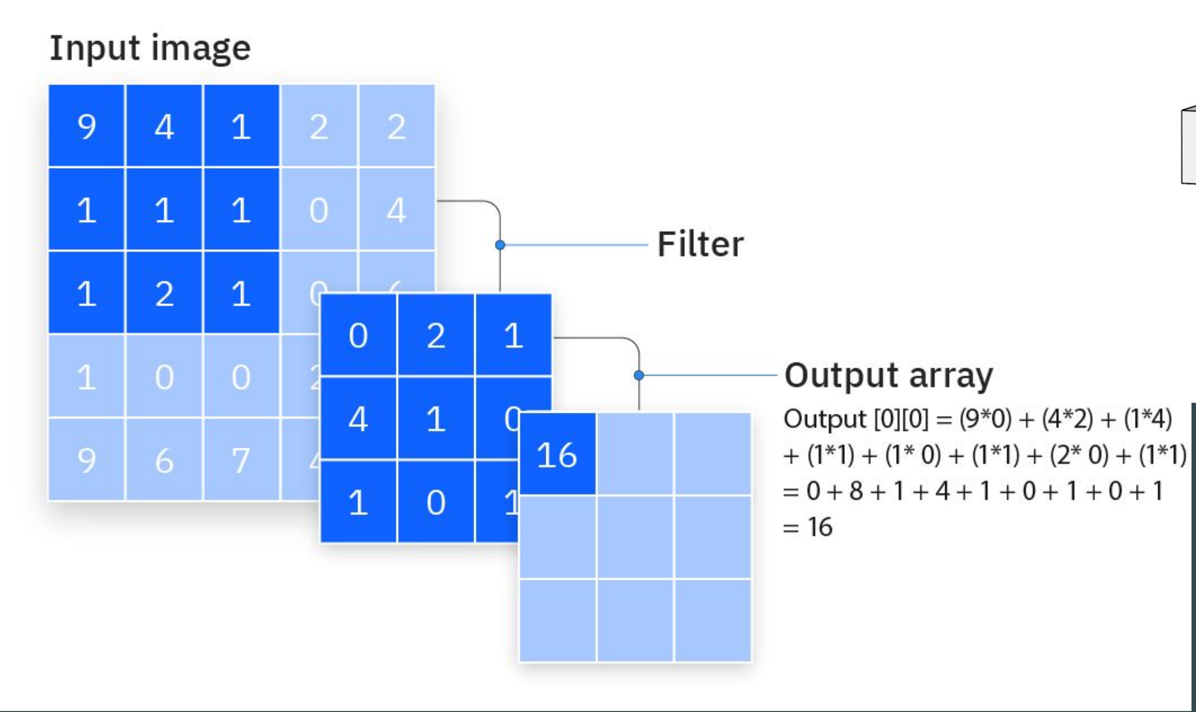
https://commons.wikimedia.org/w/index.php?curid=99461951

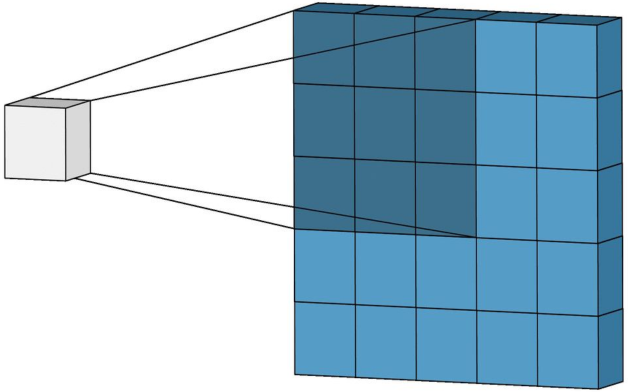
Convolutional Neural Network



https://en.wikipedia.org/wiki/Convolutional_neural_network

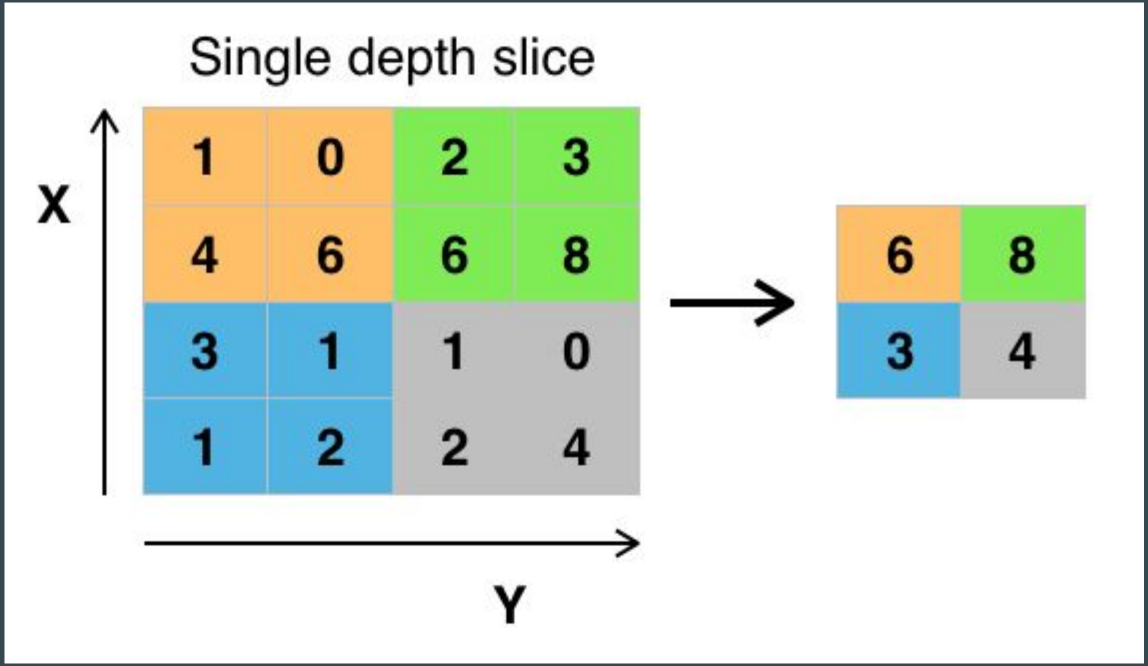
Convolutional Layer





https://www.ibm.com/topics/convolutional-neural-networks

Subsampling through Max Pooling



https://en.wikipedia.org/wiki/Convolutional_neural_network

Hyperparameters for convolutions

- Kernel size: Number of pixels processed together, expressed as kernel's dimensions, e.g., 2x2, or 3x3.
- Padding: Addition of 0-valued pixels on the borders of an image, so that the border pixels are not undervalued from the output.
- Stride: Number of pixels that the analysis window moves on each iteration.
- Dilation: Ignoring pixels, increases kernels
- Number of filters: Since feature map size decreases with depth, layers near the input layer tend to have fewer filters while higher layers can have more.
- Filter size: Chosen based on data set
- Pooling type and size: Typically used max pooling with 2x2 dimension

CNNs with PyTorch – torch.nn

X = 1, 2 or 3

- Conv*X*d: 1-3D convolutions over an input signal composed of several input planes
- ConvTranspose*X*d: 1-3D transposed convolutions; can be seen as gradient of Conv*X*d with respect to its input
- LazyConv(Transpose)*X*d: derive shape of parameters from their first input to the forward method
- Unfold: Extracts sliding local blocks from a batched input tensor.
- Fold: Combines an array of sliding local blocks into a large containing tensor.

Convolution is equivalent to Unfold + MatMul + Fold

Experimenting with MNIST

- Adapt the previous notebook by replacing the model with another neural network architecture of your choice
- Example: stacks of 2D convolutional layers (Conv2d) + ReLU + MaxPool2d

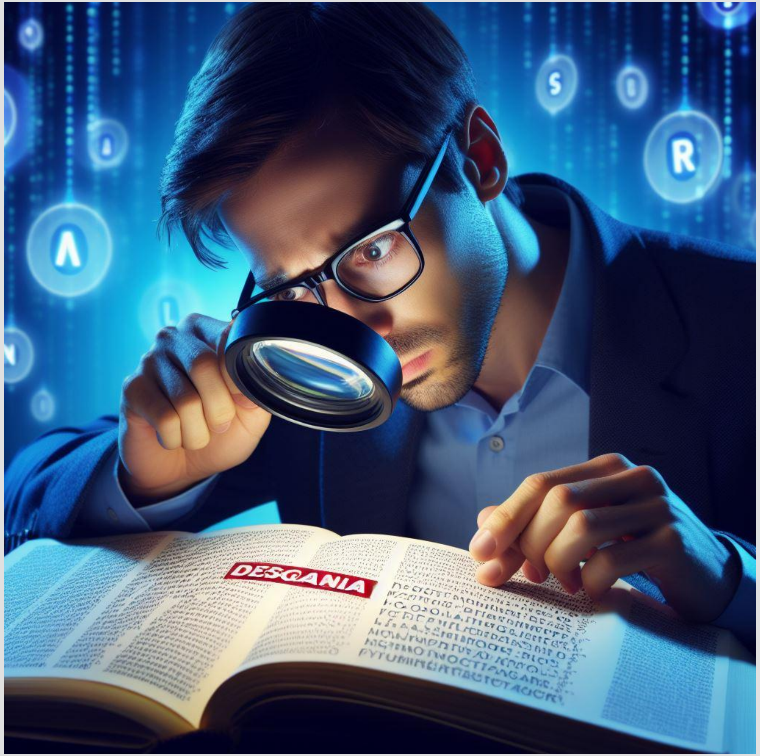
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

- Feel free to experiment with layers, optimizers, losses, activations, normalizations, regularizations!

Let’s see the results!

created with https://designer.microsoft.com/image-creator

Classifying names
with RNNs



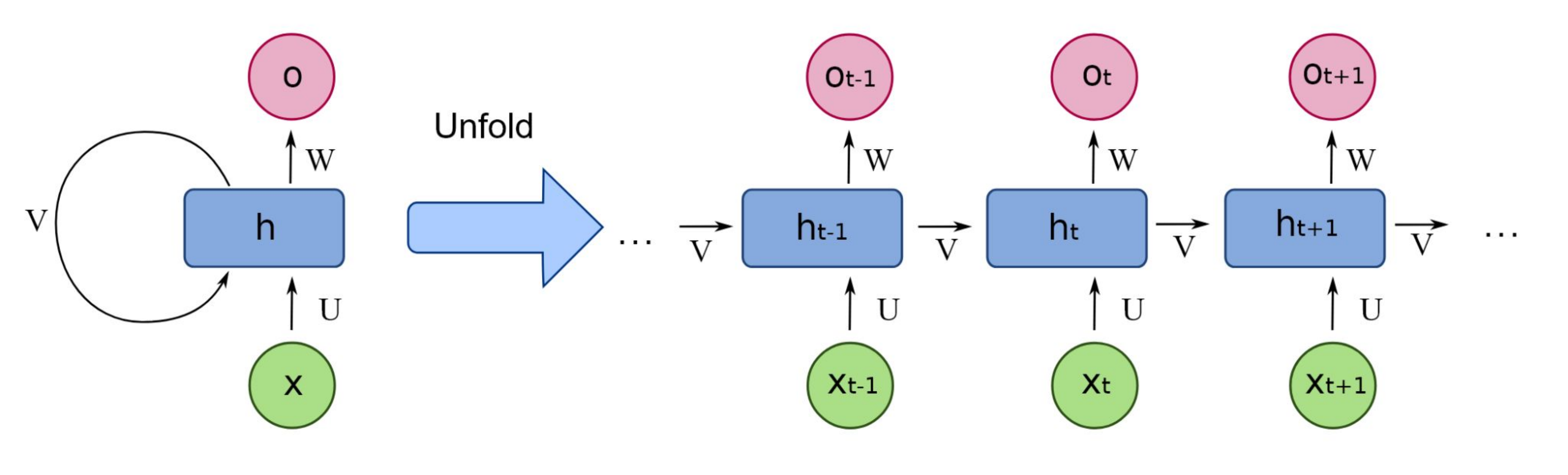
created with https://designer.microsoft.com/image-creator

