AI In a Nutshell:
How To Build a Machine Learning Model

Aobo Li
Halıcıoğlu Data Science Institute
Department of Physics
UC San Diego

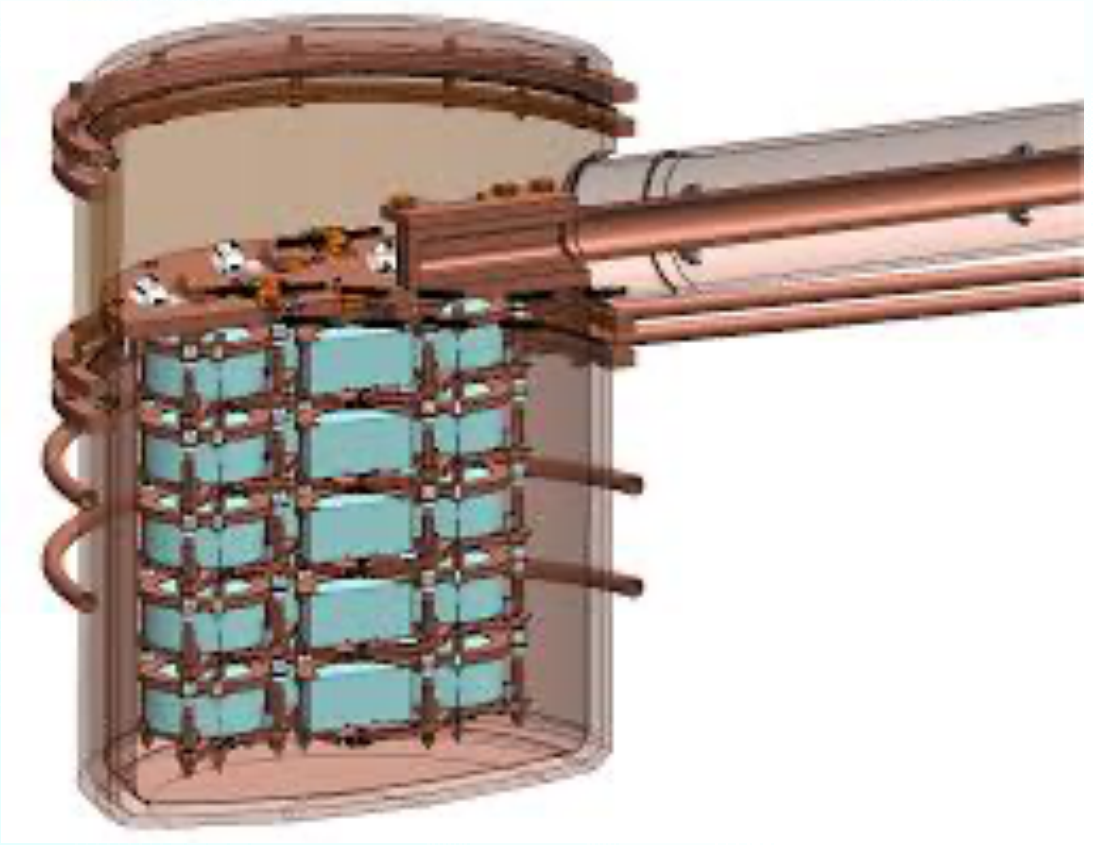
National Nuclear Physics Summer School, 07/22/2024

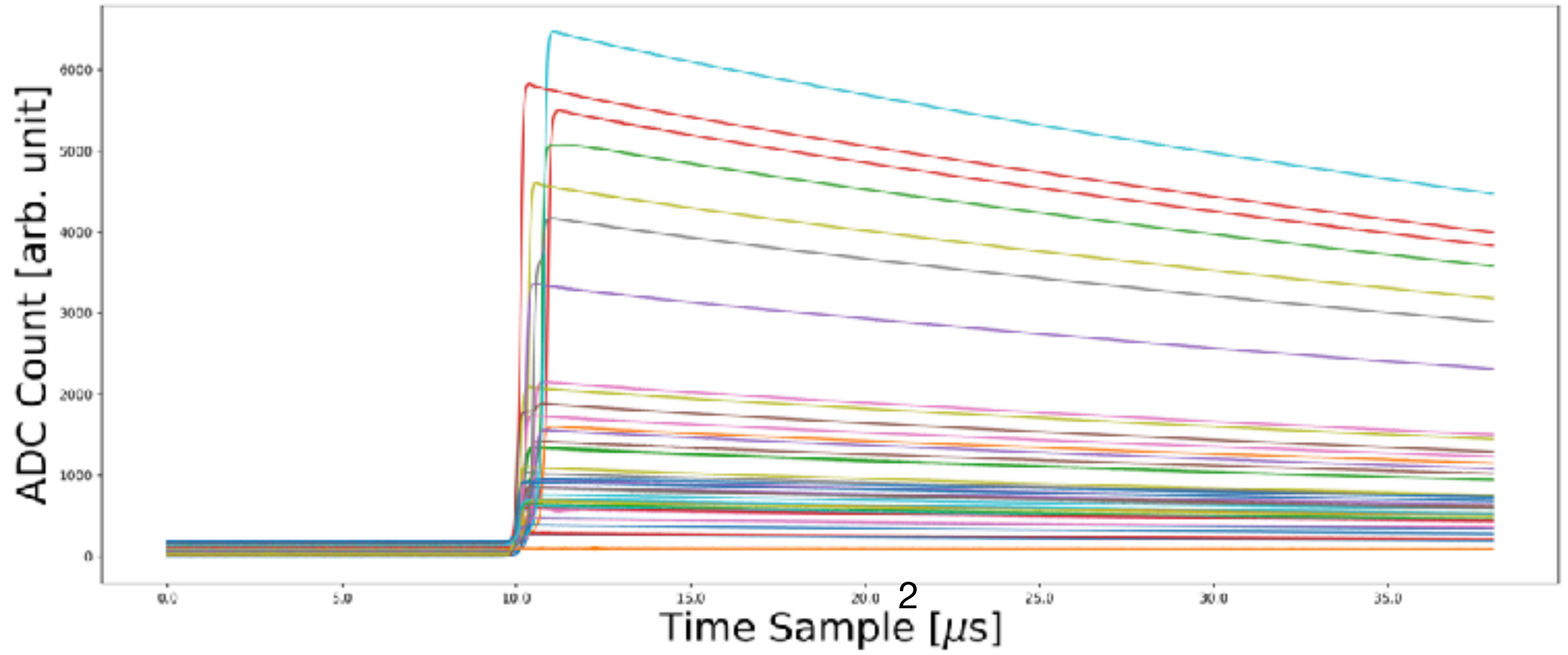
UC San Diego
HALICIOĞLU DATA SCIENCE INSTITUTE

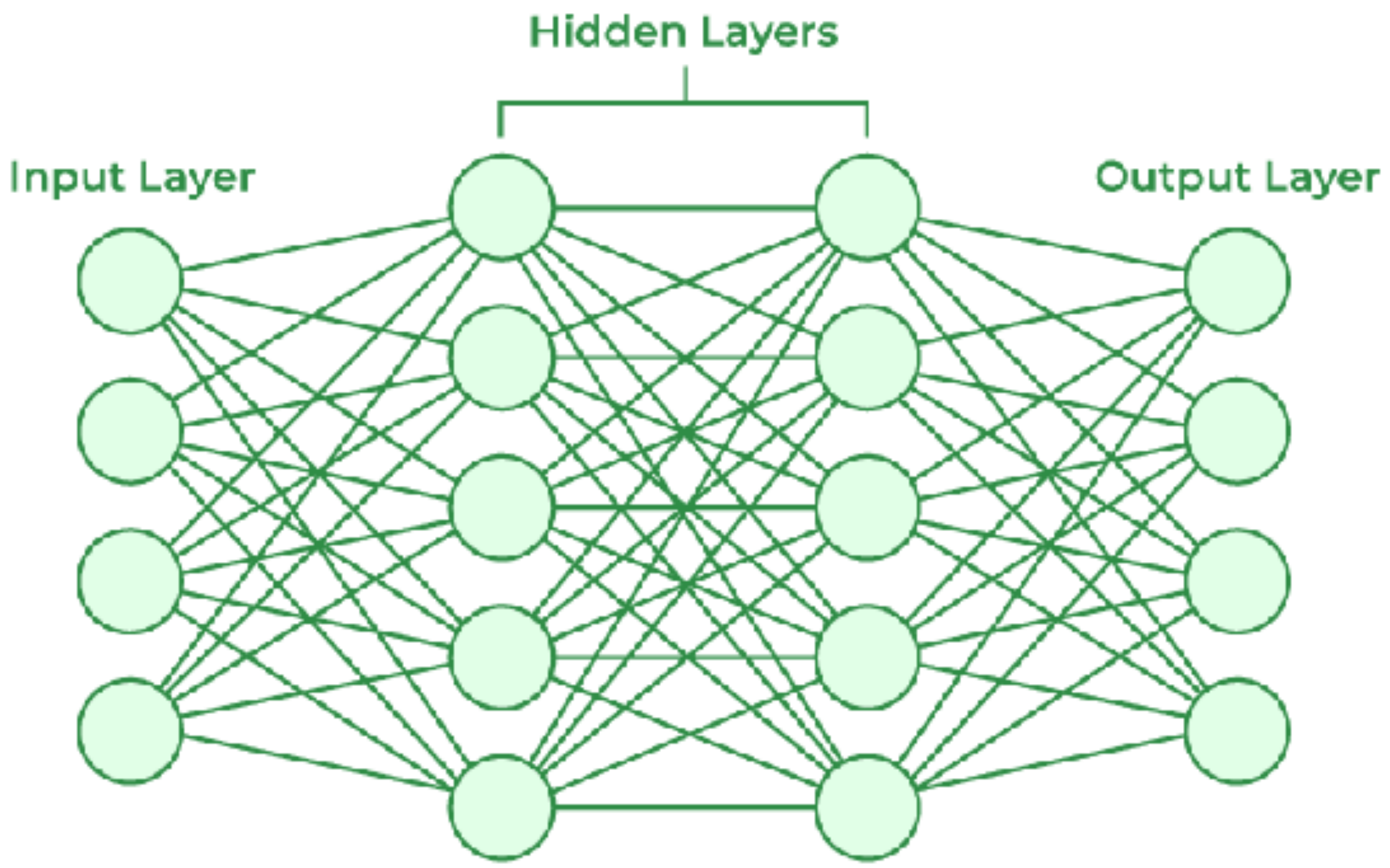
UC San Diego
PHYSICS

Course Objectives

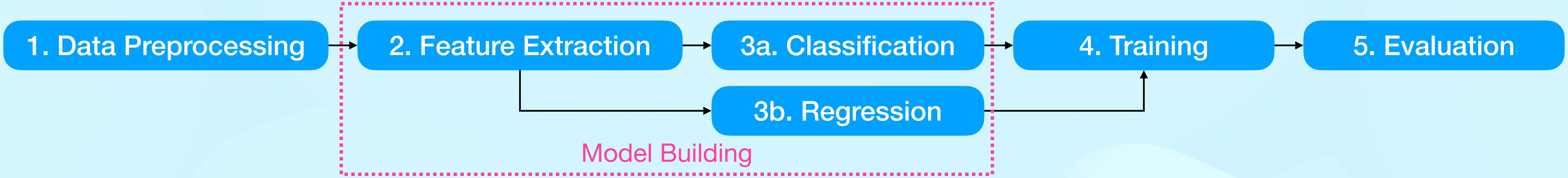
1. Understand Machine Learning from 0 background
    - Assuming some background in Python programming
2. Solve a nuclear physics problem using real detector data
    - Open time series data from the Majorana Demonstrator experiment
3. Building blocks for more advanced models and other experiments
    - Next lecture: connecting dots between AI and NP







Course Outline



Theory

Important equations/theoretical derivations you should understand

Concept

Fundamental concepts that appears everywhere in AI/ML paper and textbook

Code

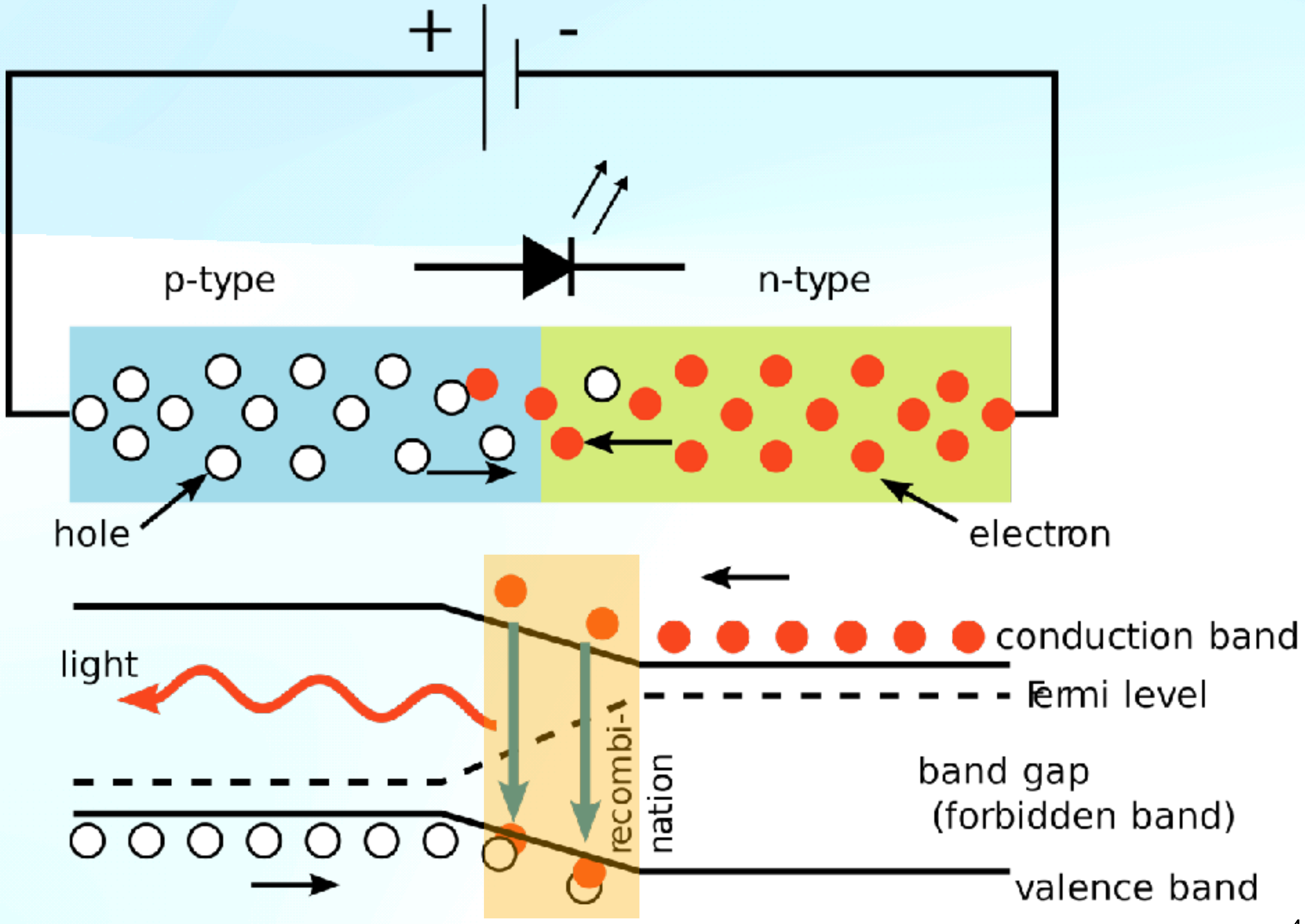
The actual PyTorch implementation, useful for building your own models

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

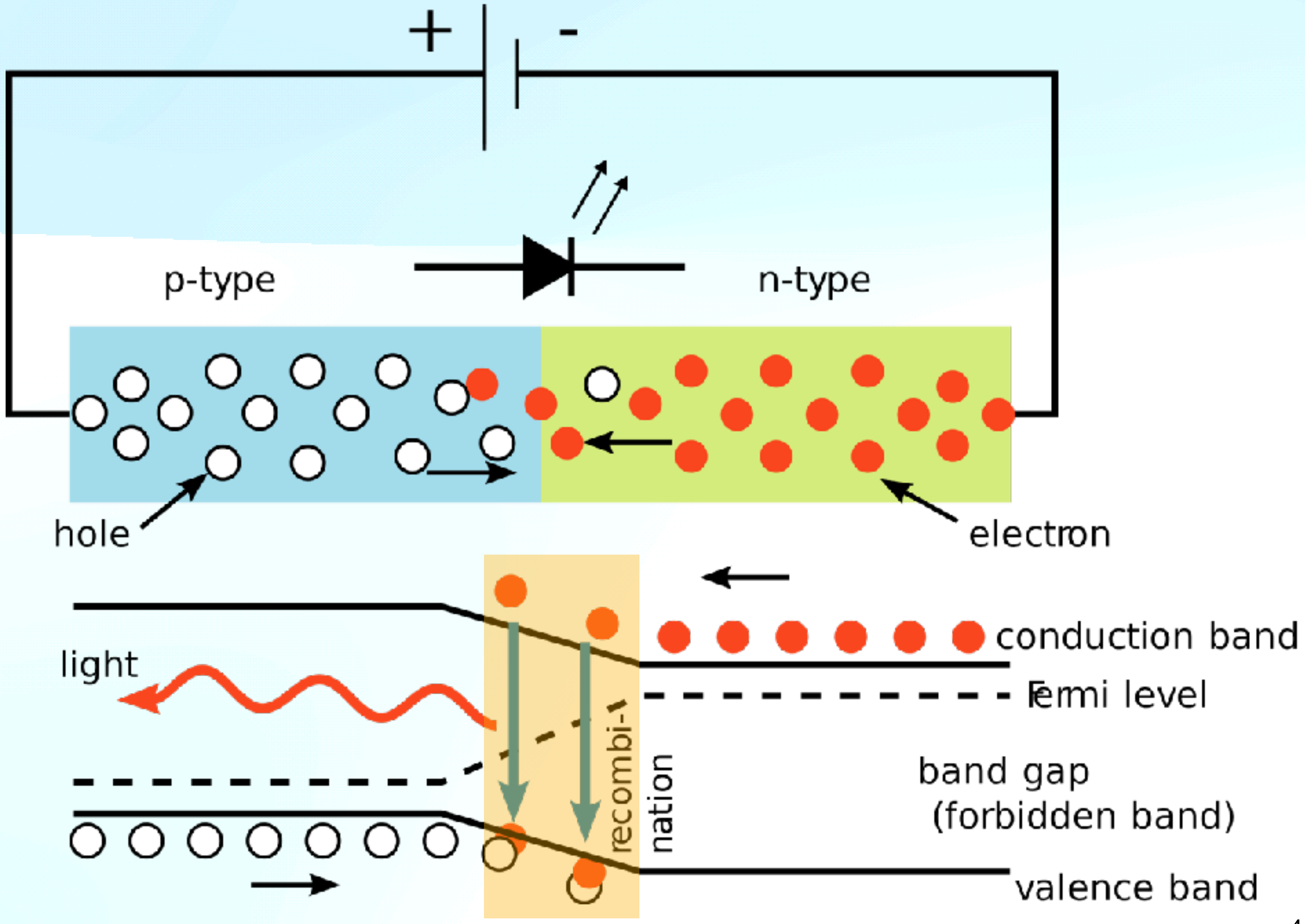
Light Emitting Diode (LED)

Semiconductor device with a **depletion region**



Light Emitting Diode (LED)

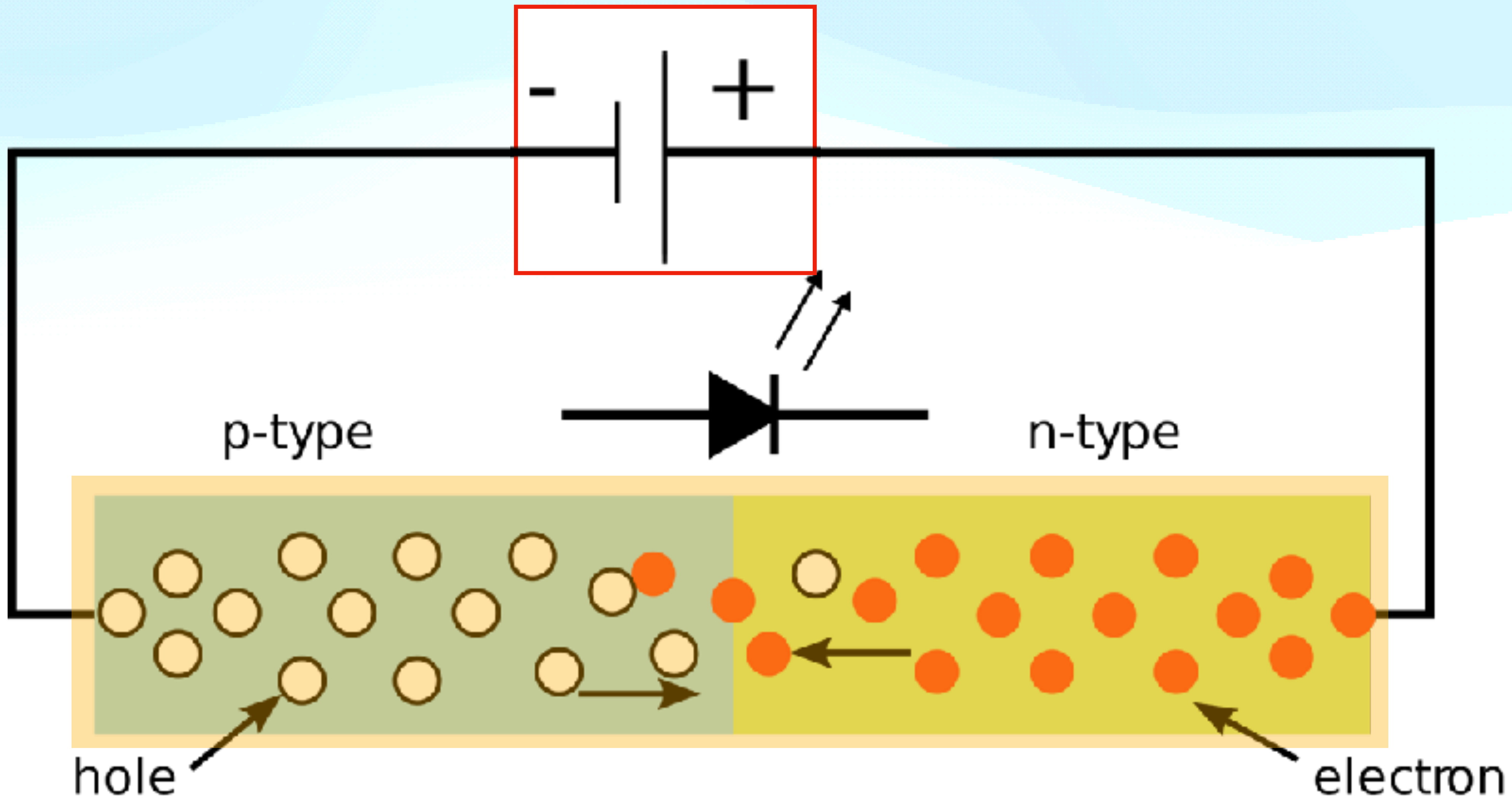
Semiconductor device with a **depletion region**

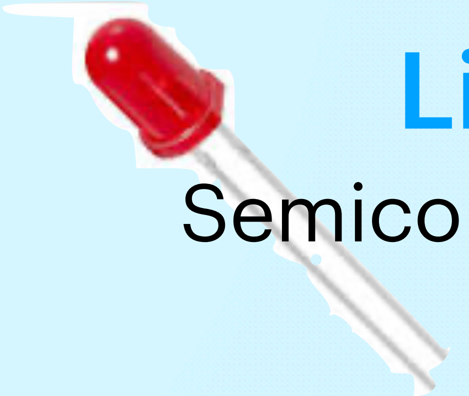


High Purity Ge Detector (HPGe)

^{76}Ge is a double-beta decay isotope

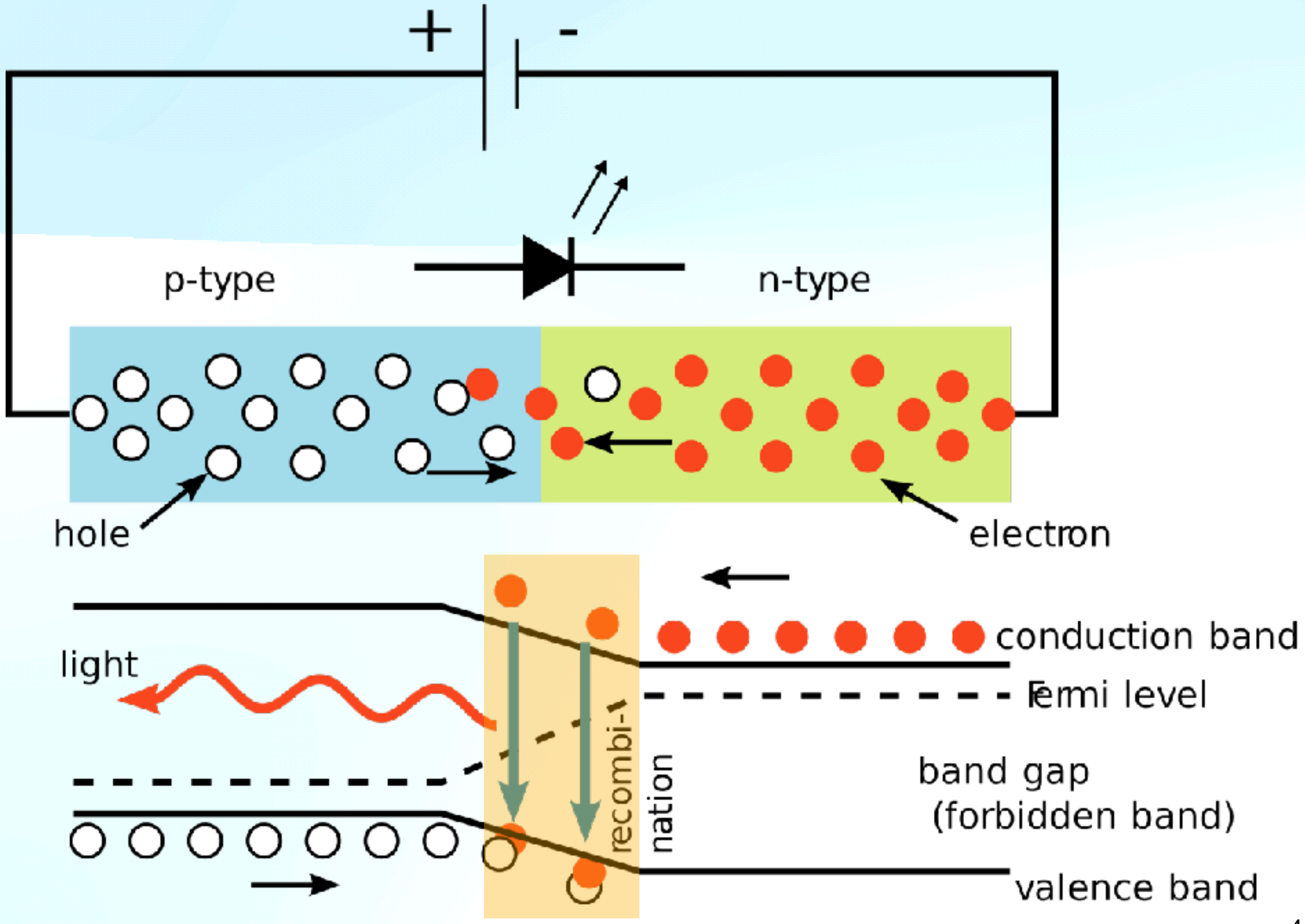
Reverse Bias: increase the size of **depletion region**





Light Emitting Diode (LED)

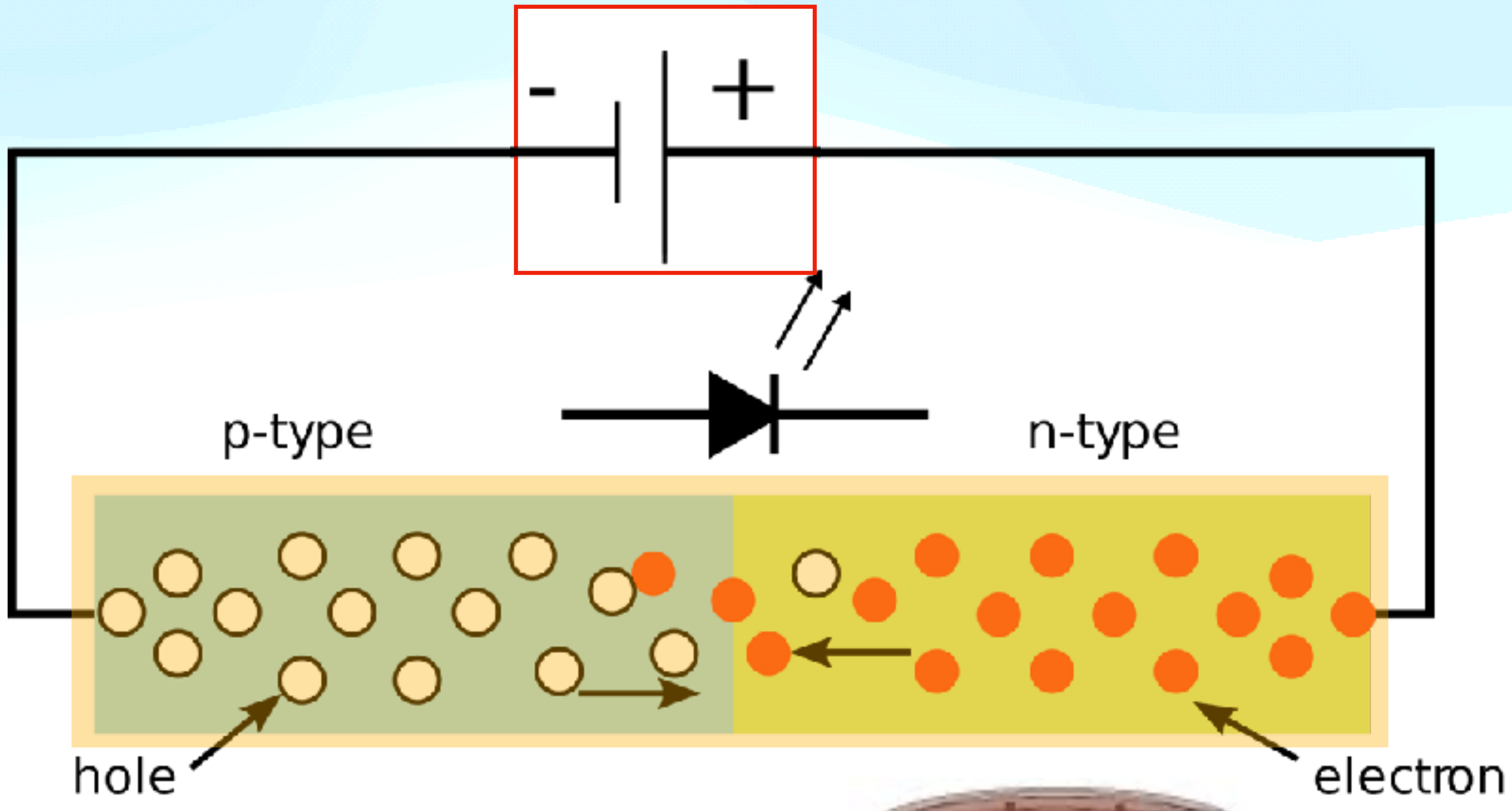
Semiconductor device with a **depletion region**



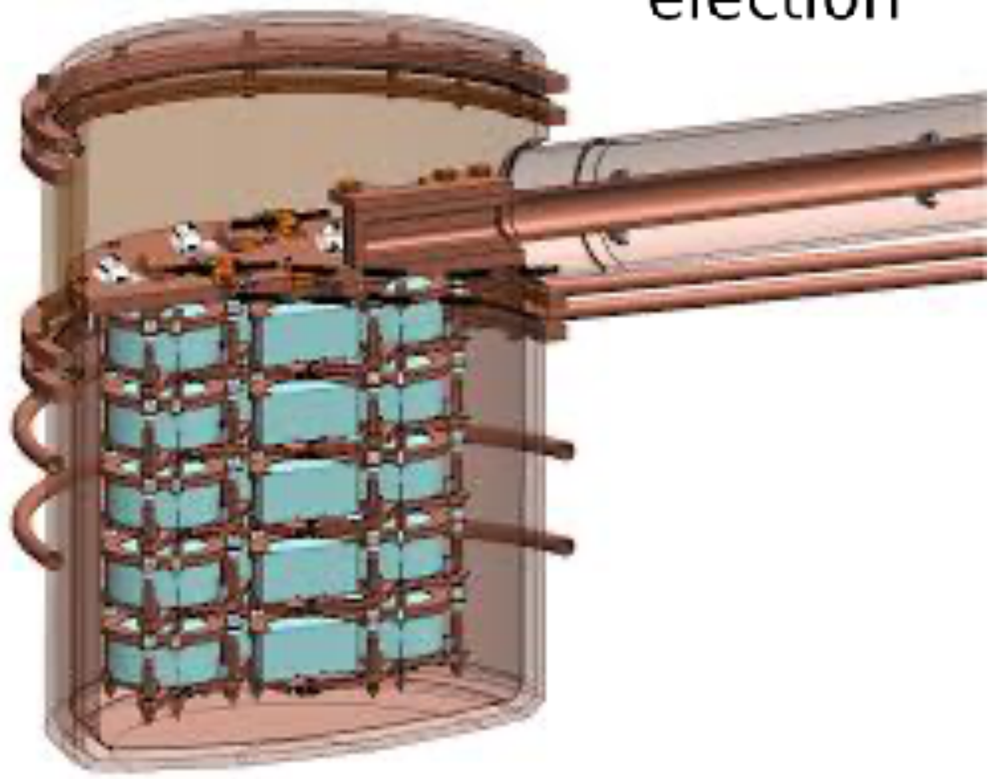
High Purity Ge Detector (HPGe)

^{76}Ge is a double-beta decay isotope

Reverse Bias: increase the size of **depletion region**



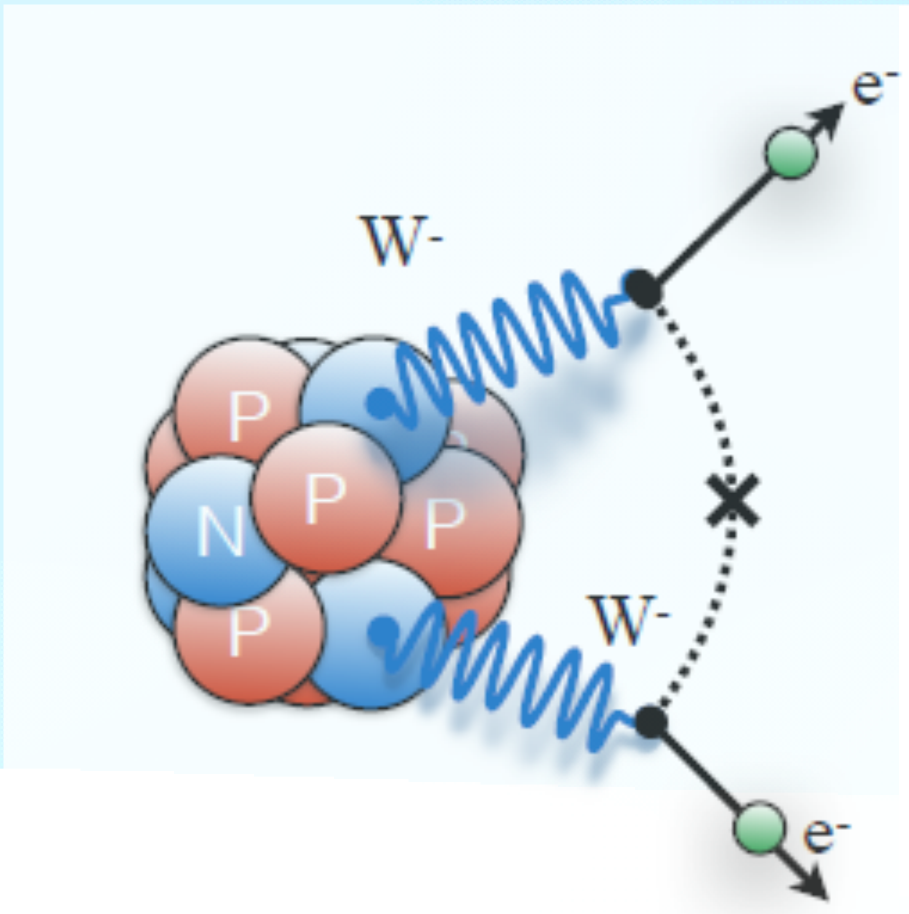
Detector Array

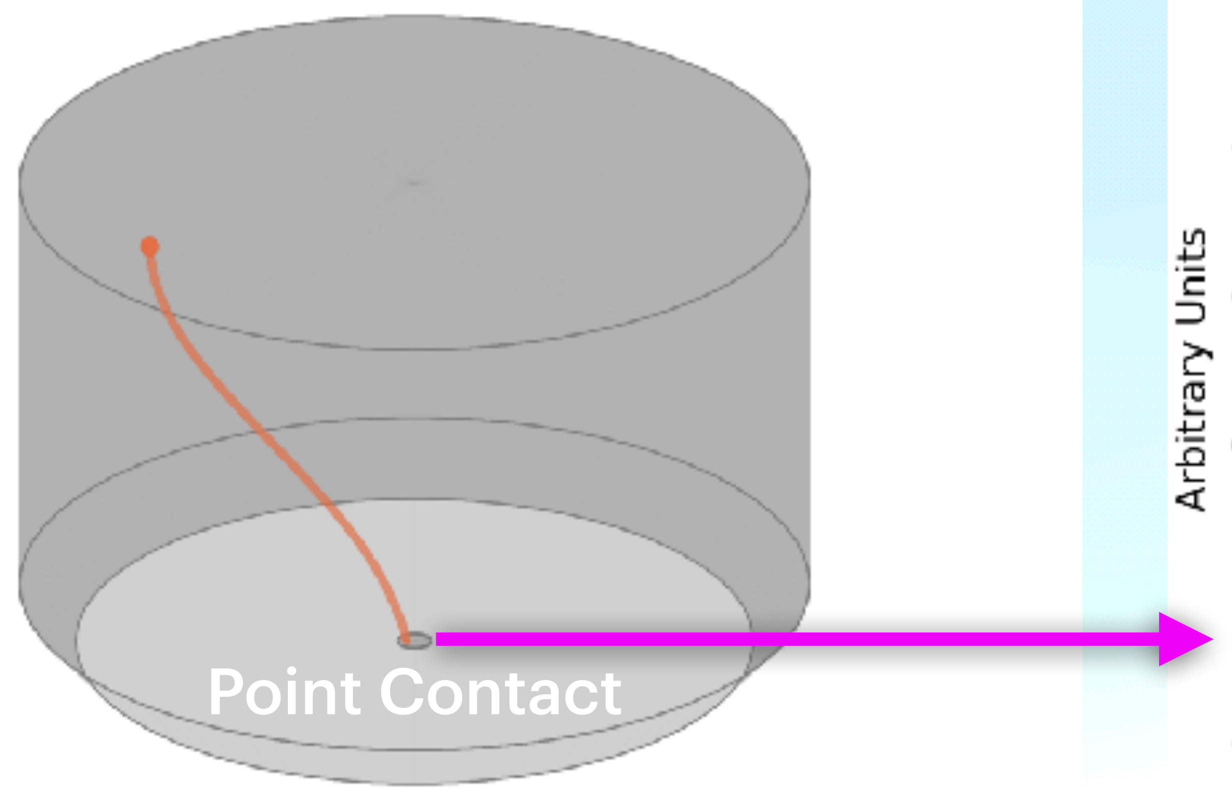


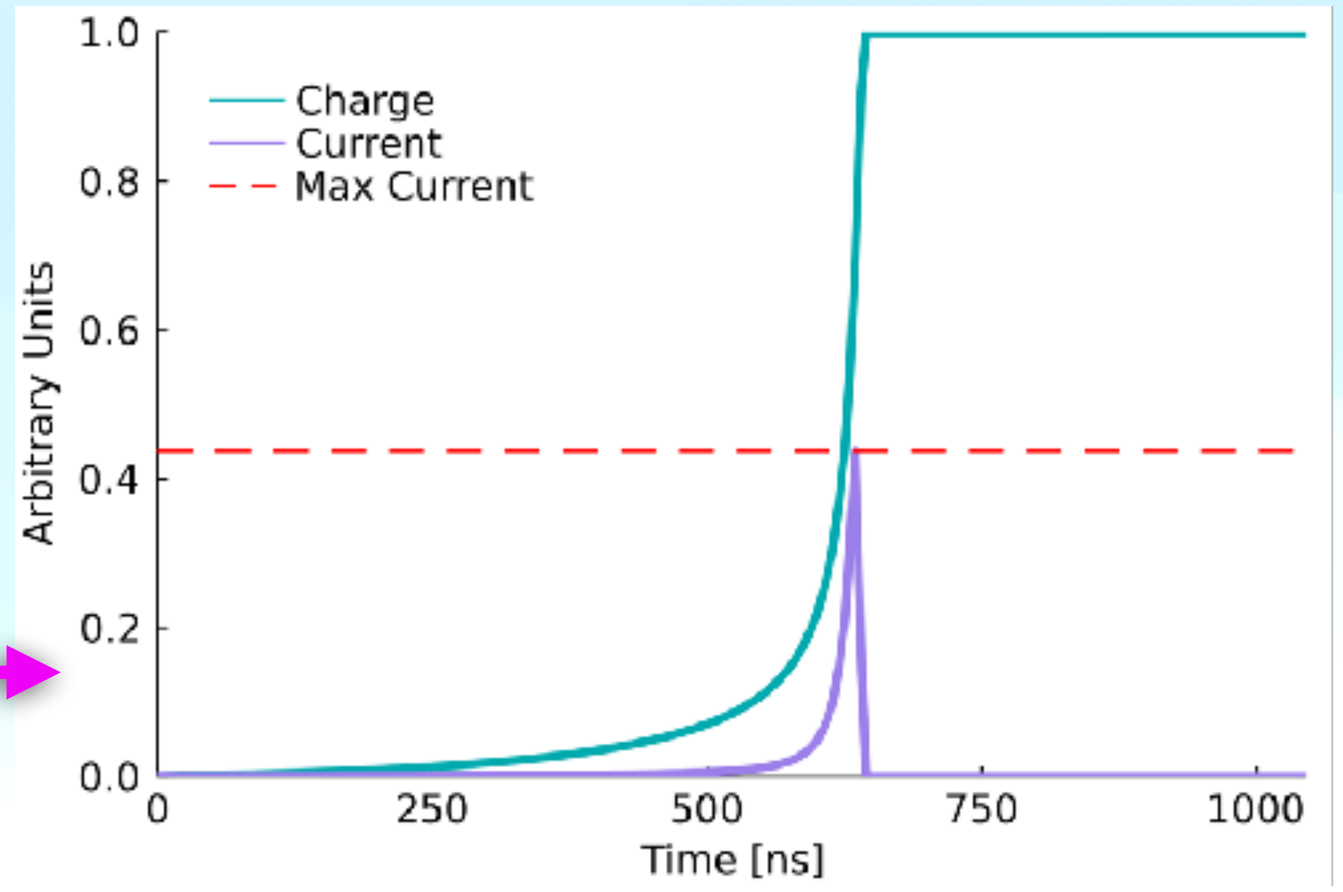
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

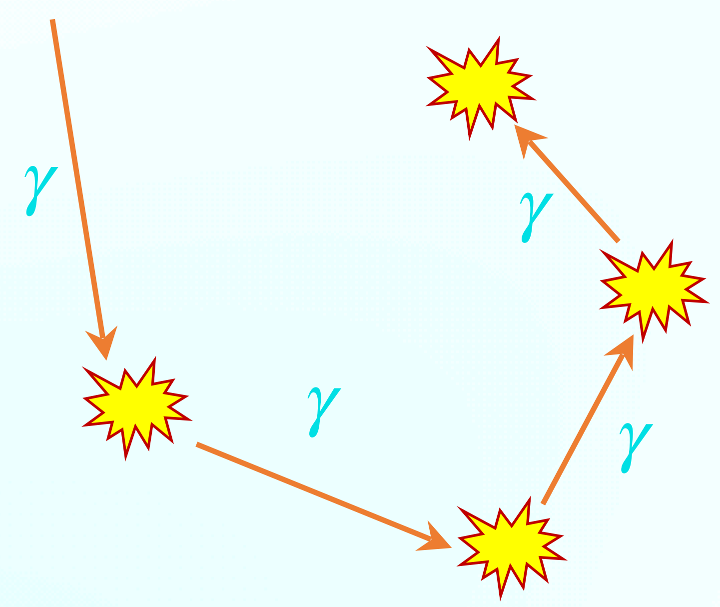
MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\upsilon\beta\beta$ Search

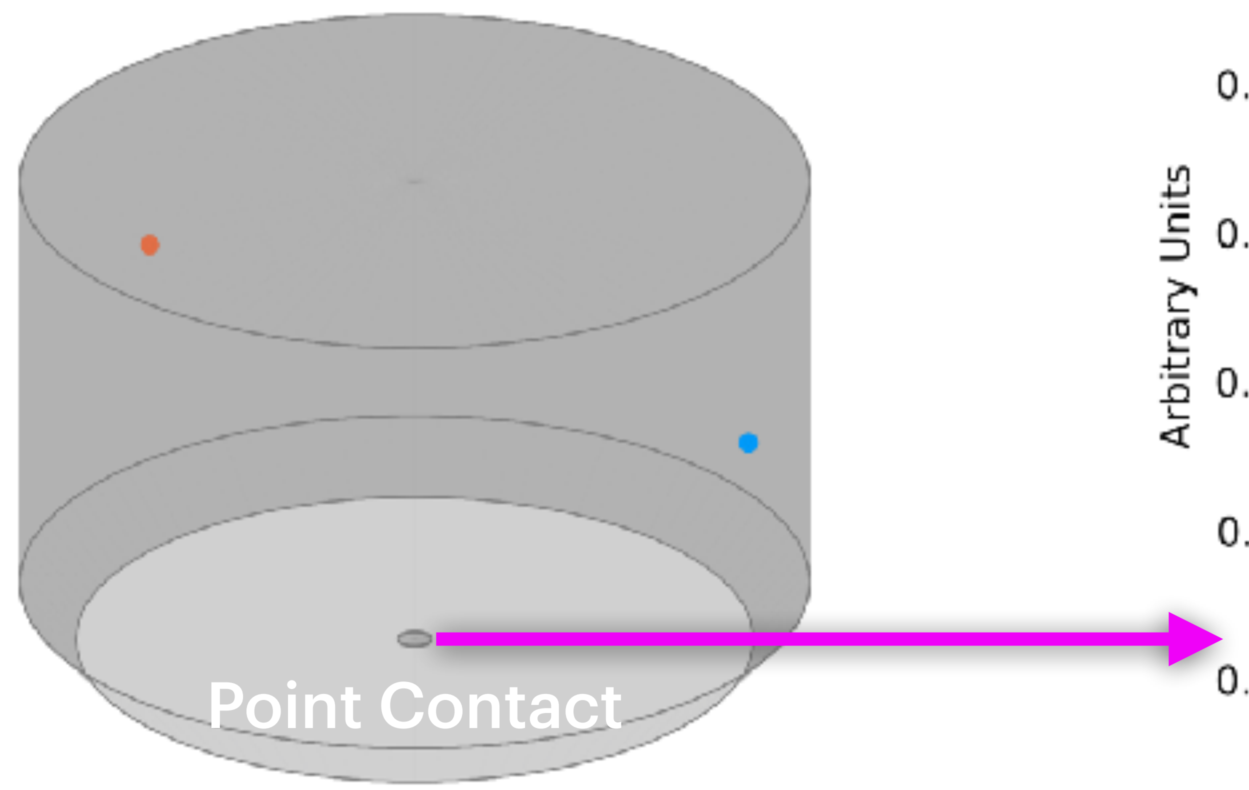


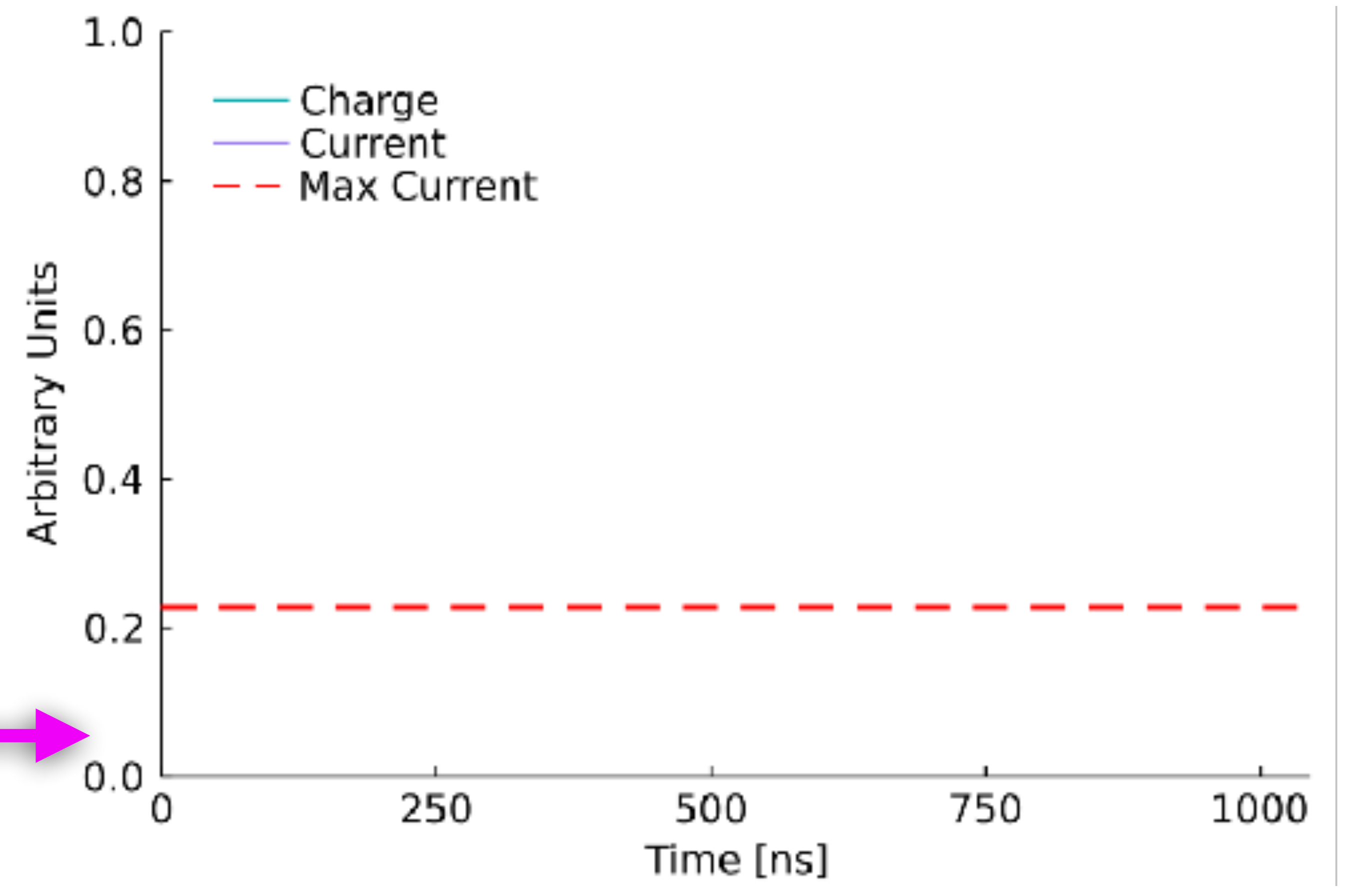




Single-Site Waveform





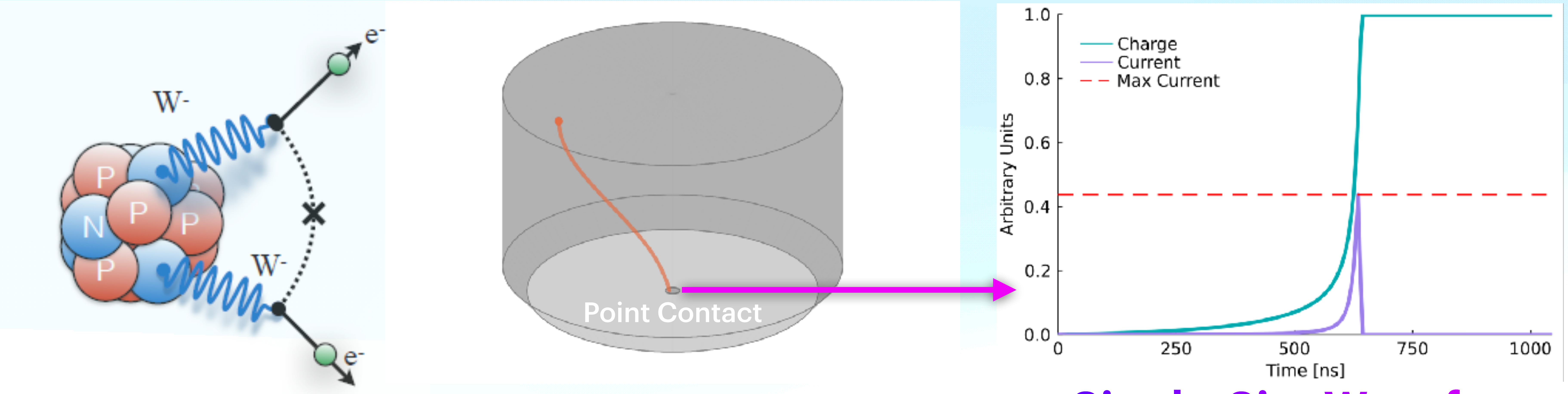


 Multi-Site Waveform

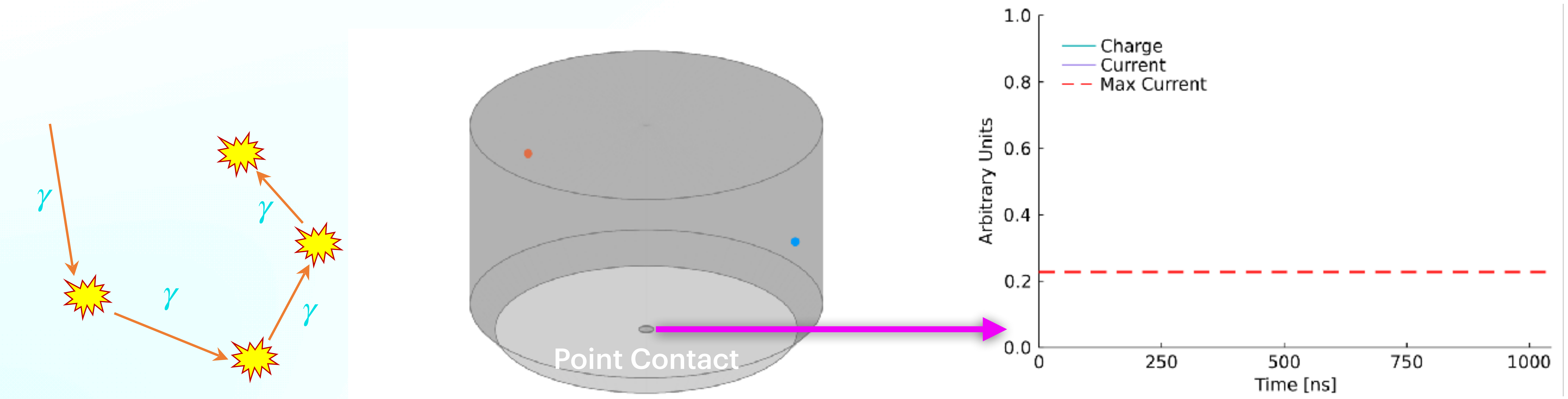
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\upsilon\beta\beta$ Search



Single-Site Waveform

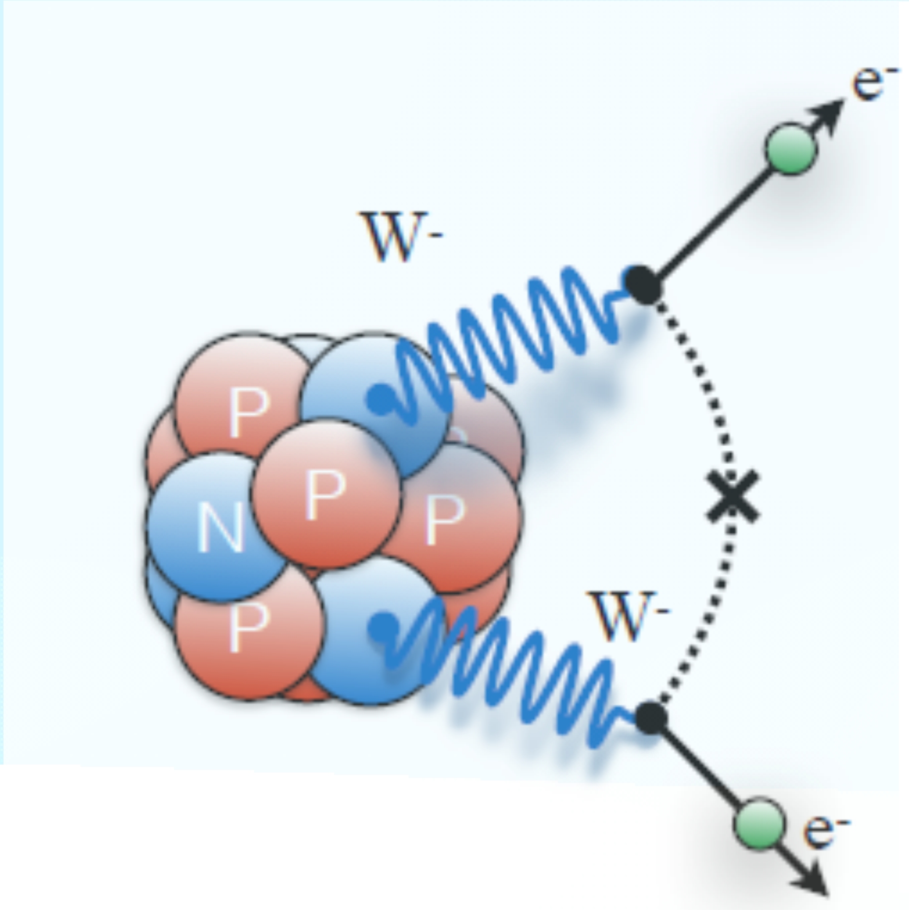


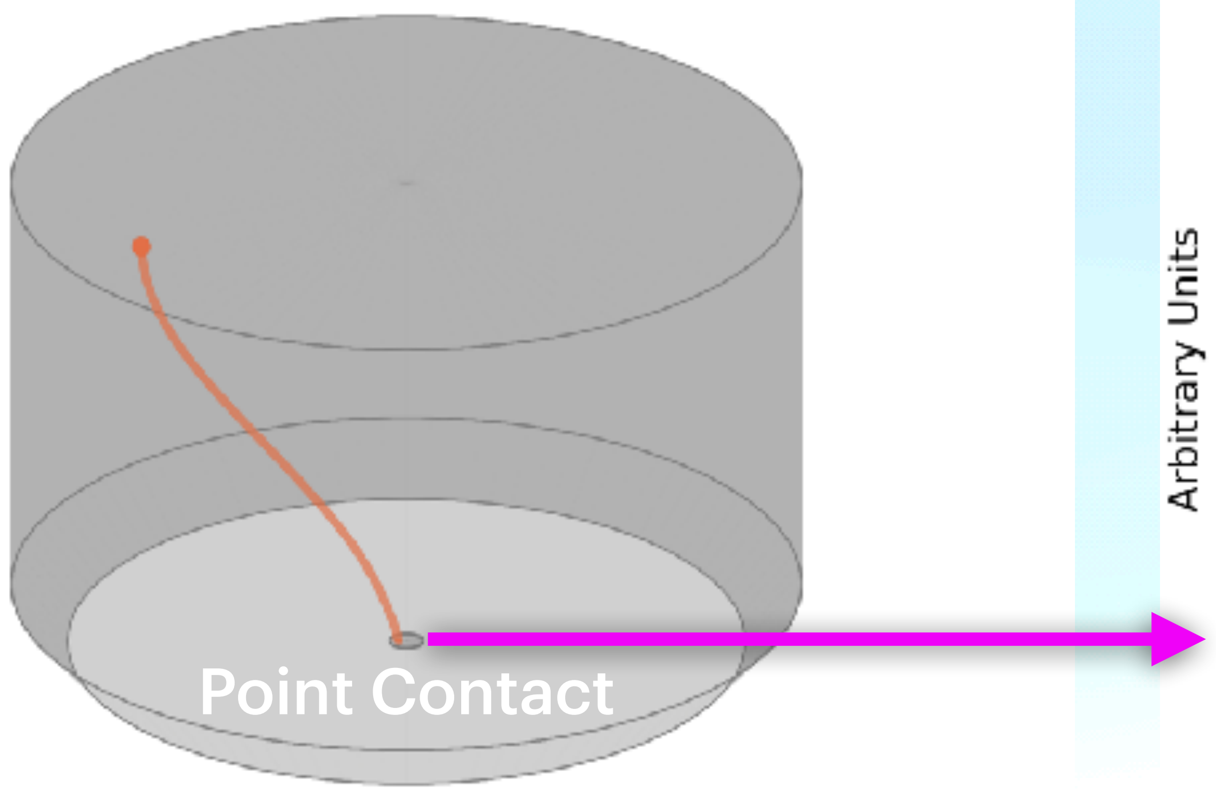
Multi-Site Waveform

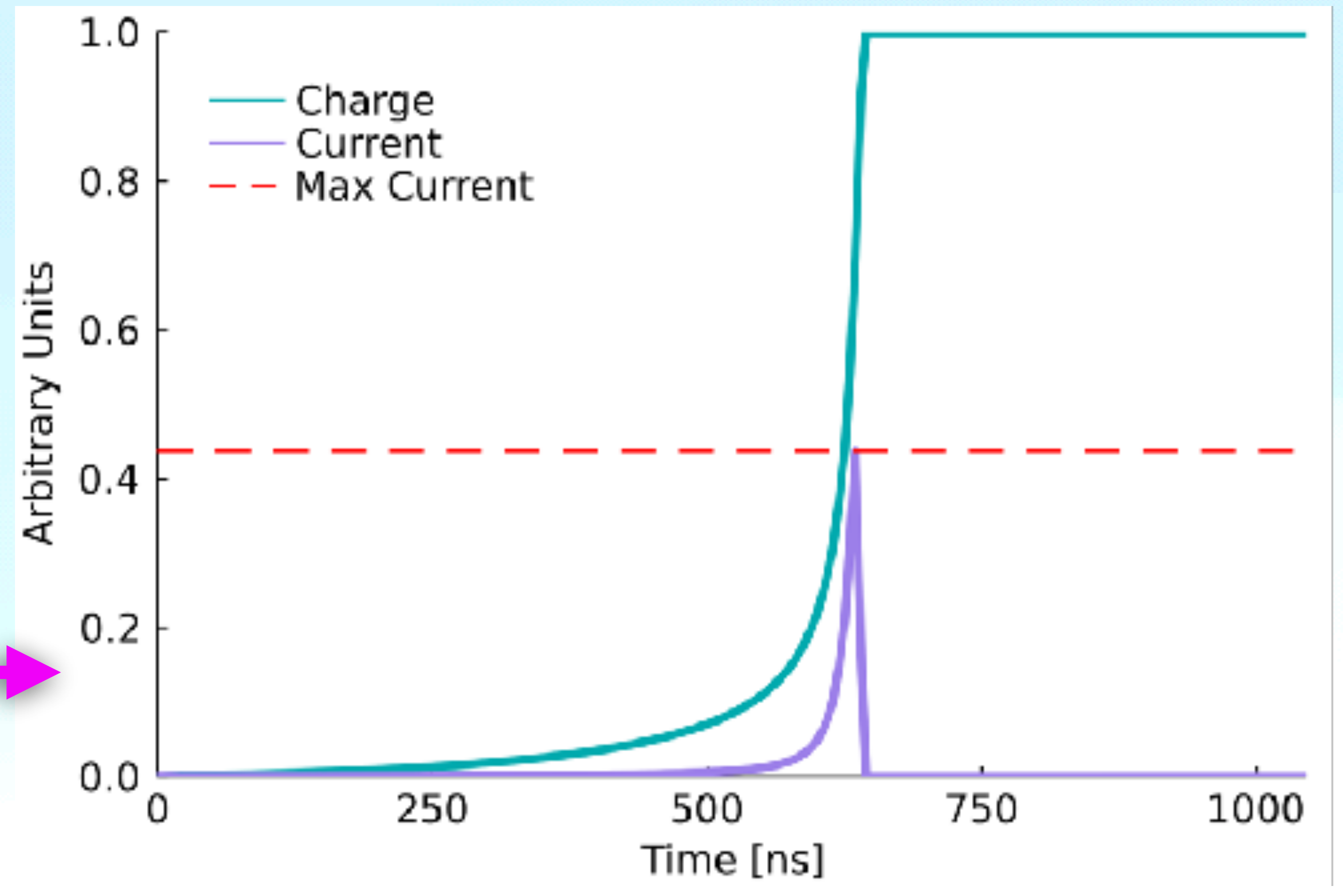
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

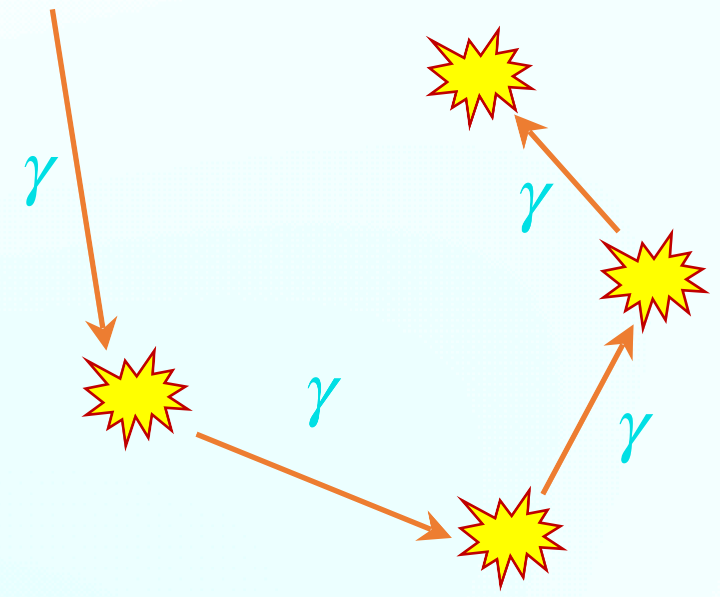
MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for 0υββ Search

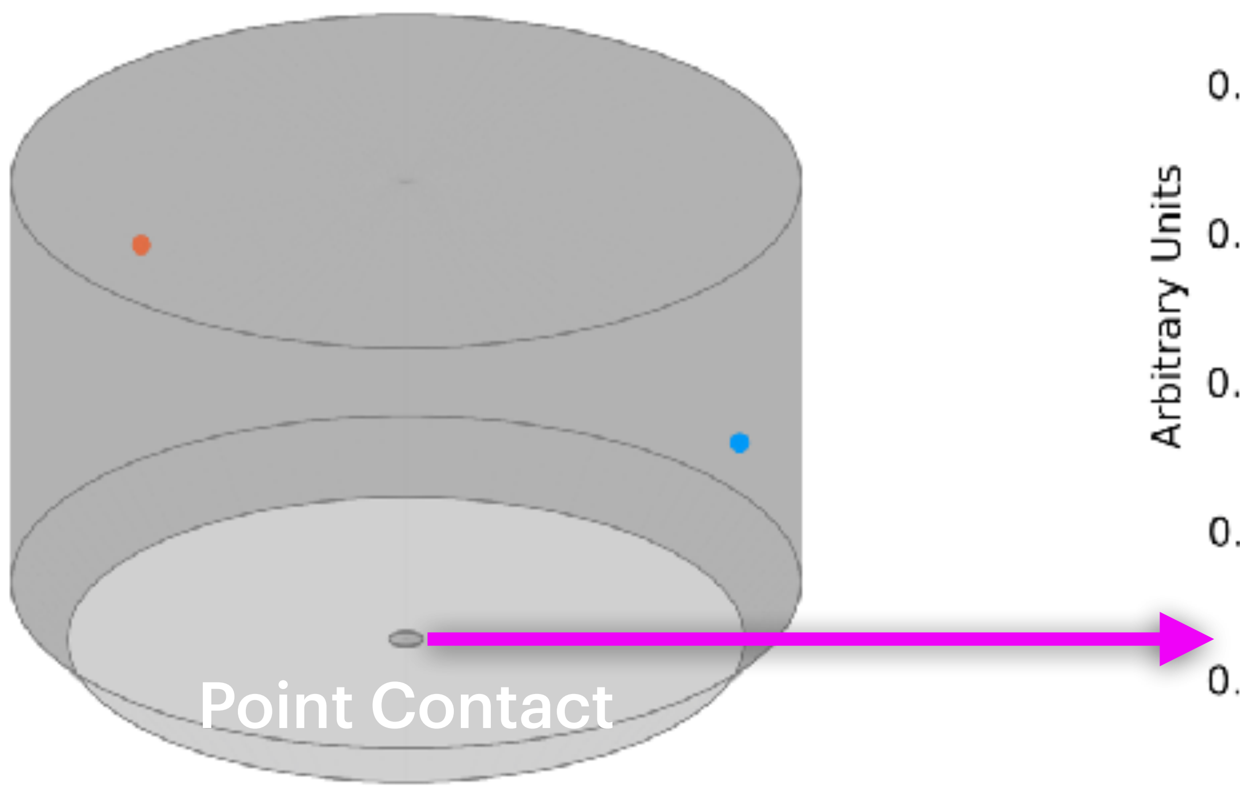


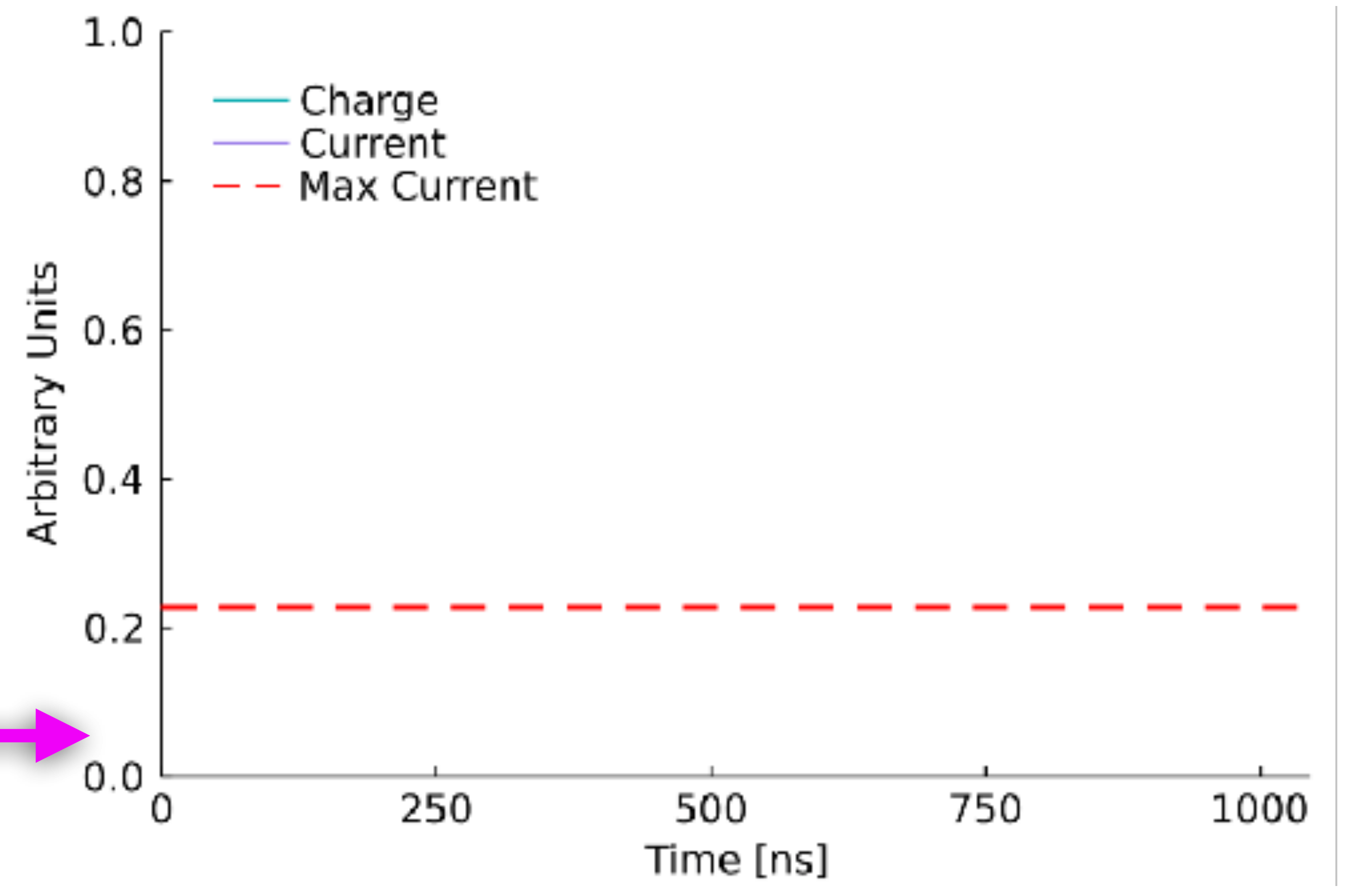




Single-Site Waveform





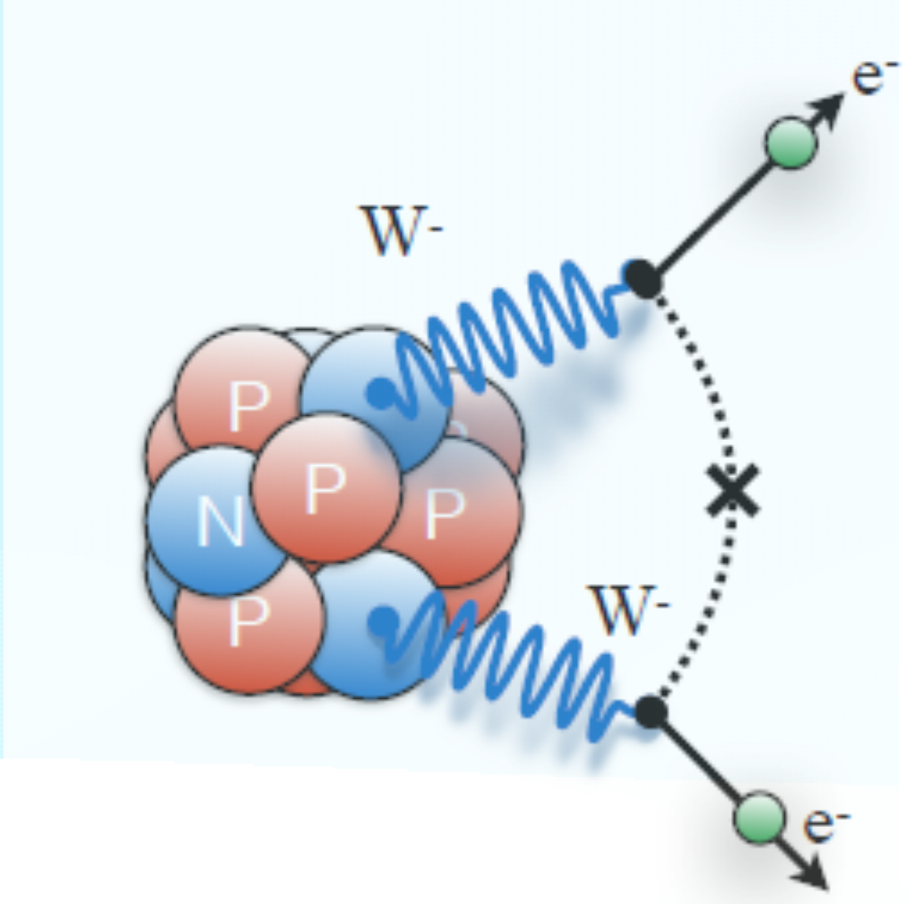


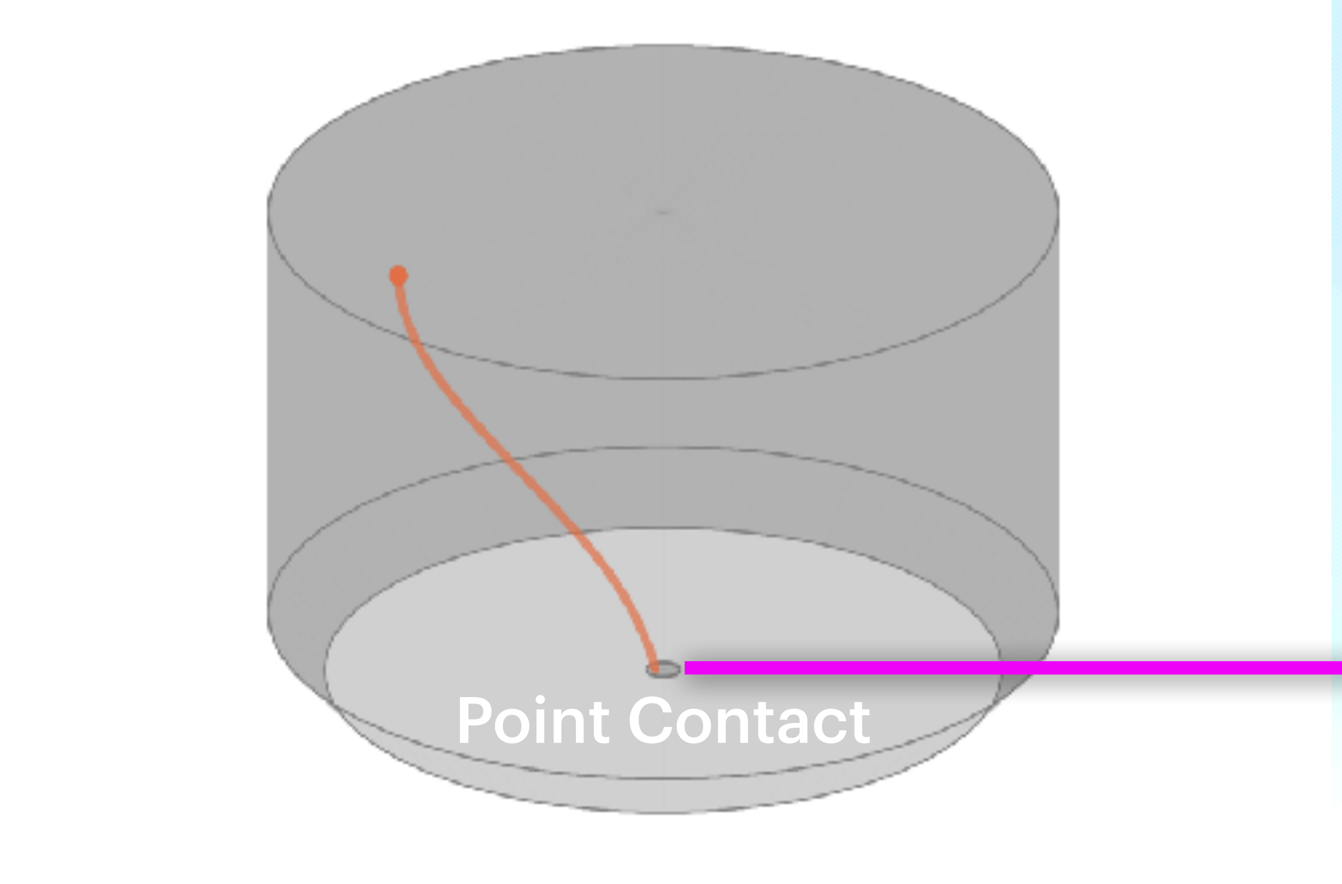
 Multi-Site Waveform

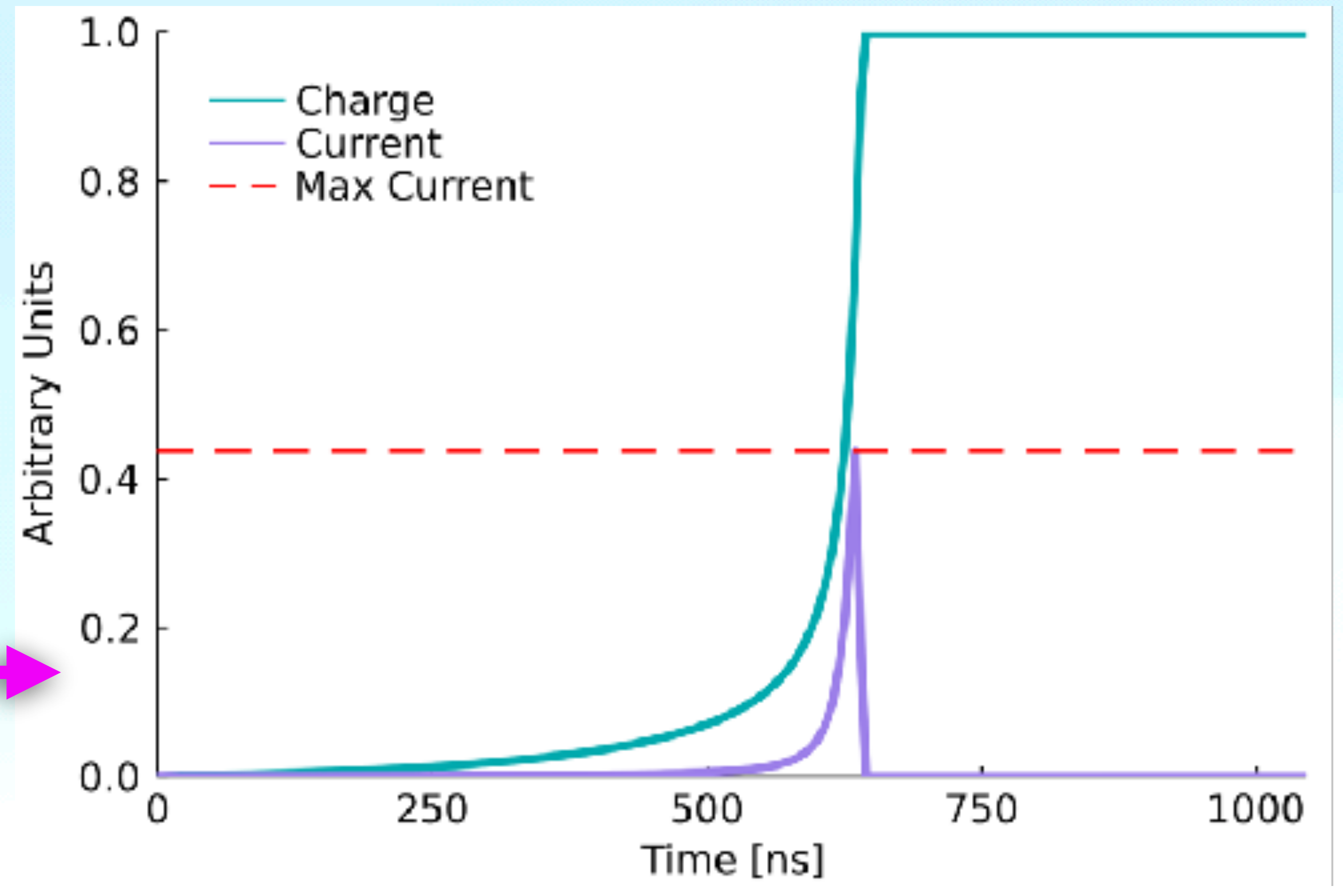
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