Recurrent Neural Network

- Bi-directional neural network: allows output from some nodes to affect subsequent input to the same nodes (temporal, sequential flow)
- Use internal state (= memory) to process arbitrary sequences of inputs
- Applications in
  - Handwriting recognition
  - Speech recognition
  - Natural language processing
- Various architectures:
  - Fully recurrent: outputs of all neurons to inputs of all neurons
  - Long short-term memory (LSTM): avoids vanishing gradient problem, augmented by “forget gates”

https://www.youtube.com/watch?v=cUTMhmVh1qs&t=1780s

Recurrent Neural Network

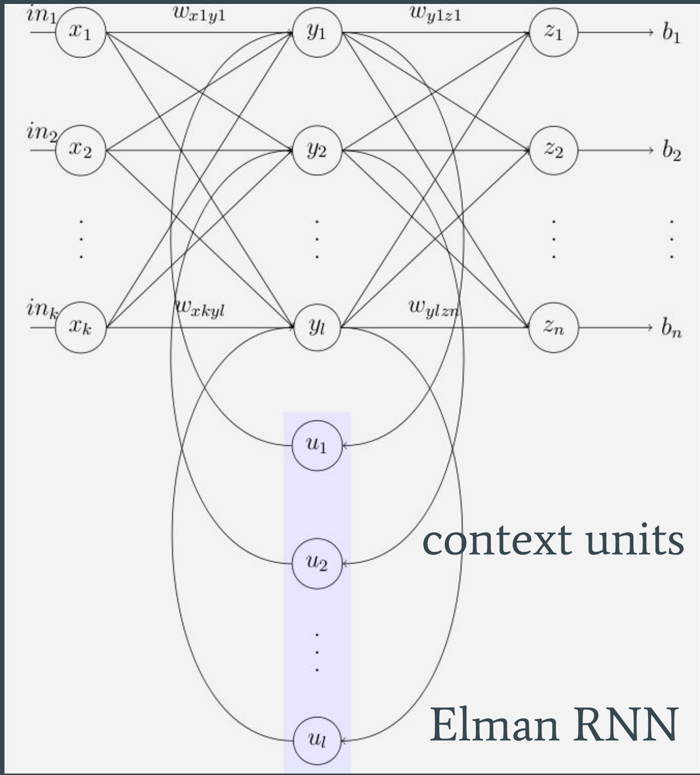


time

https://commons.wikimedia.org/w/index.php?curid=60109157

RNNs with PyTorch – torch.nn

- RNNBase: aspects shared by RNN, LSTM, GRU; no forward
- RNN: multi-layer Elman RNN with *tanh* or *ReLU*
- LSTM: Long Short-Term Memory, 3 gates
  (input, forget, output)
- GRU: Gated Recurrent Unit, simplified compared
  to LSTM, 2 gates (update + reset)
  less prone to overfitting, on smaller datasets
- and their individual cells



https://commons.wikimedia.org/w/index.php?curid=583704

Experimenting with RNNs

- Task: Classifying names with a character-level RNN
- Checkout the notebook, it includes a simple hand-made RNN model
- Data contains a few thousand surnames from 18 languages of origin
- Can the model predict your last name correctly?
- Experiment with model parameters, different RNN models, optimizers, ...
  Can you improve the performance?

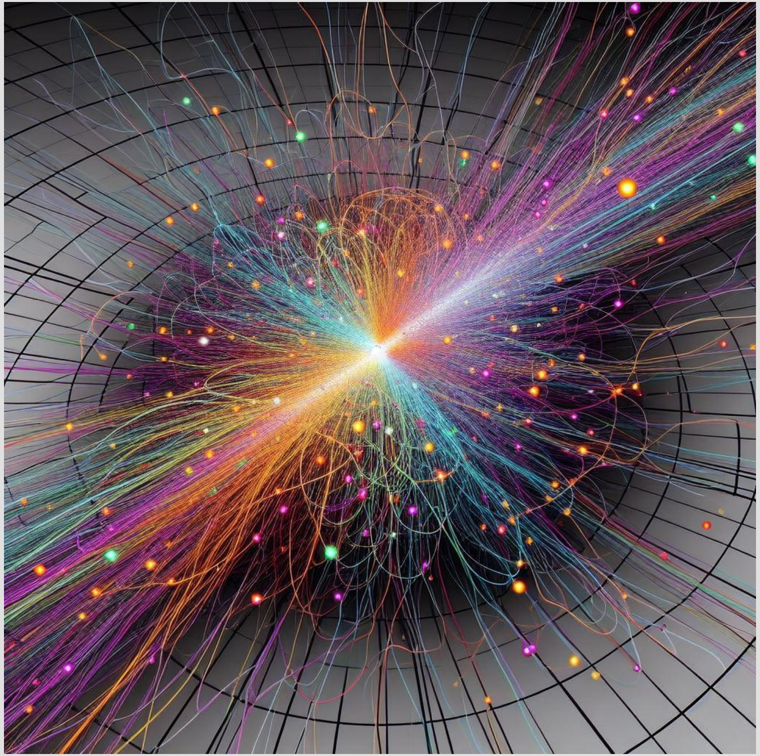
Let’s see the results!

created with https://designer.microsoft.com/image-creator

What can you expect in the coming days?

Monday, 07.04.2025	The Basics *MNIST, Linear Regression*
Tuesday, 08.04.2025	*A deeper dive* *CNNs @ MNIST, RNNs @ names*
Today, 09.04.2025	The Problem *Tracking, TrackML kNN search*
Thursday, 10.04.2025	The Solution *Graph Neural Networks, PyTorch Geometric, TrackML GNN*
Friday, 11.04.2025	The Add-On *Fun & Games*

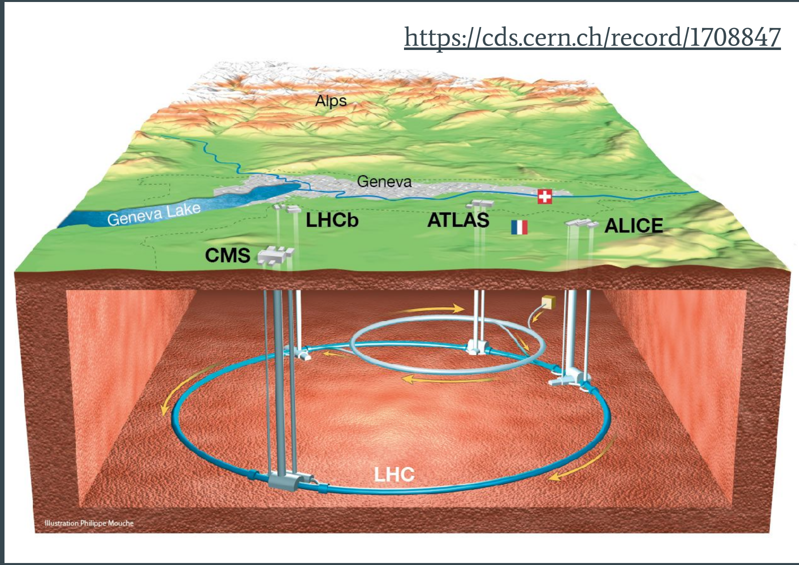
The Problem
Tracking @ HL-LHC
TrackML Challenge



created with https://designer.microsoft.com/image-creator

The Large Hadron Collider

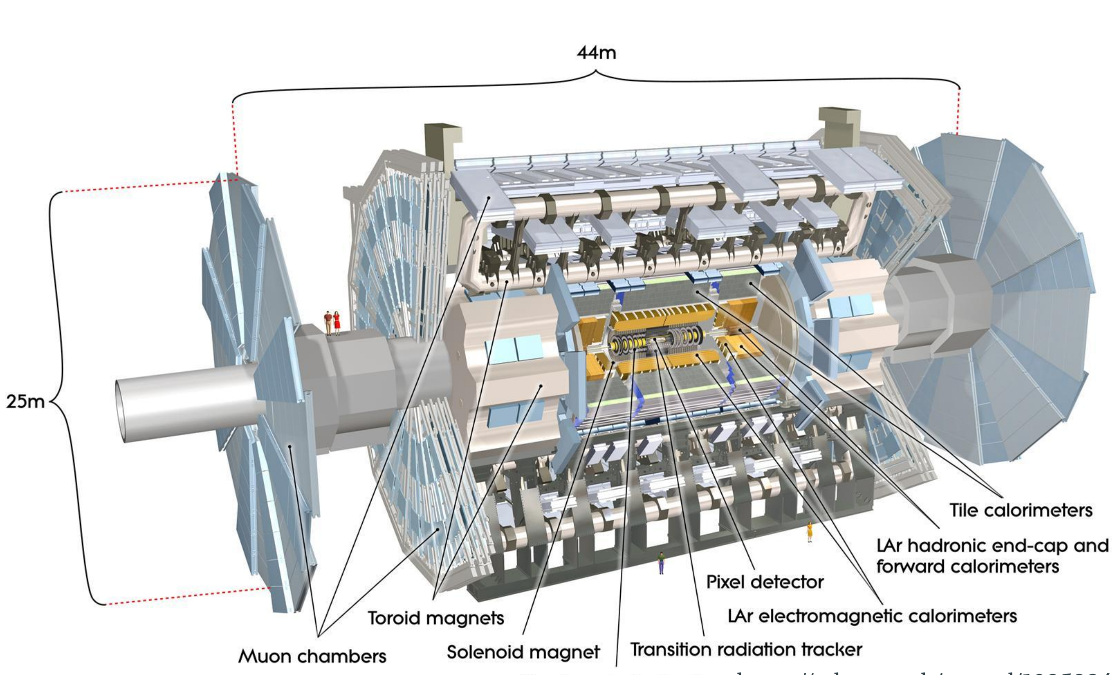
The most powerful particle accelerator ever built!



Quantity	Number (Run 2)
Circumference	26 659 m
Dipole operating temperature	1.9 K (-271.3°C)
Number of magnets	9593
Number of main dipoles	1232
Number of main quadrupoles	392
Number of RF cavities	8 per beam
Nominal energy, protons	6.5 TeV
Nominal energy, ions	2.56 TeV/u (energy per nucleon)
Nominal energy, protons collisions	13 TeV
No. of bunches per proton beam	2808
No. of protons per bunch (at start)	1.2×10^{11}
Number of turns per second	11245
Number of collisions per second	1 billion

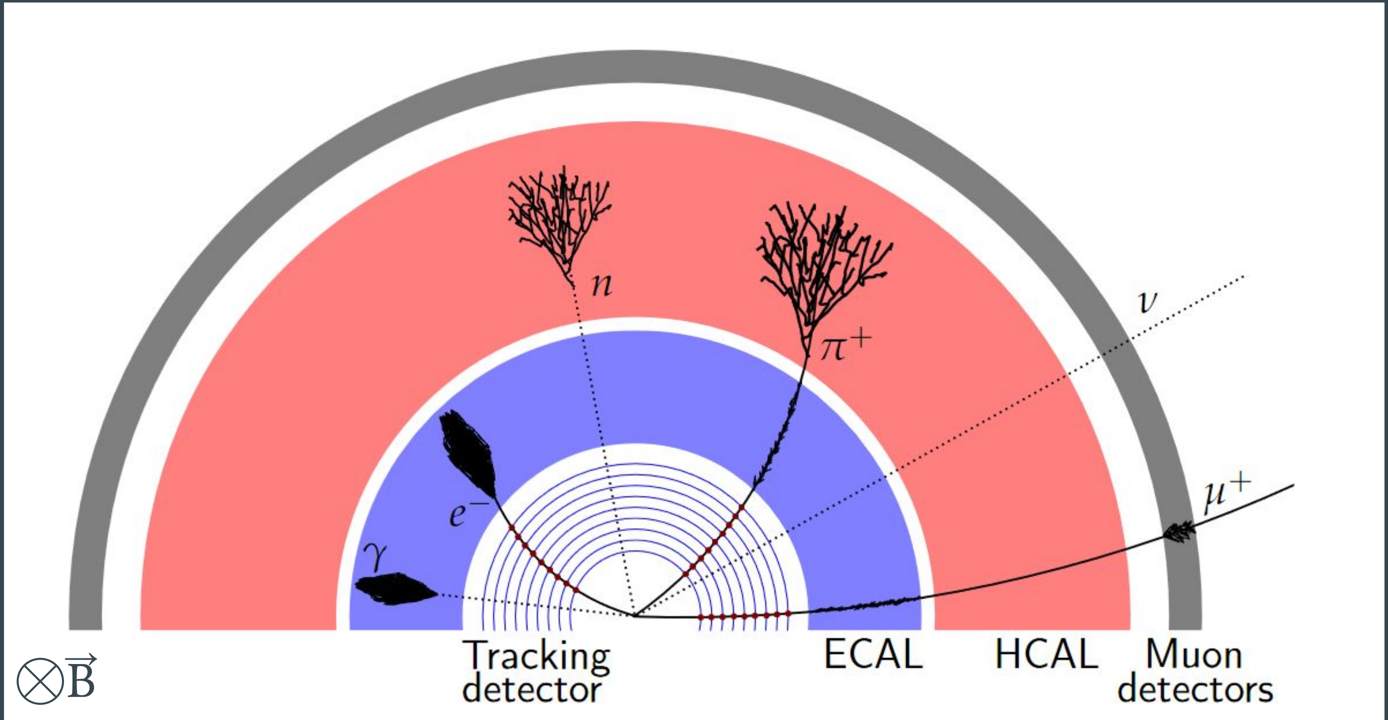
The ATLAS Experiment

- ATLAS is one of two general-purpose detectors at the LHC
- Wide range of physics
    - Higgs boson properties
    - Standard Model parameters
    - Physics beyond the Standard Model
- Beams of particles collide at the centre of the ATLAS detector
- Six subdetectors and huge magnets: measure the paths, momentum, and energy of the particles



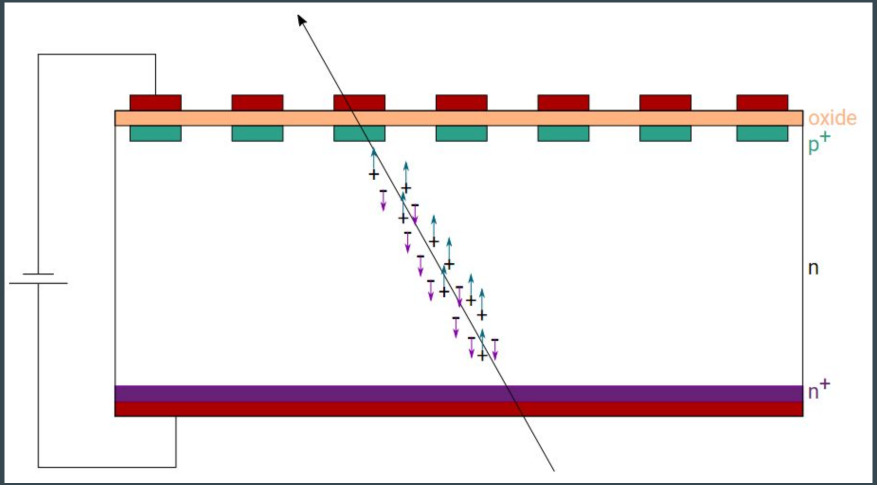
https://cds.cern.ch/record/1095924

General Purpose Collider Detector Concept

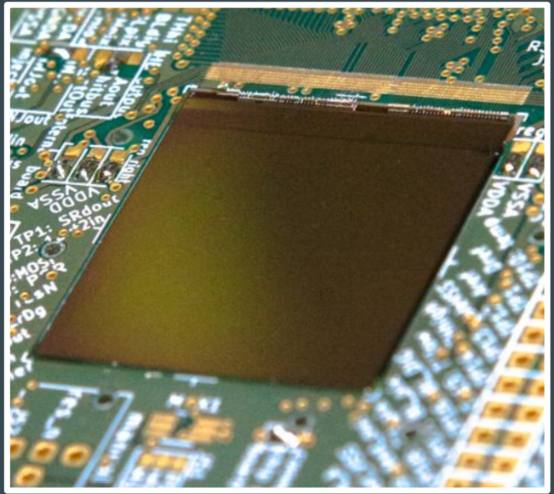


Silicon Tracking Detectors

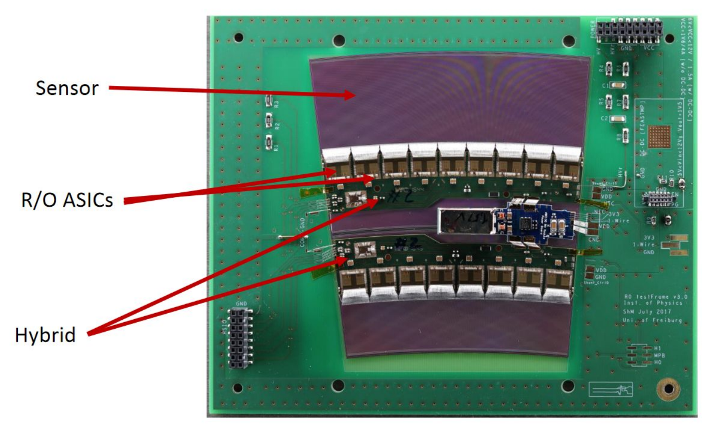
Principle of a semiconductor detector



A silicon pixel sensor (MuPix10, Mu3e)



A silicon strip module (ATLAS ITk)



Track Reconstruction

"Track reconstruction is the task of finding and estimating the trajectory of a charged particle, usually embedded in a static magnetic field to determine its momentum and charge."

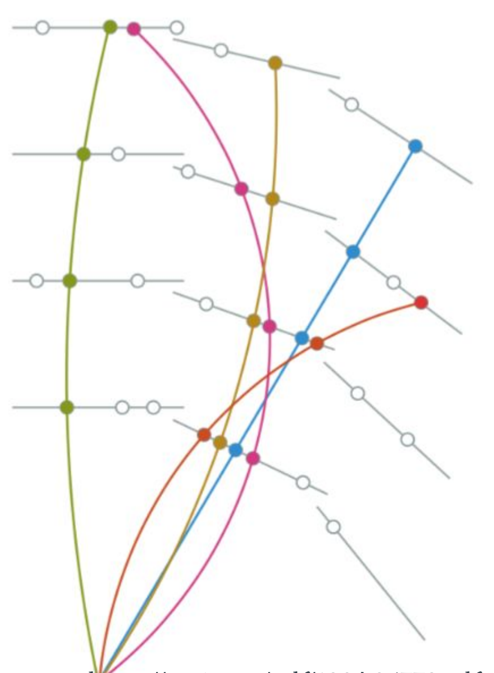
Frühwirth, Brondolin, Strandlie

Involves pattern recognition and statistical estimation methods

- **Pattern recognition / Track finding**
- Track parameter estimation / Track fitting
- Track hypothesis test

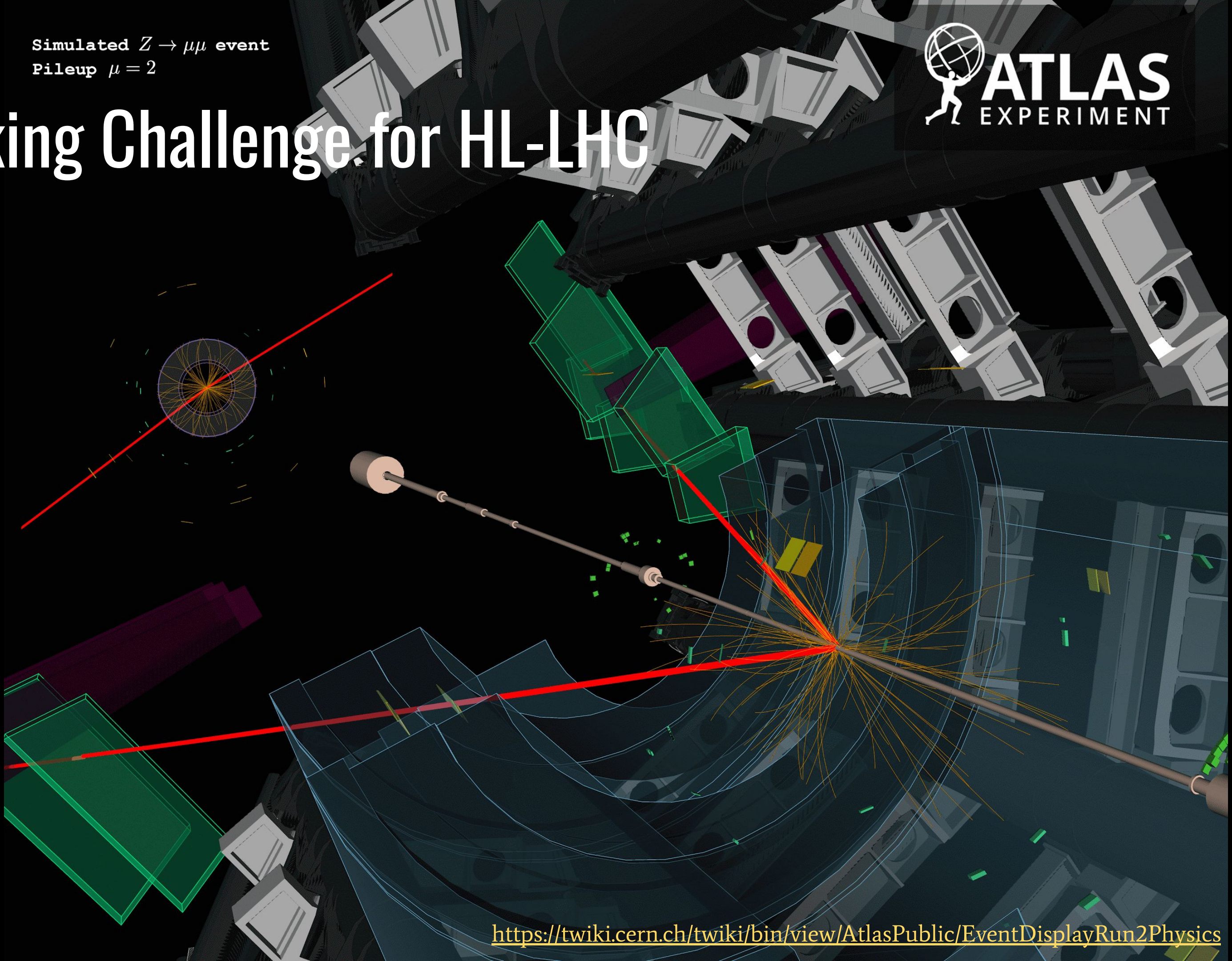
Track Finding

- Task: associate points into tracks
- Currently conveniently solved by combinatorial optimization methods (based on Kalman filters)
- But: CPU time increases (worse than linearly) with number of simultaneous proton collisions
- This is where Machine Learning may help us!



https://arxiv.org/pdf/1904.06778.pdf

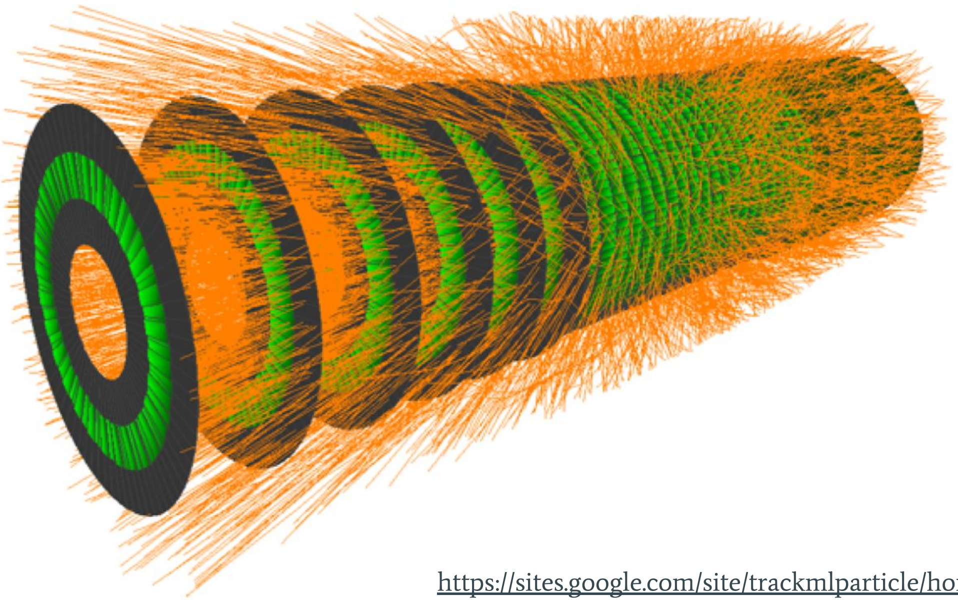
The Tracking Challenge for HL-LHC



https://twiki.cern.ch/twiki/bin/view/AtlasPublic/EventDisplayRun2Physics

The TrackML Challenge

- Machine Learning Challenge in 2018, using the power of the "crowd"
- 100'000 points from 10'000 particles from very high energy proton collisions



https://sites.google.com/site/trackmlparticle/home

Setup (I)

- An event is a set of particle measurements (hits) in the detector
- The detector is formed of discrete layers
- An event has ~ 100’000 hits, corresponding to 10’000 particles.
  - Each particle is created close to, but not exactly, at the center of the detector.
  - Each hit is a 3D measurement in Cartesian coordinates (*x*– , *y*,– *z*).
  - For each particle, the number of hits is on average 12, but as low as 4 and as large as 20.
  - Target: associate the hits created by each particle together, to form tracks.
    At least 90% of the true tracks should be recovered.
  - The tracks are slightly distorted arc of helices with axes parallel to the *z*-axis, and pointing approximately to the interaction center.