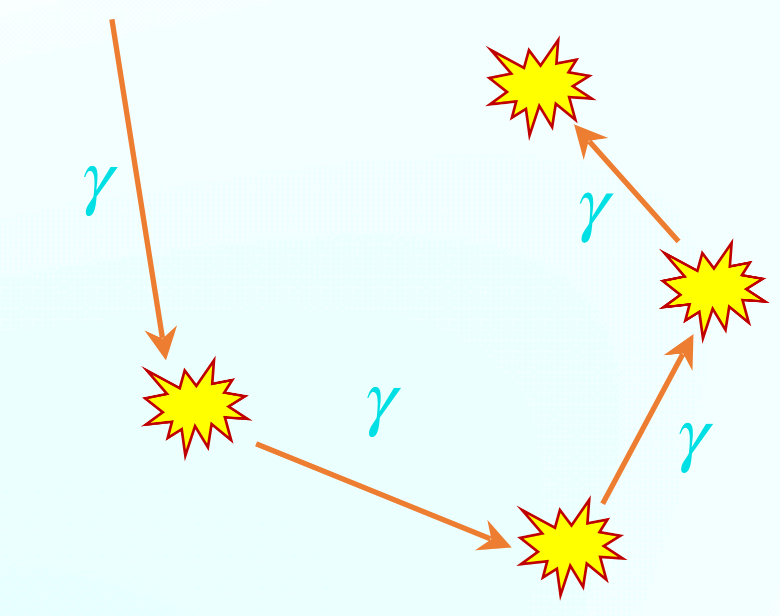
MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\nu\beta\beta$ Search

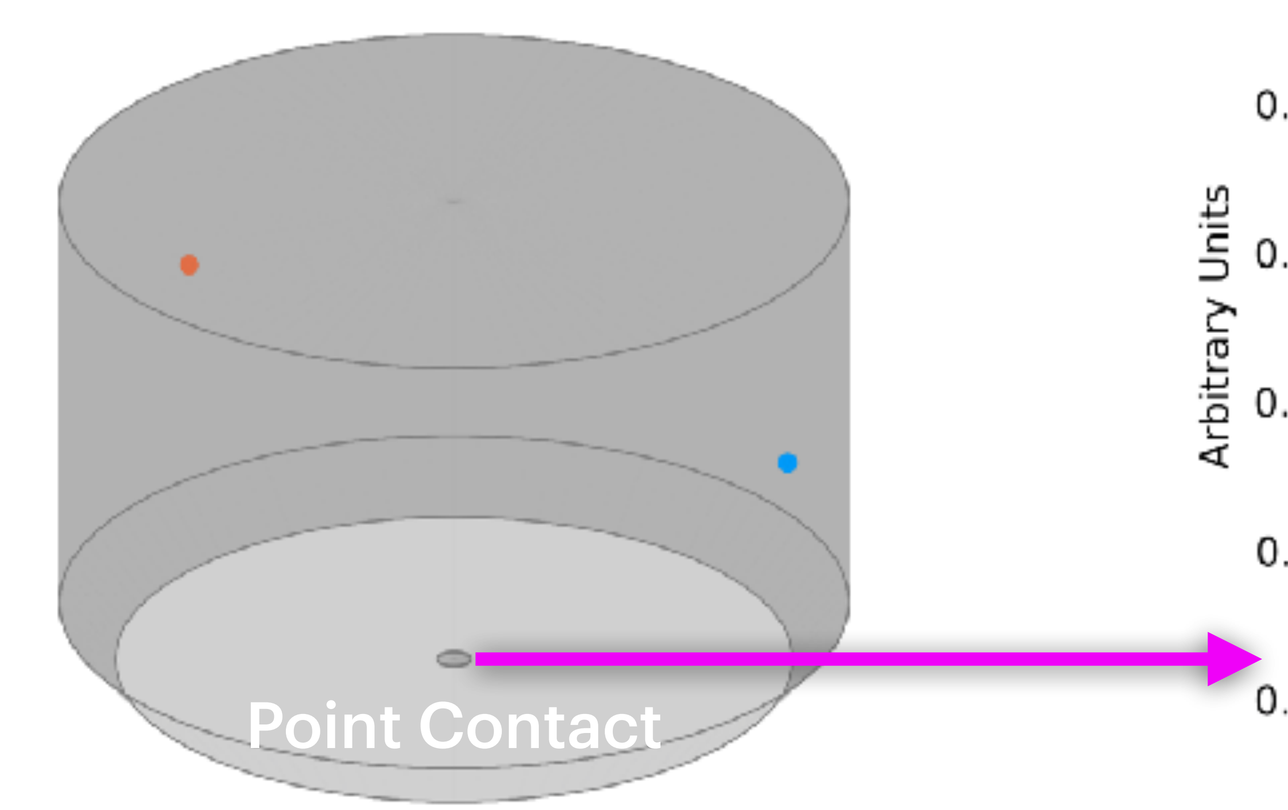


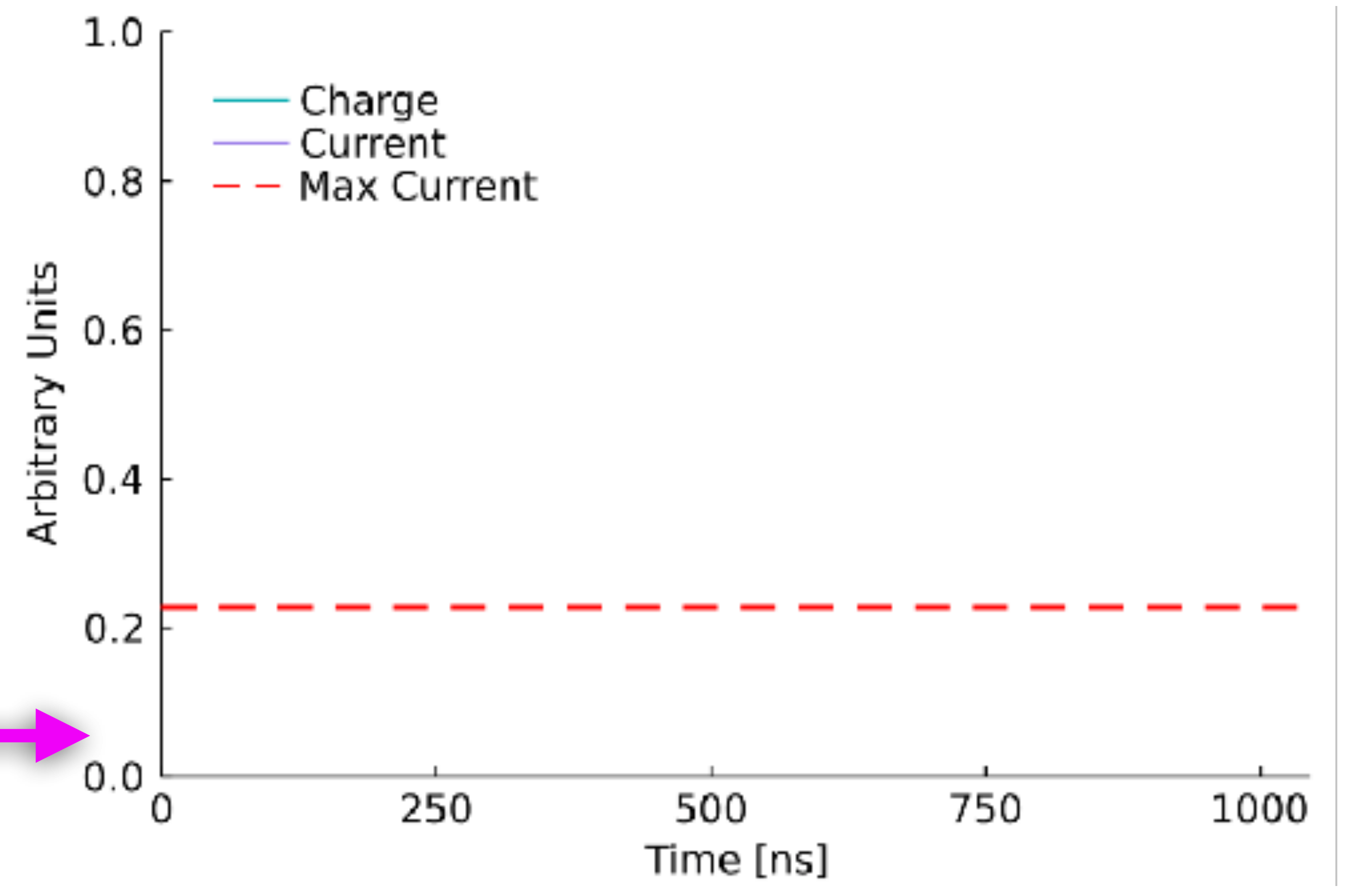




Single-Site Waveform





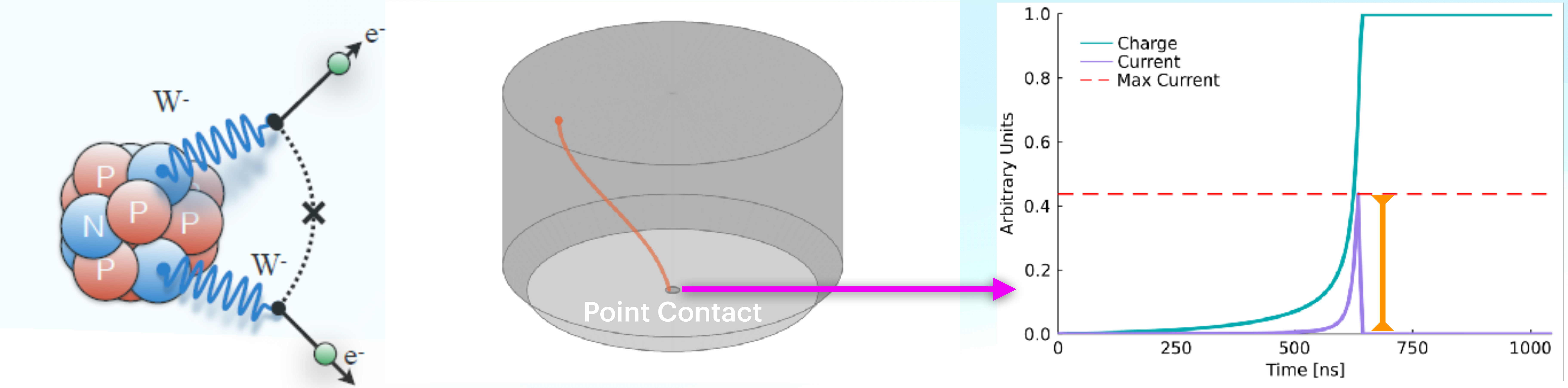


 Multi-Site Waveform

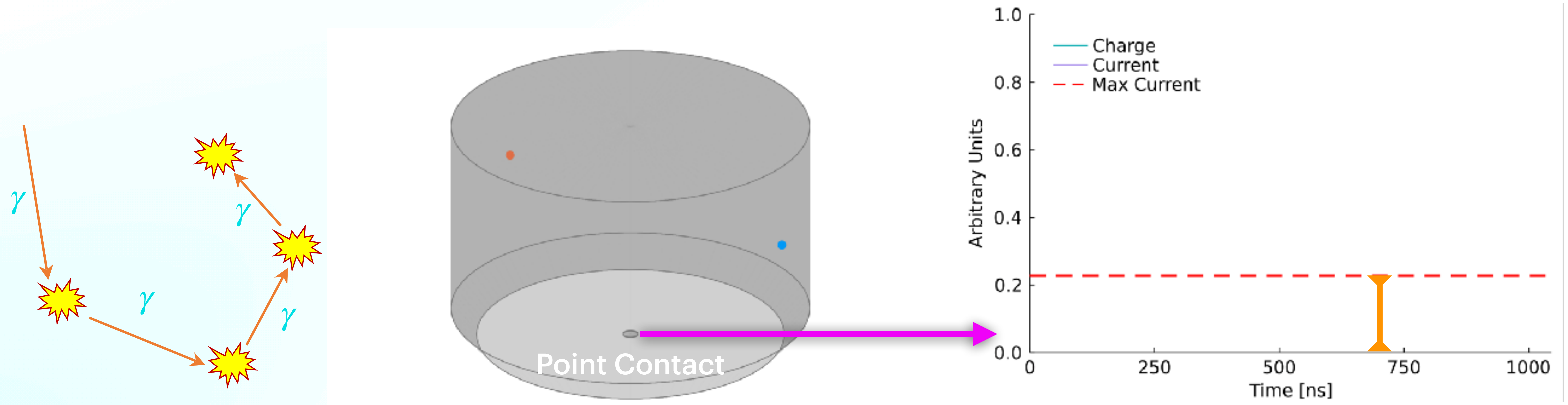
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for 0υββ Search



Single-Site Waveform



AvsE

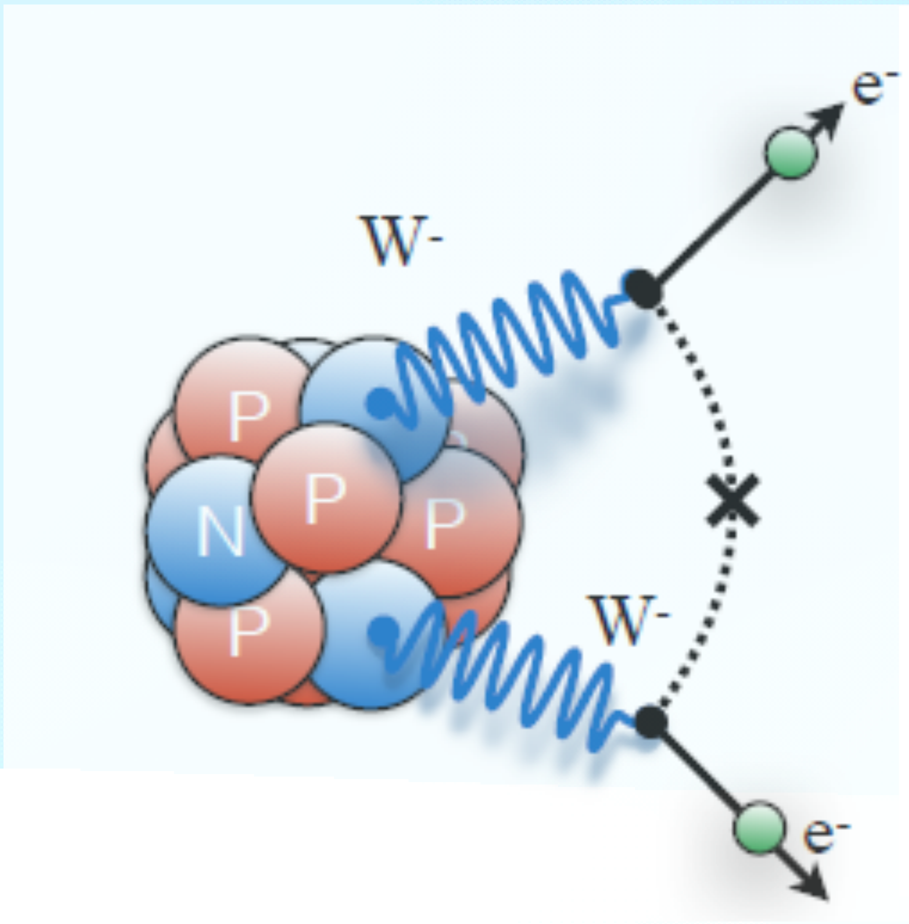
For **multi-site background** rejection

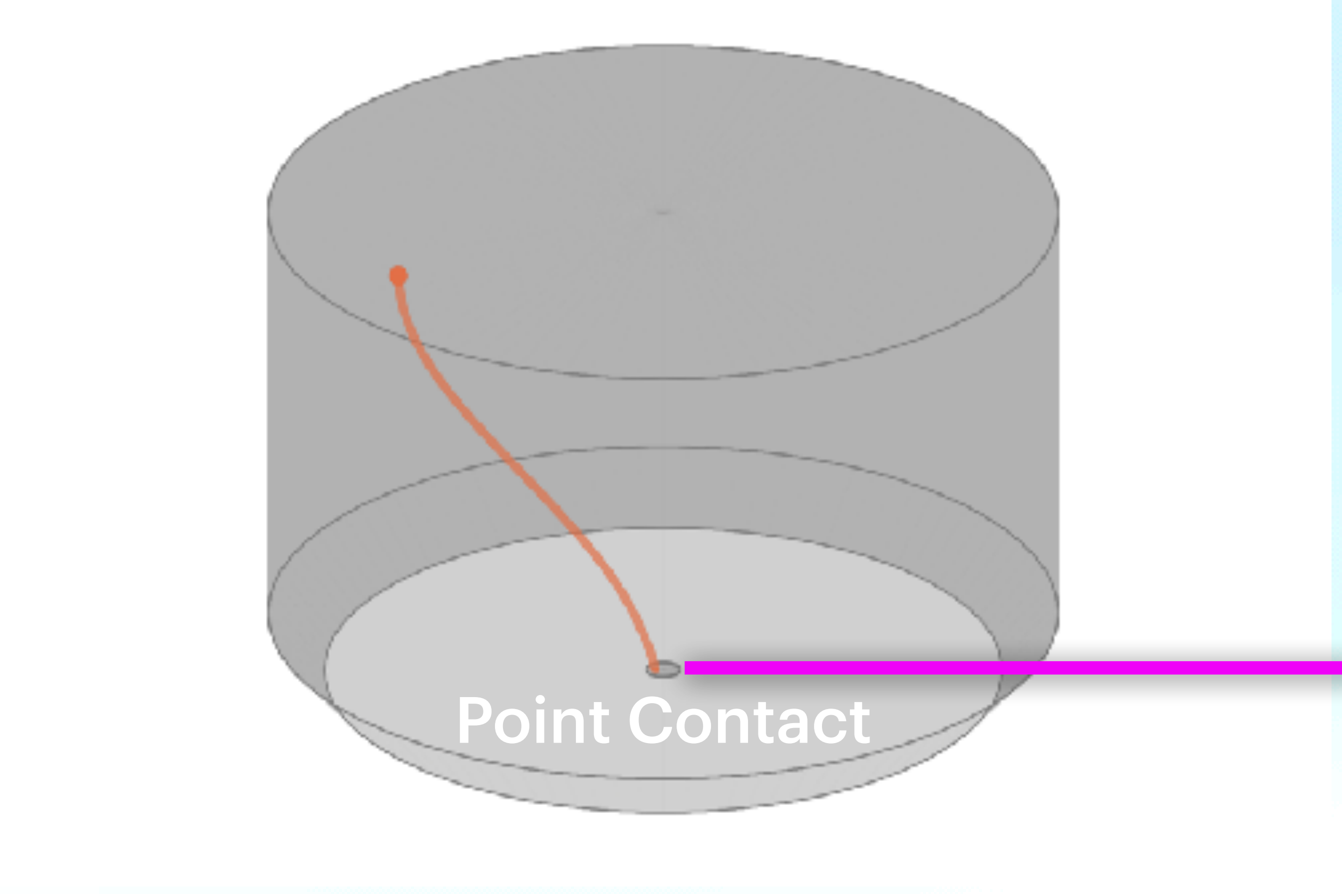
 Multi-Site Waveform

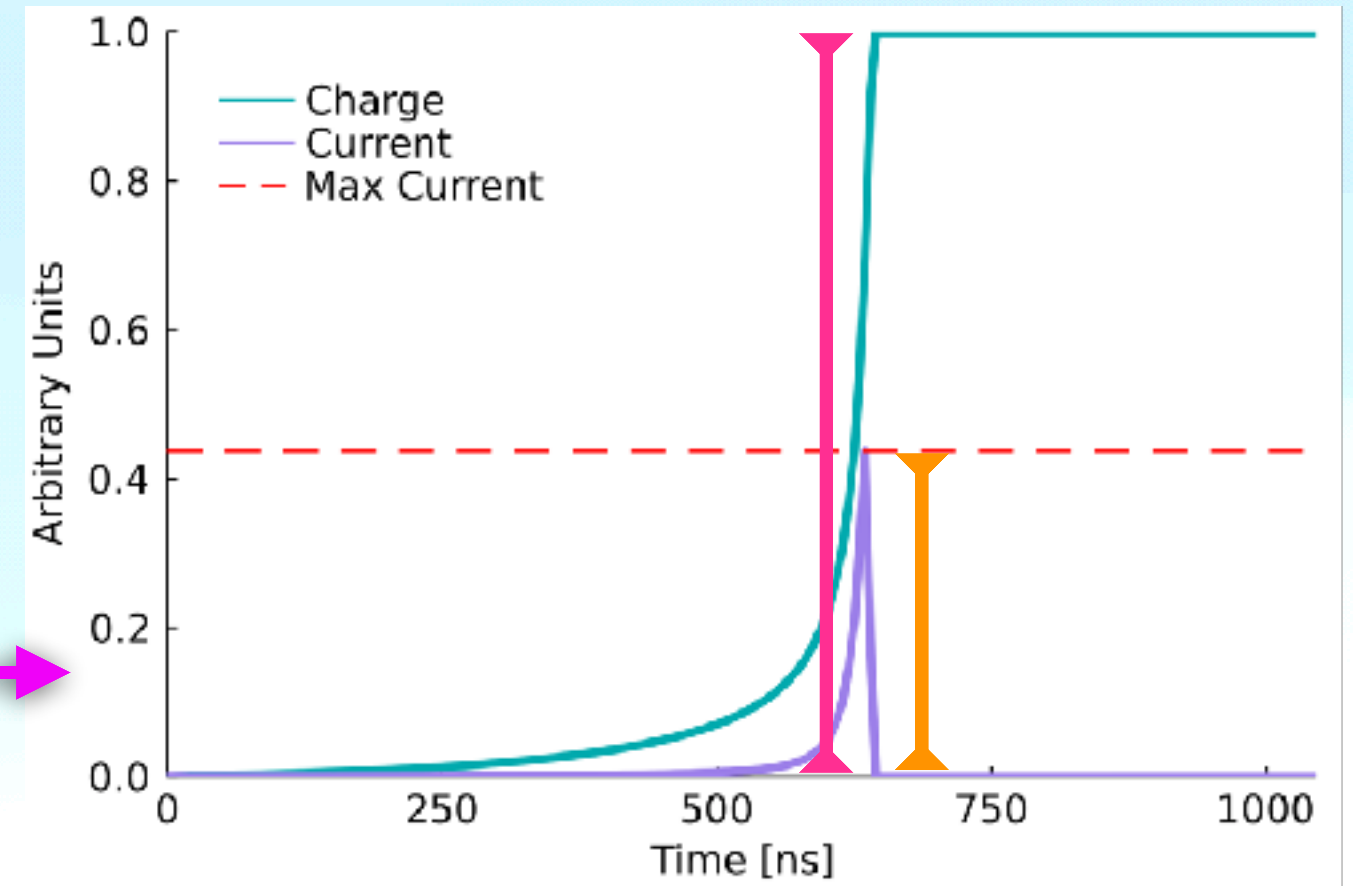
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\upsilon\beta\beta$ Search

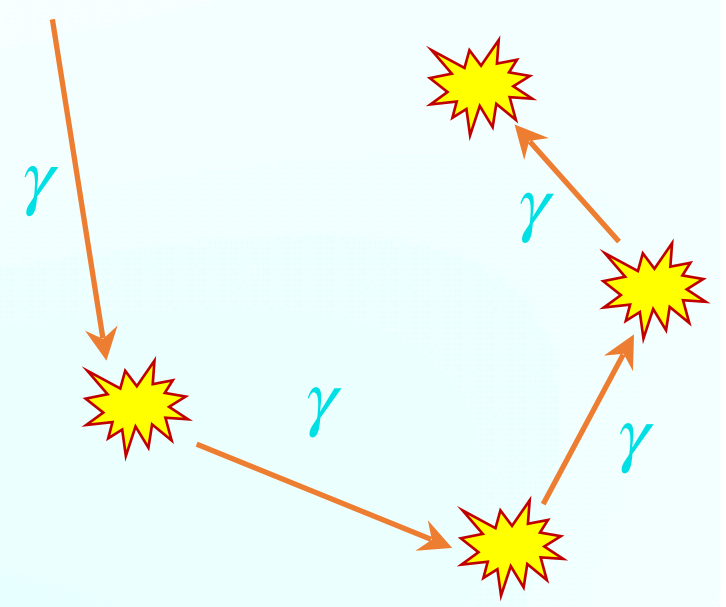


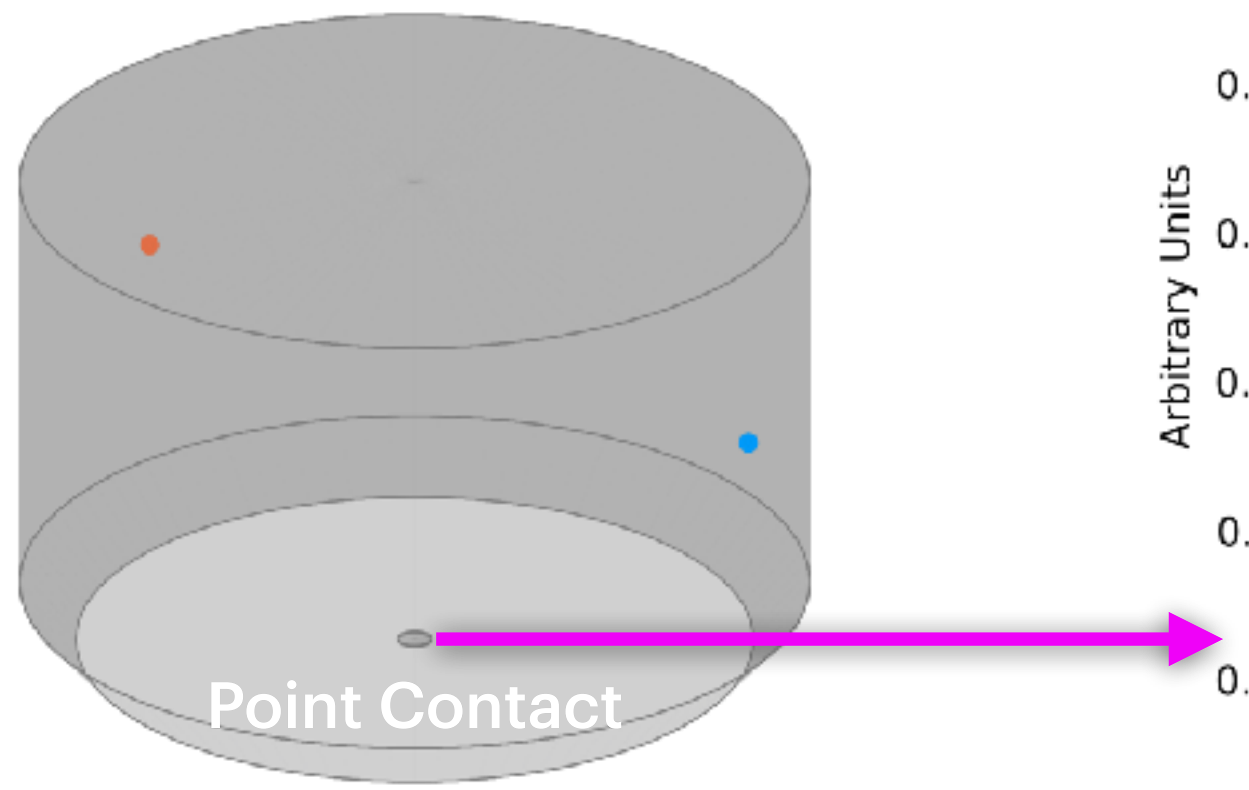


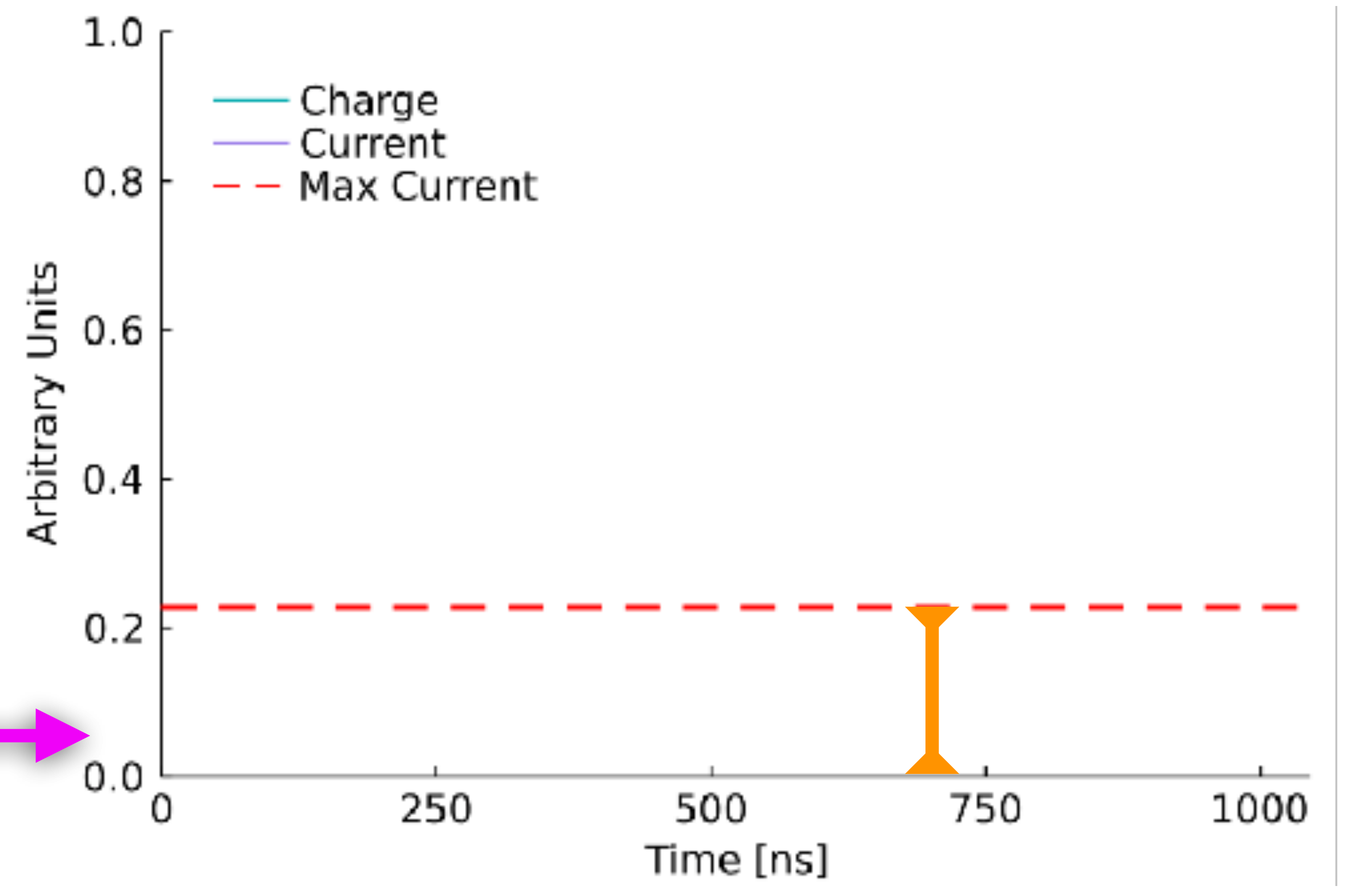


Single-Site Waveform

Energy







AvsE

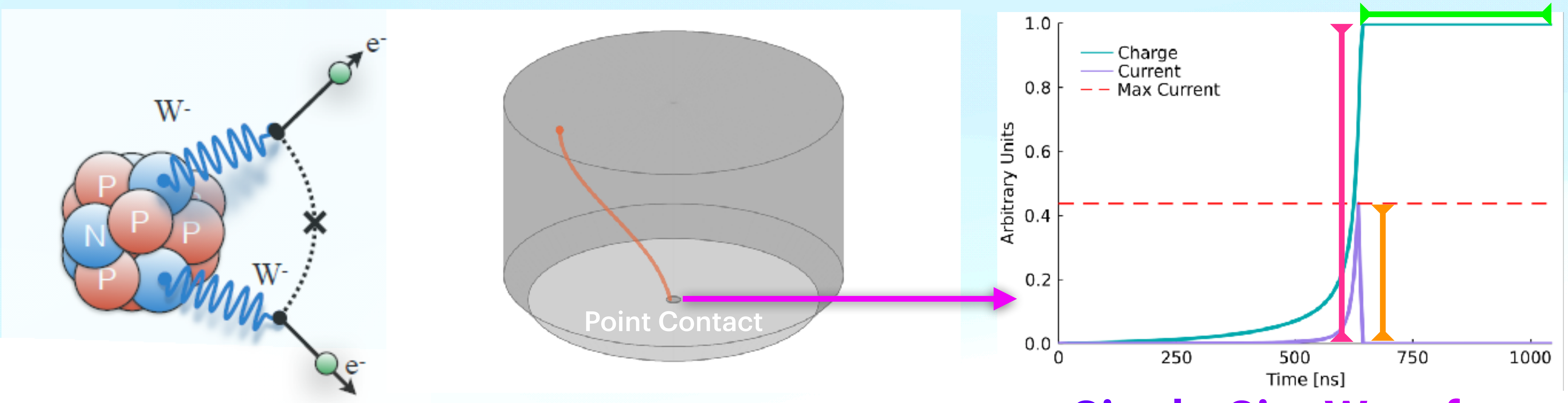
For **multi-site background** rejection

Multi-Site Waveform

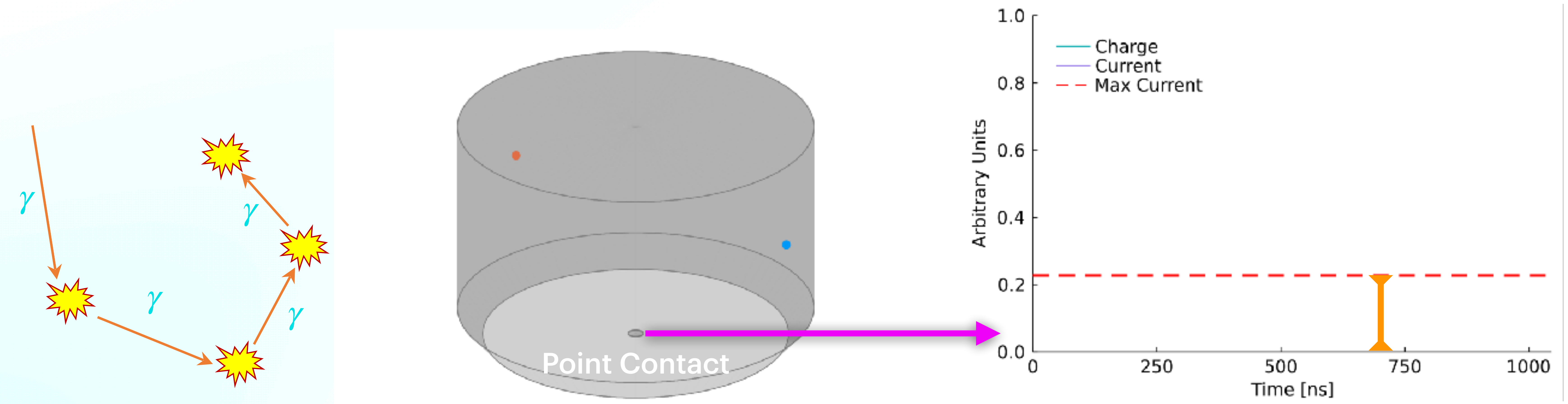
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\upsilon\beta\beta$ Search



Single-Site Waveform



 Multi-Site Waveform

Tail Slope

For **surface background** rejection

Energy

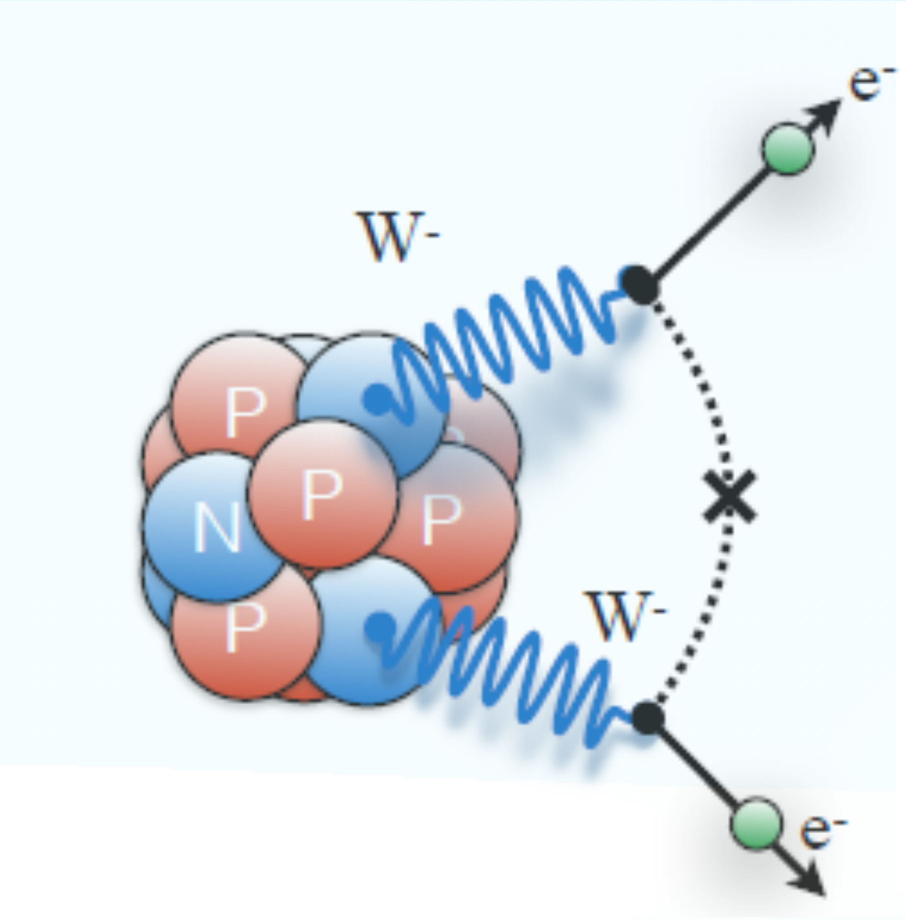
AvsE

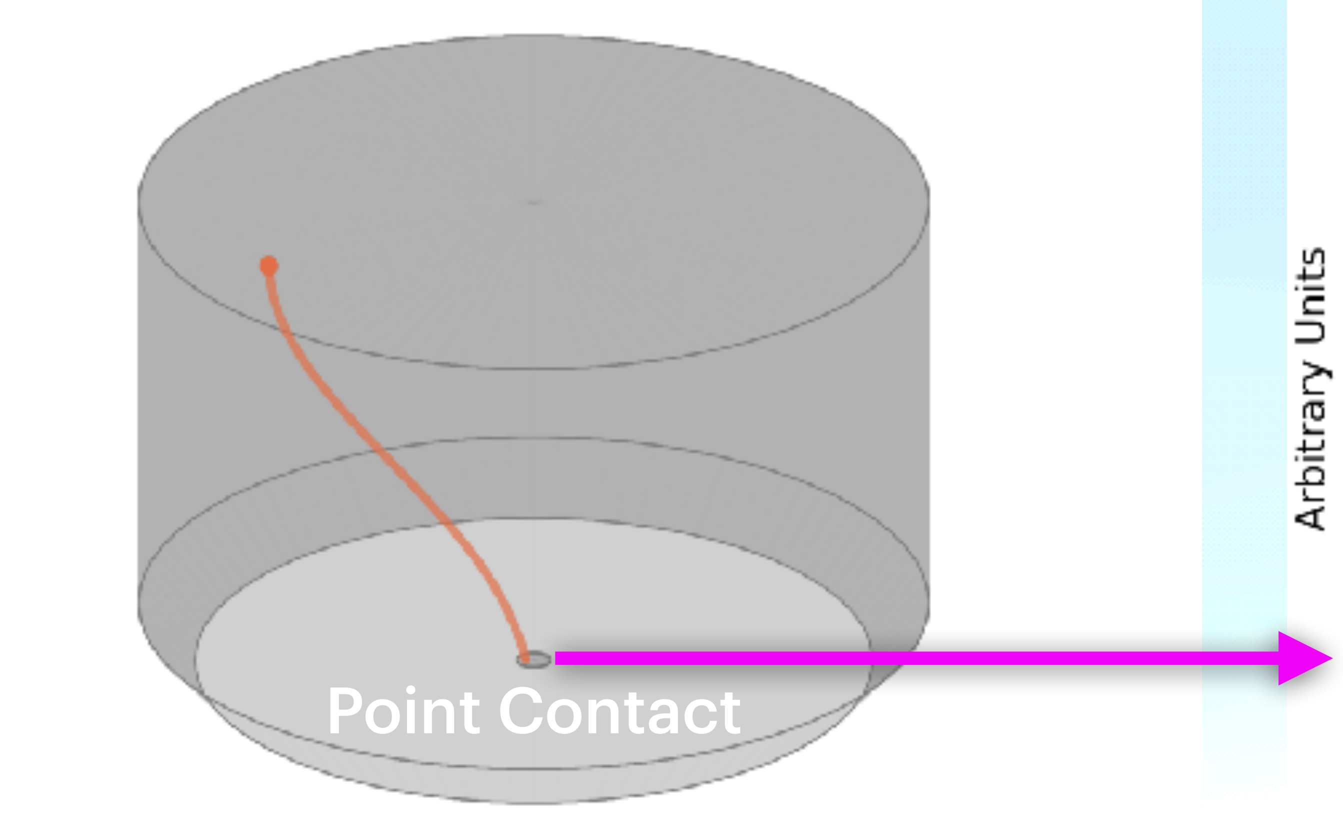
For **multi-site background** rejection

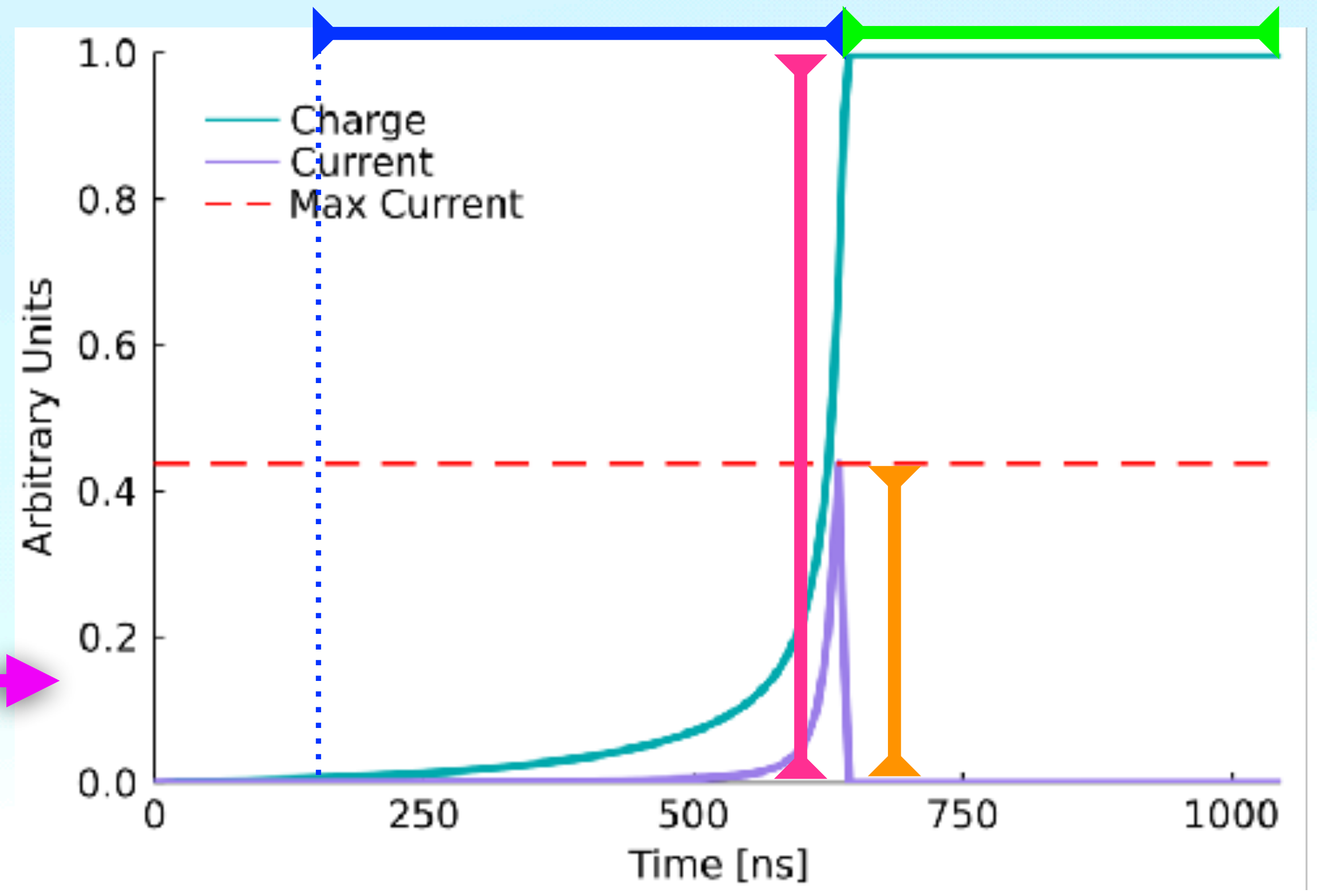
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

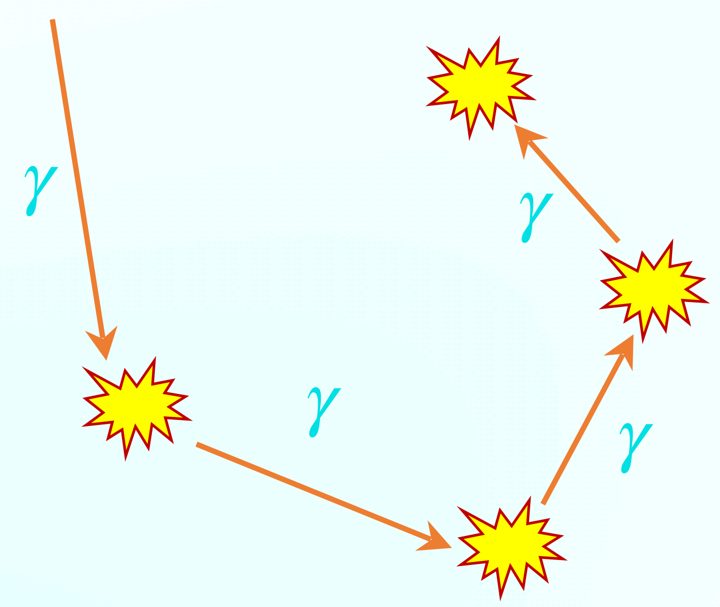
MAJORANA DEMONSTRATOR: HPGe Detector Array Experiment for $0\nu\beta\beta$ Search

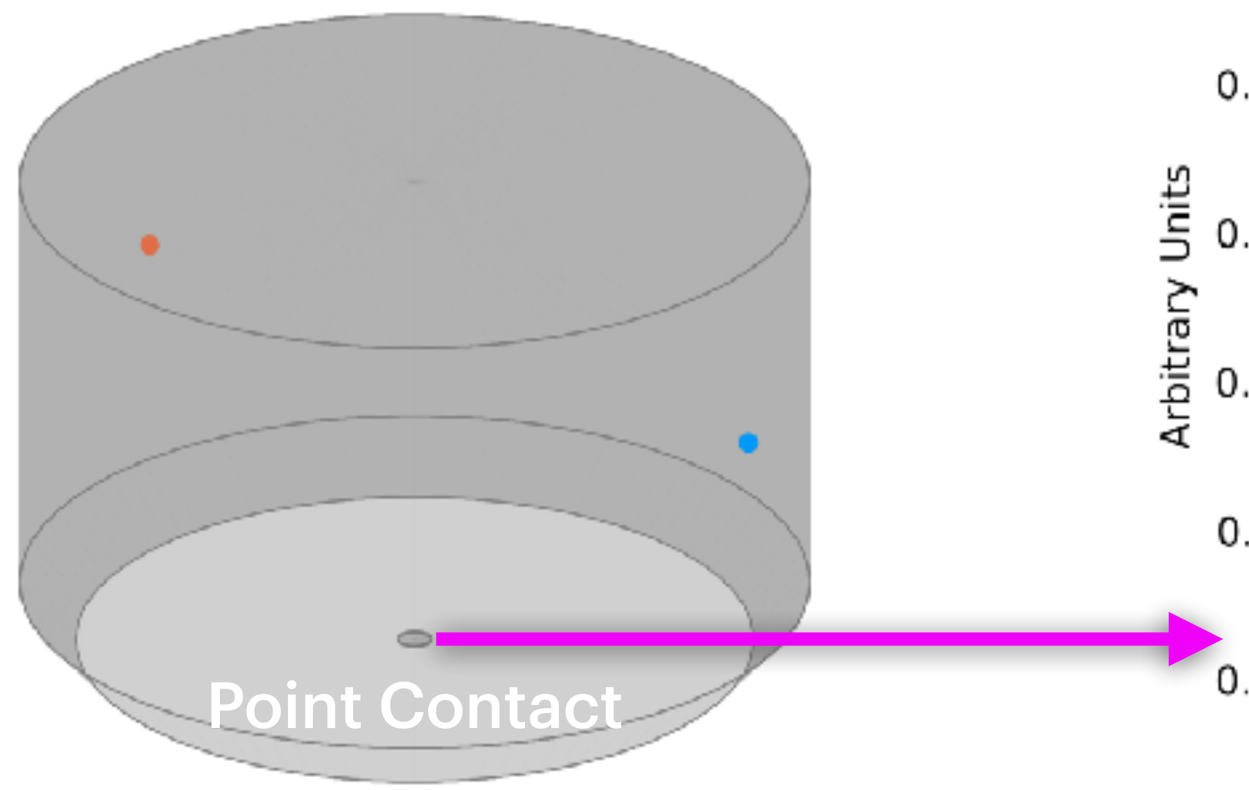


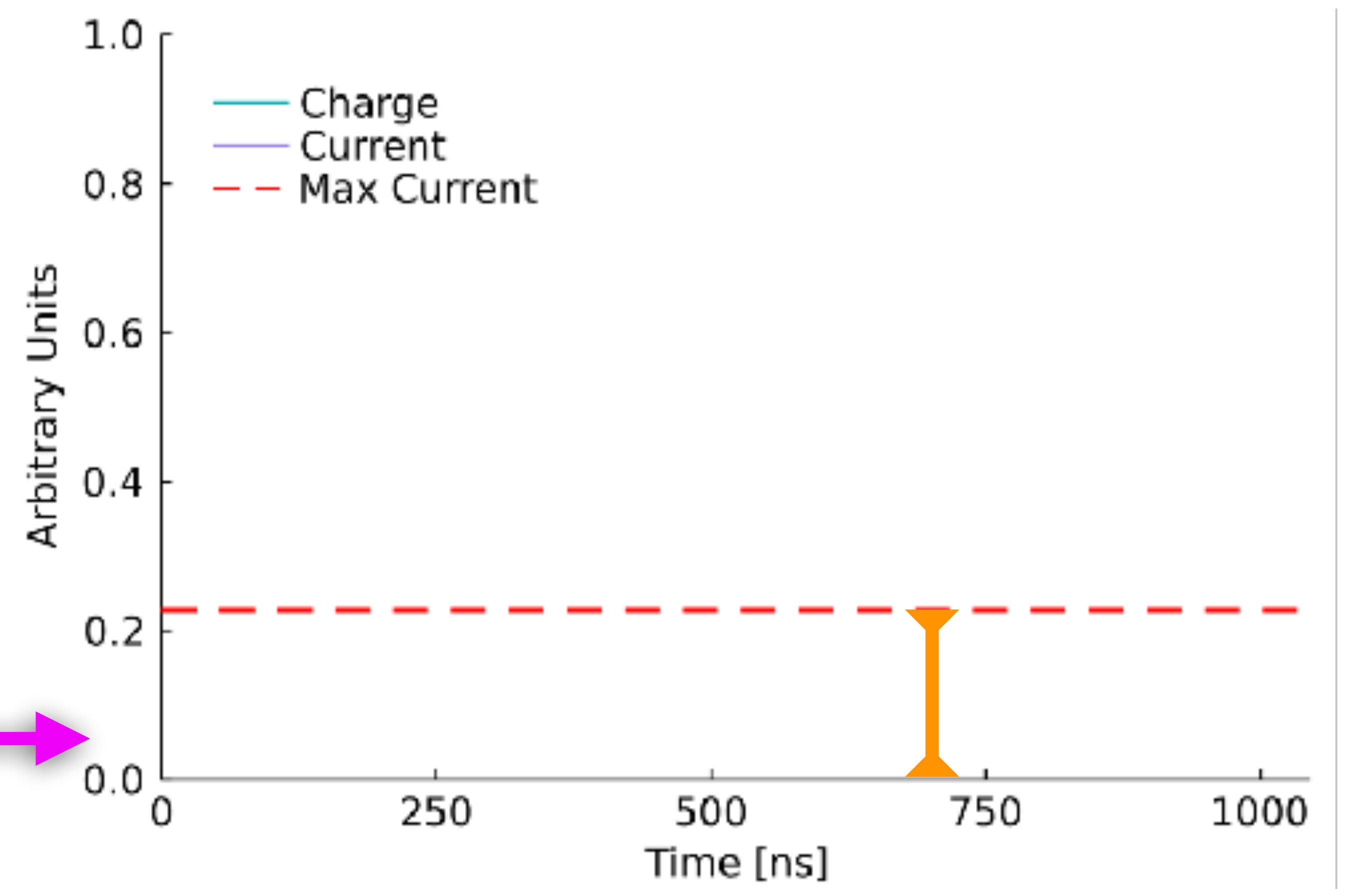




Single-Site Waveform







Multi-Site Waveform

Tail Slope

For **surface background** rejection

Energy

AvsE

For **multi-site background** rejection

Drift Time

Reflect the location of incident particle

Dataset: https://zenodo.org/records/8257027
Document: https://arxiv.org/abs/2308.10856

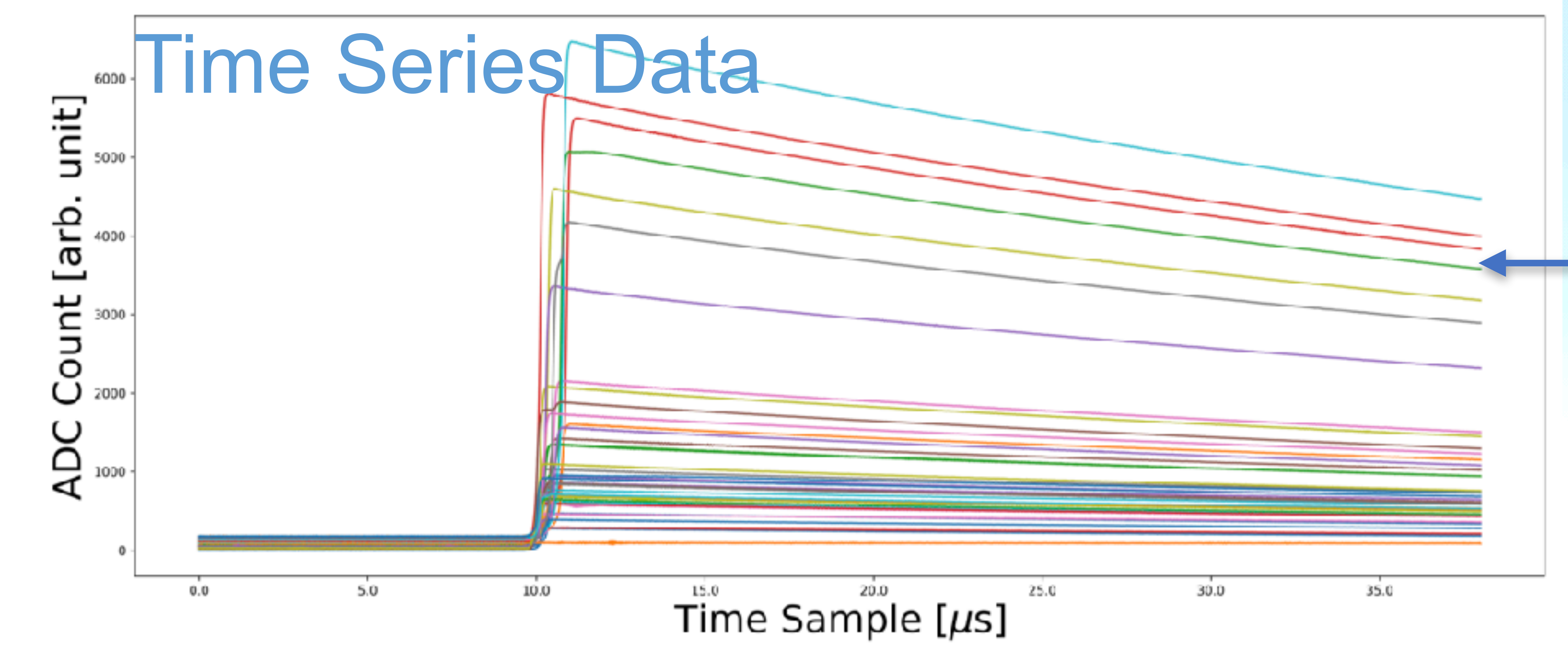
This is real data from a state-of-the-art physics experiment, feel free to play with it to do your own machine learning project!

Field	Description	Data Type
raw_waveform	Detector Waveform	array(size=(3800,) dtype=float)
energy_label	Analysis Label	float
psd_label_low_avse	Analysis Label	binary
psd_label_high_avse	Analysis Label	binary
psd_label_dcr	Analysis Label	binary
psd_label_lq	Analysis Label	binary
tp0	Analysis Parameter	integer
detector	Metadata	integer
run_number	Metadata	integer
id	Metadata	integer

Energy
AvsE
Tail Slope

More on MJD: https://www.energy.gov/science/np/articles/majorana-demonstrator-gives-its-final-answer-about-rare-nuclear-decay

Dataset: https://zenodo.org/records/8257027
Document: https://arxiv.org/abs/2308.10856



This is real data from a state-of-the-art physics experiment, feel free to play with it to do your own machine learning project!

Field	Description	Data Type
raw_waveform	Detector Waveform	array(size=(3800,) dtype=float)
energy_label	Analysis Label	float
psd_label_low_avse	Analysis Label	binary
psd_label_high_avse	Analysis Label	binary
psd_label_dcr	Analysis Label	binary
psd_label_lq	Analysis Label	binary
tp0	Analysis Parameter	integer
detector	Metadata	integer
run_number	Metadata	integer
id	Metadata	integer

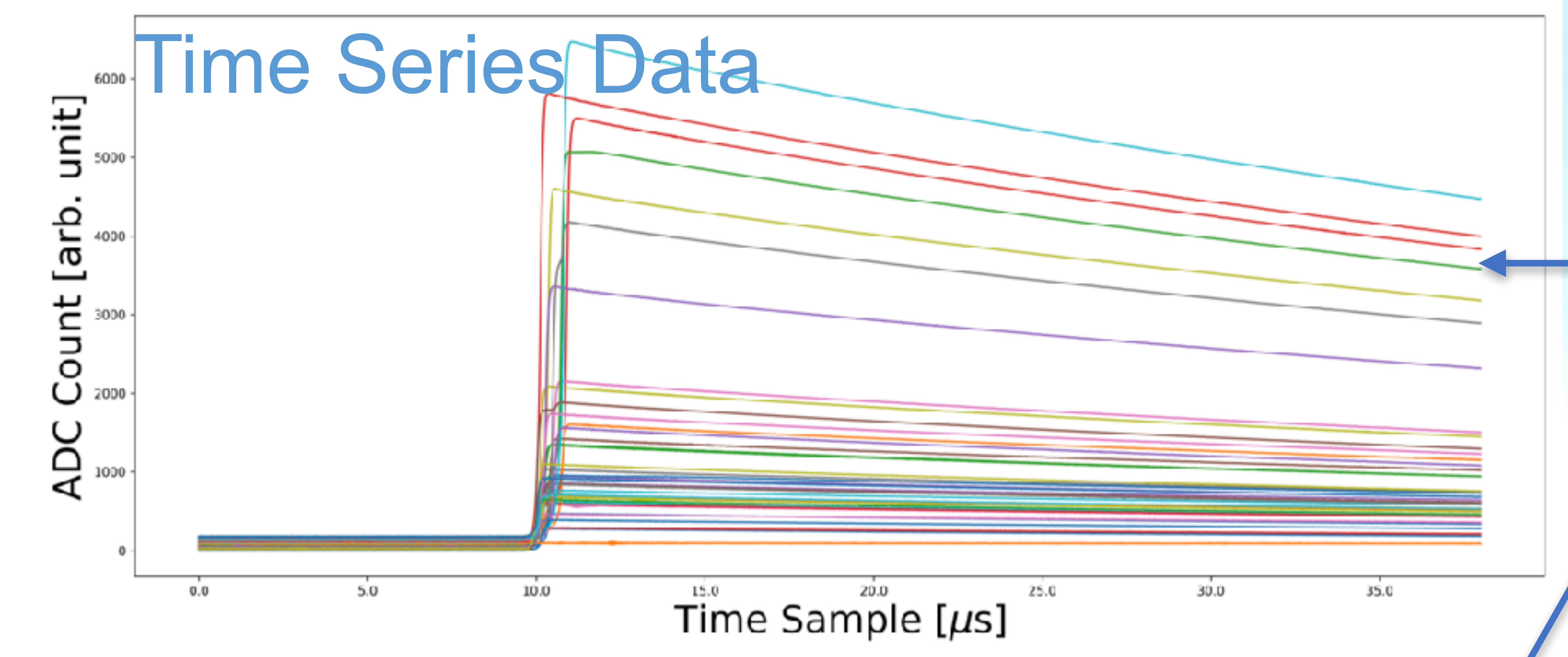
Energy
AvsE
Tail Slope

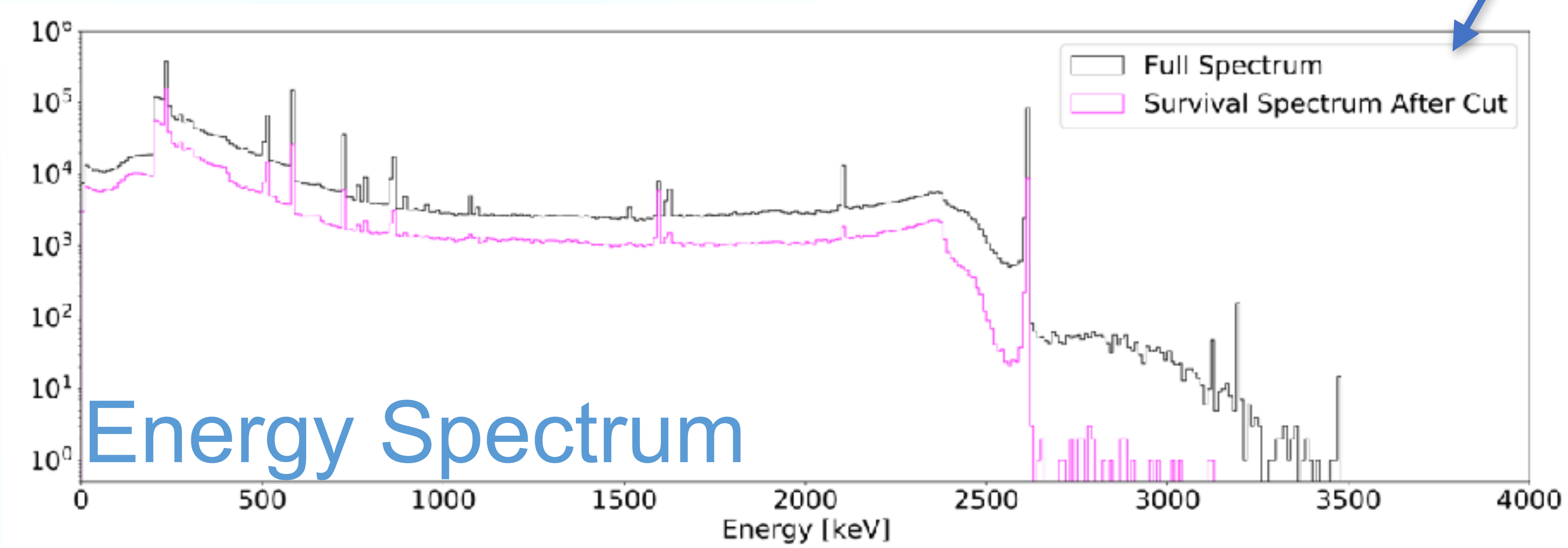
More on MJD: https://www.energy.gov/science/np/articles/majorana-demonstrator-gives-its-final-answer-about-rare-nuclear-decay

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Dataset: https://zenodo.org/records/8257027
Document: https://arxiv.org/abs/2308.10856





This is real data from a state-of-the-art physics experiment, feel free to play with it to do your own machine learning project!

Field	Description	Data Type
raw_waveform	Detector Waveform	array(size=(3800,) dtype=float)
energy_label	Analysis Label	float
psd_label_low_avse	Analysis Label	binary
psd_label_high_avse	Analysis Label	binary
psd_label_dcr	Analysis Label	binary
psd_label_lq	Analysis Label	binary
tp0	Analysis Parameter	integer
detector	Metadata	integer
run_number	Metadata	integer
id	Metadata	integer

Energy

AvsE

Tail Slope

More on MJD: https://www.energy.gov/science/np/articles/majorana-demonstrator-gives-its-final-answer-about-rare-nuclear-decay

All files are stored in .hdf5 format

More readings on HDF5: https://web.mit.edu/fwtools_v3.1.0/www/H5.intro.html

```
MJD_Test_2.hdf5
├── detector (65000,)
├── energy_label (65000,)
├── id (65000,)
├── psd_label_dcr (65000,)
├── psd_label_high_avse (65000,)
├── psd_label_low_avse (65000,)
├── psd_label_lq (65000,)
├── raw_waveform (65000, 3800)
├── run_number (65000,)
└── tp0 (65000,)
```

Read file with h5py.File command

Code

Read “energy” field with indexing

Read “waveform” field with indexing

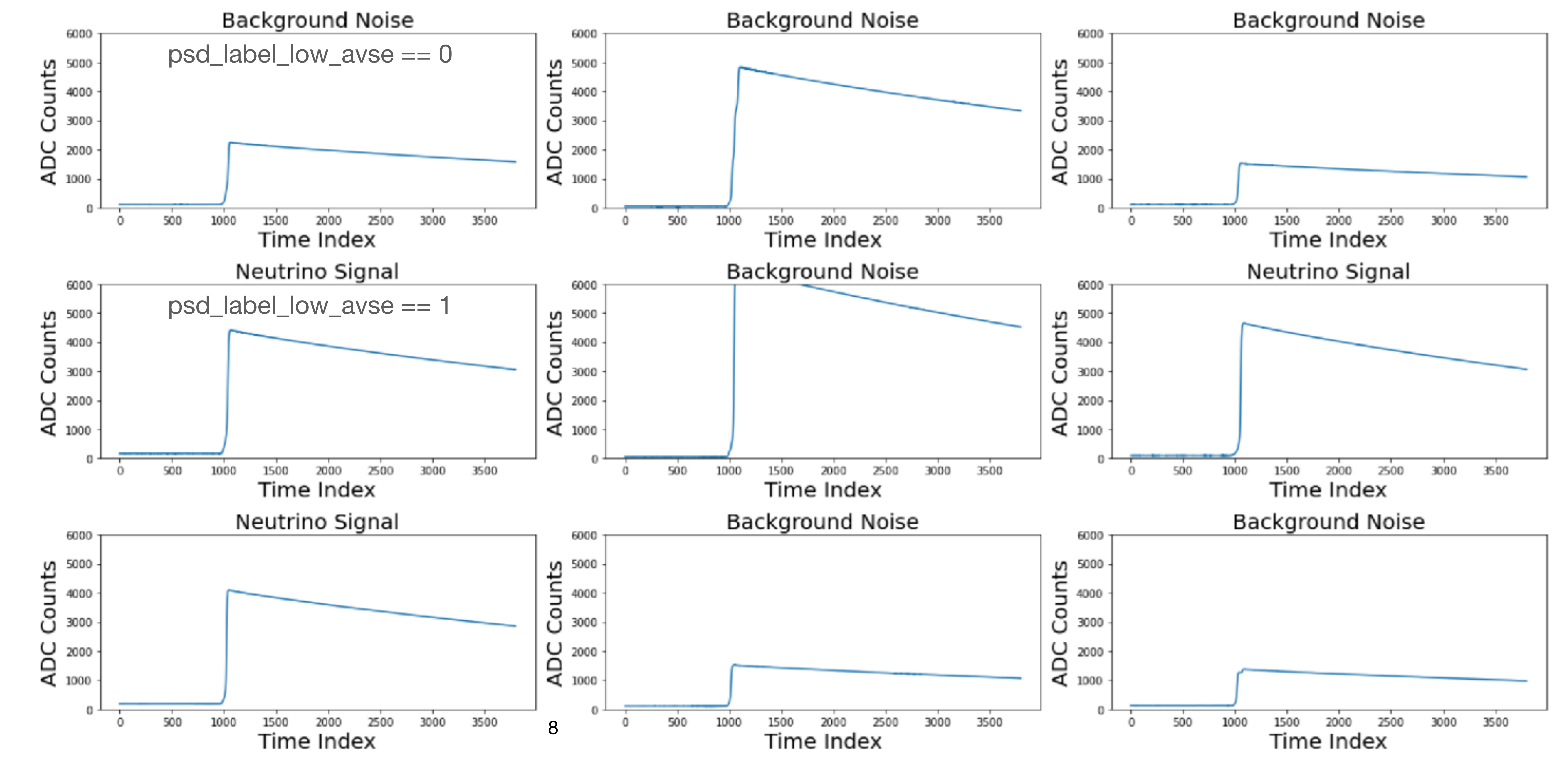
......

```python
with h5py.File(file_path, 'r') as file:
    energy = np.array(file["energy_label"])
    #only selecting higher energy event to make the task easier
    selection_flage = energy>self.energy_threshold
    randind = np.arange(selection_flage.sum())
    np.random.shuffle(randind)
    data = np.array(file["raw_waveform"])[selection_flage][randind][:,500:1500]
    label = np.array(file["psd_label_low_avse"])[selection_flage][randind]
    energy = np.array(file["energy_label"])[selection_flage][randind]
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Take a look at raw waveforms:

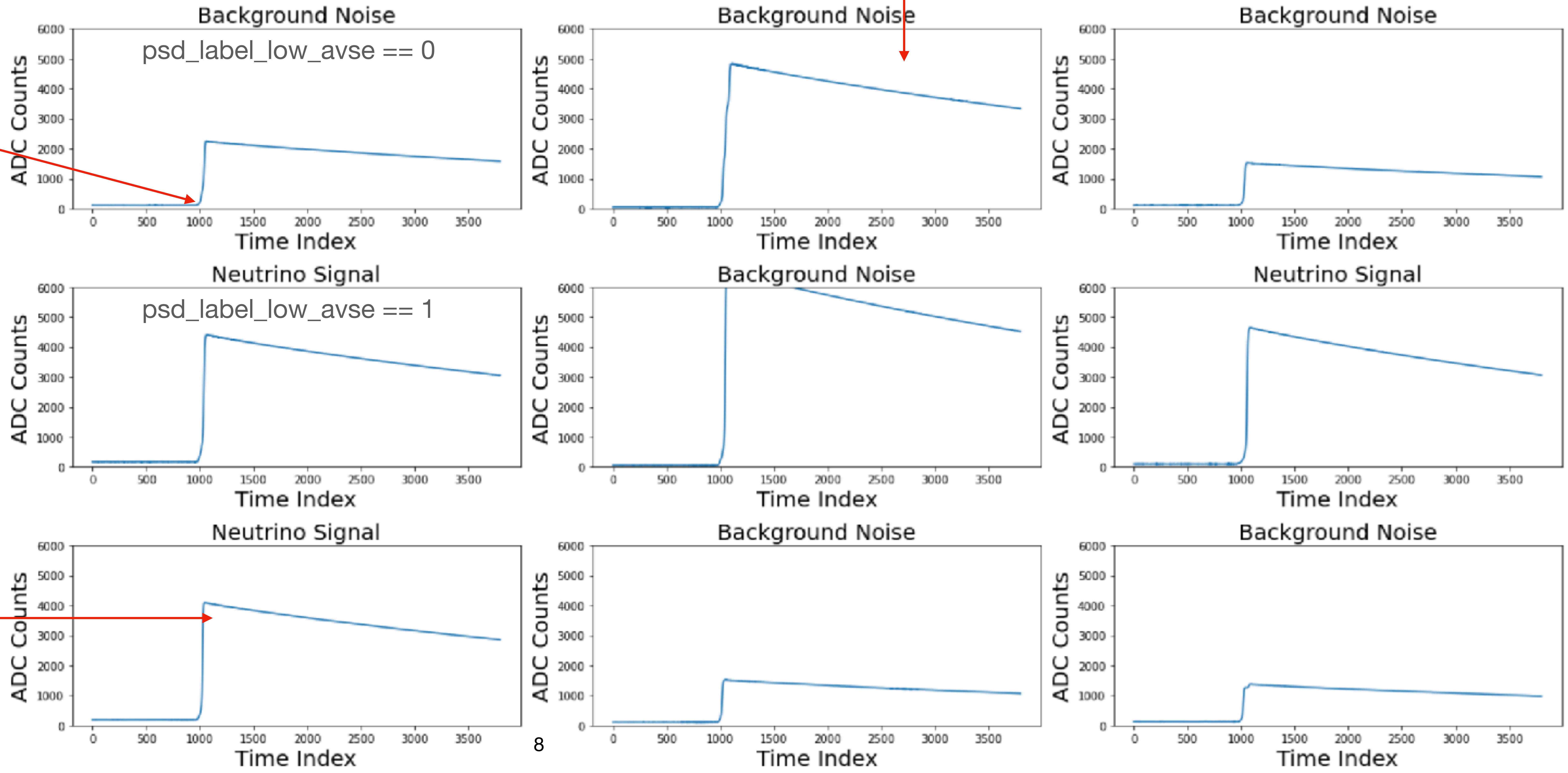


Take a look at raw waveforms:

Alarm: tail region is too long, but it does not contain physics

Alarm: Waveforms may starts rising at different positions

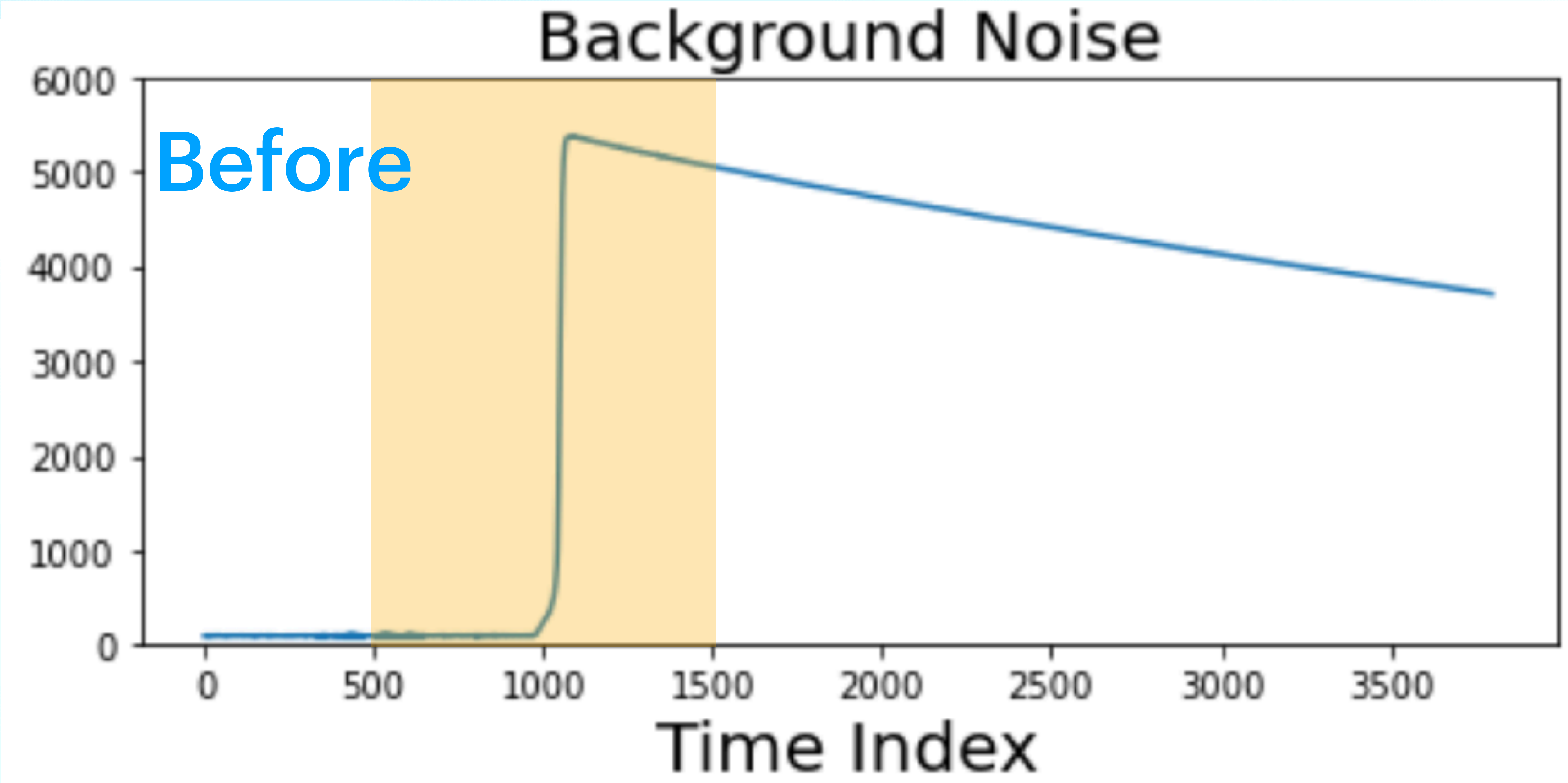
Alarm: Wavefomrs have different hieight due to energy difference.

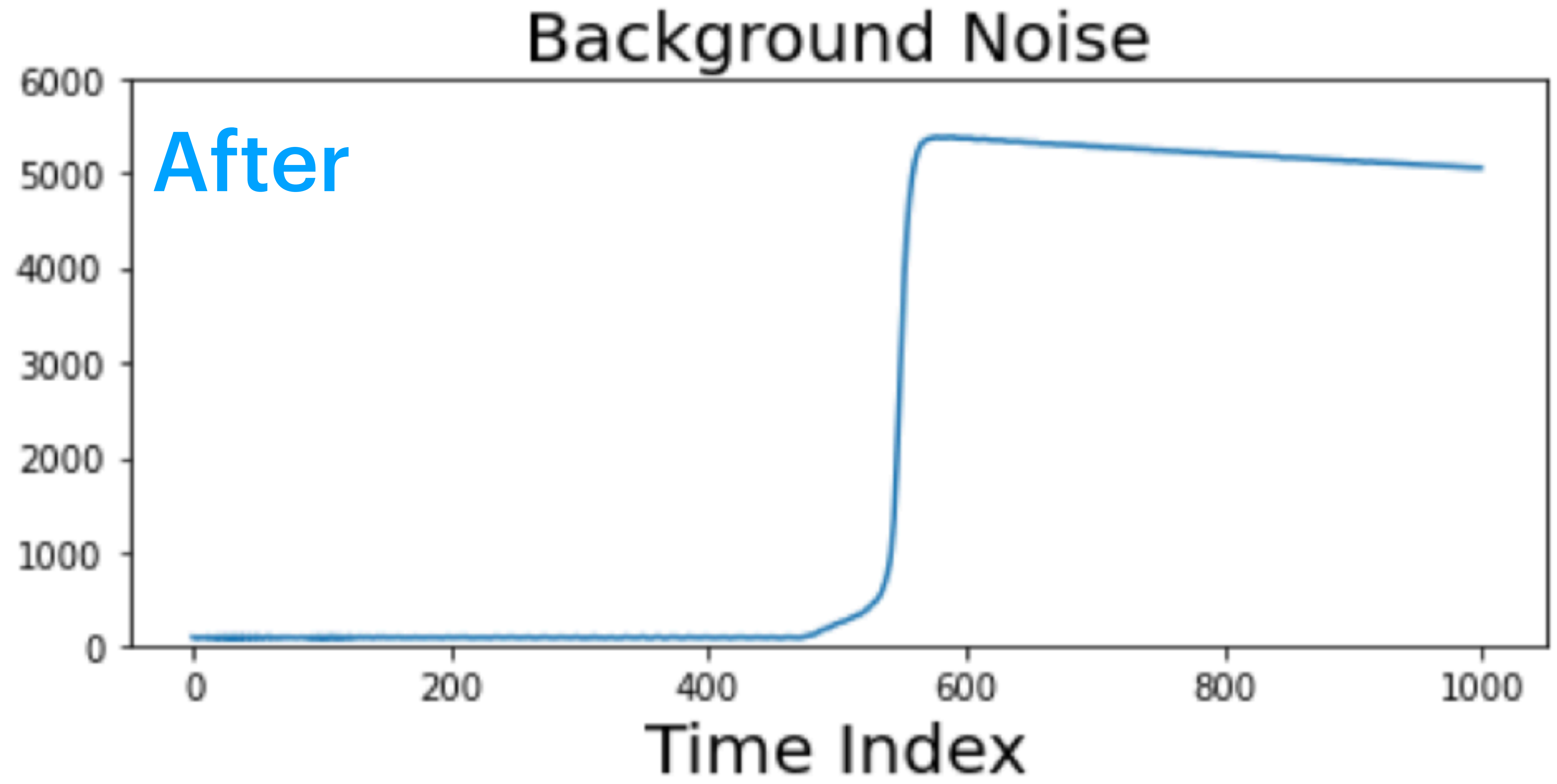


Concept

Data Preprocessing: Transform our data in certain way so that unwanted features does not affect model training

Windowing: zoom in to the rising edge of the waveform that contains most of the physics





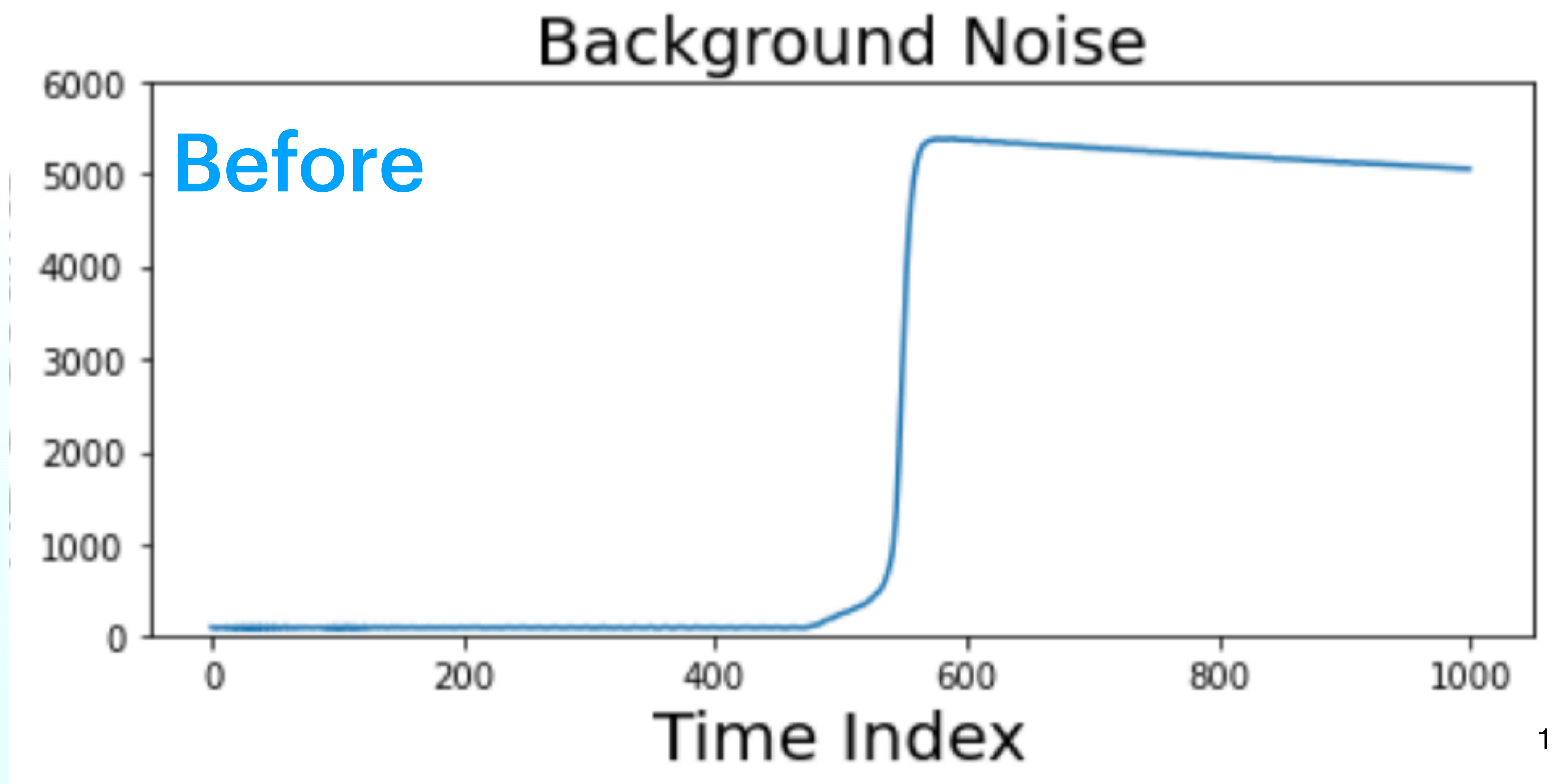
Concept

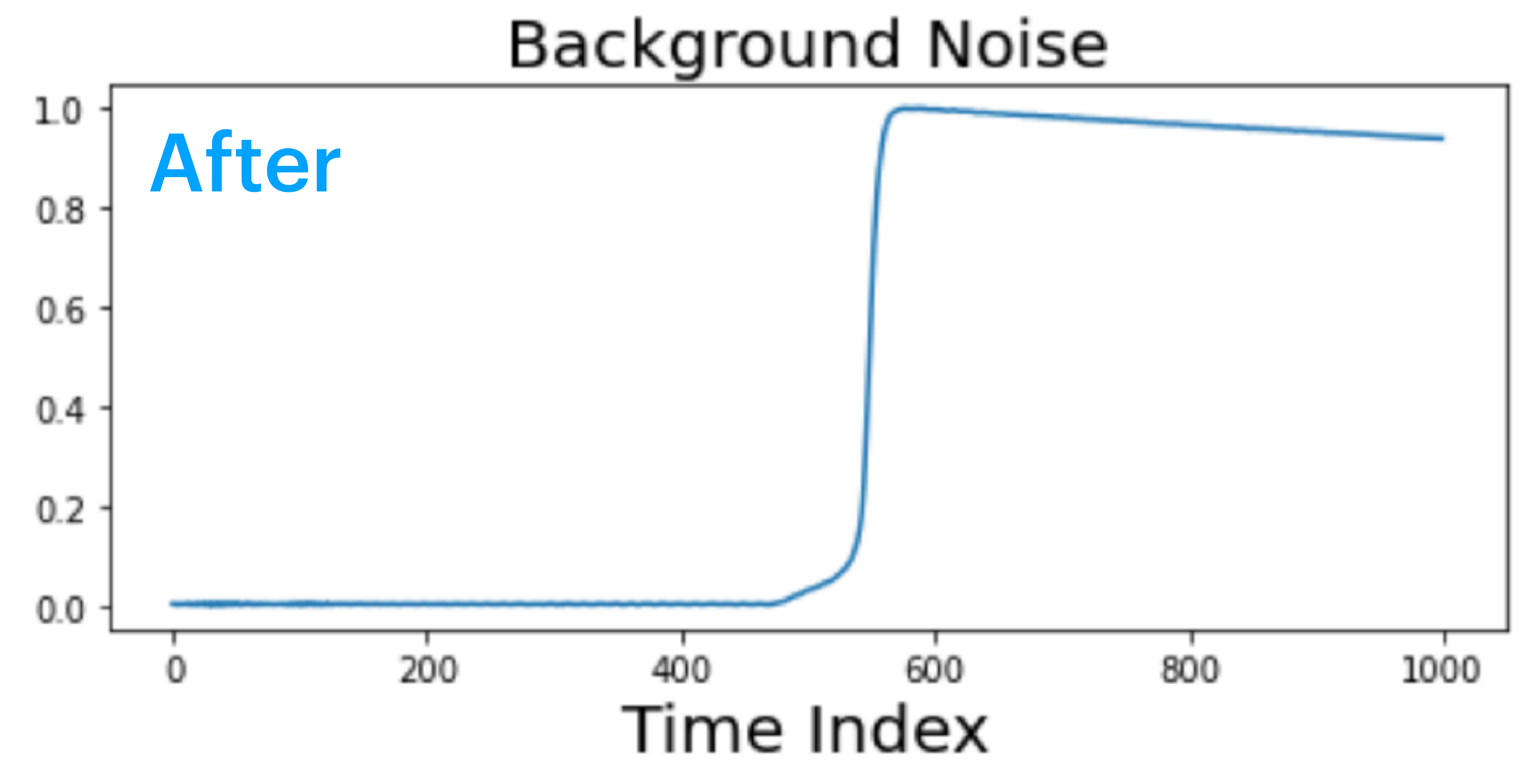
Data Preprocessing: Transform our data in certain way so that unwanted features does not affect model training

Normalizing: make sure all waveforms are of the same height (remove energy dependency)

Code