https://arxiv.org/pdf/1904.06778.pdf

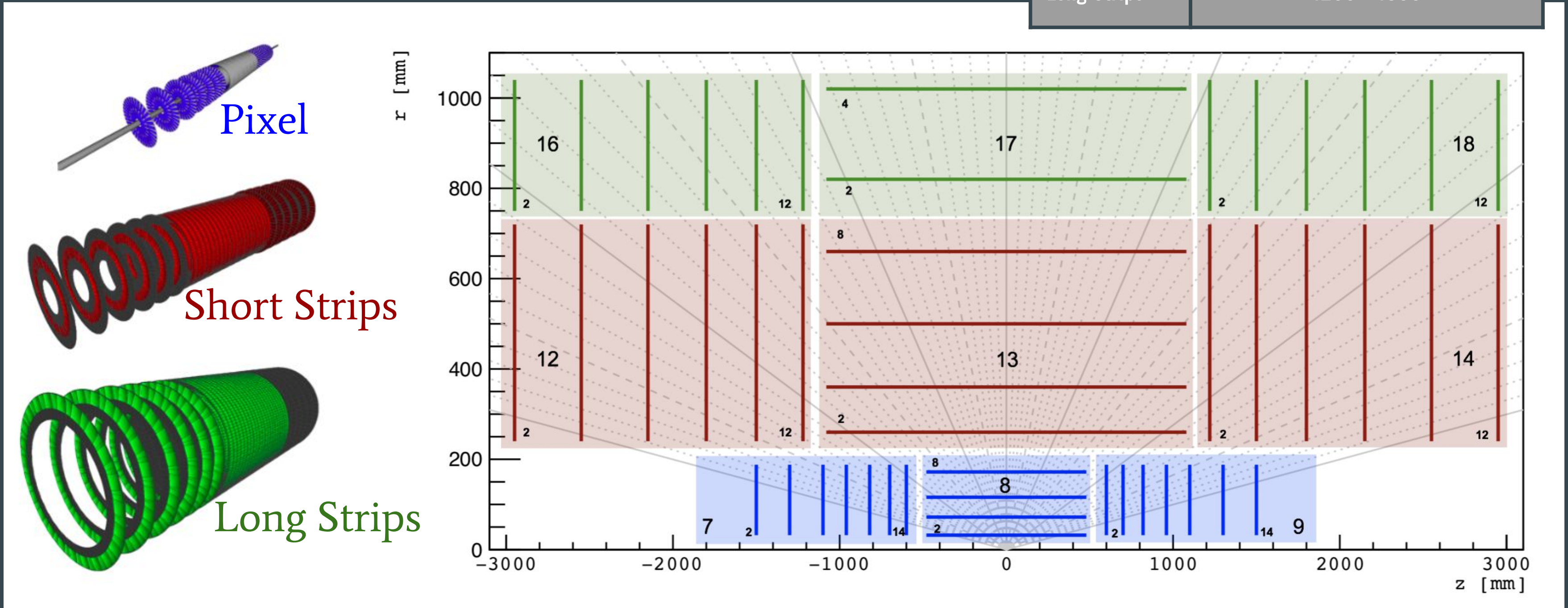
Setup (II)

- In an ideal world:
  - Each particle would leave one and only one hit on each layer of the detector
  - The trajectories would be exact arcs of helices
  - The (x, y, z) coordinates would be exact.
  - In this ideal world, fitting the parameters of the helices suffices to solve the problem.
- Subtleties:
  - Depending of the local geometry, each particle may leave multiple hits in a layer, and the layer may not record anything at all.
  - The arcs are often slightly distorted.
  - The measurements have some non isotropic uncertainty

https://arxiv.org/pdf/1904.06778.pdf

TrackML Detector

Detector	Spatial resolution ($\mu m \times \mu m$)
Pixel	50 × 50
Short Strips	80 × 1200
Long Strips	1200 × 1800



$\vec{B}$

Dataset (I)

- Data is stored per event. Events are statistically independent
- Hits
  - **hit_id** : Unique hit identifier
  - **x, y, z** : Cartesian coordinates in millimetres
  - **volume_id** : numerical identifier of the detector group.
  - **layer_id** : numerical identifier of the detector layer inside the group.
  - **module_id** : numerical identifier of the detector module inside the layer.
- Hit truth
  - **hit_id** : Unique hit identifier
  - **particle_id** : Particle identifier (0 = non-reconstructible)
  - **tx, ty, tz** : Truth hit positions
  - **tpx, tpy, tpz** : Truth particle momentum at hit (in GeV/c)
  - weight: don’t care for us

Dataset (II)

- Data is stored per event. Events are statistically independent
- Particles truth
  - **particle_id** : Particle identifier
  - **vx, vy, vz** : Truth initial position (vertex) in millimetres
  - **px, py, pz** : Truth initial particle momentum (in GeV/c)
  - **q:** Particle charge (in units of *e*)
  - **nhits:** Number of hits
- Cells: additional information per hit (individual pixels or strips)
  - **hit_id:** Hit identifier
  - **ch0, ch1:** coordinates within detector module
  - **value:** deposited charge within cell
- Detector geometry information

Coding Time

TrackML kNN search

created with https://designer.microsoft.com/image-creator

Task Description: Getting Started with TrackML (I)

Get the Data

Download the Data
(100 events, split 80, 10, 10 in trainset, valset, testset)

You can load events / dataset using the trackml-library

Visualize the Data of an event

Create a Dataset

Include particle p_T with `add_momentum_quantities`

Allow for a lower bound cut on particle p_T

Instantiate Datasets with cut $p_T > 2$ (GeV)

DataLoaders:
batch size: 1 event

Build a kNN search

Goal: hits belonging to a track should be near, others far away

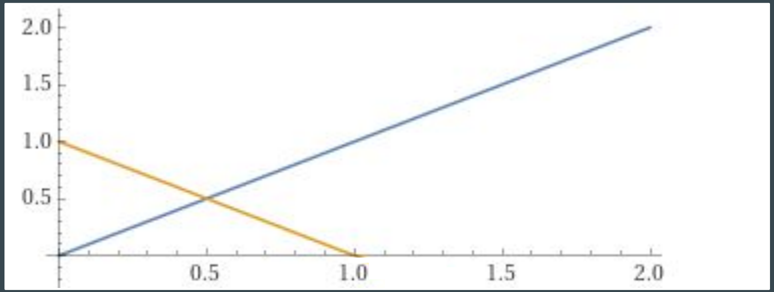
We can achieve this using an HingeEmbeddingLoss

Add label tensor to dataset
y [Nhits, Nhits]

Hinge loss function

"maximum-margin" classification

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, margin - x_n\}, & \text{if } y_n = -1, \end{cases}$$

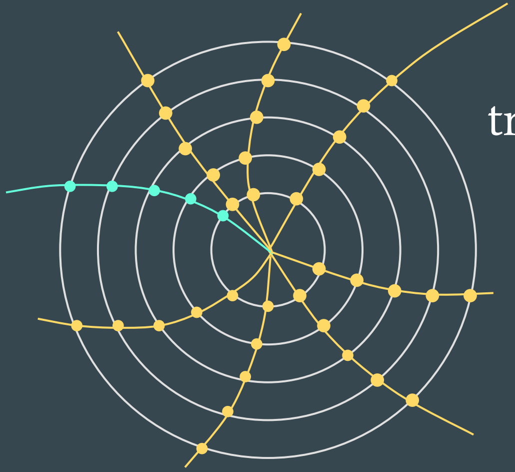


y_n = 1 for hits from same particle,
y_n = -1 for all other hits

kNN search with TrackML

Goal: embed all hits belonging to a track such that they form a cluster in latent space

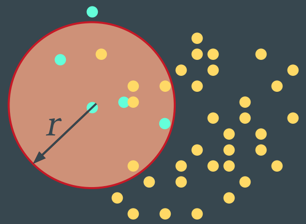
real space



train with hinge loss

Neural Network

latent space



search for k Nearest Neighbors within radius r

Task Description: Getting Started with TrackML (II)

Build a Model

Experiment here!

Choose a model architecture to embed hits into a latent space of arbitrary dimension

Choose which features you will use as input to your model (no truth!)

Train loop

Calculate pairwise distances between all embedded hits (=prediction)

→ input for loss function, together with labels

Set up Optimizer

Test loop evaluation

Add a kNN search NearestNeighbors to evaluate efficiency and purity; remove neighbors outside of radius

efficiency:
true hits in circle / all true hits
purity:
true hits in circle / all hits in circle

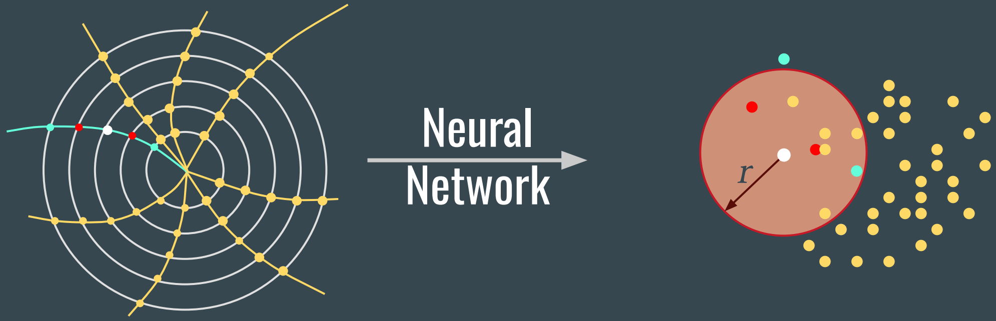
Starting Notebooks

- We have plenty of time now for coding!
- Notebooks prepared for usage on Google Colab
- Minimal notebook → Link to Notebook
    - Installs external dependencies
    - Downloads and unpacks the data
    - Freedom to implement the way you want → Enjoy!
- If you want a little more help from the start (or to get some inspiration) → Link to Notebook
    - You can spend more time in trying to find good model architectures
- Train and evaluate → and visualize your results!

If you have spare time...

You can relax the p_T cut (1 GeV or remove it completely)
→ this may require that you create the labels tensor in the training loop (slow), or reduce number of events, due to memory constraints

Or you can change the definition of the labels:
Only hits from same particle in neighboring layers are to be put close together
→ This is what we do in Metric Learning (Graph Construction for GNN Tracking)



Let’s get started!

created with https://designer.microsoft.com/image-creator

Let's see the results!

created with https://designer.microsoft.com/image-creator

Shortcomings of our kNN search

- We cluster around every hit (multiple times per track)
  - Which hit should be the center of the cluster? → Object condensation!
- We try to cluster full tracks in the latent space
  - We end up with a lot of wrong hits within the clusters, if we want to be efficient
  - Reduce task complexity by clustering only consecutive layers → metric learning
- Labels tensor is large (scales N^2)
  - Make use of sparsity
  - Hard negative mining (wrong hits outside of margin don’t contribute)
  - Custom hinge loss → label wrong combinations with 0 instead of -1

We are actually creating here a set of hits connected with edges → a graph!
We can use similar techniques to construct a graph and apply a graph neural network for edge labeling → then we can later cut the graph

What can you expect in the coming days?

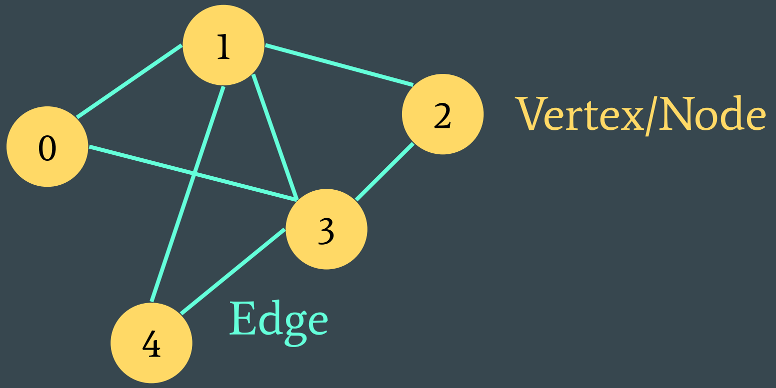
Monday, 07.04.2025	The Basics *MNIST, Linear Regression*
Tuesday, 08.04.2025	*A deeper dive* *CNNs @ MNIST, RNNs @ names*
Wednesday, 09.04.2025	The Problem *Tracking, TrackML kNN search*
Today, 10.04.2025	The Solution *Graph Neural Networks, PyTorch Geometric, TrackML GNN*
Friday, 11.04.2025	The Add-On *Fun & Games*

ML with Graphs
Graph Neural Networks

created with https://designer.microsoft.com/image-creator

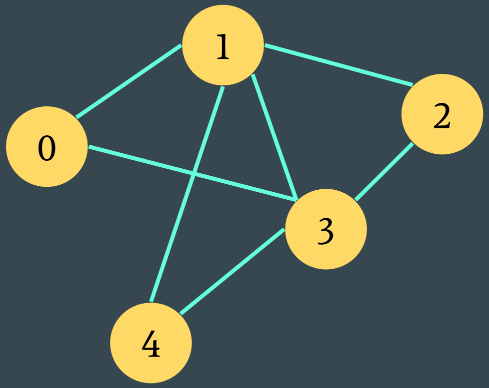
What is a Graph?

A network that helps define and visualize relationships between various components.