```
wf_norm = (wf - wf.min()) / (wf.max() - wf.min())  # wf is a 1-d numpy array
```





1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Concept

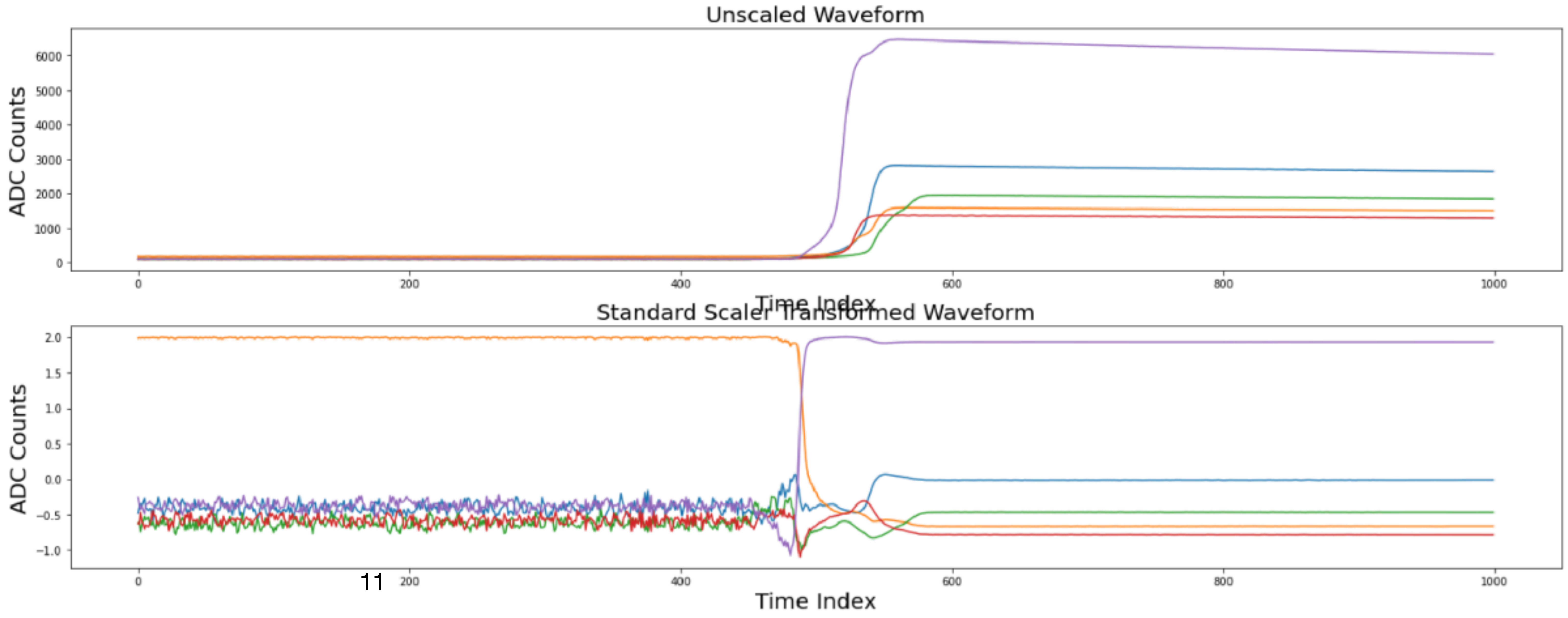
Data Preprocessing: Transform our data in certain way so that unwanted features does not affect model training

Concept

Standard Scaler: transform each dimension of data so that its mean = 0 and standard dev. = 1

Code

```
# sklearn package contains Standard Scaler
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# Has to be applied over multiple wfs
wf_norm = scaler.fit_transform(wf_arr)
```



Code

Putting everything into a PyTorch Dataset class

```python
class NeutrinoDataset(torch.utils.data.Dataset):

    def __init__(self,plot=True):
        '''
        REQUIRED function
        This function initializes the dataset object
        '''


    def __len__(self):
        '''
        REQUIRED function
        This function returns the size of overall dataset
        '''
        return


    def __getitem__(self, idx):
        '''
        REQUIRED function
        This function extract a single waveform from the dataset at the given index idx
        '''
        return

    def plot_data(self):
        '''
        OPTIONAL functions
        '''
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Putting everything into a PyTorch Dataset class

- **Only called once** when initializing the object
- Read information from the .hdf5 file into array(s)
    - apply dataset-level pre-processing like StandardScaler

```
class NeutrinoDataset(torch.utils.data.Dataset):

    def __init__(self,plot=True):
        '''
        REQUIRED function
        This function initializes the dataset object
        '''


    def __len__(self):
        '''
        REQUIRED function
        This function returns the size of overall dataset
        '''
        return


    def __getitem__(self, idx):
        '''
        REQUIRED function
        This function extract a single waveform from the dataset at the given index idx
        '''
        return

    def plot_data(self):
        '''
        OPTIONAL functions
        '''
```

self.waveform: (65000, 1000)
self.energy_label: (65000,)
self.psd_label_low_avse: (65000,)

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Putting everything into a PyTorch Dataset class

- **Only called once** when initializing the object
- Read information from the .hdf5 file into array(s)
  - apply dataset-level pre-processing like StandardScaler

- Return length of the dataset
- Called whenever we run len(dataset_object)

```
class NeutrinoDataset(torch.utils.data.Dataset):

    def __init__(self,plot=True):
        '''
        REQUIRED function
        This function initializes the dataset object
        '''

    def __len__(self):
        '''
        REQUIRED function
        This function returns the size of overall dataset
        '''
        return 65000

    def __getitem__(self, idx):
        '''
        REQUIRED function
        This function extract a single waveform from the dataset at the given index idx
        '''
        return

    def plot_data(self):
        '''
        OPTIONAL functions
        '''
```

self.waveform: (65000, 1000)
self.energy_label: (65000,)
self.psd_label_low_avse: (65000,)

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Putting everything into a PyTorch Dataset class

- **Only called once** when initializing the object
- Read information from the .hdf5 file into array(s)
  - apply dataset-level pre-processing like StandardScaler

- Return length of the dataset
- Called whenever we run len(dataset_object)

- Return **one data point** at input index **idx**
- Datapoint-level preprocessing can go here (i.e. Normalizing)
- Will be **called very frequently** during network training and validation
  - Don't put slow operations here!

```python
class NeutrinoDataset(torch.utils.data.Dataset):

    def __init__(self,plot=True):
        '''
        REQUIRED function
        This function initializes the dataset object
        '''

    def __len__(self):
        '''
        REQUIRED function
        This function returns the size of overall dataset
        '''
        return 65000

    def __getitem__(self, idx):
        '''
        REQUIRED function
        This function extract a single waveform from the dataset at the given index idx
        '''
        return self.waveform[idx], self.psd_label_low_avse[idx], self.energy_label[idx]

    def plot_data(self):
        '''
        OPTIONAL functions
        '''
```

self.waveform: (65000, 1000)
self.energy_label: (65000,)
self.psd_label_low_avse: (65000,)

Code

Putting everything into a PyTorch Dataset class

- **Only called once** when initializing the object
- Read information from the .hdf5 file into array(s)
  - apply dataset-level pre-processing like StandardScaler

- Return length of the dataset
- Called whenever we run len(dataset_object)

- Return **one data point** at input index **idx**
- Datapoint-level preprocessing can go here (i.e. Normalizing)
- Will be **called very frequently** during network training and validation
  - Don't put slow operations here!

- Define as many helper functions as you like

```
class NeutrinoDataset(torch.utils.data.Dataset):

    def __init__(self,plot=True):
        '''
        REQUIRED function
        This function initializes the dataset object
        '''

    def __len__(self):
        '''
        REQUIRED function
        This function returns the size of overall dataset
        '''
        return 65000

    def __getitem__(self, idx):
        '''
        REQUIRED function
        This function extract a single waveform from the dataset at the given index idx
        '''
        return self.waveform[idx], self.psd_label_low_avse[idx], self.energy_label[idx]

    def plot_data(self):
        '''
        OPTIONAL functions
        '''
```

self.waveform: (65000, 1000)
self.energy_label: (65000,)
self.psd_label_low_avse: (65000,)

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Define **dataset** Object

Obtain **training** and **validation** indices

Shuffling indices

Create **sampler** based on the indices list

loader = **dataset** + **sampler**

Code

Used the pre-defined class on last slides to construct training and testing data loader

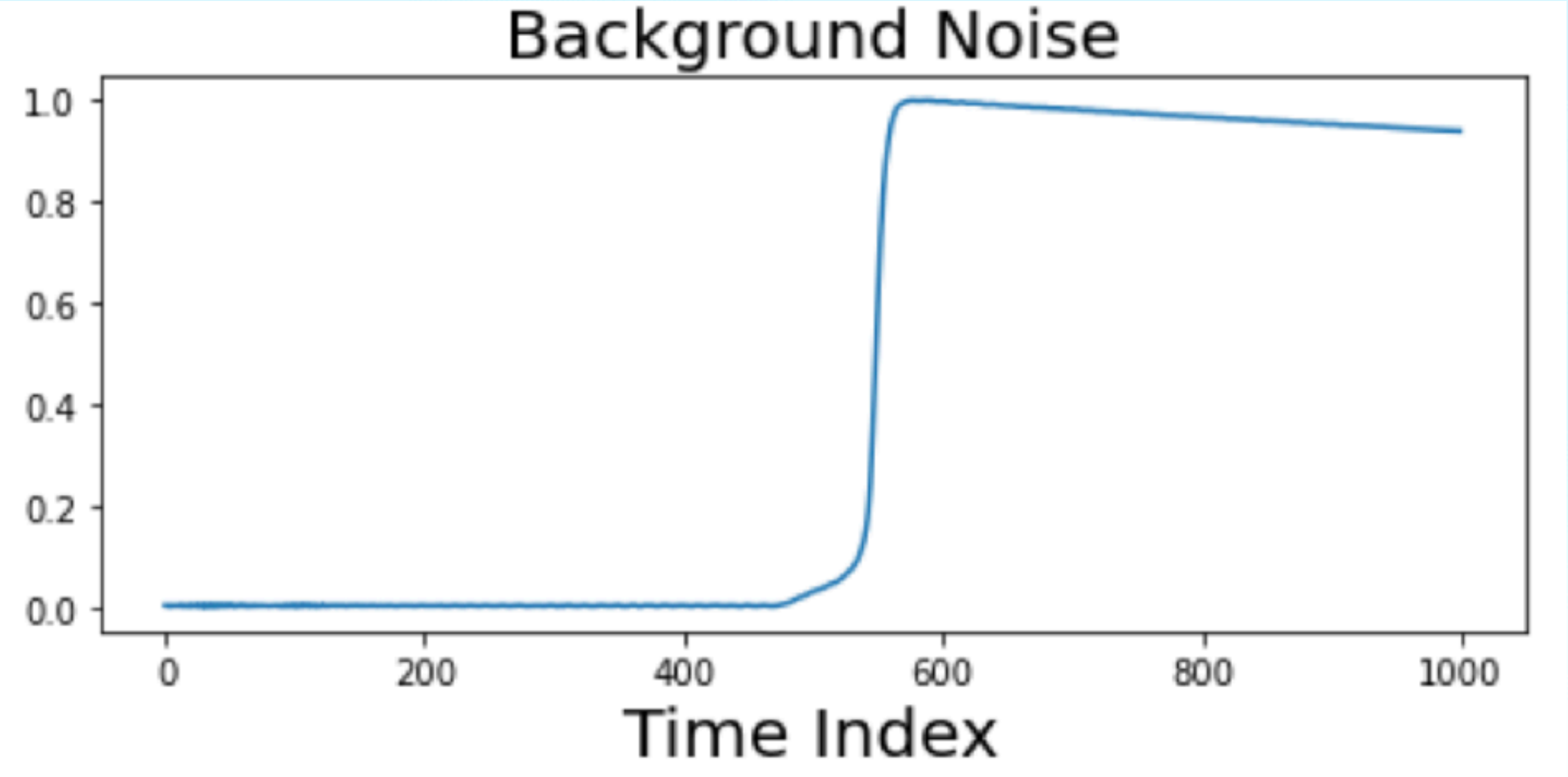
```
dataset = NeutrinoDataset(plot=False)
'''
Get the indices of train dataset and test dataset correspondingly,
indices [0:train_test_split] is the training dataset,
indices [train_test_split, len(dataset)] is the test dataset.
'''
train_test_split = int(0.7*len(dataset))
train_indices, val_indices = list(range(train_test_split)), list(range(train_test_split,len(dataset)))

#Shuffle the two indices list
np.random.shuffle(train_indices)
np.random.shuffle(val_indices)

# Define two subset random sampler to sample events according to the training indices
train_sampler = SubsetRandomSampler(train_indices)
valid_sampler = SubsetRandomSampler(val_indices)
'''
Finally, define the loader by passing in the dataset, batch size and corresponding sampler
Note that the number of data in each sub-dataset might not be divisibe by the batch size,
so drop_last=True drops the last batch with all the residual events.
'''
train_loader = data_utils.DataLoader(dataset, batch_size=BATCH_SIZE, sampler=train_sampler, drop_last=True)
test_loader = data_utils.DataLoader(dataset, batch_size=BATCH_SIZE,sampler=valid_sampler,  drop_last=True)
```

Batch: group n waveforms together into a 2D array
(1000,) → (BATCH_SIZE, 1000)

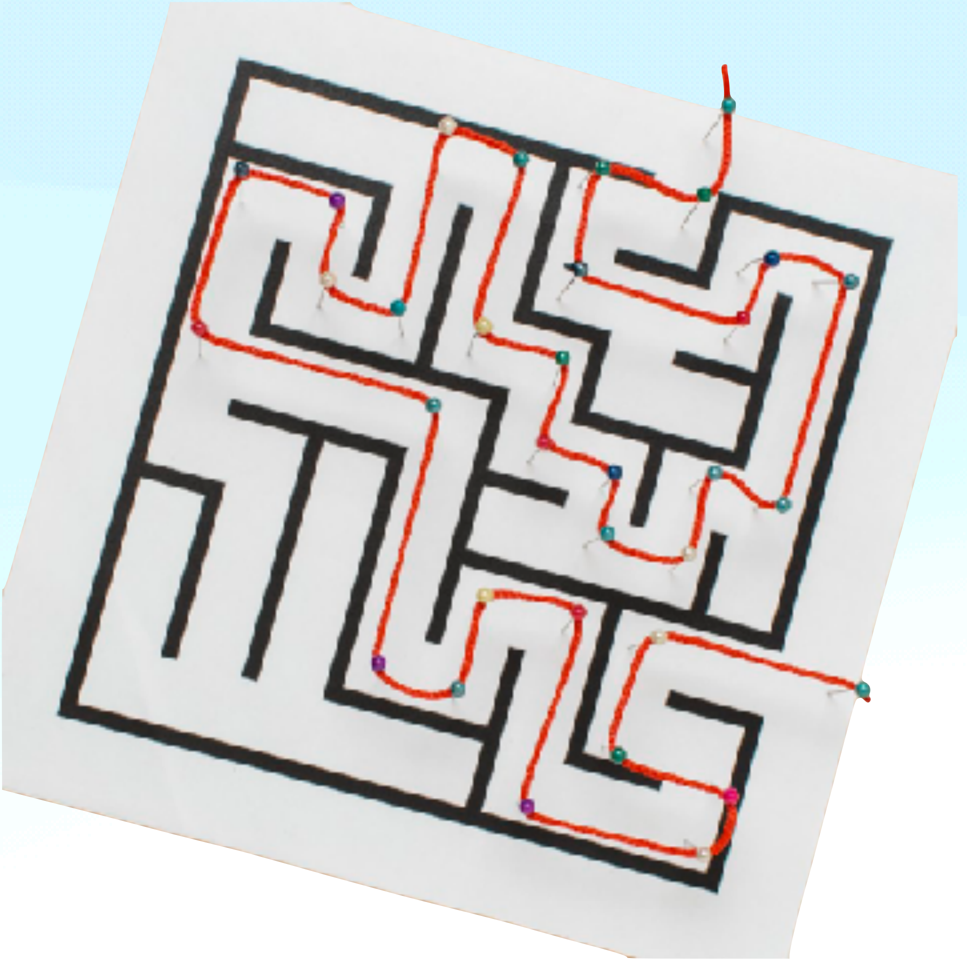
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Time Series Data
(BATCH_SIZE, 1000)



Model
Map input to output



→ **Signal or Background**
(batch_size, 1)

psd_label_low_avse
(batch_size, 1)

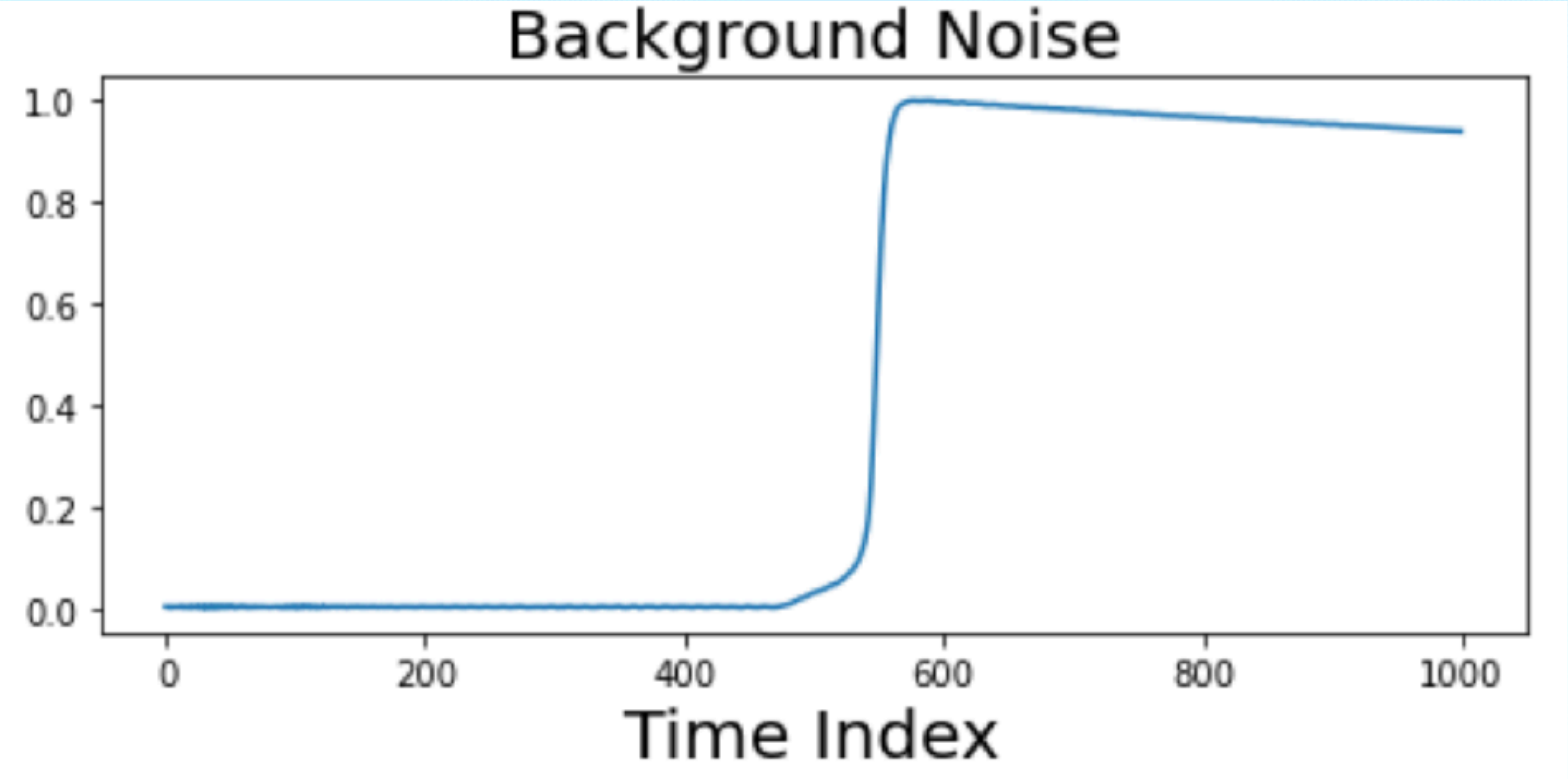
→ **Event Energy**
(batch_size, 1)

energy_label
(batch_size, 1)

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Time Series Data
(batch_size, 1000)

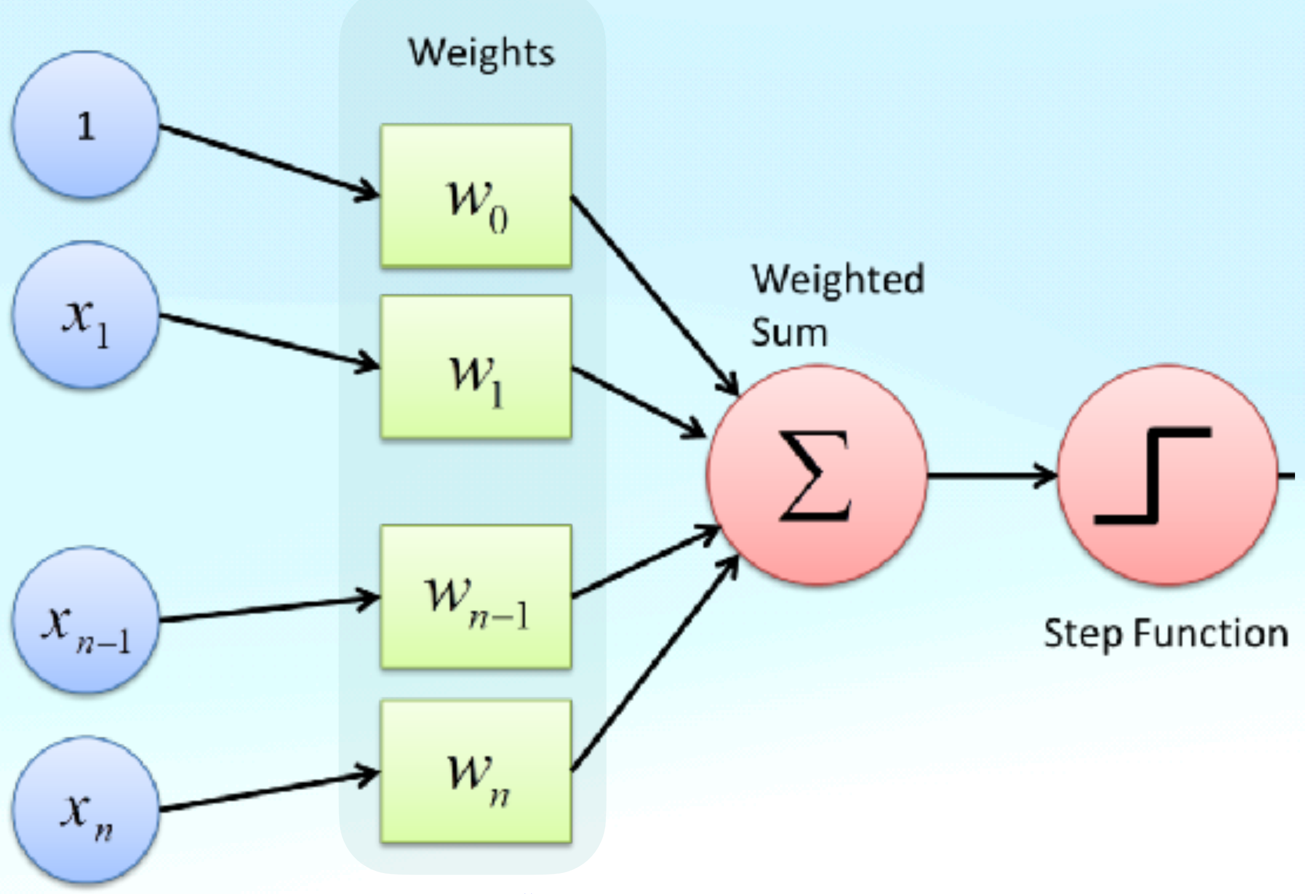


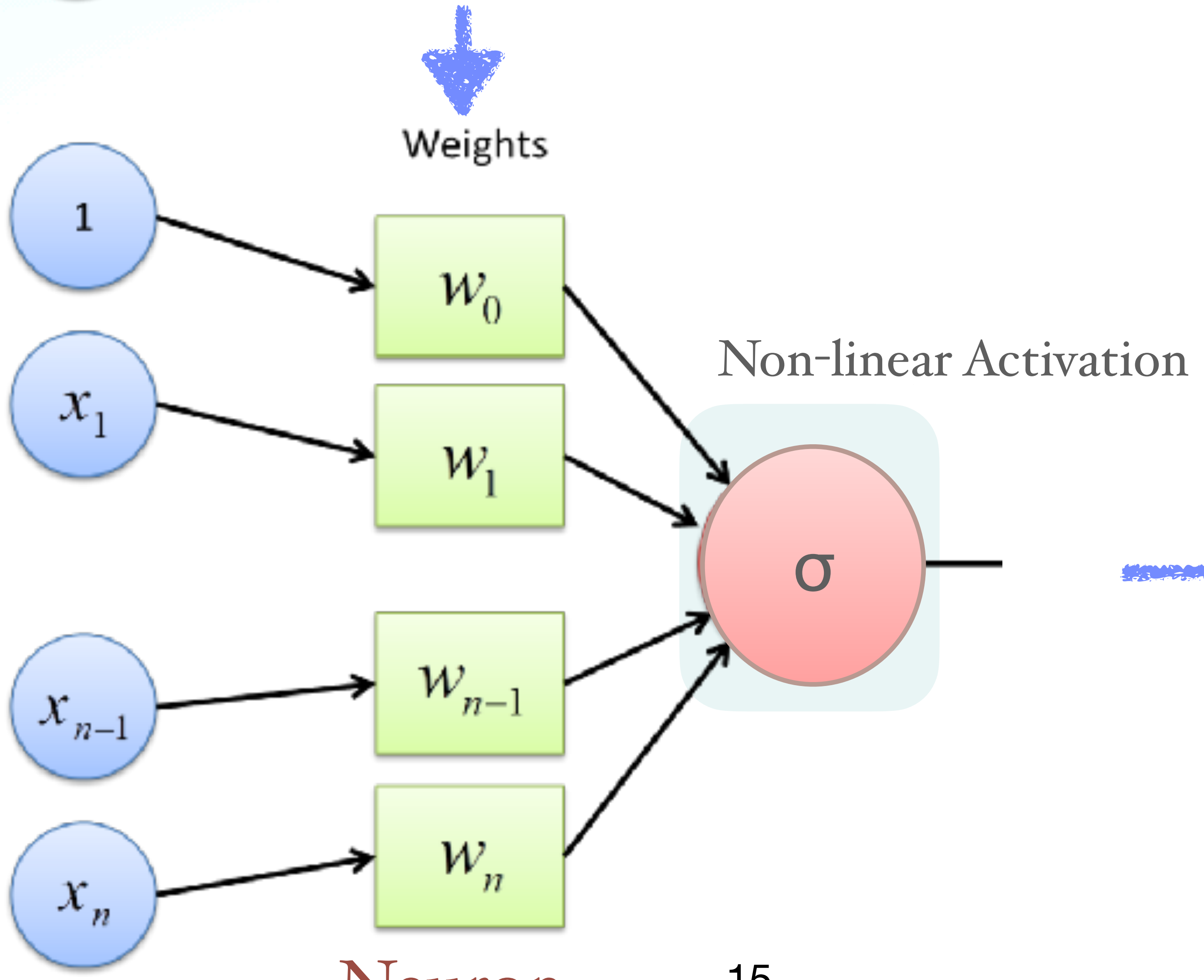
Concept

Elements of NN Layers:

- Input dimension
- Output dimension
- Activation function

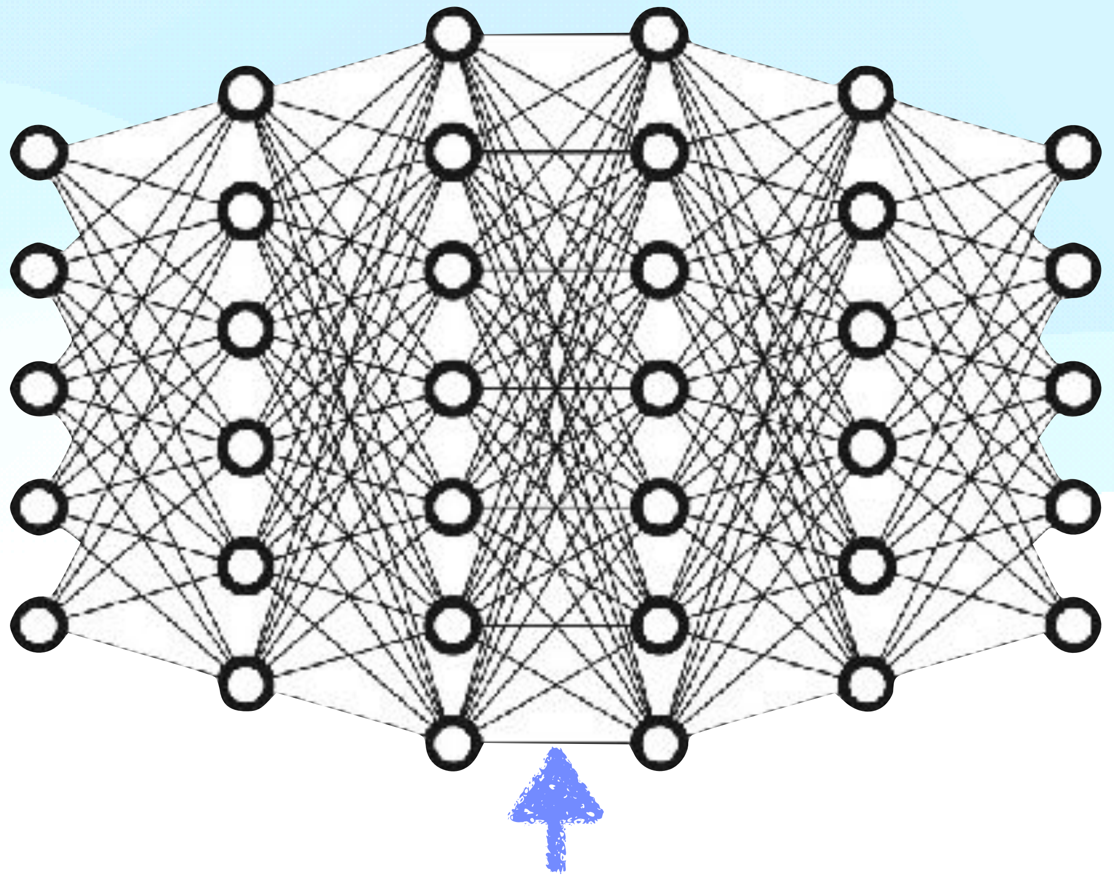
Perceptron Learning
(Linear Classifier)



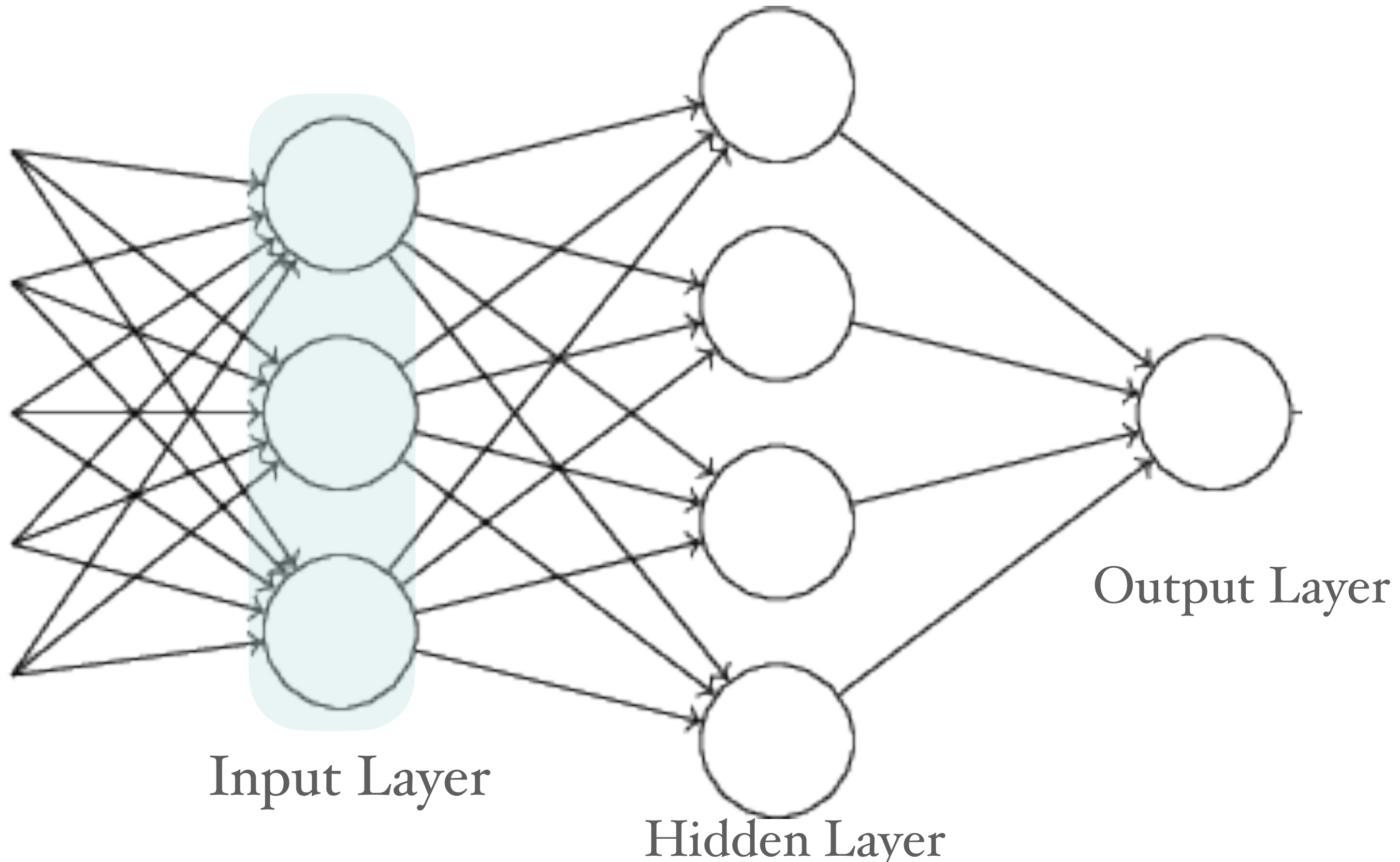


Neuron

Deep Neural Networks
Adding more layers!