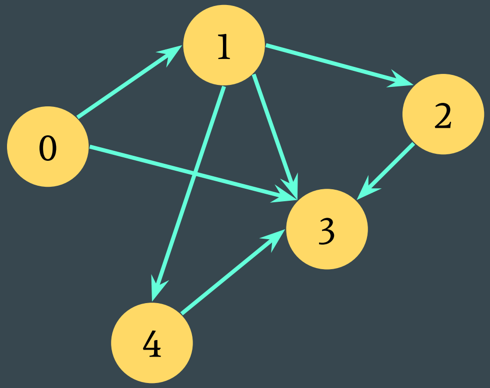
A graph $G = (V, E)$ is a set of Vertices V and edges E, where each edge (u,v) is a connection between vertices, $u, v \in V$

Types of Graphs



Undirected Graph

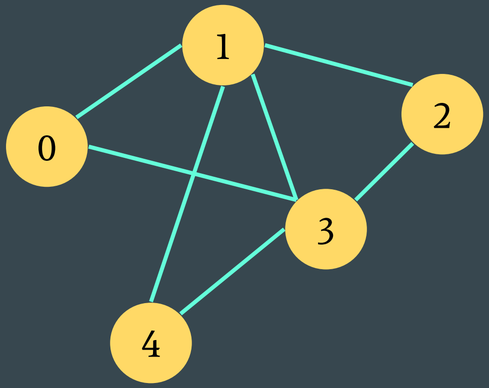
Edge (u,v) implies (v,u)

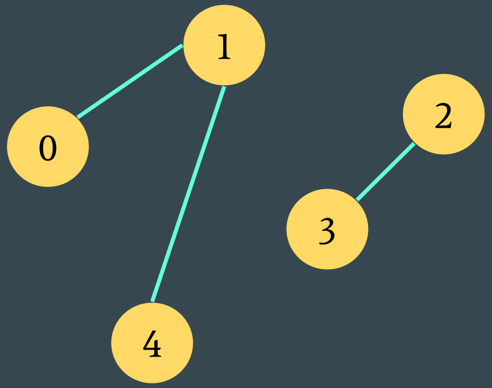


Directed Graph

Edges are unidirectional

Connectivity





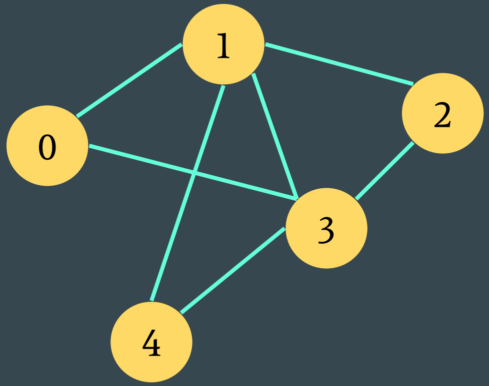
Connected graph

All vertices are connected

Edges are unidirectional

Connected components (subsets of vertices)

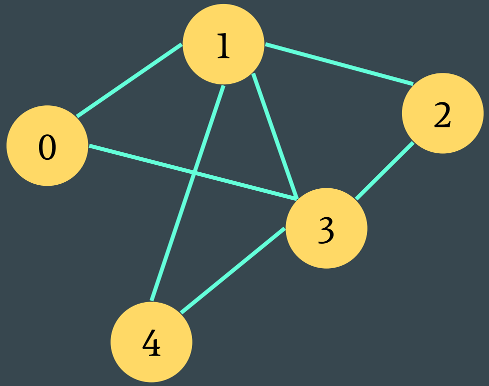
Graph Representations



Adjacency Matrix

	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	1	1
2	0	1	0	1	0
3	1	1	1	0	1
4	0	1	0	1	0

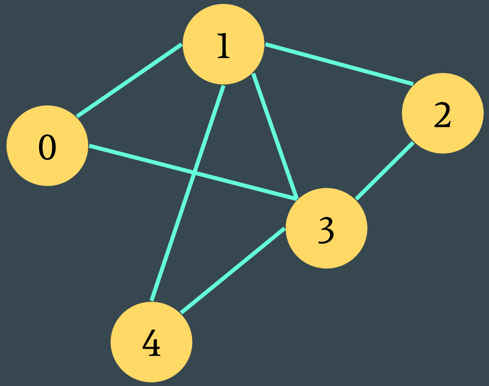
Graph Representations



Edge Set

{ (0,1), (0,3),
(1,2), (1,3), (1,4)
(2,3),
(3,4) }

Graph Representations

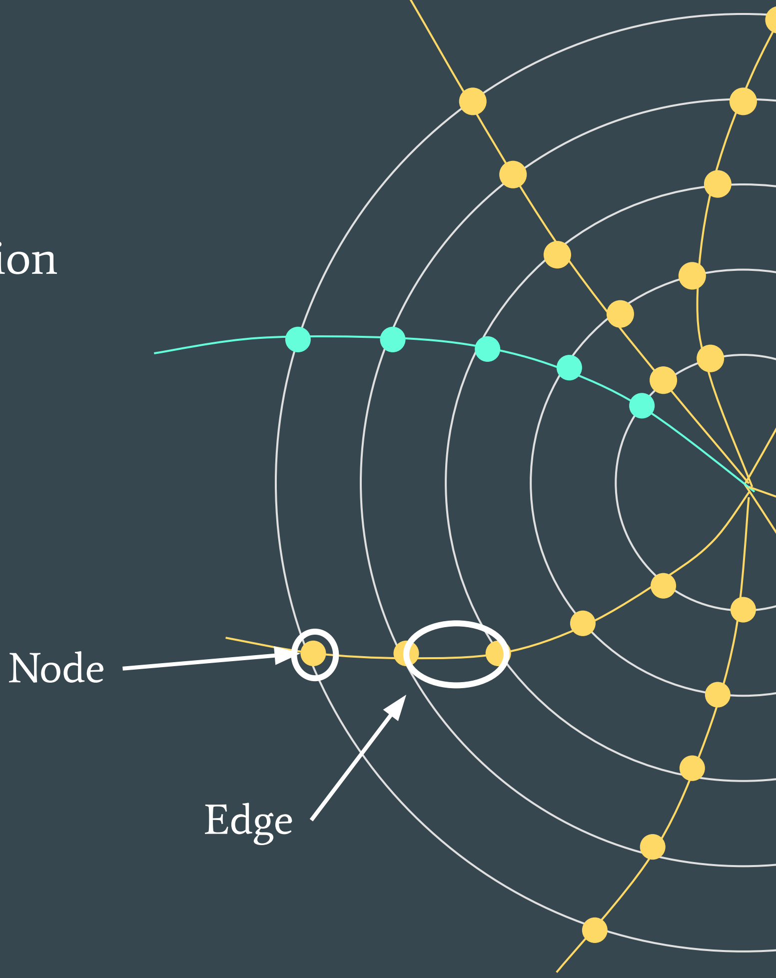


Adjacency List

0 →	1	3		
1 →	0	2	3	4
2 →	1	3		
3 →	0	1	2	4
4 →	1	3		

How does this apply to tracking?

- A **graph** is a **natural representation** for a collision event in a tracking detector
- Graphs consist of a set of **nodes** and **edges**
  - Represent each **hit** as a **node**
  - **Edges** suggest **two hits** belong to the **same track**
- Levels of information:
  - Node: position, energy deposited, ...
  - Edge: geometric info, belongs to track, ...
  - Graph: event, detector region, ...
- Predictions possible with a GNN on each level
  - **Track reconstruction uses edge-level predictions**



Graph Neural Networks

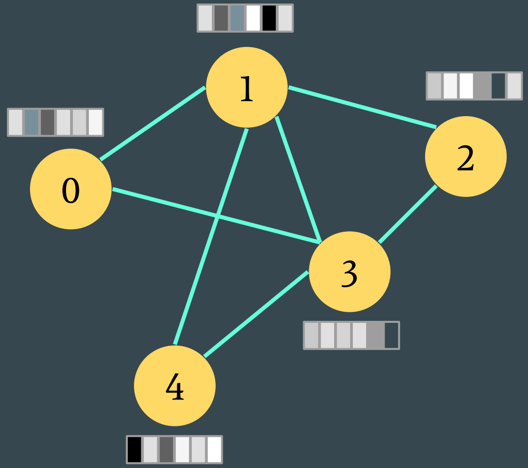
Aim:

- Generalize classical deep learning concepts to irregular structured data (in contrast to images or text)
- Enable neural networks to reason about objects and their relations

How it's done:

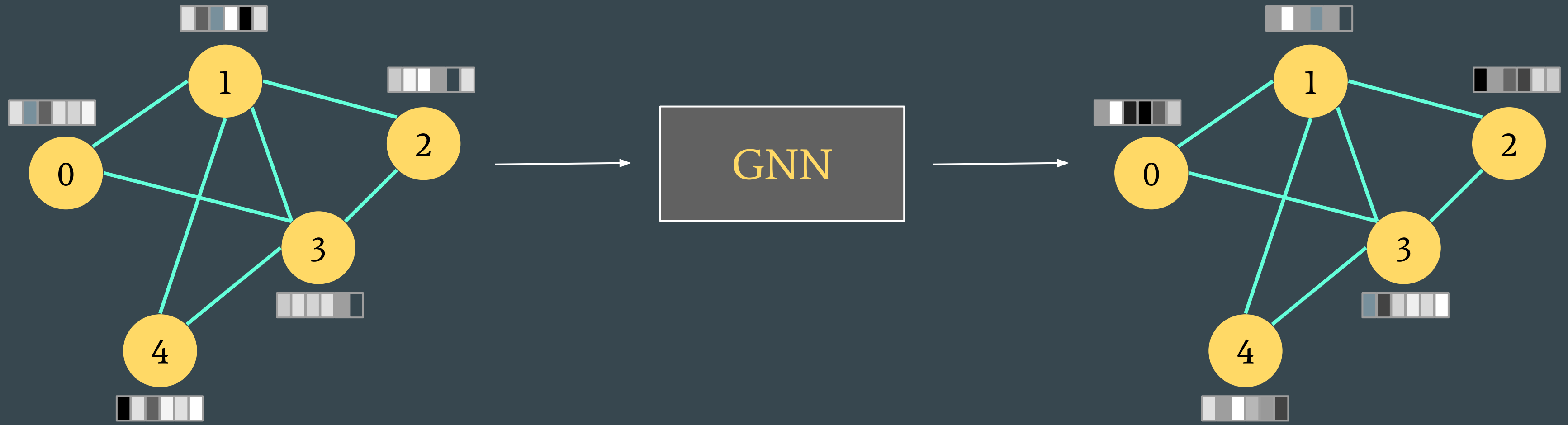
- Neural message passing scheme, where node features are iteratively updated by aggregating localized information from their neighbors

Graph Neural Networks



Initial node representation

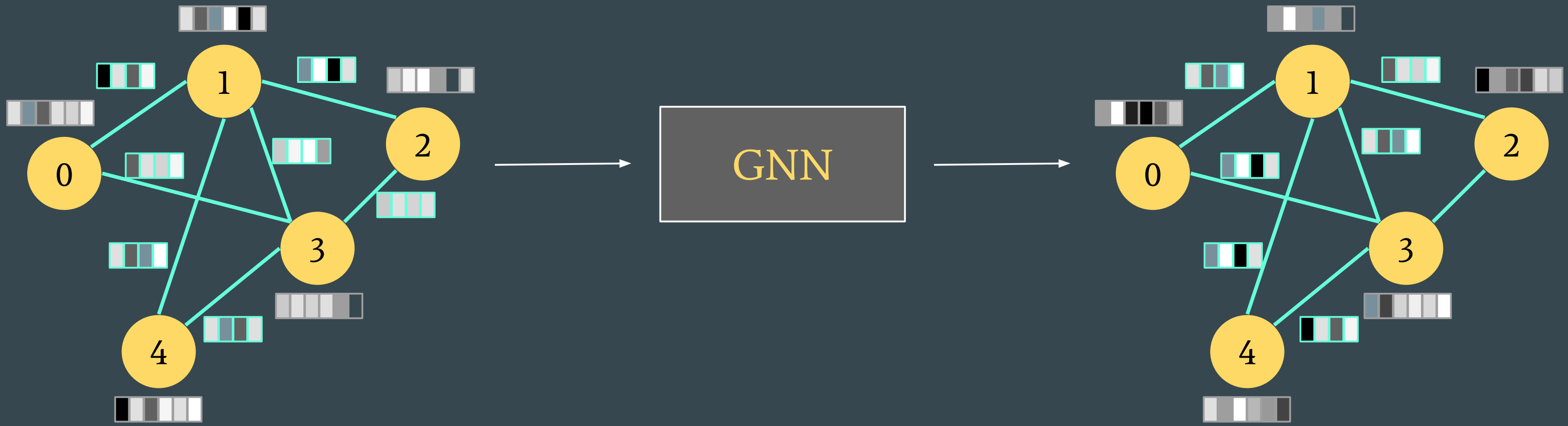
Graph Neural Networks



Initial node representation

Output representations of nodes
How they belong in graph context

Graph Neural Networks

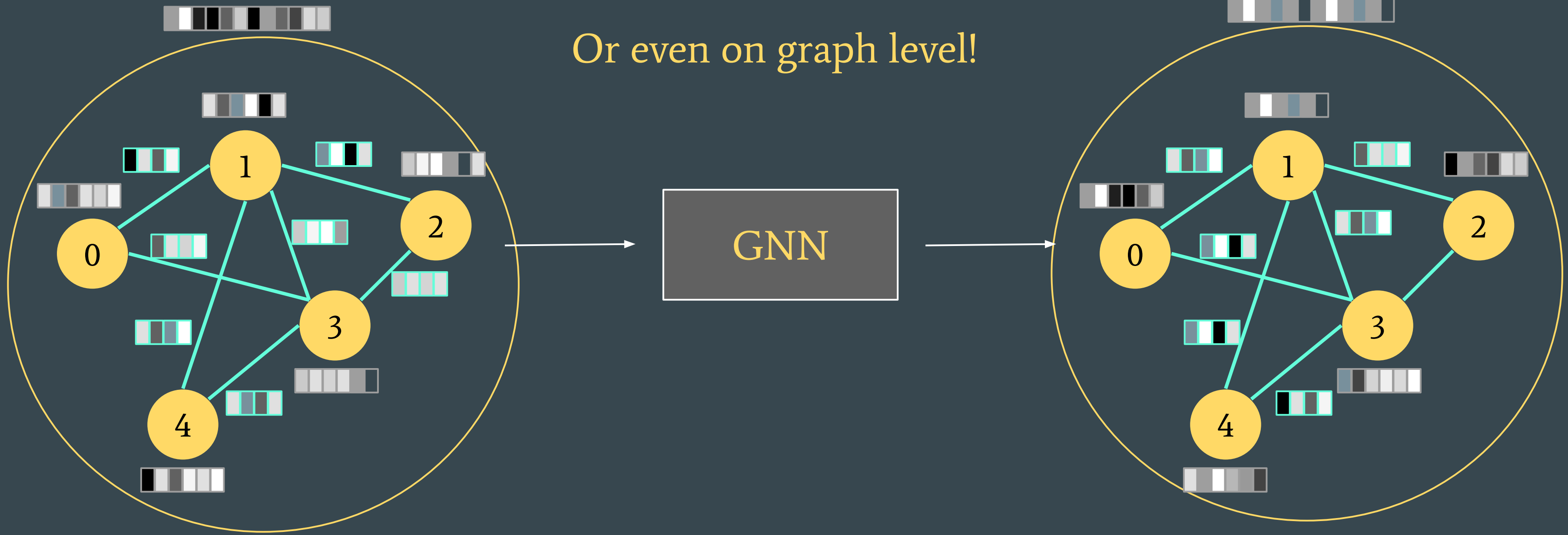


Initial node representation
Also for edges

Output representations of nodes/edges
How they belong in graph context

Graph Neural Networks

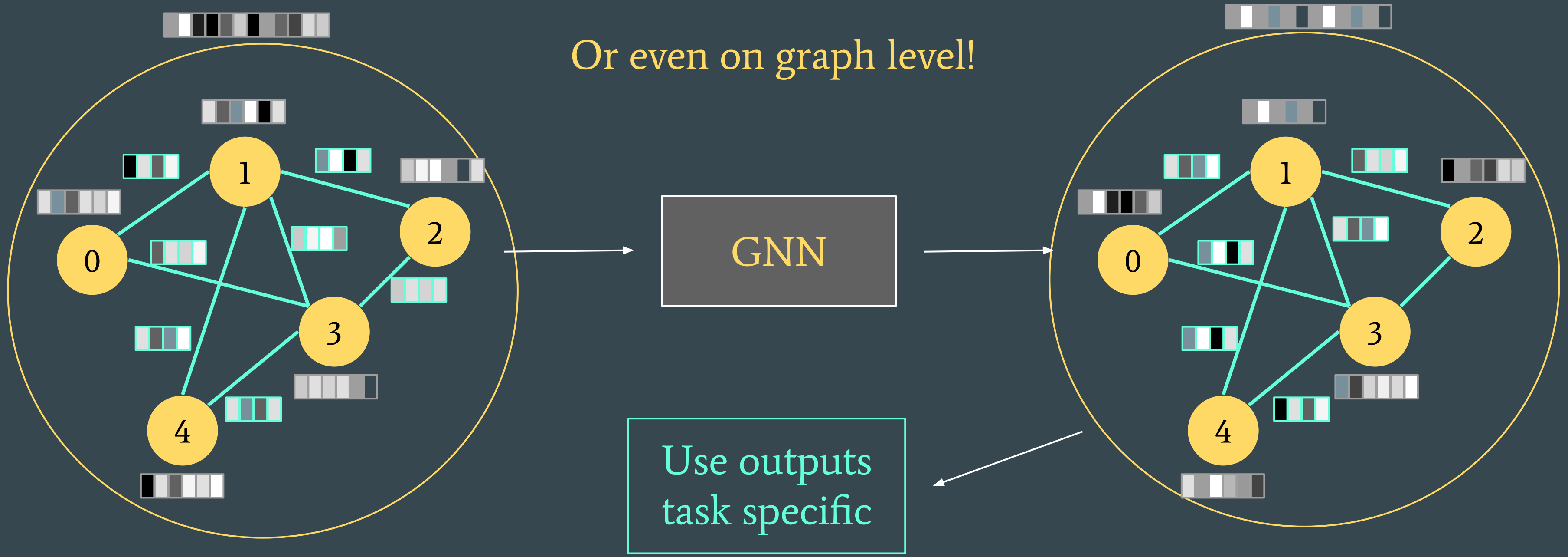
Or even on graph level!



Initial node representation
Also for edges

Output representations of nodes/edges
How they belong in graph context

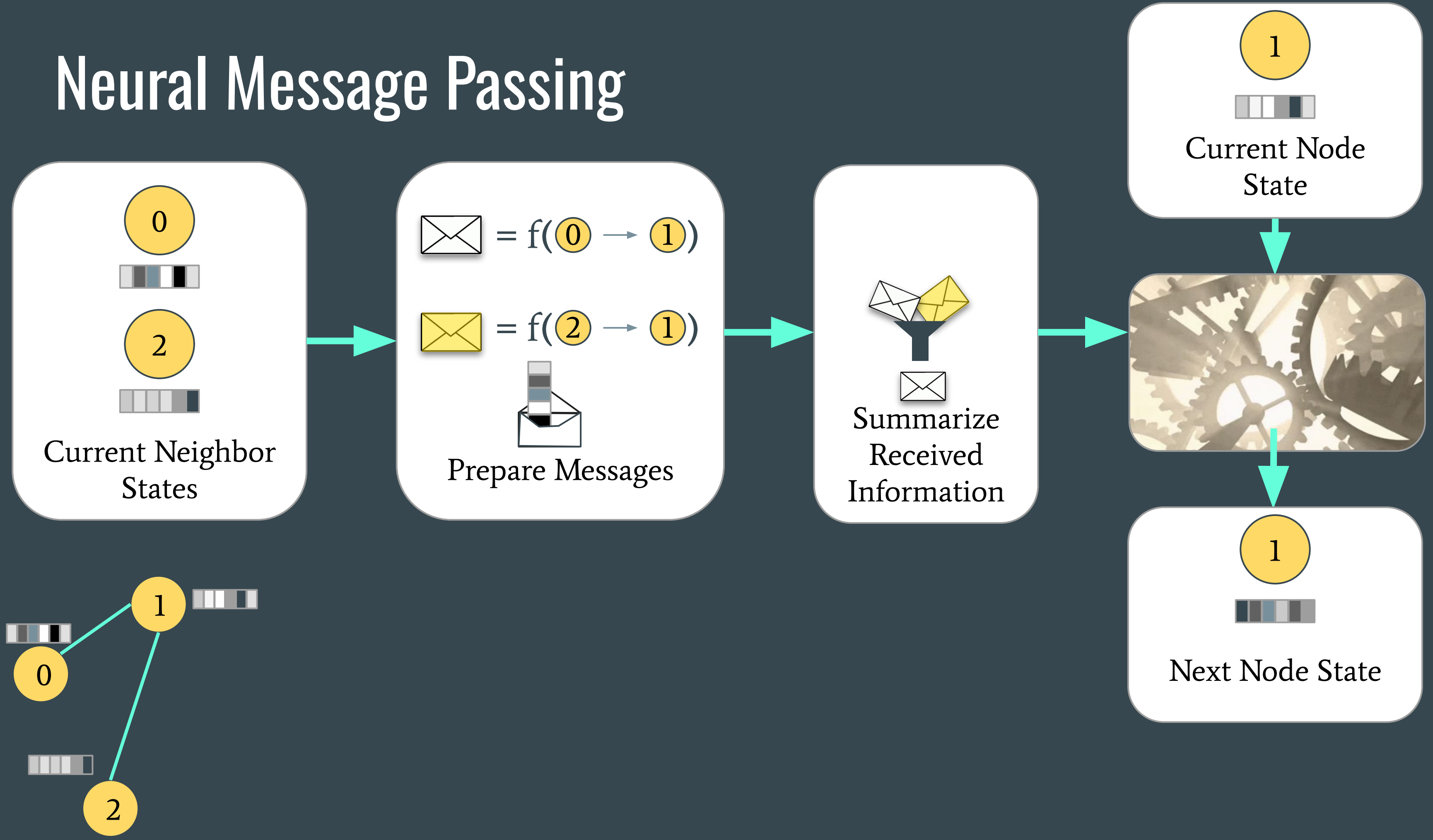
Graph Neural Networks



Initial node representation
Also for edges

Output representations of nodes/edges
How they belong in graph context

Neural Message Passing



Neural Message Passing – Adjacency Matrix

0

2

Current Neighbor States

= f(0 → 1)

= f(2 → 1)

Prepare Messages

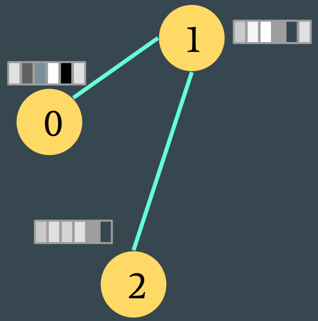
Summarize Received Information

1

Current Node State

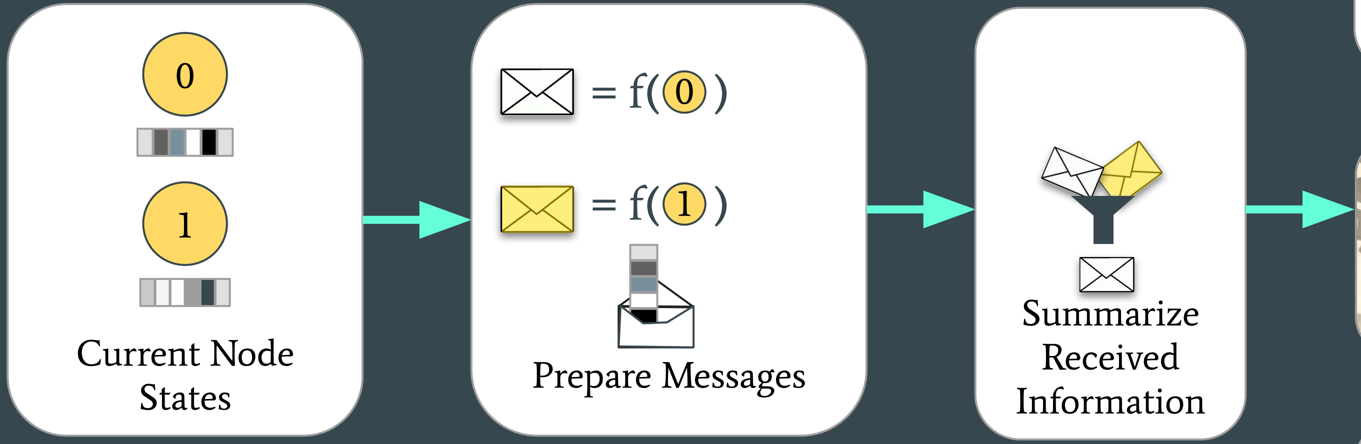
1

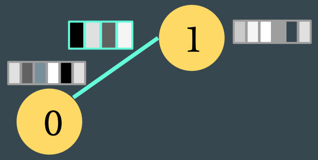
Next Node State



$$A=\begin{bmatrix}0 & 1 & 0\\1 & 0 & 1\\0 & 1 & 0\end{bmatrix},\ N=\begin{bmatrix}a\\b\\c\end{bmatrix},\ A\cdot N=\begin{bmatrix}b\\a+c\\b\end{bmatrix}$$

Interaction Network (Edge Update)