Layers
Many parallel neurons

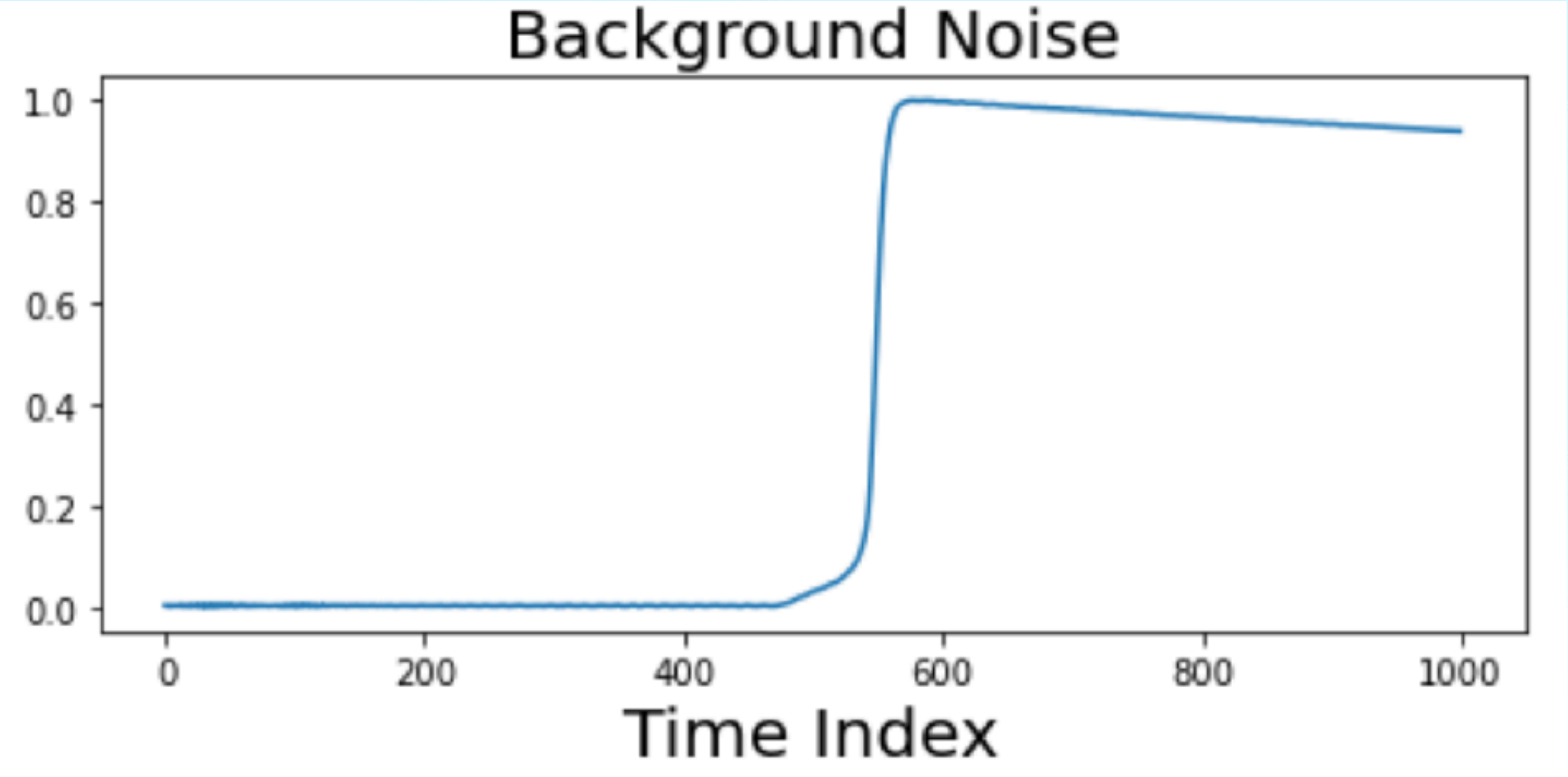


1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

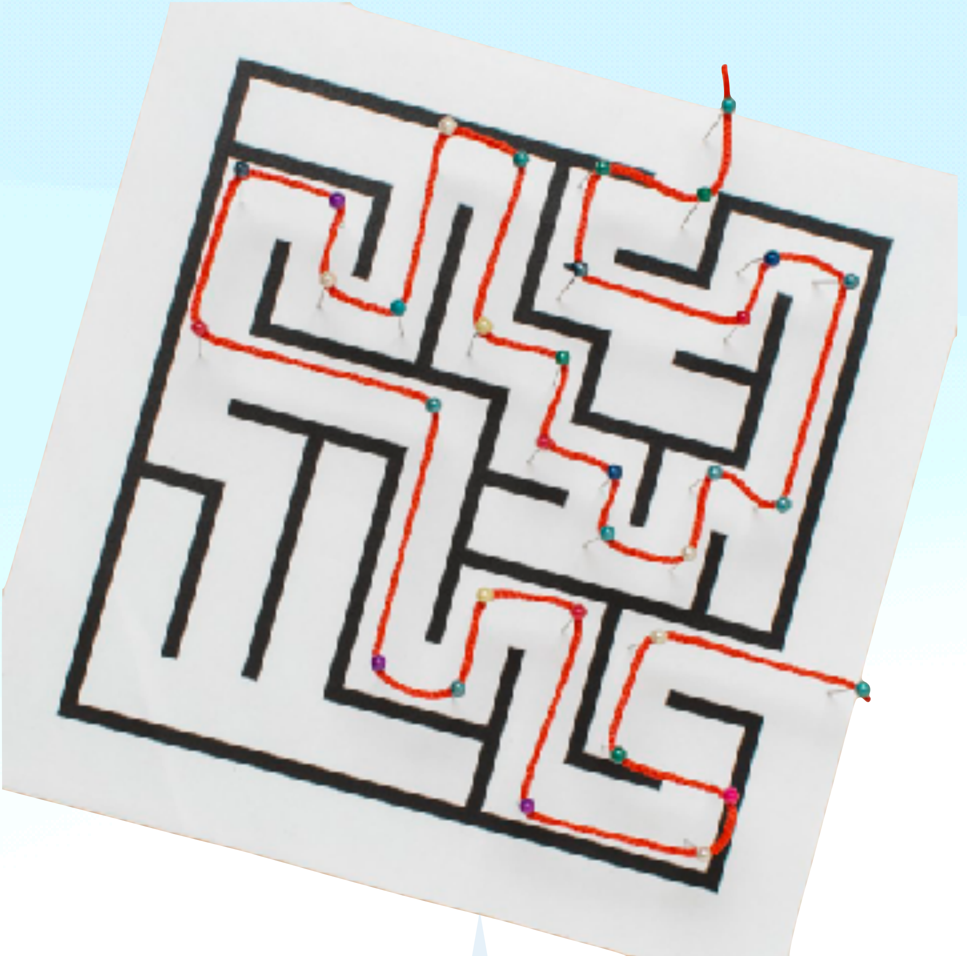
Time Series Data

(BATCH_SIZE, 1000)



Model

Map input to output



→ **Signal or Background** (batch_size, 1) — **psd_label_low_avse** (batch_size, 1)

→ **Event Energy** (batch_size, 1) — **energy_label** (batch_size, 1)

Concept

$$NNLayer(1000,512) \rightarrow \sigma$$

$$\rightarrow NNLayer(512, 256) \rightarrow \sigma$$

$$\rightarrow NNLayer(256, 32) \rightarrow \sigma$$

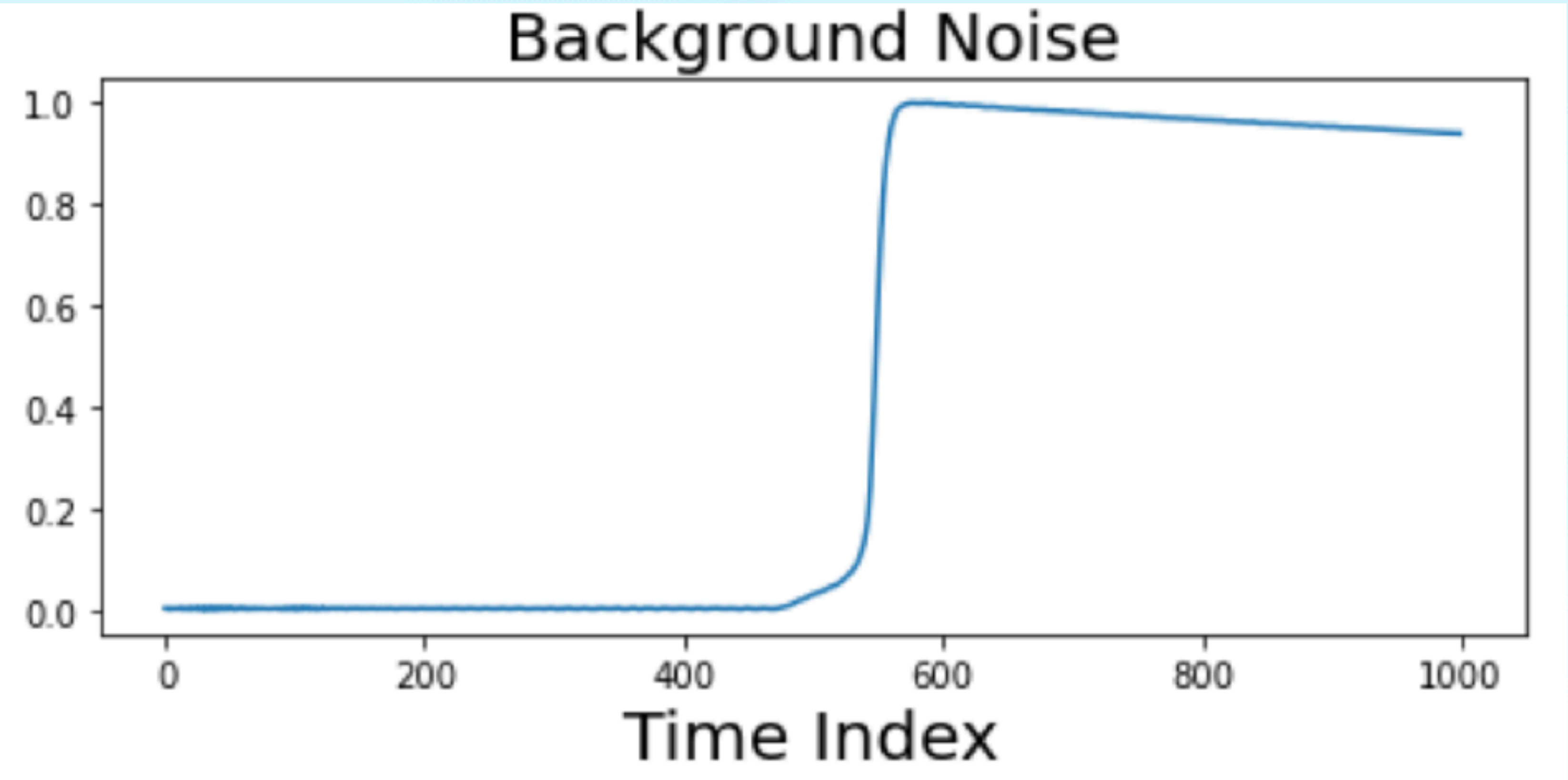
$$\rightarrow NNLayer(32, 1) \rightarrow \sigma$$

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

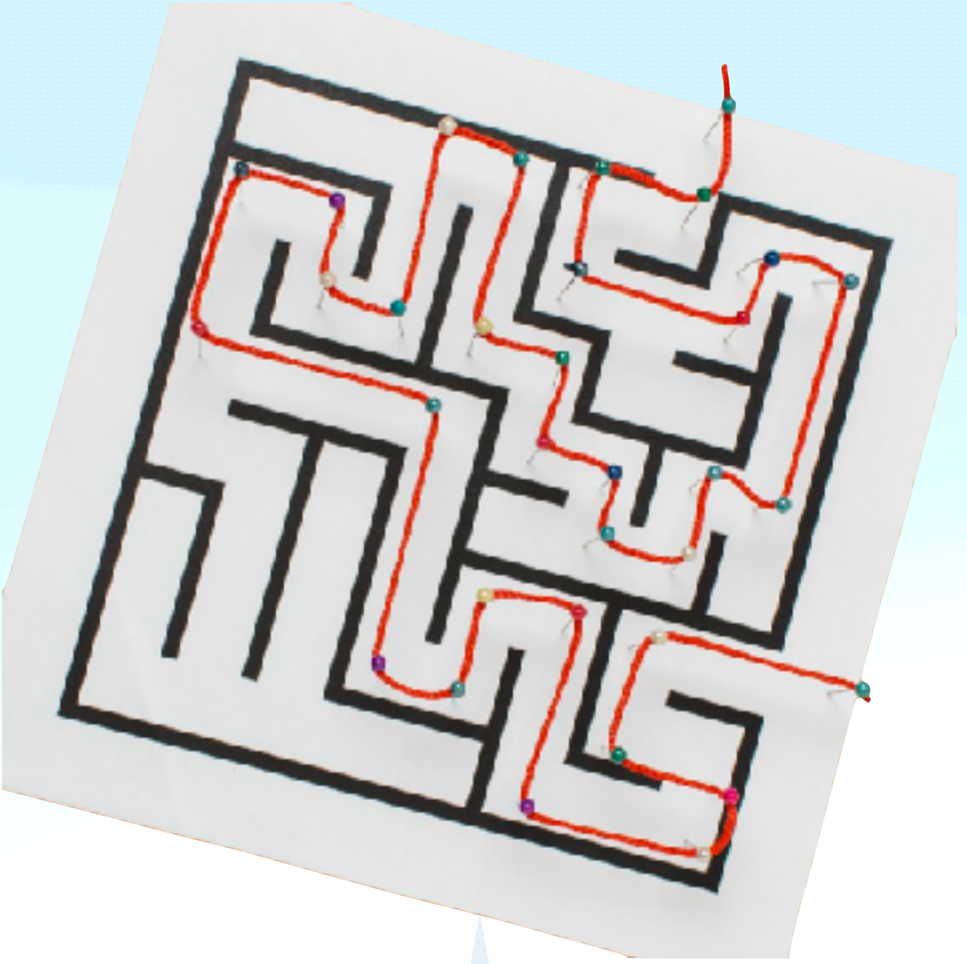
Time Series Data

(BATCH_SIZE, 1000)



Model

Map input to output



→ **Signal or Background**
(batch_size, 1)

psd_label_low_avse
(batch_size, 1)

→ **Event Energy**
(batch_size, 1)

energy_label
(batch_size, 1)

Concept

Feature Extractor Network

Take raw data as the input and output a low-dimensional vector

NNLayer(1000,512) → σ

→ *NNLayer(512, 256)* → σ

→ *NNLayer(256, 32)* → σ

→ *NNLayer(32, 1)* → σ

Code

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = nn.Sequential(
            torch.nn.Linear(1000, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 32),
            torch.nn.ReLU(),
        )

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)

        return x
```

- **Only called once** when initializing the neural network object
- Define network structures

Code

```
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = nn.Sequential(
            torch.nn.Linear(1000, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 32),
            torch.nn.ReLU(),
        )

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)

        return x
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

- **Only called once** when initializing the neural network object
- Define network structures

Linear Layer:

- Defined with input & output dim

Code

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = nn.Sequential(
            torch.nn.Linear(1000, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 32),
            torch.nn.ReLU(),
        )

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)

        return x
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

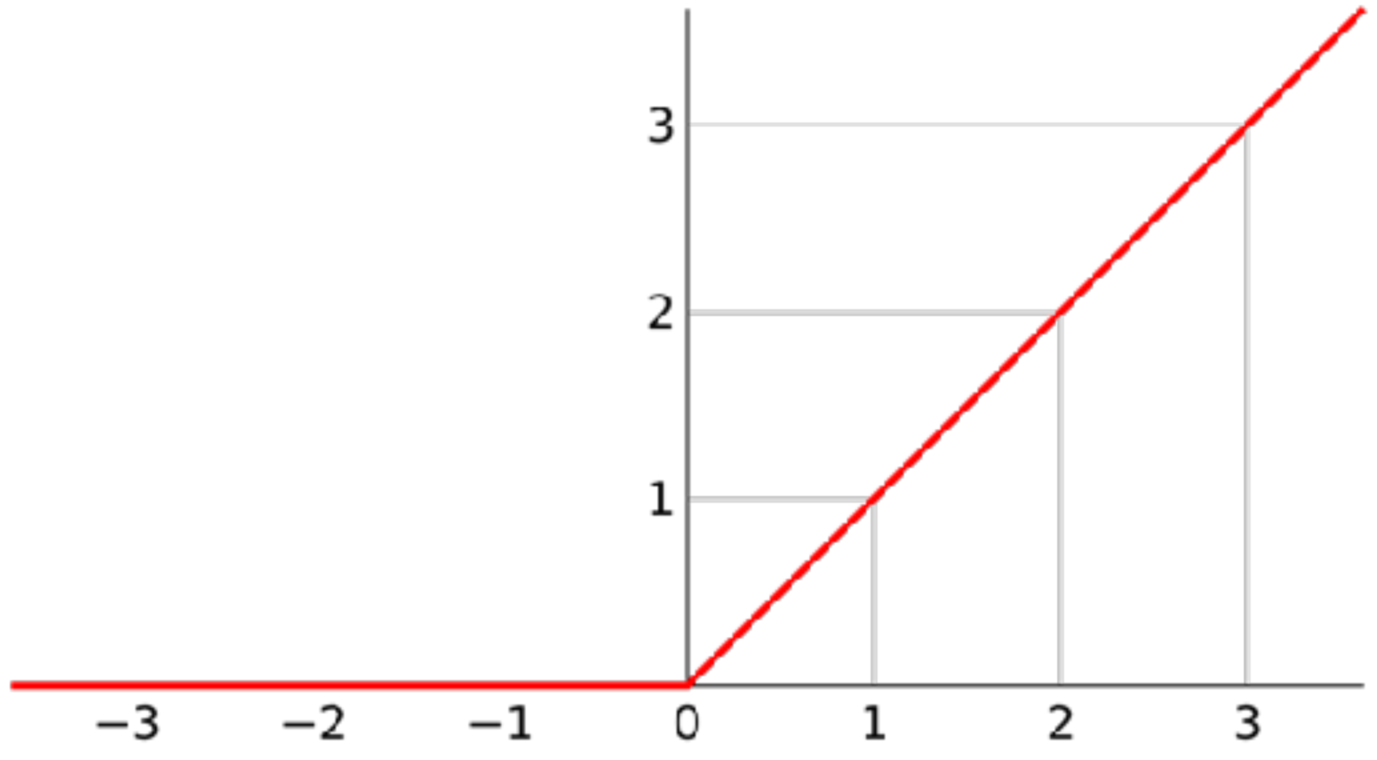
- **Only called once** when initializing the neural network object
- Define network structures

Linear Layer:
- Defined with input & output dim

Code

```
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = nn.Sequential(
            torch.nn.Linear(1000, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 32),
            torch.nn.ReLU(),
        )

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)

        return x
```

Activation Function:
- Adding non-linearity to NN
- ReLU is most commonly used



1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

- **Only called once** when initializing the neural network object
- Define network structures

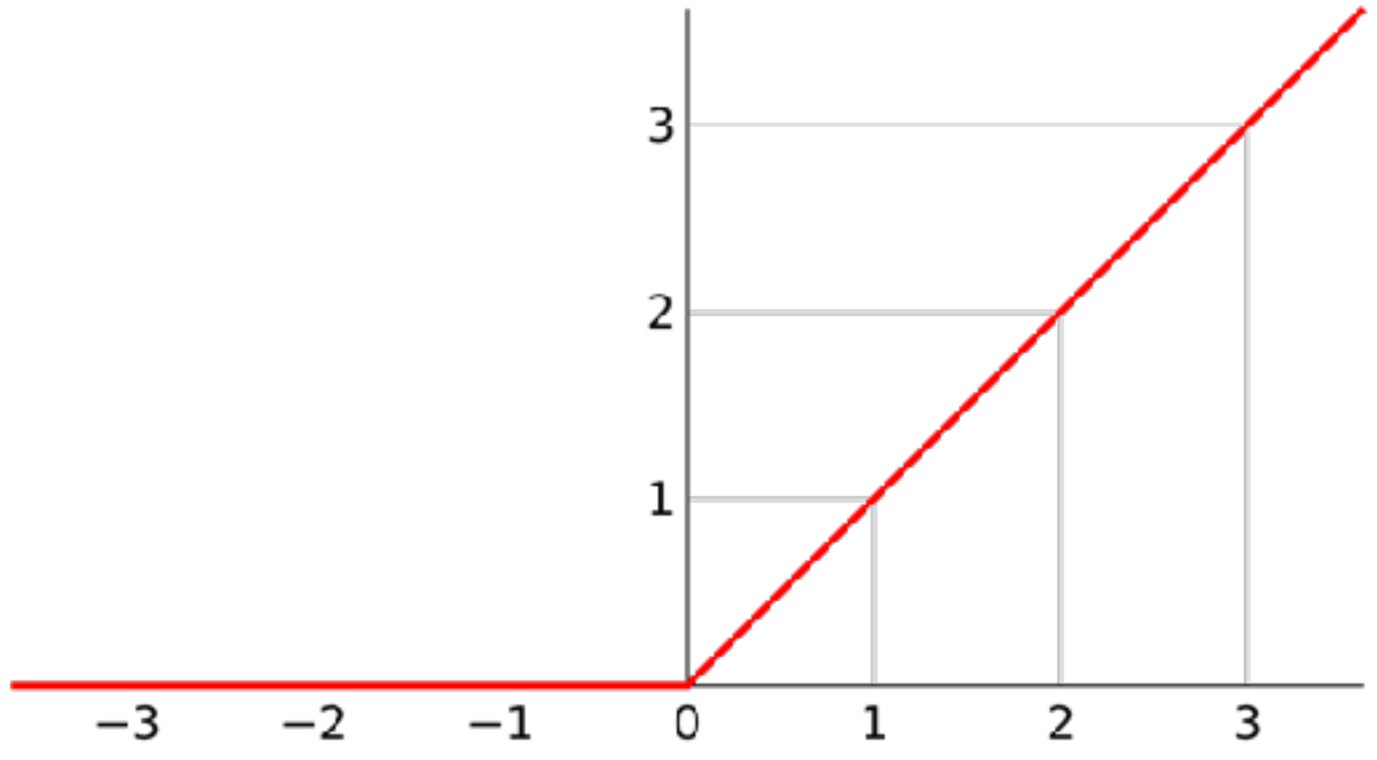
Linear Layer:
- Defined with input & output dim

Forward Pass:
- Feed data through the neural network to obtain desired outputs
  - (BS, 1000,)→(BS, 1)
- **Called multiple times** during training and validation

Backward Pass will be discussed in Sec. 4!

```
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = nn.Sequential(
            torch.nn.Linear(1000, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 32),
            torch.nn.ReLU(),
        )

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)

        return x
```

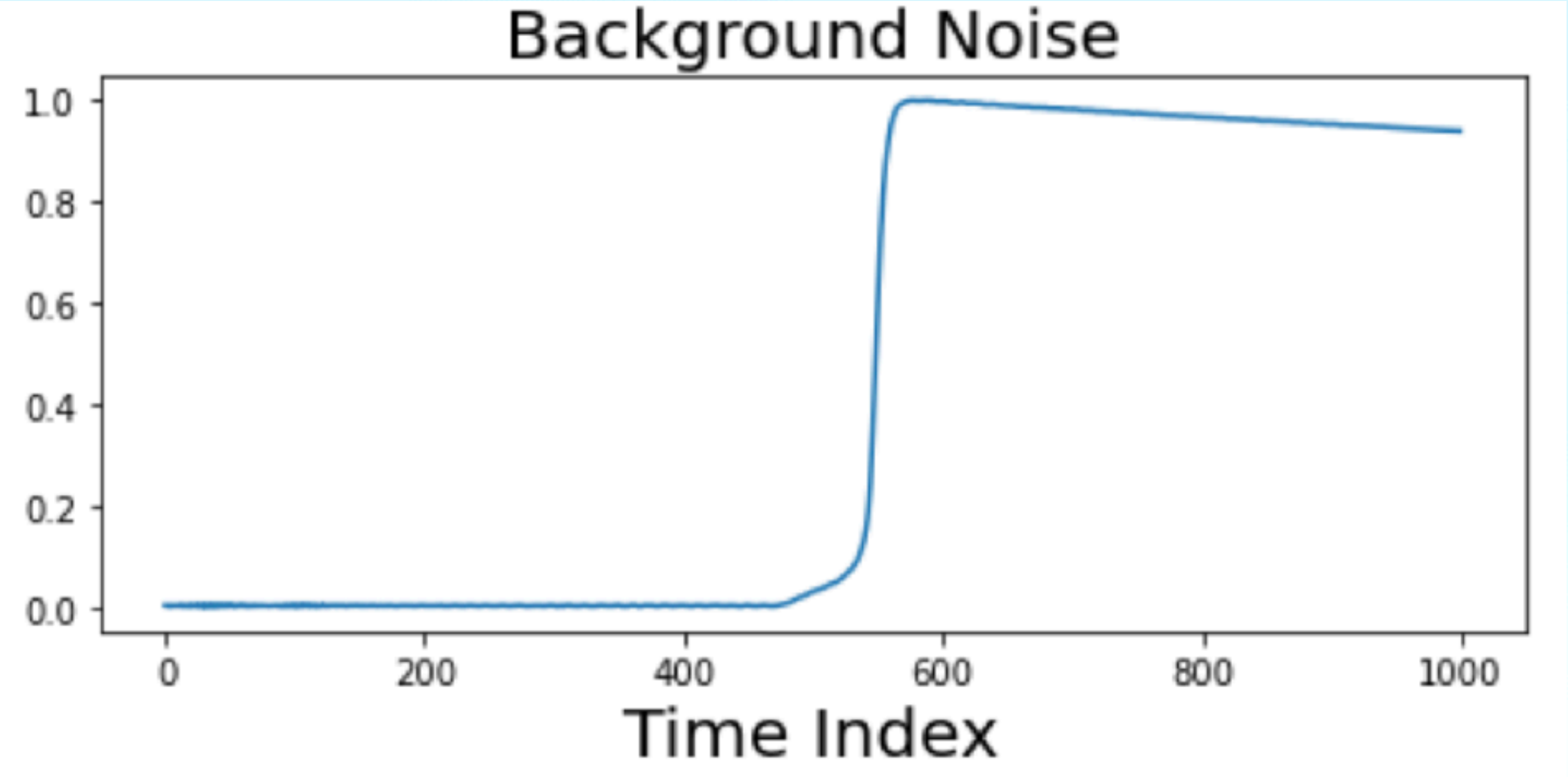
Activation Function:
- Adding non-linearity to NN
- ReLU is most commonly used



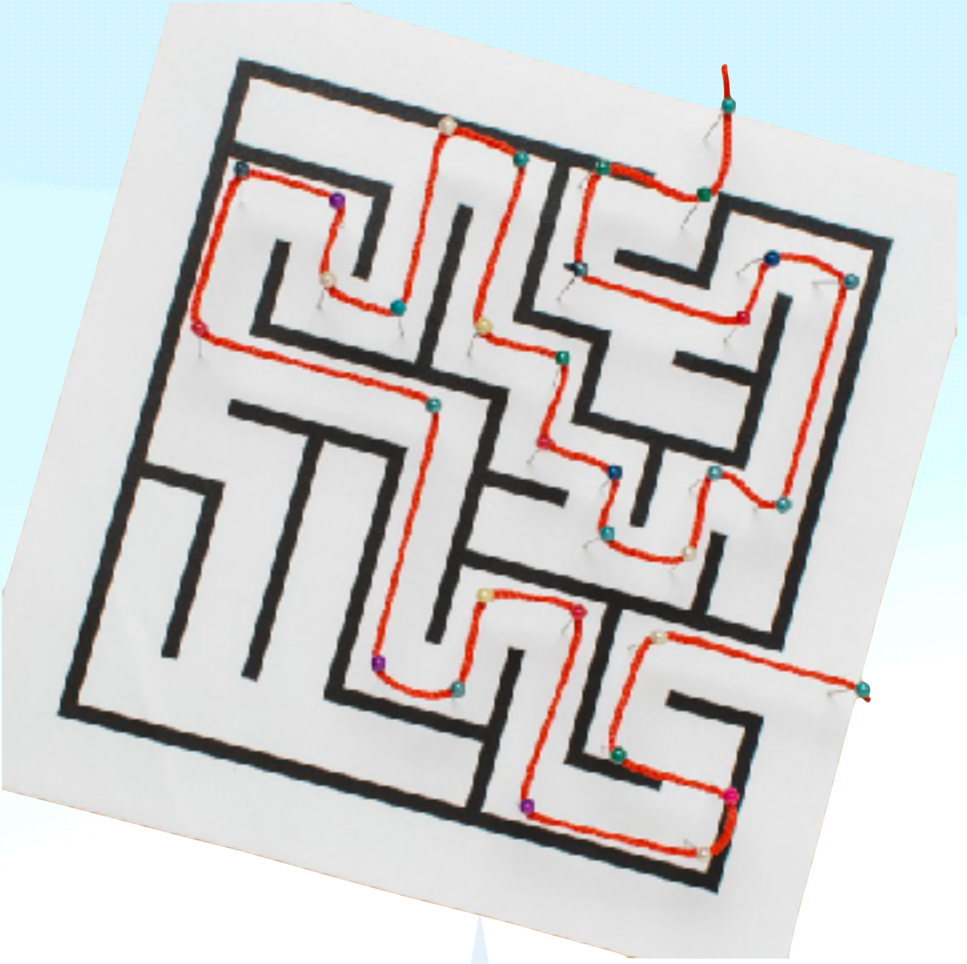
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Time Series Data
(BATCH_SIZE, 1000)



Model
Map input to output



→ **Signal or Background**
(batch_size, 1)

psd_label_low_avse
(batch_size, 1)

→ **Event Energy**
(batch_size, 1)

energy_label
(batch_size, 1)

Concept

Feature Extractor Network

Take raw data as the input and output a low-dimensional vector

NNLayer(1000,512) → σ
→ *NNLayer(512, 256)* → σ
→ *NNLayer(256, 32)* → σ

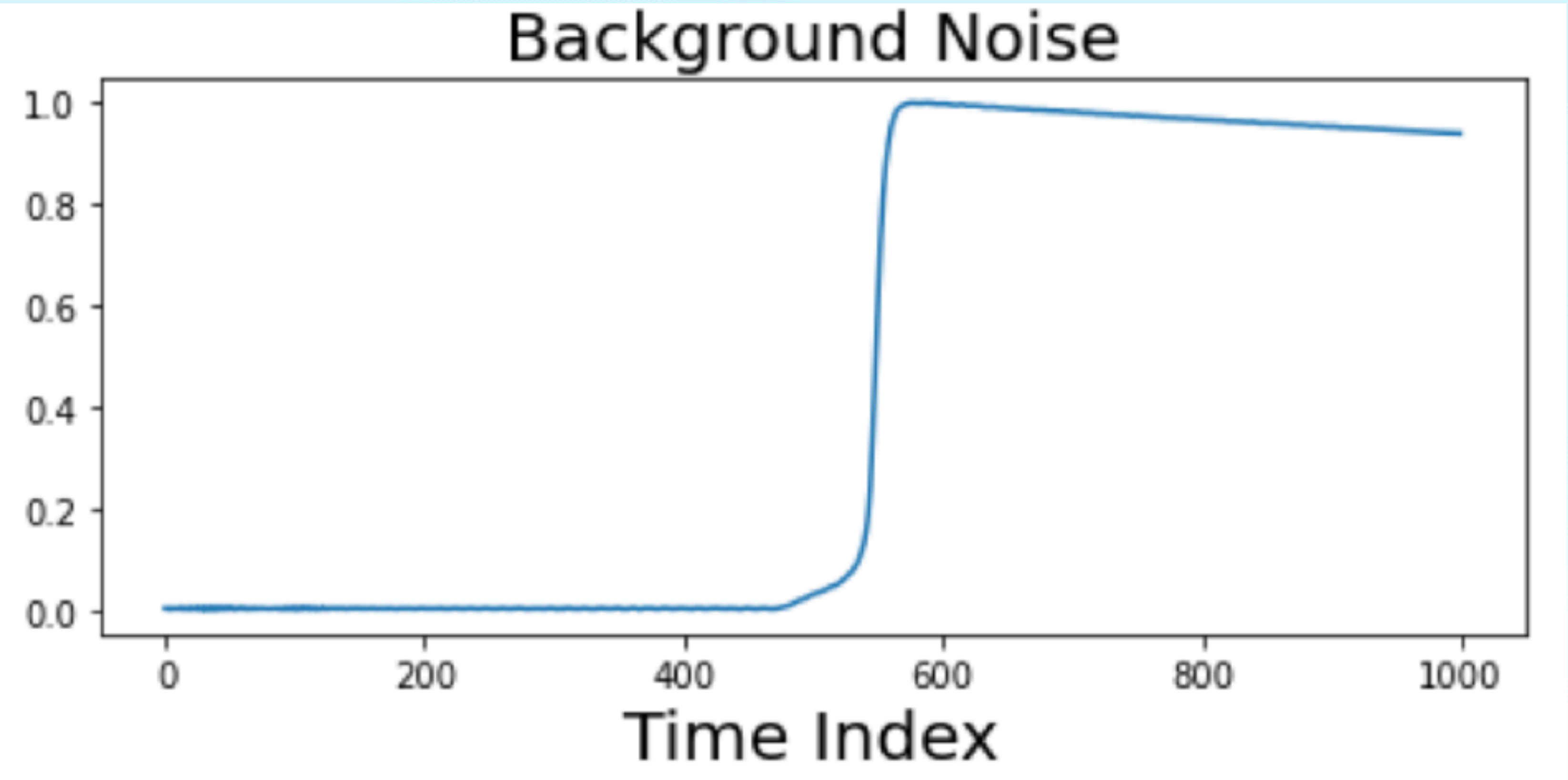
→ *NNLayer(32, 1)* → σ

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

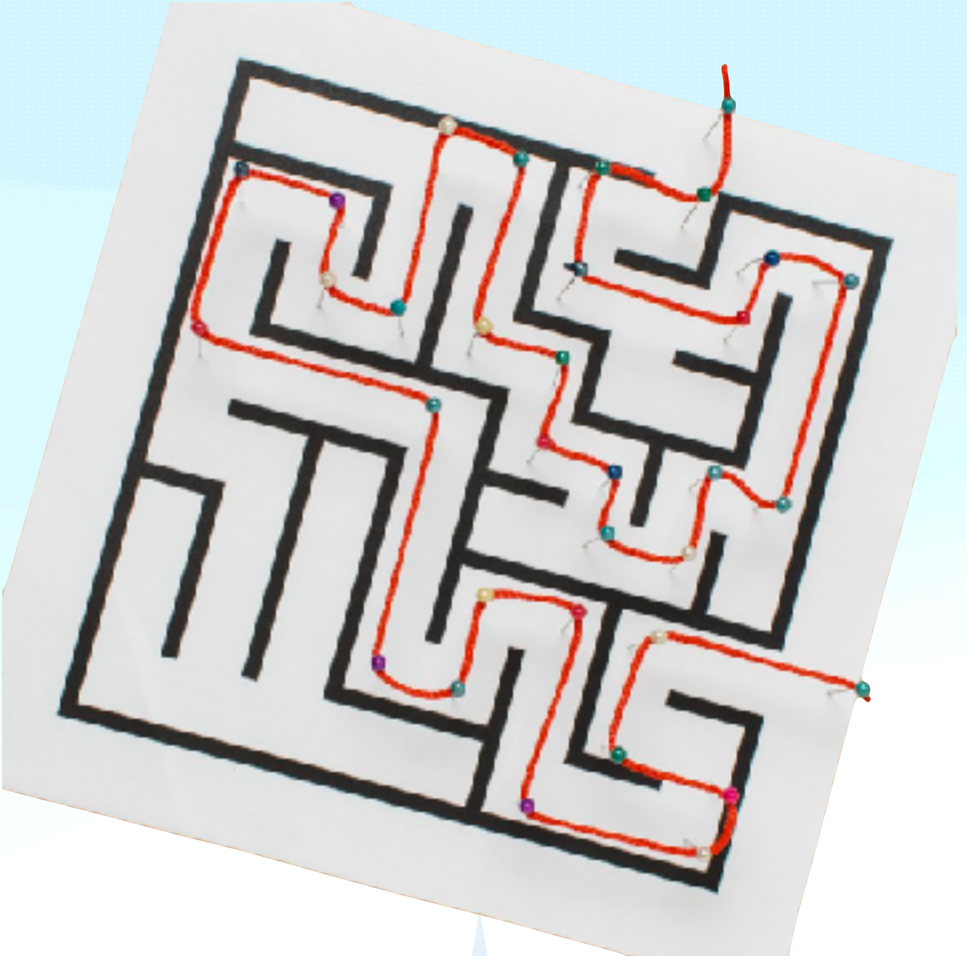
Time Series Data

(BATCH_SIZE, 1000)