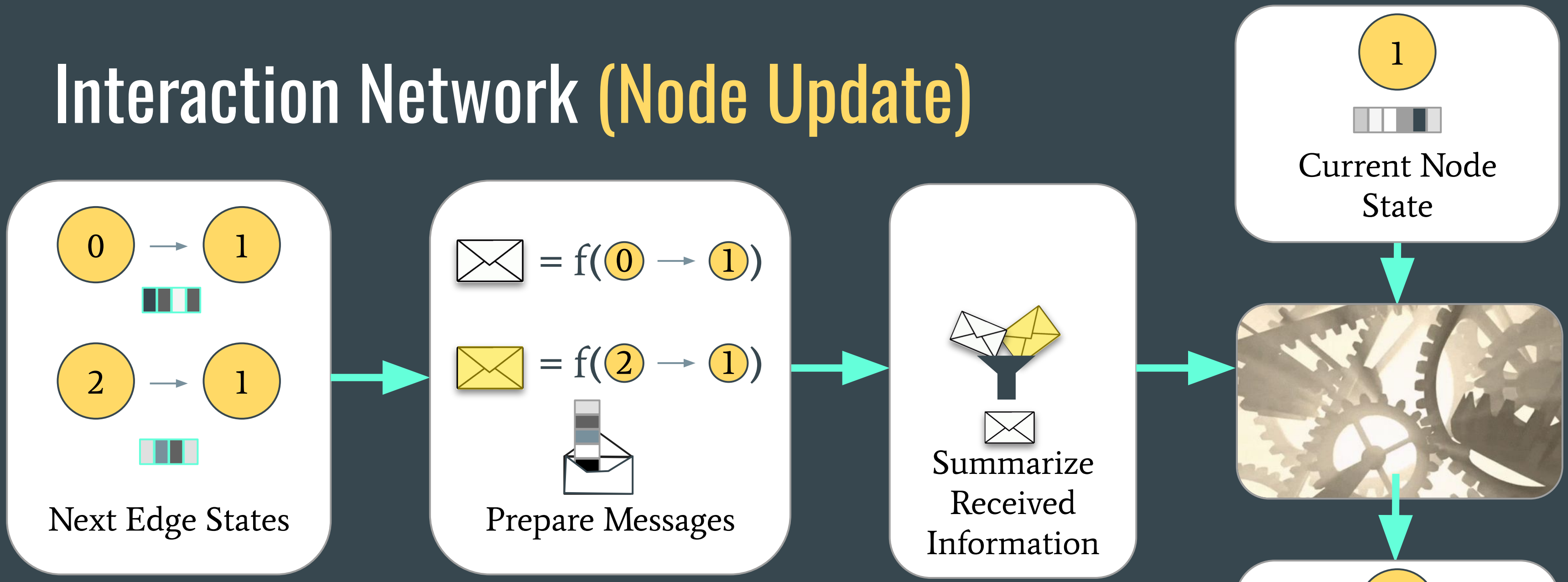


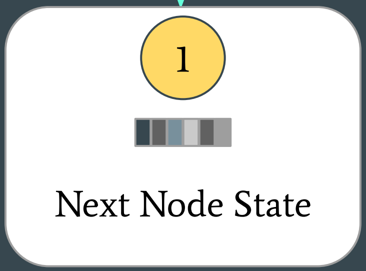
$$e_{x,y}^{1} = \phi_e(v_x^{0}, v_y^{0}, e_{x,y}^{0})$$

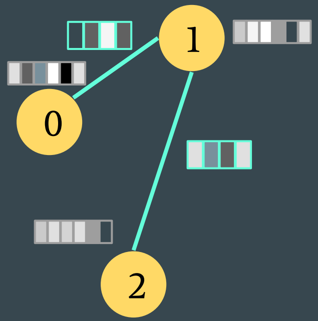
ϕ_e: Edge update MLP

v_x^{k} = features of node x at iteration k
$e_{x,y}^{k}$ = features of edge between nodes x and y at iteration k

Interaction Network (Node Update)







$$v_x^{1} = \phi_n(v_x^{0}, \sum e_{x,y}^{1})$$

ϕ_n: Node update MLP

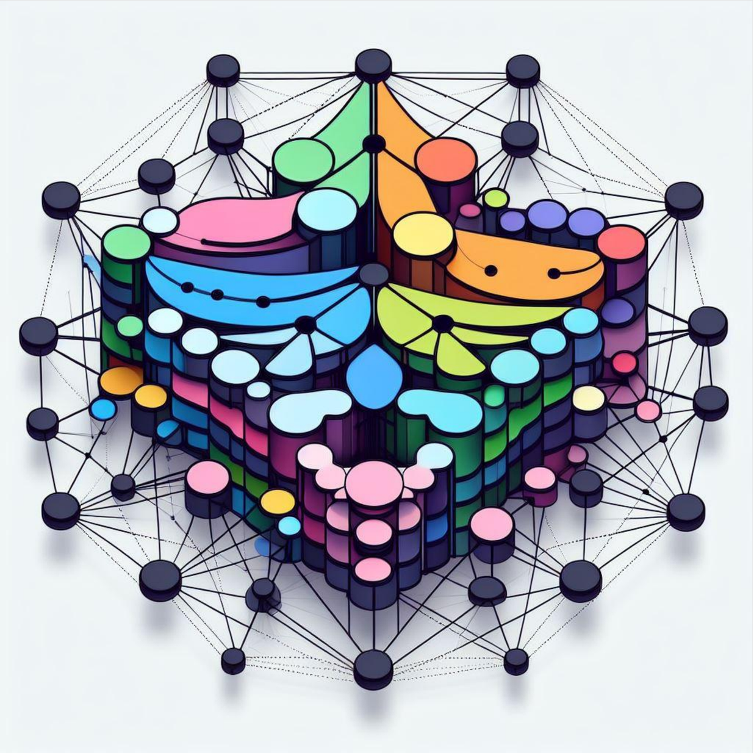
$\sum$: Aggregation function

v_x^k = features of node x at iteration k

$e_{x,y}^k$ = features of edge between nodes x and y at iteration k

PyG

ML with Graphs
PyTorch Geometric



created with https://designer.microsoft.com/image-creator

GNNs with PyG

- Can all be done with plain PyTorch
  - Matrix multiplications with adjacency matrix, as seen here
- PyG provides some neat utilities for message passing
- We'll do an introductory example: Karate Club

→ Link to Notebook

→ Link to documentation

```
# Install required packages.
import os
import torch
os.environ['TORCH'] = torch.__version__
print(torch.__version__)

!pip install -q torch-scatter -f
https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q torch-sparse -f
https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q
git+https://github.com/pyg-team/pytorch_geometric.git

# Helper function for visualization.
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
```

GNNs with PyG

- Can all be done with plain PyTorch
  - Matrix multiplications with adjacency matrix, as seen here
- PyG provides some neat utilities for message passing
- We'll do an introductory example: Karate Club

→ Link to Notebook

→ Link to documentation

```
def visualize_graph(G, color):
    plt.figure(figsize=(7,7))
    plt.xticks([])
    plt.yticks([])
    nx.draw_networkx(G, pos=nx.spring_layout(G, seed=42), with_labels=False,
                     node_color=color, cmap="Set2")
    plt.show()


def visualize_embedding(h, color, epoch=None, loss=None):
    plt.figure(figsize=(7,7))
    plt.xticks([])
    plt.yticks([])
    h = h.detach().cpu().numpy()
    plt.scatter(h[:, 0], h[:, 1], s=140, c=color, cmap="Set2")
    if epoch is not None and loss is not None:
        plt.xlabel(f'Epoch: {epoch}, Loss: {loss.item():.4f}', fontsize=16)
    plt.show()
```

GNNs with PyG

- Loading the dataset
- Property inspection
- Detailed look at the data

```
from torch_geometric.datasets import KarateClub

dataset = KarateClub()
print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

data = dataset[0]  # Get the first graph object.

print(data)
print('=====================================================
=========')

# Gather some statistics about the graph.
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges /
data.num_nodes:.2f}')
print(f'Number of training nodes: {data.train_mask.sum()}')
print(f'Training node label rate:
{int(data.train_mask.sum()) / data.num_nodes:.2f}')
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
print(f'Has self-loops: {data.has_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

[→ Link to Notebook]

[→ Link to documentation]

GNNs with PyG

- edge_index holds a tuple of two node indices for each edge
- Edges are stored in COO format (coordinate format)
- Visualization

→ Link to Notebook

→ Link to documentation

```
edge_index = data.edge_index
print(edge_index.t())


from torch_geometric.utils import to_networkx

G = to_networkx(data, to_undirected=True)
visualize_graph(G, color=data.y)
```

GNNs with PyG

- Implementing a Graph Neural Network
- GCN layer
(Graph Convolutional Network)

$$\mathbf{x}_v^{(\ell+1)} = \mathbf{W}^{(\ell+1)} \sum_{w \in \mathcal{N}(v) \cup \{v\}} \frac{1}{c_{w,v}} \cdot \mathbf{x}_w^{(\ell)}$$

W: trainable weight matrix
c: fixed normalization coefficient per edge

[→ Link to Notebook]

[→ Link to documentation]

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv


class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        torch.manual_seed(1234)
        self.conv1 = GCNConv(dataset.num_features, 4)
        self.conv2 = GCNConv(4, 4)
        self.conv3 = GCNConv(4, 2)
        self.classifier = Linear(2, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh()  # Final GNN embedding space.

        # Apply a final (linear) classifier.
        out = self.classifier(h)

        return out, h

model = GCN()
print(model)
```

GNNs with PyG

- Visualization of embedding

```
model = GCN()

_, h = model(data.x, data.edge_index)
print(f'Embedding shape: {list(h.shape)}')

visualize_embedding(h, color=data.y)
```

→ Link to Notebook

→ Link to documentation

GNNs with PyG

- Loss
- Optimizer
- And training :)

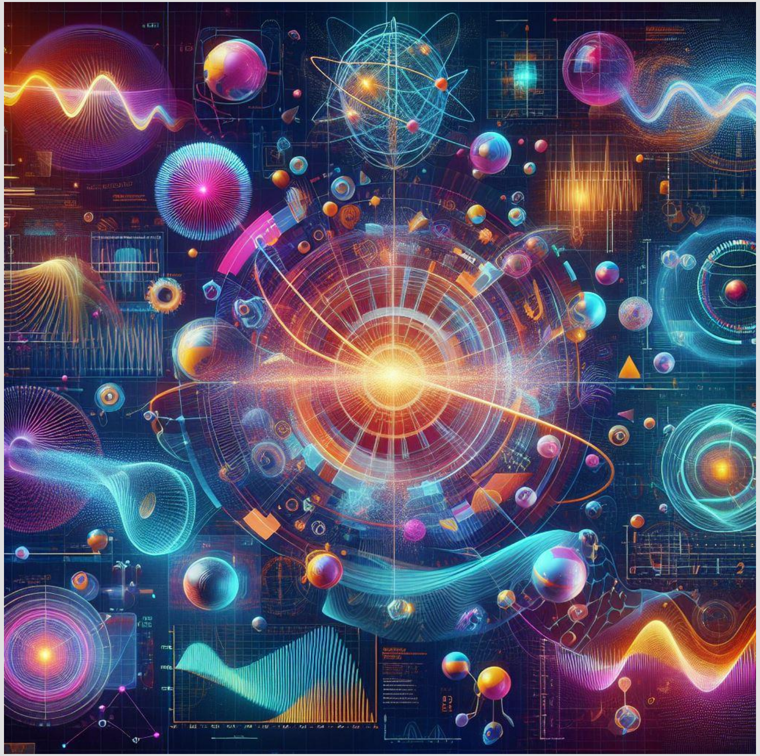
```
import time

model = GCN()
criterion = torch.nn.CrossEntropyLoss()  # Define loss
criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)  #
Define optimizer.

def train(data):
    optimizer.zero_grad()  # Clear gradients.
    out, h = model(data.x, data.edge_index)  # Perform a
single forward pass.
    loss = criterion(out[data.train_mask],
data.y[data.train_mask])  # Compute the loss solely based on
the training nodes.
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on
gradients.
    return loss, h

for epoch in range(1001):
    loss, h = train(data)
    if epoch % 10 == 0:
        visualize_embedding(h, color=data.y, epoch=epoch,
loss=loss)
        time.sleep(0.3)
```

→ Link to Notebook

→ Link to documentation

Coding Time

TrackML with graphs



created with https://designer.microsoft.com/image-creator

Tracking with Graph Neural Networks

- Task: Classifying track edges with GNNs
- Checkout the zip file, which contains a notebook and some utility files → all files are needed on Colab (Jupyterhub)
- We make use of graphs created with a p_T > 2 GeV cut, constructed with about 99.7 % edge efficiency and 30 % edge purity
  → we want to improve purity, and keep a high efficiency!
- We will walk through the notebook together
- Experiment! Change the networks, parameters, weightings, ... as you like!
- If you want, you can make use of larger graphs created without p_T cut!
  99.0 % efficiency, 1.6 % purity → takes longer to train! may require too much memory...

Let’s see the results!

created with https://designer.microsoft.com/image-creator

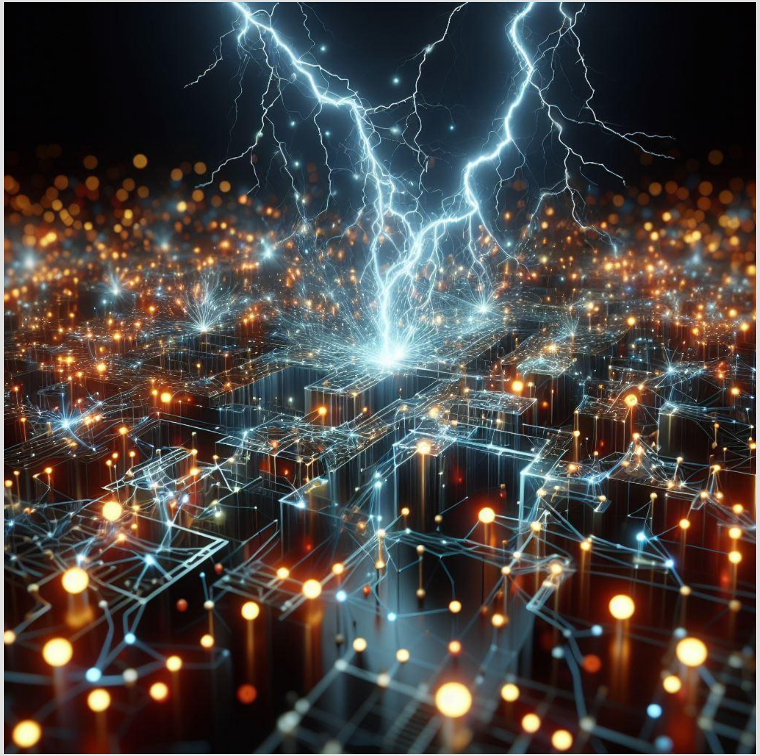
What can you expect in the coming days?

Monday, 07.04.2025	The Basics *MNIST, Linear Regression*
Tuesday, 08.04.2025	*A deeper dive* *CNNs @ MNIST, RNNs @ names*
Wednesday, 09.04.2025	The Problem *Tracking, TrackML kNN search*
Thursday, 10.04.2025	The Solution *Graph Neural Networks, PyTorch Geometric, TrackML GNN*
Today, 11.04.2025	The Add-On *PyTorch Lightning, Fun & Games*

Lightning AI

Creators of PyTorch Lightning

PyTorch Lightning



created with https://designer.microsoft.com/image-creator

What is Lightning?

- Lightning organizes PyTorch code to remove boilerplate and unlock scalability
- 7 steps to translate PyTorch to Lightning
  - Computational code goes into LightningModule (model architecture in __init__)
  - Set forward hook
  - Optimizes go into configure_optimizers hook
  - Training logic goes into training_step
  - Validation logic goes into validation_step
  - Remove device calls → lightning modules are hardware agnostic
  - Override more LightningModule hooks (if needed, +20 hooks for full flexibility)
- Lightning Trainer
  - Automates engineering of loops, hardware calls, train, eval, zero_grad, ...
  - Takes PyTorch DataLoaders
  - More functionalities via callbacks
  - Choose device for training

Let's take a look at an example

→ Link to Notebook

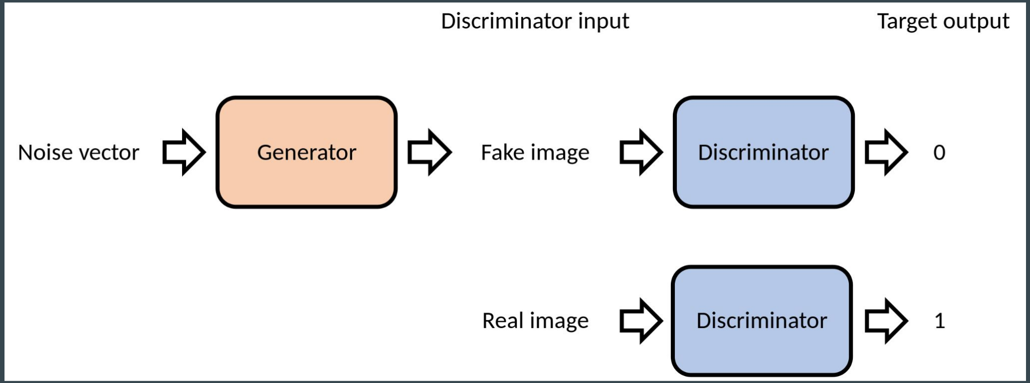
compare to plain PyTorch

Generative AI

created with https://designer.microsoft.com/image-creator

Generative Adversarial Networks

- Two neural networks, Generator and Discriminator, contest each other in a zero-sum game
- The Discriminator tries to distinguish true images from fake images generated by the Generator
- The Generator tries to fool the Discriminator, such that it cannot distinguish anymore between true and fake images



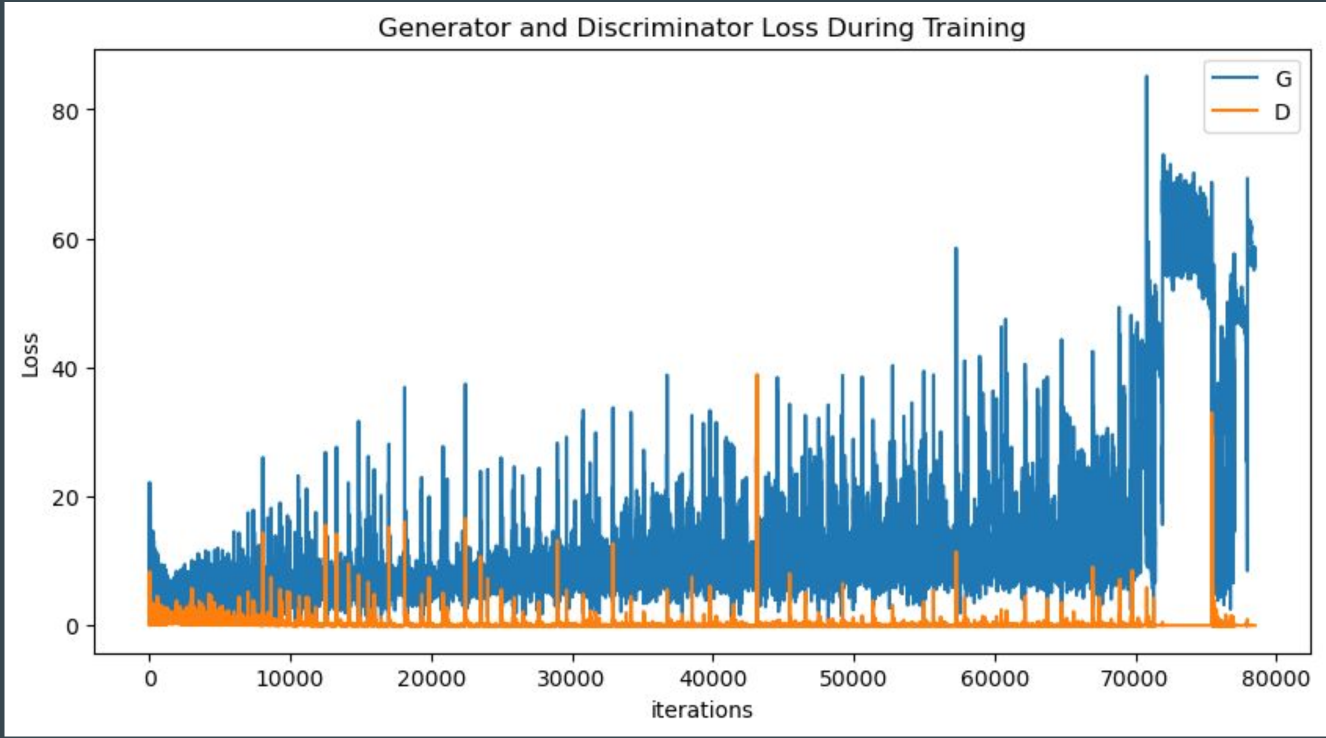
https://commons.wikimedia.org/w/index.php?curid=133747411

Generating new Pokemon with a DCGAN

- Checkout the notebook and the data
- This notebook is an adaption from the original PyTorch tutorial (which is about generating new celebrity faces)
- Play with it, let's see some new shiny Pokemons!
- You can explore the code during training (this may take some time, especially without a GPU)
- Feel free to search for new image datasets, change the neural networks, hyperparameters, …
  You may need to tweak parameters for good results
  → see in 2 slides what can happen

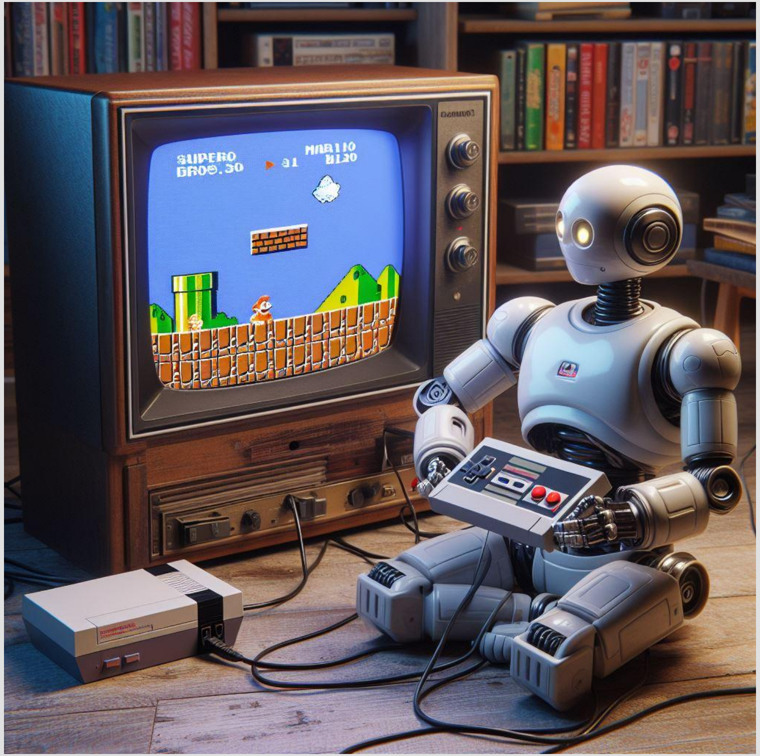
Can you create better Pokemons?

What can go wrong with GANs?



What can go wrong with GANs?

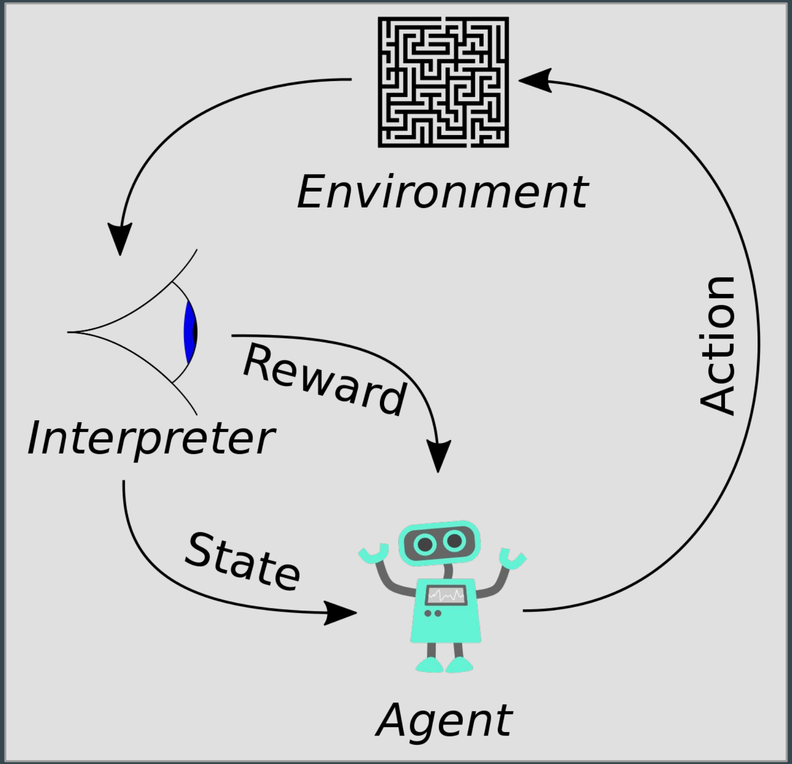
Reinforcement Learning



created with https://designer.microsoft.com/image-creator

Reinforcement Learning

- Machine Learning + optimal control
- Intelligent agent to take actions in a dynamic environment to maximize cumulative reward
- Markov decision process:
  - Set of environment and agent states S
  - Set of actions A
  - Probabilities $P_a(s,s')$ to transition from state s to s' under action a
  - Immediate reward $R_a(s,s')$
  - Optimization objective: find best action in state s



https://commons.wikimedia.org/w/index.php?curid=57895741

Reinforcement Learning Agent playing Mario

- Check out the official PyTorch example notebook
- You can increase epochs to see how good your agent actually gets
- And checkout the code during training

Revisit previous notebooks

created with https://designer.microsoft.com/image-creator

To wrap things up

created with https://designer.microsoft.com/image-creator

Please provide feedback! Thanks :)

Final Remarks

- I hope you learned something over the course of this week
- And feel ready to implement Machine Learning with PyTorch for any of your upcoming projects
- There are many more examples and tutorials around
- For instance, we did not touch transformers

- If you are intrigued by the application of GNNs (ML) for track reconstruction and you are looking for a thesis project
  → get in touch! (gfazzino@physi.uni-heidelberg.de, dittmeier@physi.uni-heidelberg.de)