Model

Map input to output



→ **Signal or Background** (batch_size, 1) — **psd_label_low_avse** (batch_size, 1)

→ **Event Energy** (batch_size, 1) — **energy_label** (batch_size, 1)

Concept

Feature Extractor Network

Take raw data as the input and output a low-dimensional vector

$NNLayer(1000,512) \to \sigma$

$\to NNLayer(512, 256) \to \sigma$

$\to NNLayer(256, 32) \to \sigma$

$\to NNLayer(32, 1) \to \sigma$

Concept

Task Module: Task Layer + Loss Function

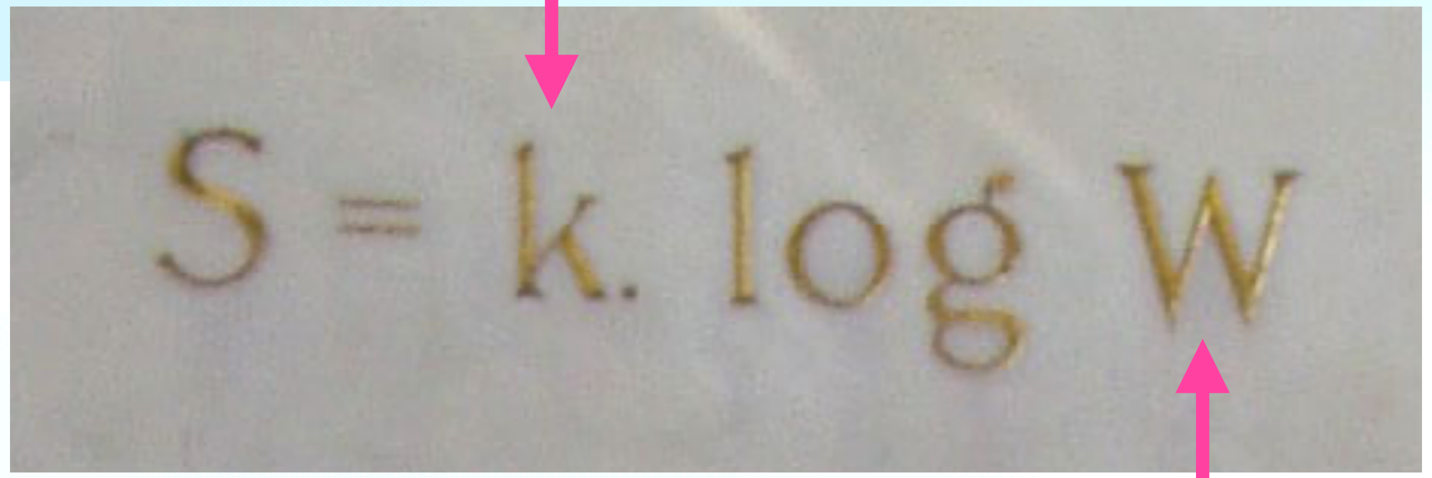
- **Task Layer:** Take low-dimensional vector as input and produce a value/vector as **output**
- **Loss Function:** produce a quantitative measure evaluating how well your algorithm models your dataset
- Minimizing the loss function means that the **model output** reproduces the **label** better

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

BOLTZMANN ENTROPY

Boltzmann's Constant



The number of real microstates corresponding to the gas's macrostate

INFORMATION ENTROPY

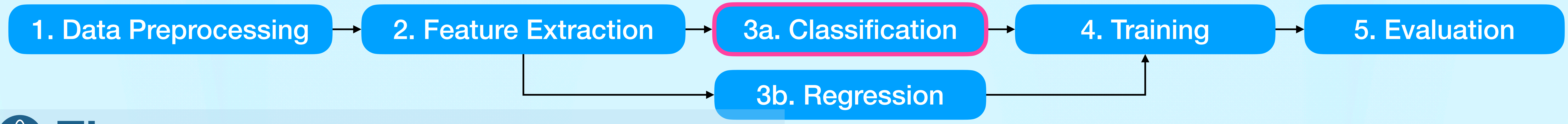
Probability of state i

Theory

$$H = \sum_i p_i \log \frac{1}{p_i} = -\sum_i p_i \log p_i$$

Optimal coding length

The number of states in state i assuming all states have equal probability



Theory

Binary entropy: $H(x) = p\log\frac{1}{p} - (1-p)log\frac{1}{1-p}$

Theory

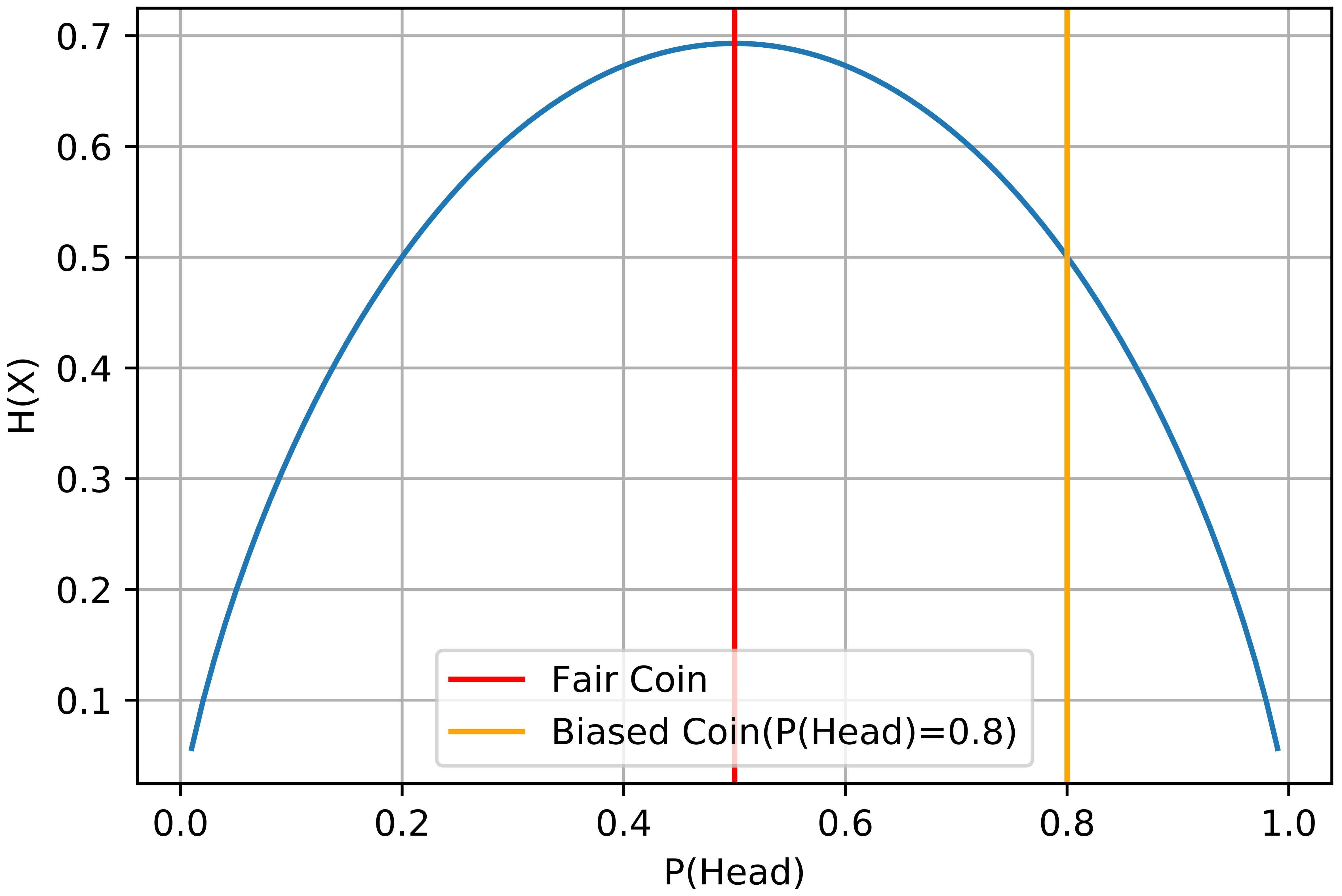
Binary entropy: $H(x) = p\log\frac{1}{p} - (1-p)log\frac{1}{1-p}$

Suppose we are tossing a **fair coin:**

- That is, the probability of head (H) and tail (T) are equal
- $H(X) = \frac{1}{2}log\frac{1}{1/2} + \frac{1}{2}log\frac{1}{1/2} = log2 \approx 0.6931$

Suppose we have an **unfair coin** where $P(Head) = 0.8$:

- $H(X) = 0.8log\frac{1}{0.8} + 0.2log\frac{1}{0.2} \approx 0.5$



Theory

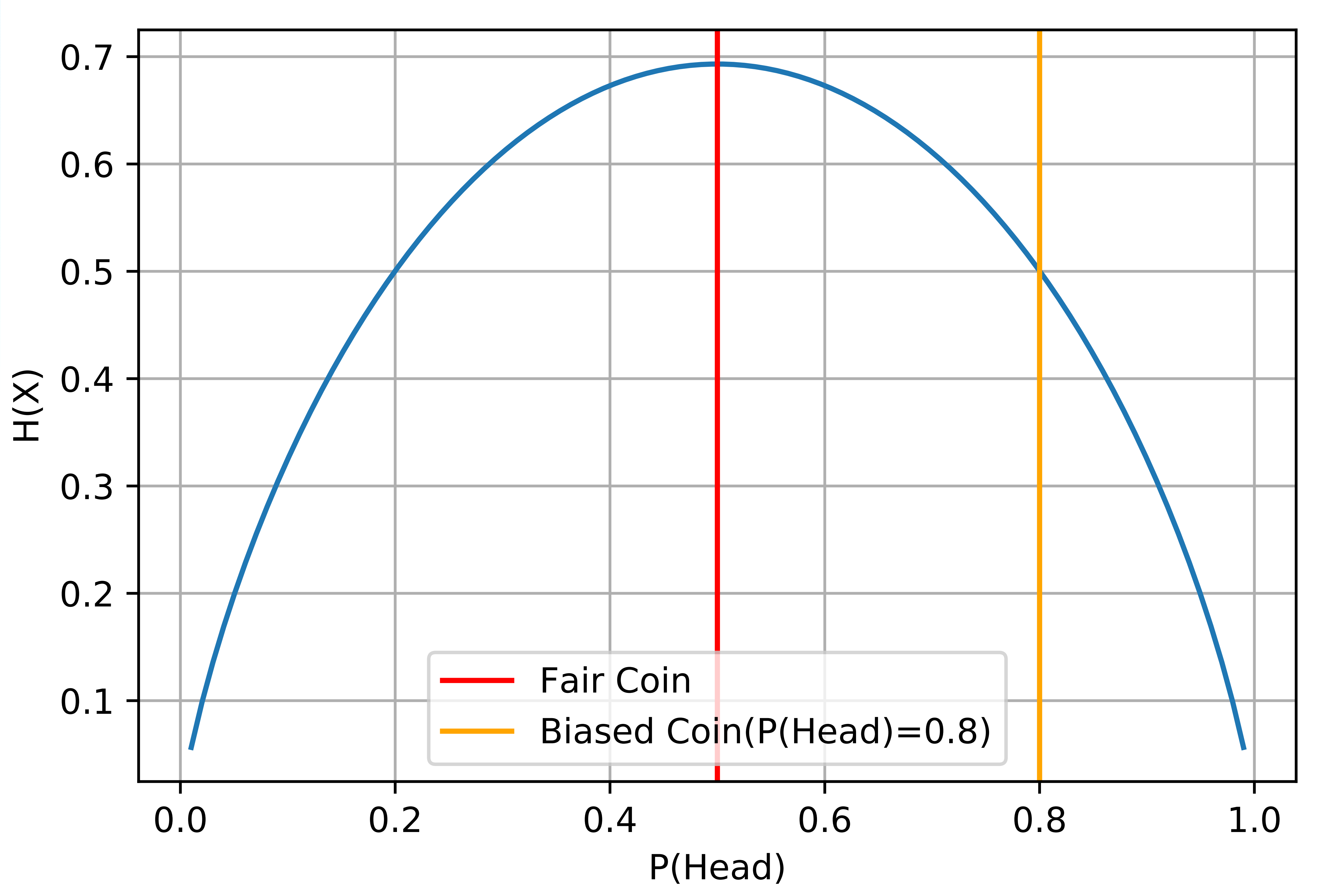
Binary entropy: $H(x) = p\log\frac{1}{p} - (1-p)log\frac{1}{1-p}$

Suppose we are tossing a **fair coin:**

- That is, the probability of head (H) and tail (T) are equal
- $H(X) = \frac{1}{2}log\frac{1}{1/2} + \frac{1}{2}log\frac{1}{1/2} = log2 \approx 0.6931$

Suppose we have an **unfair coin** where $P(Head) = 0.8$:

- $H(X) = 0.8log\frac{1}{0.8} + 0.2log\frac{1}{0.2} \approx 0.5$



Concept

Information entropy is the **average amount of surprise** we measure

- For a fair coin, it is hard to predict the outcome of next toss, so the average surprisal is **high** and thus **high entropy**
- For a unfair coin, the next outcome is likely to be a head, so the average surprisal is **low** and thus **low entropy**

Theory

Cross Entropy $H(p, q)$ is calculated over two distributions $p(X)$ and $q(X)$:

$$H(p, q) = \sum p(X) log \frac{1}{q(X)}$$

- Suppose we have a classification task to separate neutrino signals from backgrounds
- The **ground truth distribution** $p(X)$ follows the distribution of labels
- On the other hand, we build up a neural network, which analyze every single waveform to produce an **output distribution** $q(X)$

Theory

Cross Entropy $H(p, q)$ is calculated over two distributions $p(X)$ and $q(X)$:

$$H(p, q) = \sum p(X) log \frac{1}{q(X)}$$

- Suppose we have a classification task to separate neutrino signals from backgrounds
- The **ground truth distribution** $p(X)$ follows the distribution of labels
- On the other hand, we build up a neural network, which analyze every single waveform to produce an **output distribution** $q(X)$

Concept

Binary Cross Entropy: label is either 1 or 0

$$H = -[p \cdot log(q) + (1 - p) \cdot log(1 - q)]$$

Concept

Cross Entropy: label contains n classes

$$H = -\sum \vec{p} \cdot log(\vec{q}) \qquad \vec{p} = \{1,0,0...0\}$$

Information Entropy measures the average amount of surprise

Information Entropy measures the average amount of surprise

Cross Entropy measure the average amount of surprise from **network output distribution** $q(X)$**,** given that the **ground truth distribution** follows $p(X)$

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Information Entropy measures the average amount of surprise

↓

Cross Entropy measure the average amount of surprise from **network output distribution** $q(X)$**,** given that the **ground truth distribution** follows $p(X)$

↓

Information Entropy measures the average amount of surprise

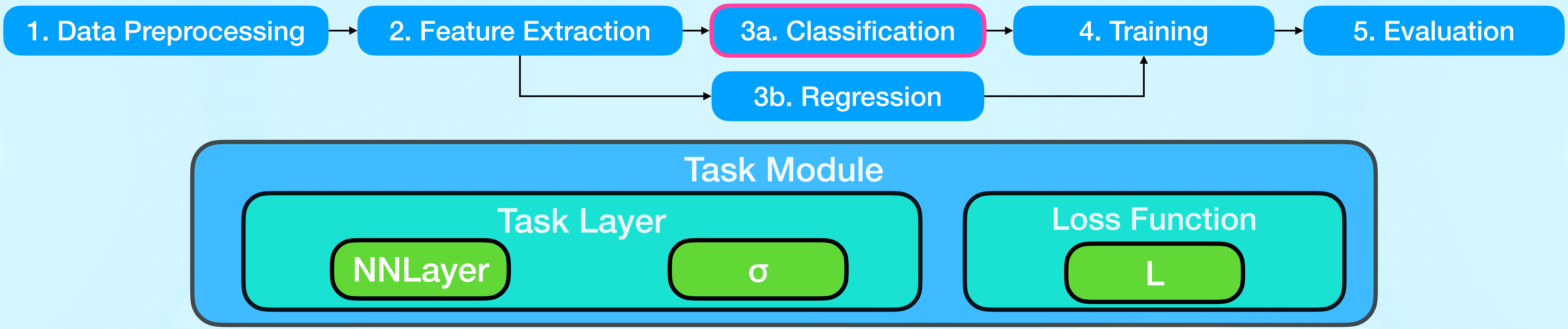
Cross Entropy measure the average amount of surprise from **network output distribution** $q(X)$**,** given that the **ground truth distribution** follows $p(X)$

Concept

Cross Entropy Loss: Minimize Cross Entropy is identical to minimizing the surprise of NN output with respect to the ground truth label

- When $p(\text{Waveform}) = \text{Neutrino}, \quad q(\text{Waveform}) \rightarrow \text{Neutrino}$
- When $p(\text{Waveform}) = \text{Background}, \quad q(\text{Waveform}) \rightarrow \text{Background}$

Lower Cross Entropy loss means better separation between neutrino signal and background



Concept

Binary Classification 1

Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

- After training, it can be onsidered as a **“classification score”:**
    - Higher score means the answer is more likely “yes”
    - Lower score means the answer is more likely “no”
    - A **threshold** is need to distinguish “yes” from “no”

σ: *torch.sigmoid(x)*

- Sigmoid function $\sigma(x) = 1/(1 + e^{x})$ maps input from [-inf, inf] to [0,1]

Loss Function: *torch.nn.BCELoss()*

- Input: has to be a number between [0,1]
- Target: has to be either 0 or 1, cannot be other number

Concept

Binary Classification 2

Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

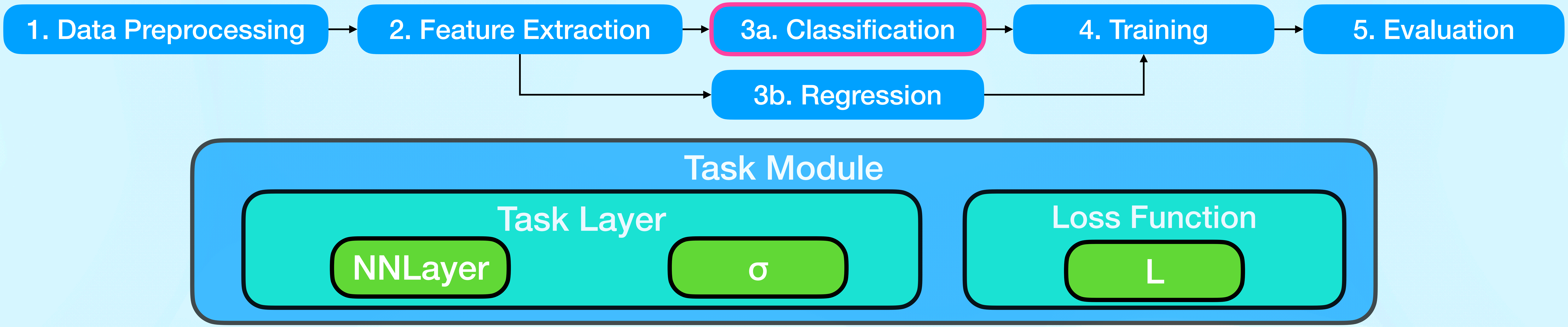
After training, it can be onsidered as a **“classification score”:**

- Higher score means the answer is more likely “yes”
- Lower score means the answer is more likely “no”
- A **threshold** is need to distinguish “yes” from “no”

σ: *None*

Loss Function: *torch.nn.BCEWithLogitsLoss()*

- Input: *any range between [-inf, inf]*
- Target: has to be either 0 or 1, cannot be other number



Concept

Binary Classification 3

Task Layer: *NNLayer(32, 2) → two float point No. between [-inf, inf]*

- After training, it can be considered as a **"classification decision"**:
    - Assign meaning to the two numbers (1st - yes, 2nd - no)
    - The larger number of the two represents the one we select
    - No **threshold** needed

σ: *None*

Loss Function: *torch.nn.CrossEntropyLoss()*

- Input: array of size 2 with two number between [-inf, inf]
- Target: the **array indices** of correct choice

Concept

Multiclass Classification

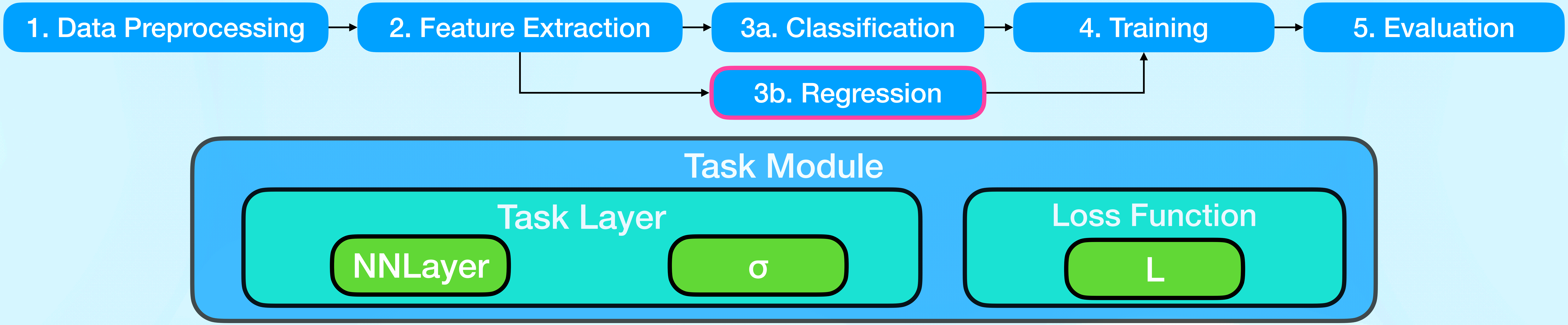
Task Layer: *NNLayer(32, <u>n</u>) → n float point No. between [-inf, inf]*

- After training, it can be considered as a **"classification decision"**
    - Assign meaning to the each numbers (1st - C1, 2nd - C2, 3rd - C3, ...)
    - The largest number represents the one we select
    - No **threshold** needed

σ: *None*

Loss Function: *torch.nn.CrossEntropyLoss()*

- Input: array of size <u>*n*</u> with two number between [-inf, inf]
- Target: the **array indices** of correct choice



Concept

Regression 1

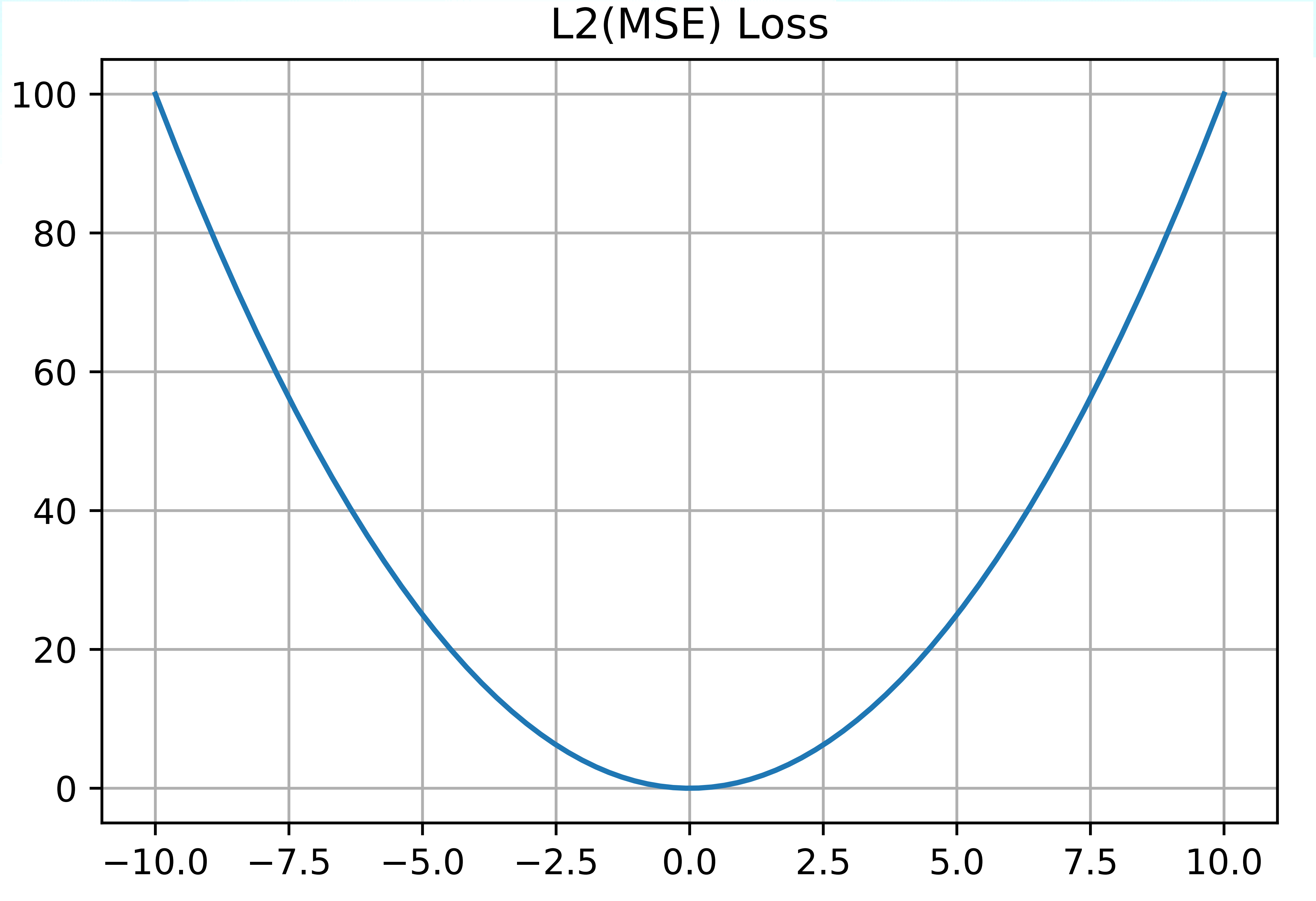
Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

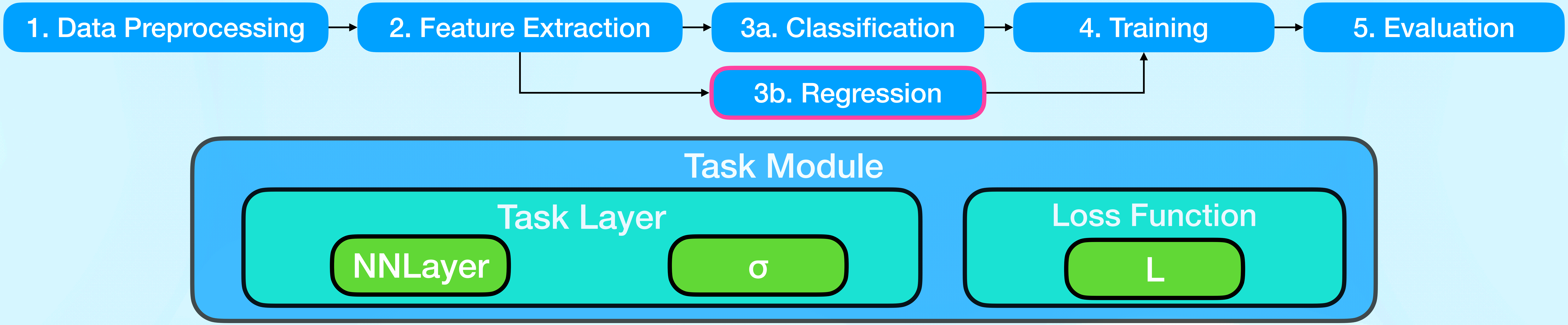
σ:

- None if you want to fit a physics quantity like energy
- torch.sigmoid(x) if you want to model a percentage like efficiency

Loss Function: *torch.nn.MSELoss()*

- $L = (\mathrm{TL\_Output} - \mathrm{Energy})^2$ for energy reconstruction
- Most commonly used, everywhere differentiable
- Gradient explosion with extreme values





Concept

Regression 2

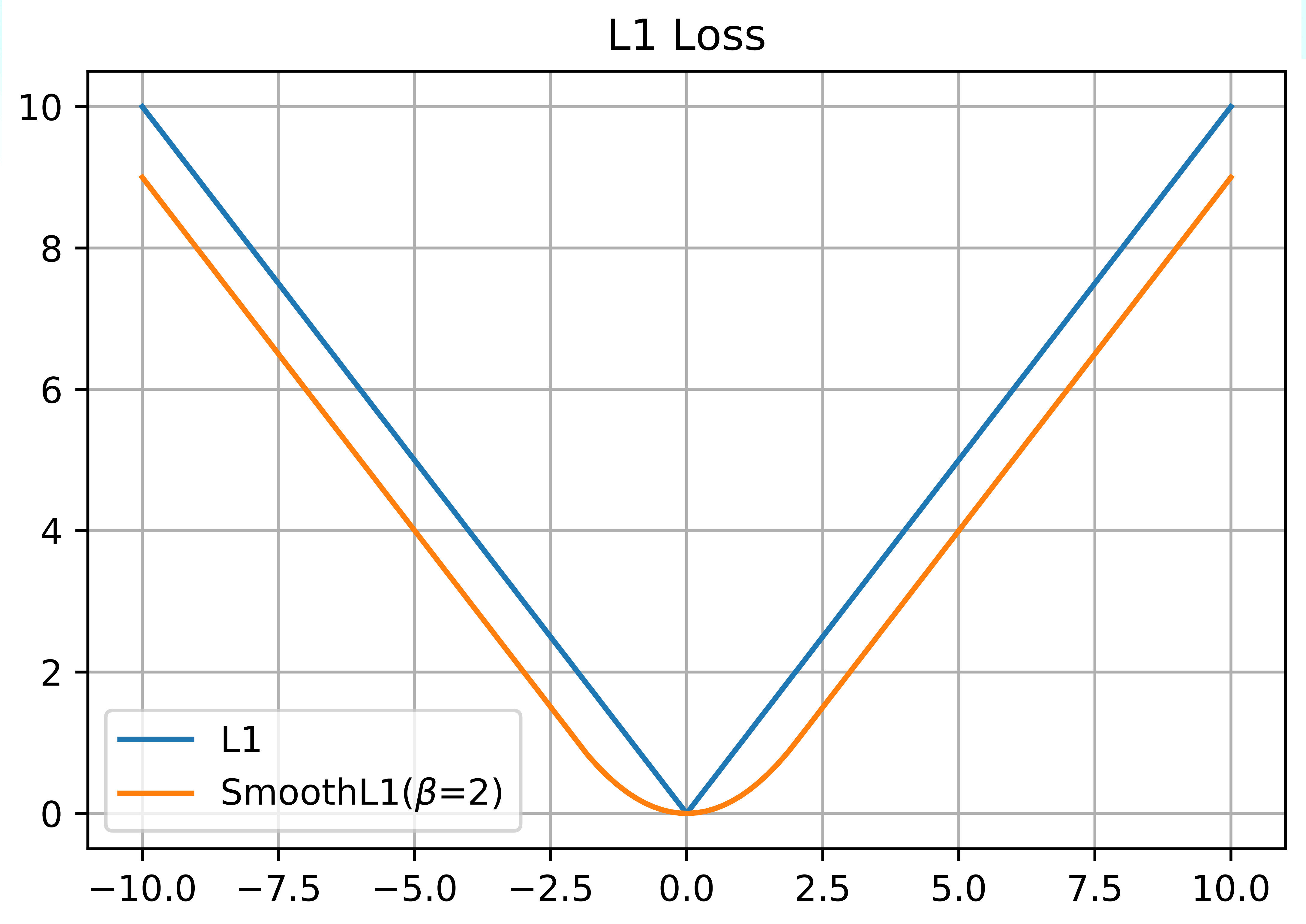
Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

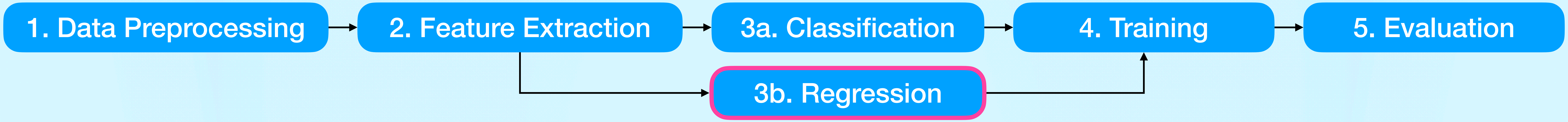
σ:

- None if you want to model a physics quantity like energy
- torch.sigmoid(x) if you want to model a percentage like efficiency

Loss Function: *torch.nn.L1Loss()*

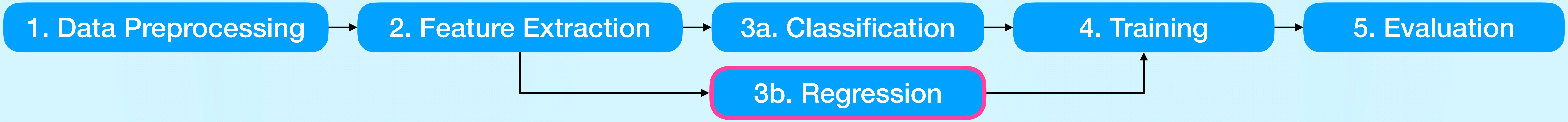
- $L = |\mathrm{TL\_Output} - \mathrm{Energy}|$ for energy reconstruction
- Generally more Robust, but not Not everywhere differentiable
- Sometimes replaced with *torch.nn.SmoothL1Loss()*





Code

```
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```



Code

Task Layer: output one float point number

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

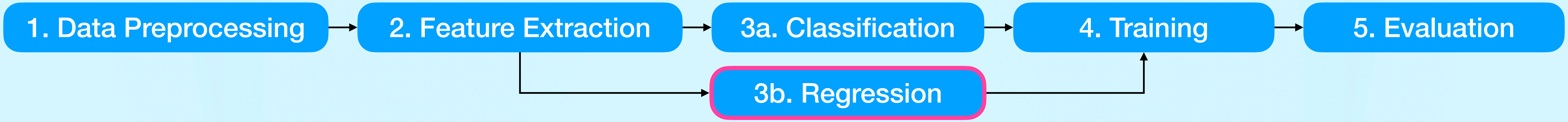
2. Feature Extraction → **3b. Regression** → 4. Training

Code

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```



Code

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

Initialize model as an regressor object

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

Initialize model as an regressor object

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

Initialize **Loss Function** as an object

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

Initialize model as an regressor object

Initialize **Loss Function** as an object

Feed waveform through the regressor object for output

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural 
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Code

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

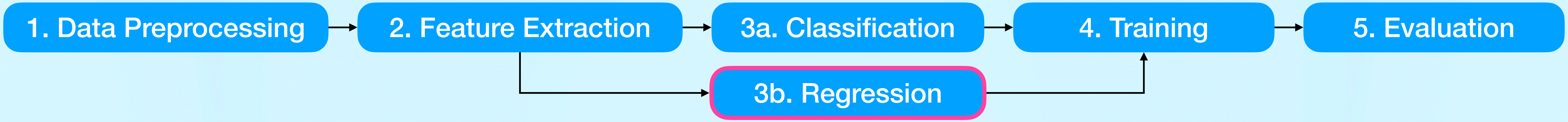
Initialize model as an regressor object

Calculate regression loss between regressor output and true energy

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural 
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

Initialize **Loss Function** as an object

Feed waveform through the regressor object for output



Code

```python
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()

        self.feature_extractor = #Already Discussed

        self.task_layer = torch.nn.Linear(32, 1)

    def forward(self, x):
        '''
        The forward operation of each training step of the neural
        '''
        x = self.feature_extractor(x)
        x = self.task_layer(x)
        #x = torch.sigmoid(x)

        return x

regressor = FCNet() # Change here to try different networks
criterion = torch.nn.L1Loss()

for i, (waveform, label, energy) in tqdm(enumerate(train_loader)):
    #Pull out 1 event from the dataset
    outputs  =  regressor (waveform)
    loss = criterion(outputs, energy)
```

Task Layer: output one float point number

σ: activation function, None for energy regression task since energy could be any values

Initialize model as an regressor object

Calculate regression loss between regressor output and true energy

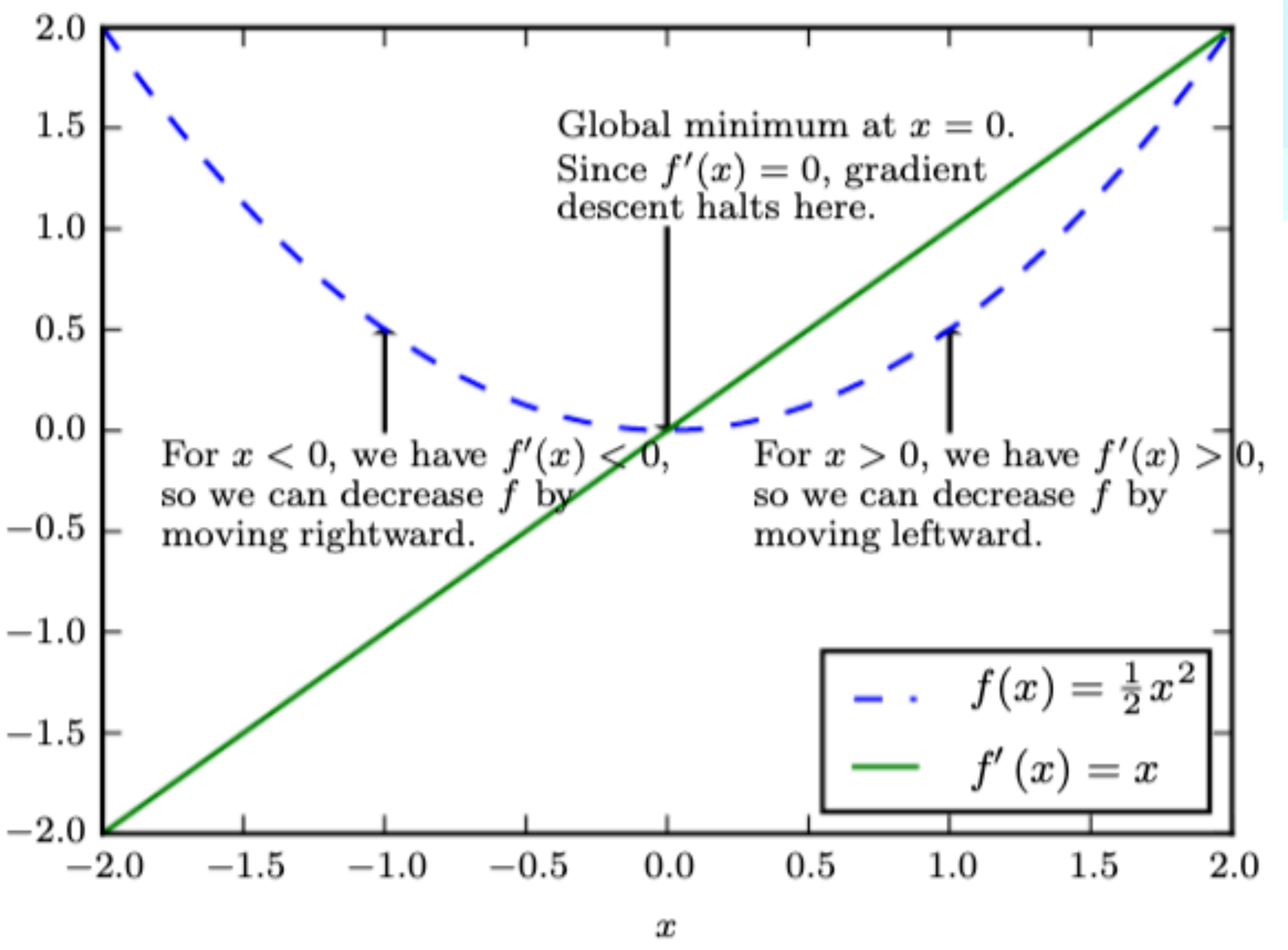
Replace **Task Layer**, **σ**, and **Loss Function** using the previously-provided concept templates to achieve different tasks!

Initialize **Loss Function** as an object

Feed waveform through the regressor object for output

Concept

- Training of neural network is a **gradient descent** optimization process:
  - To find $\underset{x}{\operatorname{argmax}} f(x)$, we calculate the gradient $\nabla_x f(x)$
  - Moving x in the opposite direction of gradient: $x' = x - \gamma \nabla_x f(x)$
  - γ is the **learning rate** of gradient descent



- Mainstream machine learning optimizers Adam, AdaGrad, RMSprop are first order optimization algorithms that only utilize gradient
- There are second order optimization algorithms which utilizes the Hessian matrix to avoid poorly conditioned probability space
  - $f(x^{(0)} - \gamma g) = f(x^{(0)}) - \gamma g^T g + \frac{1}{2}\gamma^2 g^T H g$

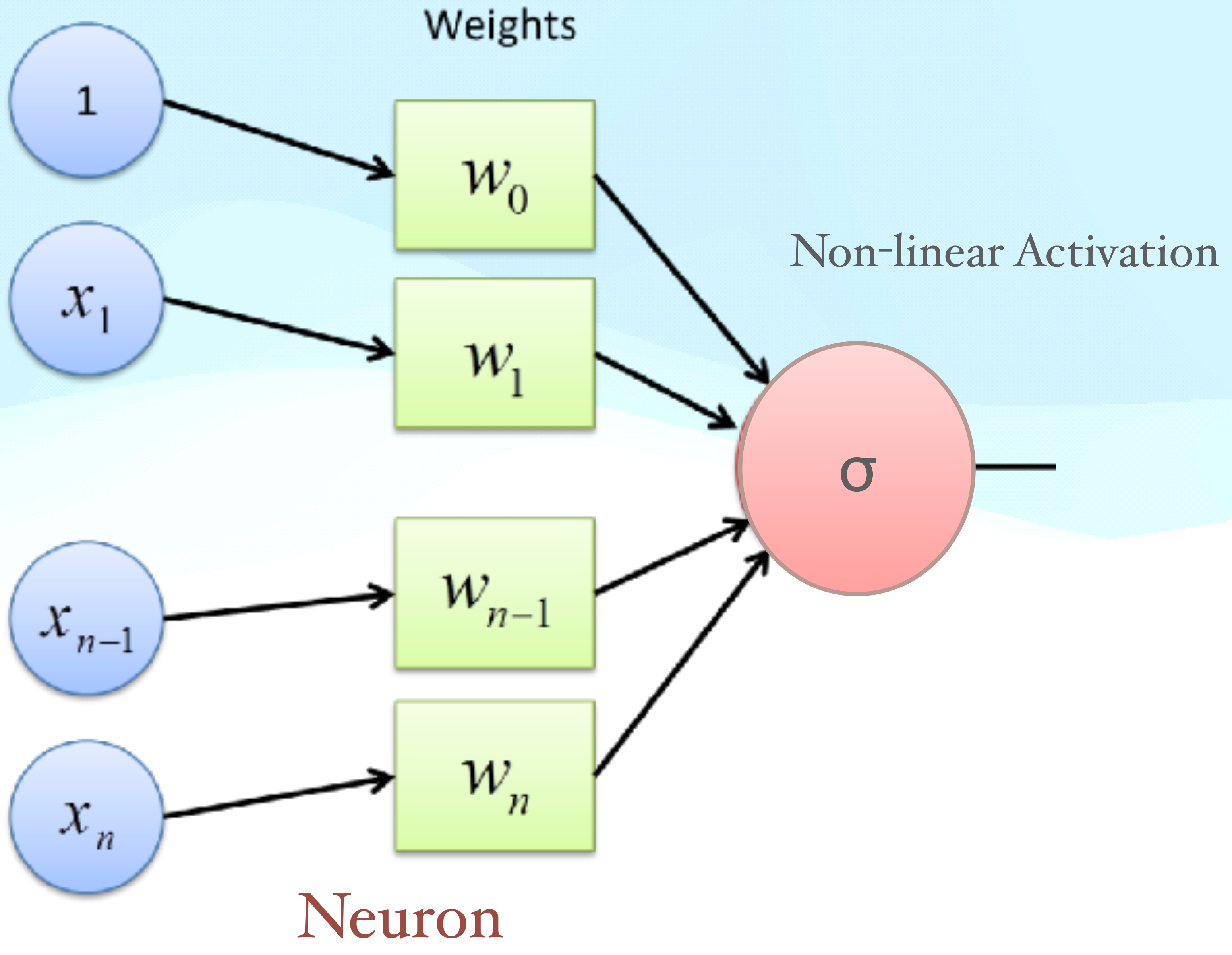
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Theory

Expression of Neural Network:

- One Layer: $\hat{y} = \sigma(\boxed{\alpha_m^T}\vec{x}^i + \boxed{\alpha_{0m}})$
- Two Layer: $\hat{y} = \sigma_2(\boxed{\beta_k^T}\boxed{\sigma(\alpha_m^T\vec{x}^i + \alpha_{0m})} + \boxed{\beta_{0k}})$
- (α_m^T, β_k^T) are the **weight matrices** of the (m^{th}, k^{th}) neuron, their values change during the training
- (α_m^T, β_k^T) is also defined as the **kernel** of neural network



Training NN = Optimizing kernel parameters (α_m^T, β_k^T) with respect to the loss function

- Calculating the gradient $\nabla_{\alpha,\beta} L(x, y \,|\, \alpha, \beta)$ is the key steps

However, the gradient is supposed to be calculated on all input data simultaneously

As we increase the size of training data, this becomes computationally impossible

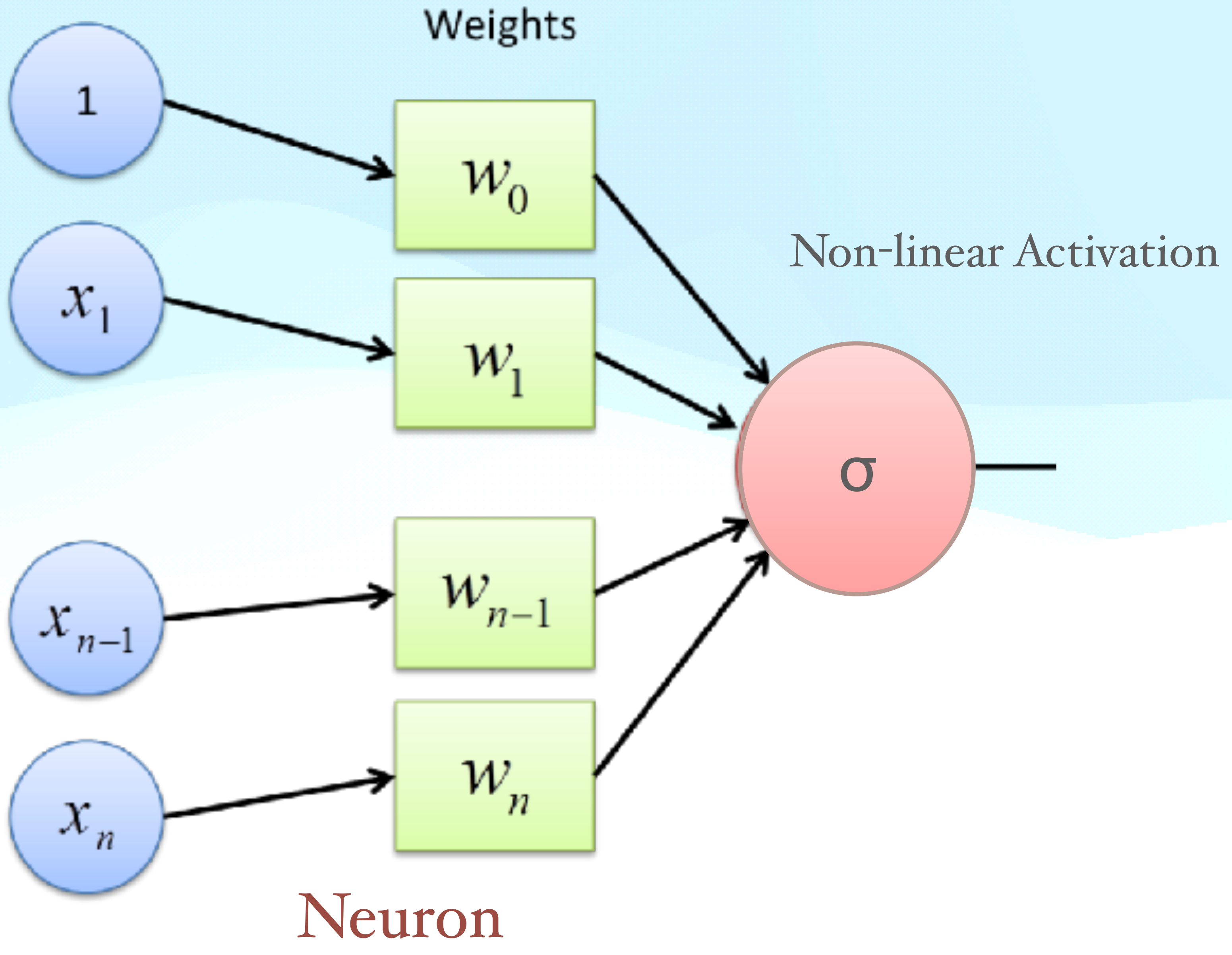
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Theory

Expression of Neural Network:

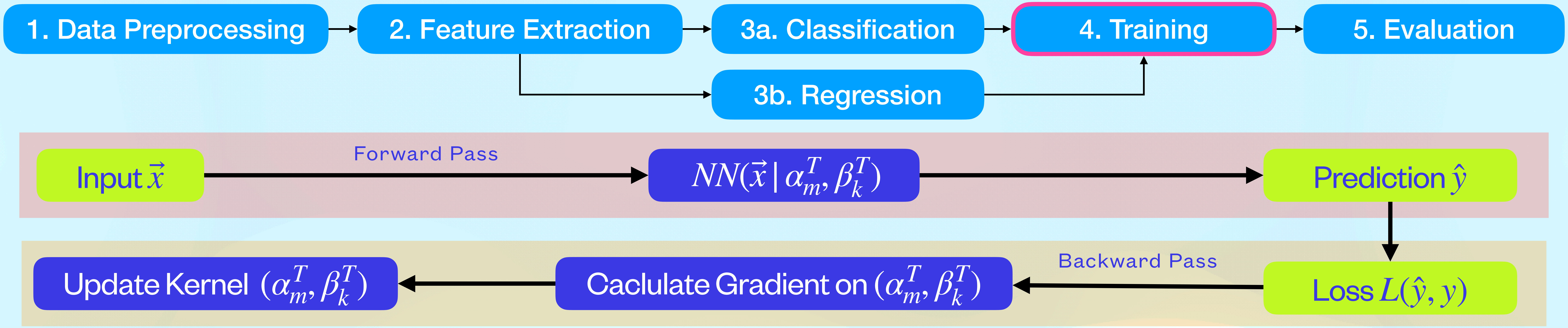
- One Layer: $\hat{y} = \sigma(\boxed{\alpha_m^T}\vec{x}^i + \boxed{\alpha_{0m}})$
- Two Layer: $\hat{y} = \sigma_2(\boxed{\beta_k^T}\boxed{\sigma(\alpha_m^T\vec{x}^i + \alpha_{0m})} + \boxed{\beta_{0k}})$
- (α_m^T, β_k^T) are the **weight matrices** of the (m^{th}, k^{th}) neuron, their values change during the training
- (α_m^T, β_k^T) is also defined as the **kernel** of neural network



Concept

Stochastic Gradient Descent (SGD):

- **Stochastic:** breaks the training data into **batches,** each batch is randomly sampled from the training dataset
- Size of batch is an important hyperparameter of NN model
- (α_m^T, β_k^T) is updated in a step-wise manner by sequentially feeding each batch into the model
- This will reproduce the effect of training using all data simultaneously
- SGD is possible because gradient is an expectation



Forward Pass:

- produce $\hat{y}$ from input $\vec{x}$ and kernel (α_m^T, β_k^T)

Concept

Backward Pass:

- Calculate loss $L(\hat{y}, y)$
- Calculate the gradient on kernel using back propagation
- Update kernel by gradient descent

Treating $NN(\vec{x} \mid \alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

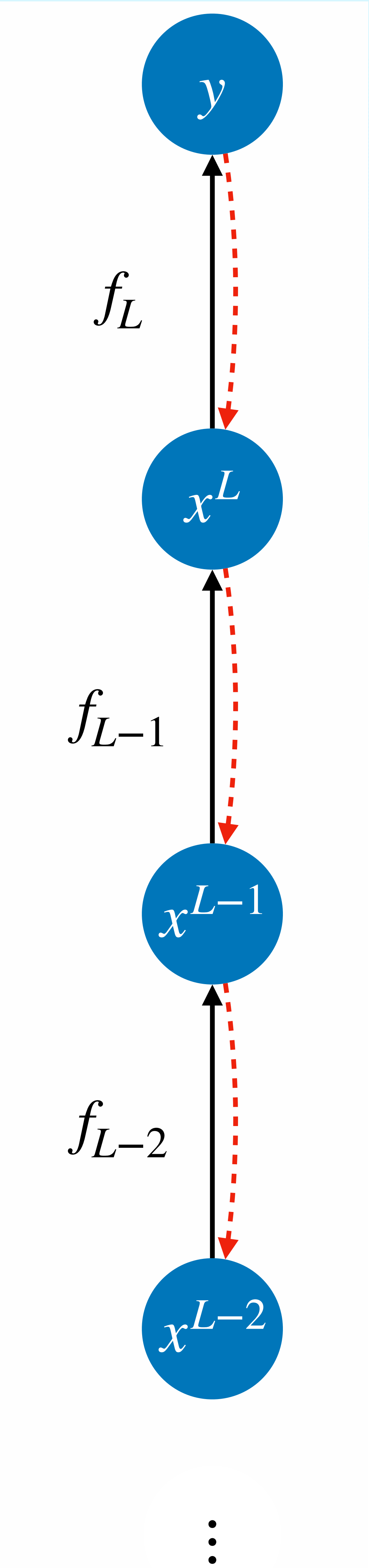
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\dfrac{\partial y}{\partial x_l}$ for all $l \in \{1, \ldots., L\}$



1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Treating $NN(\vec{x} \mid \alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

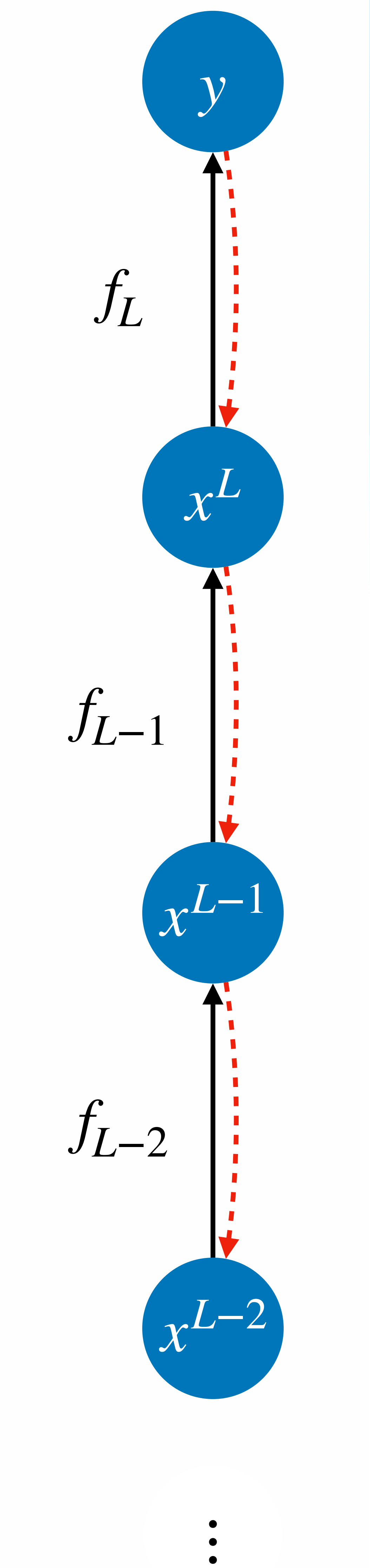
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\frac{\partial y}{\partial x_l}$ for all $l \in \{1,....,L\}$



$$\frac{\partial y}{\partial x_L}$$

$$\frac{\partial y}{\partial x_{L-1}} = \frac{\partial y}{\partial x_L} \frac{\partial x_L}{\partial x_{L-1}}$$

$$\frac{\partial y}{\partial x_{L-2}} = \frac{\partial y}{\partial x_{L-1}} \frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \frac{\partial y}{\partial x_{L-2}} \frac{\partial x_{L-2}}{\partial x^{L-3}}$$

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → **4. Training** → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Treating $NN(\vec{x}\,|\,\alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

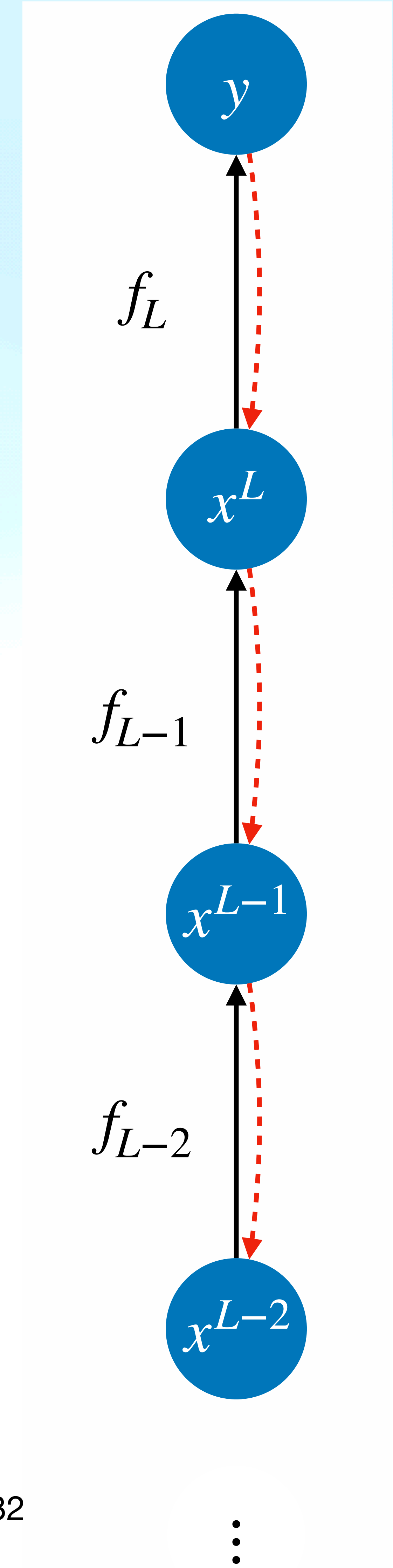
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\dfrac{\partial y}{\partial x_l}$ for all $l \in \{1,\ldots.,L\}$



$$\boxed{\frac{\partial y}{\partial x_L}}$$

$$\frac{\partial y}{\partial x_{L-1}} = \boxed{\frac{\partial y}{\partial x_L}}\frac{\partial x_L}{\partial x_{L-1}}$$

$$\frac{\partial y}{\partial x_{L-2}} = \frac{\partial y}{\partial x_{L-1}}\frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \frac{\partial y}{\partial x_{L-2}}\frac{\partial x_{L-2}}{\partial x^{L-3}}$$

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Treating $NN(\vec{x} \,|\, \alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

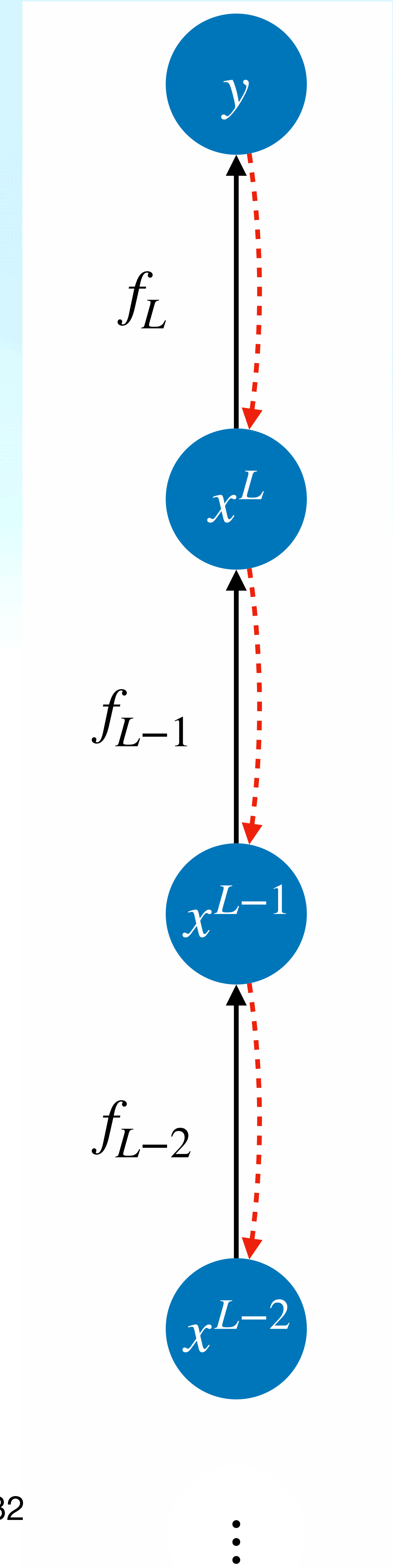
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\dfrac{\partial y}{\partial x_l}$ for all $l \in \{1,\ldots,L\}$



$$\boxed{\frac{\partial y}{\partial x_L}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-1}}} = \boxed{\frac{\partial y}{\partial x_L}} \frac{\partial x_L}{\partial x_{L-1}}$$

$$\frac{\partial y}{\partial x_{L-2}} = \boxed{\frac{\partial y}{\partial x_{L-1}}} \frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \frac{\partial y}{\partial x_{L-2}} \frac{\partial x_{L-2}}{\partial x^{L-3}}$$

Treating $NN(\vec{x} \,|\, \alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

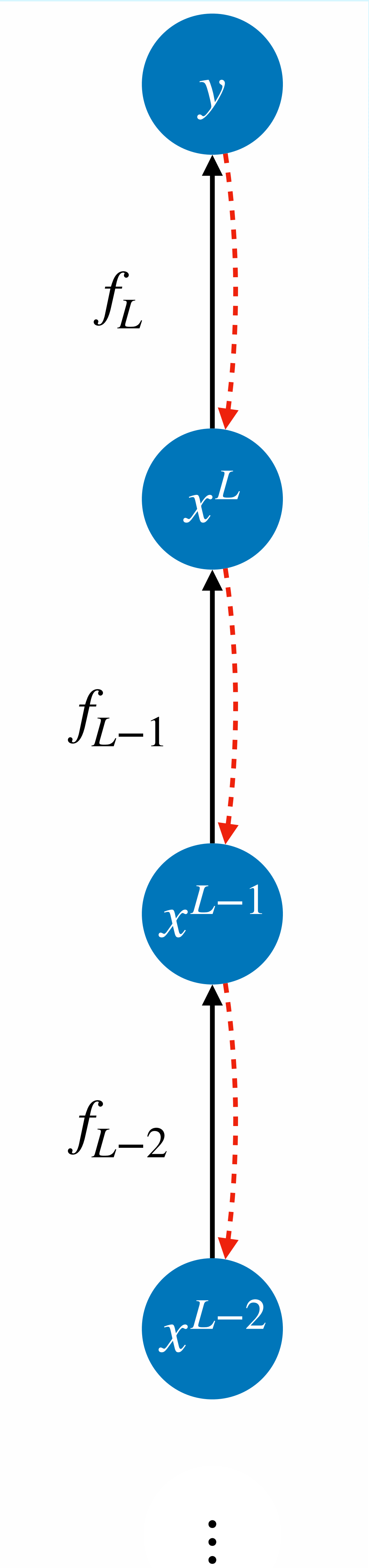
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\dfrac{\partial y}{\partial x_l}$ for all $l \in \{1,\ldots.,L\}$



$$\boxed{\frac{\partial y}{\partial x_L}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-1}}} = \boxed{\frac{\partial y}{\partial x_L}} \frac{\partial x_L}{\partial x_{L-1}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-2}}} = \boxed{\frac{\partial y}{\partial x_{L-1}}} \frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \boxed{\frac{\partial y}{\partial x_{L-2}}} \frac{\partial x_{L-2}}{\partial x^{L-3}}$$

Treating $NN(\vec{x} \,|\, \alpha_m^T, \beta_k^T)$ as a multilayer complex function, we can use **chain rule** to calculate its derivative:

$$y = f_L(\ldots f_2(f_1(x)))$$

$$\downarrow$$

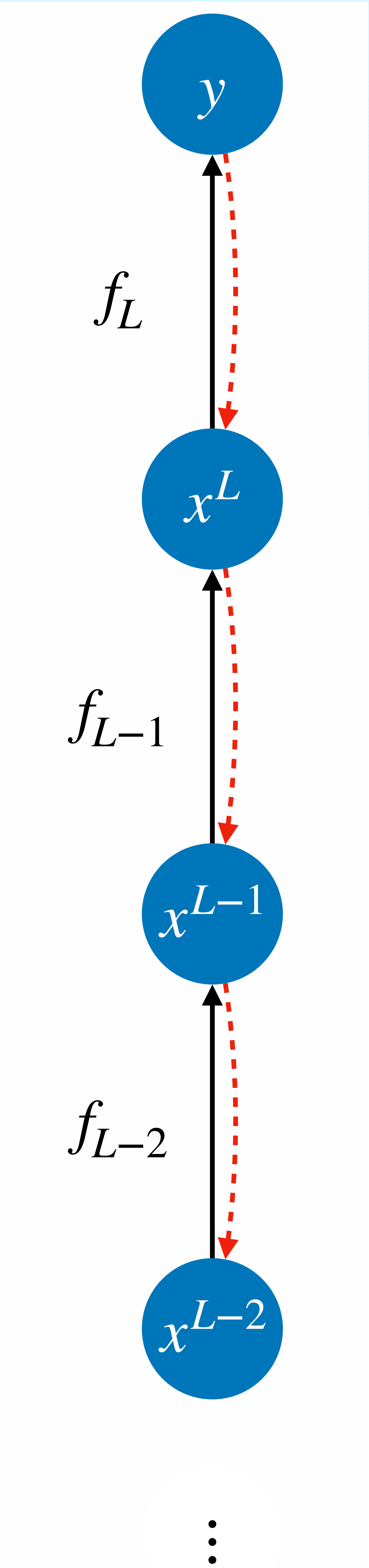
$$y = f_L(x_L)$$

$$X_L = f_{L-1}(x_{L-1})$$

$$\vdots$$

$$X_1 = f_1(x_1)$$

We wan to compute $\dfrac{\partial y}{\partial x_l}$ for all $l \in \{1,\ldots.,L\}$



$$\boxed{\frac{\partial y}{\partial x_L}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-1}}} = \boxed{\frac{\partial y}{\partial x_L}} \frac{\partial x_L}{\partial x_{L-1}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-2}}} = \boxed{\frac{\partial y}{\partial x_{L-1}}} \frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \boxed{\frac{\partial y}{\partial x_{L-2}}} \frac{\partial x_{L-2}}{\partial x^{L-3}}$$

At each step we can reuse the computation of the previous step!

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Concept

A **computation graph** is a directed graph where on each node we have an operation.

$$y = \sin(w_1 x + \log(x)) + \cos(x)$$

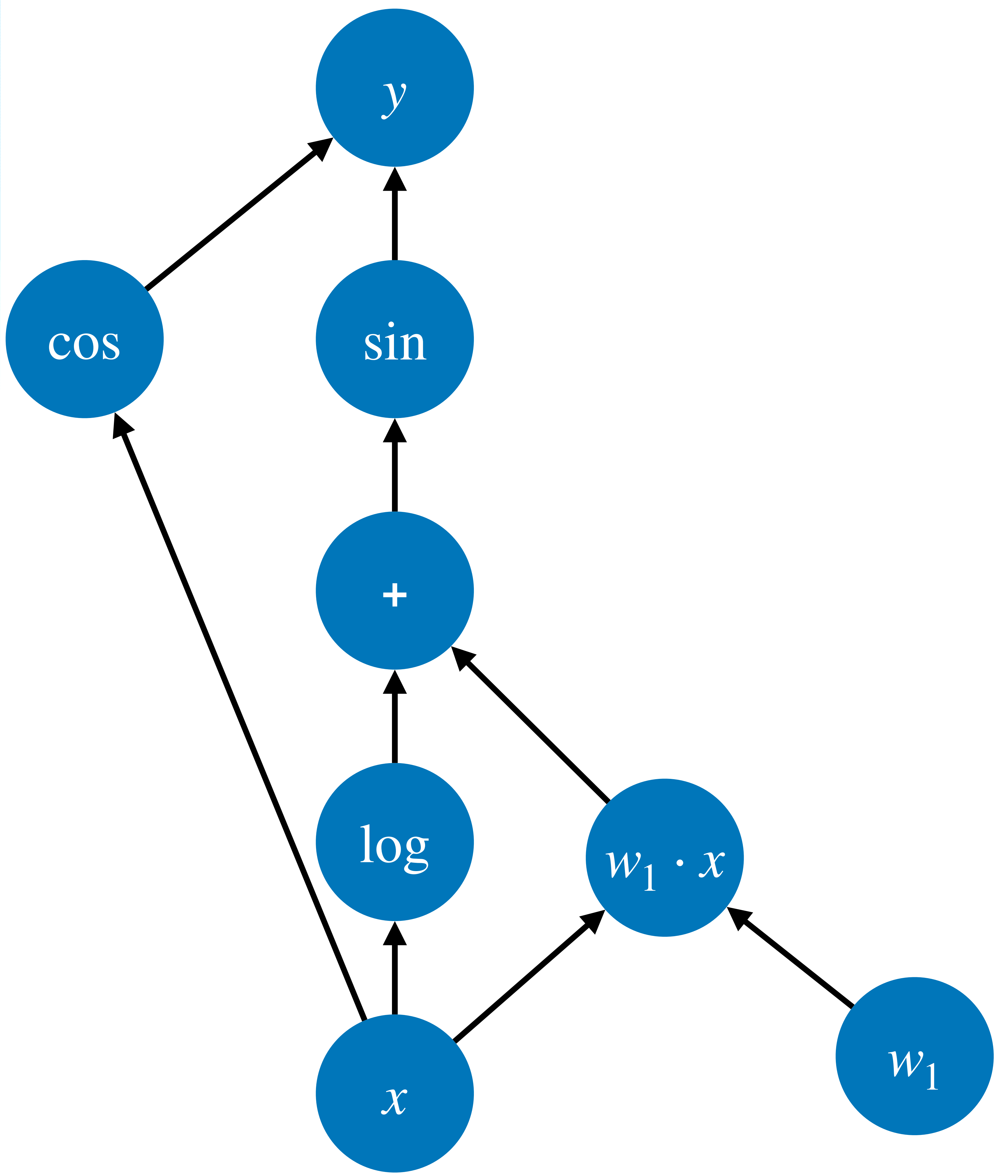
Modern deep neural networks are also computational graph! This means can calculate gradient along the graph.

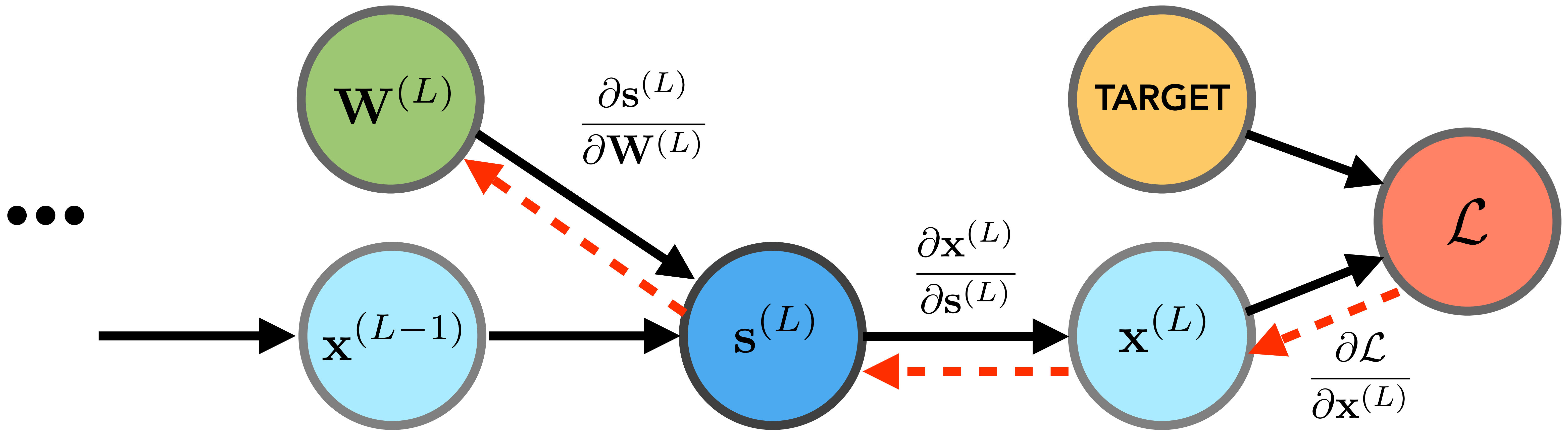
most deep learning software libraries automatically handle this for you

TensorFlow PYTORCH mxnet Keras Caffe2

and many more

just build the computational graph and define the loss





Theory

Given value of the loss, using **backpropagation equation** to calculate gradient with respect to each weight kernel parameters

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

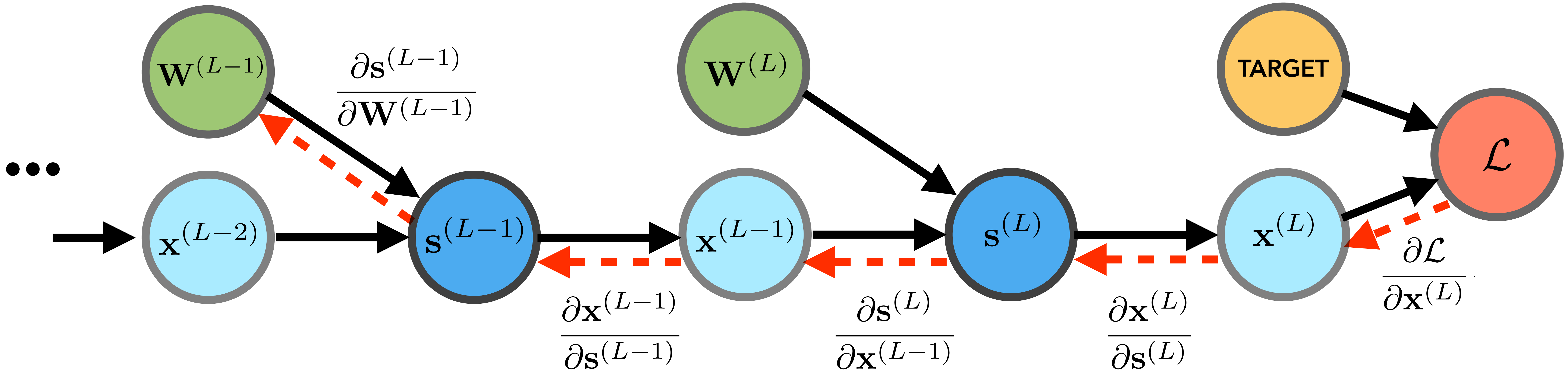
derivative of the non-linearity

$$\frac{\partial}{\partial \mathbf{W}^{(L)}}(\mathbf{W}^{(L)\intercal}\mathbf{x}^{(L-1)})$$
$$= \mathbf{x}^{(L-1)\intercal}$$

note $\nabla_{\mathbf{W}^{(L)}}\mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

now let's go back one more layer…

again we'll draw the dependency graph:



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

Code

```python
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Set up variables:

- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Set up variables:
- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Epoch: number of iterations to train through the same dataset
train_loader: iterate through each batches within the dataset

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → **4. Training** → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Set up variables:

- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Epoch: number of iterations to train through the same dataset
train_loader: iterate through each batches within the dataset

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

Forward pass: feed data through the neural network model and calculate loss value (already discussed)

Set up variables:
- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Epoch: number of iterations to train through the same dataset
train_loader: iterate through each batches within the dataset

Backpropagation: this single line of code does everything we discussed from sides 28-35!

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

Forward pass: feed data through the neural network model and calculate loss value (already discussed)

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Set up variables:

- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Epoch: number of iterations to train through the same dataset

train_loader: iterate through each batches within the dataset

Backpropagation: this single line of code does everything we discussed from sides 28-35!

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

Forward pass: feed data through the neural network model and calculate loss value (already discussed)

This line updates model parameters according to calculated gradients

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Set up variables:

- **Model:** classifier
- **Loss Function:** criterion
- **Optimizer:** can use SGD, but we recommend torch.optim.Adam()]

Epoch: number of iterations to train through the same dataset

train_loader: iterate through each batches within the dataset

Backpropagation: this single line of code does everything we discussed from sides 28-35!

This line reset the gradient to 0 at the end of current batch, so that we are ready to train the next batch of data

Code

```
classifier, criterion, optimizer = set_up_classifier()
train_loader, test_loader = get_dataloader()

loss_values = []
accuracy_values = []
y_true = []
y_pred = []

for epoch in range(NUM_EPOCHS):
    for i, (waveform, labels, energies) in tqdm(enumerate(train_loader)):
        classifier.train() # This line set the neural network to train mode,

        waveform = waveform.to(DEVICE).float()
        labels = labels.to(DEVICE).view(-1,1).float()

        #Train the RNN classifier
        outputs  = classifier(waveform).view(-1,1)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Back-propagate loss to update gradient
        loss.backward()

        # Perform gradient descent to update parameters
        optimizer.step()

        # reset gradient to 0 on all parameters
        optimizer.zero_grad()

    print('\rEpoch [{0}/{1}], Iter [{2}/{3}] Loss: {4:.4f}'.format(
        epoch+1, NUM_EPOCHS, i+1, len(train_loader),
        loss.item(), end=""),end="")
    loss_values.append(loss.item())
```

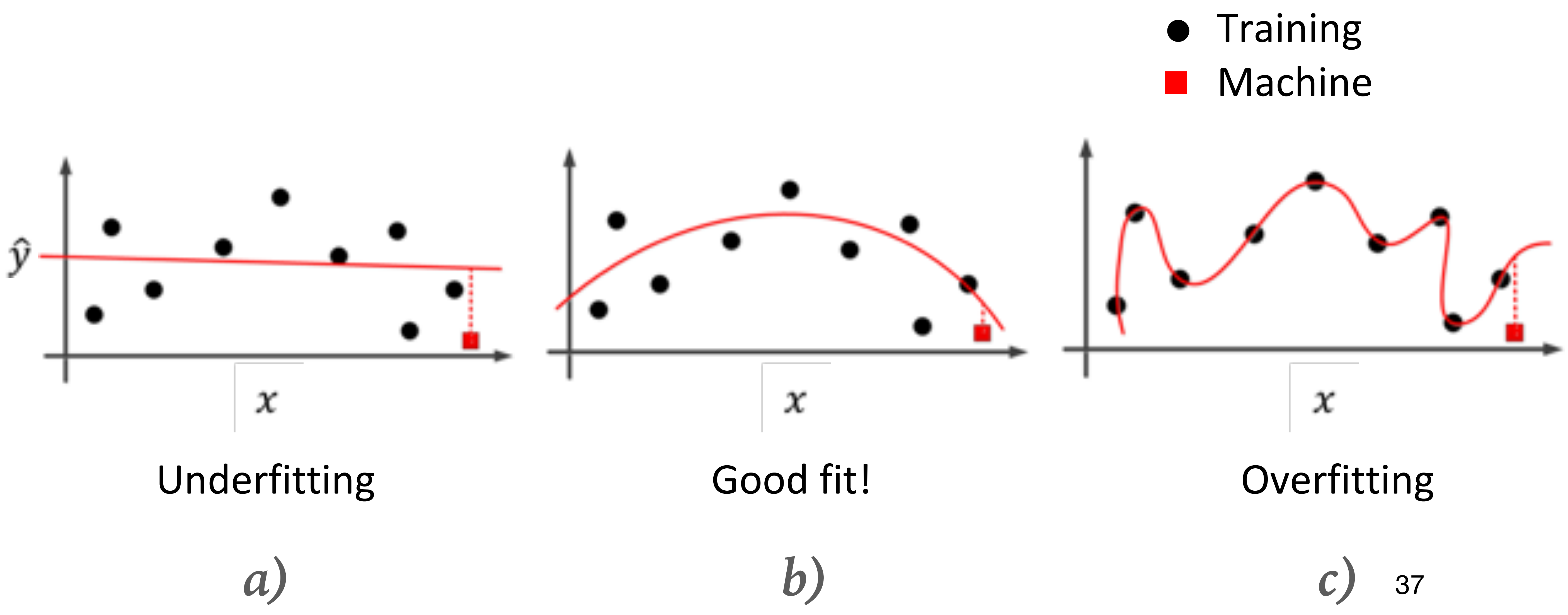
Forward pass: feed data through the neural network model and calculate loss value (already discussed)

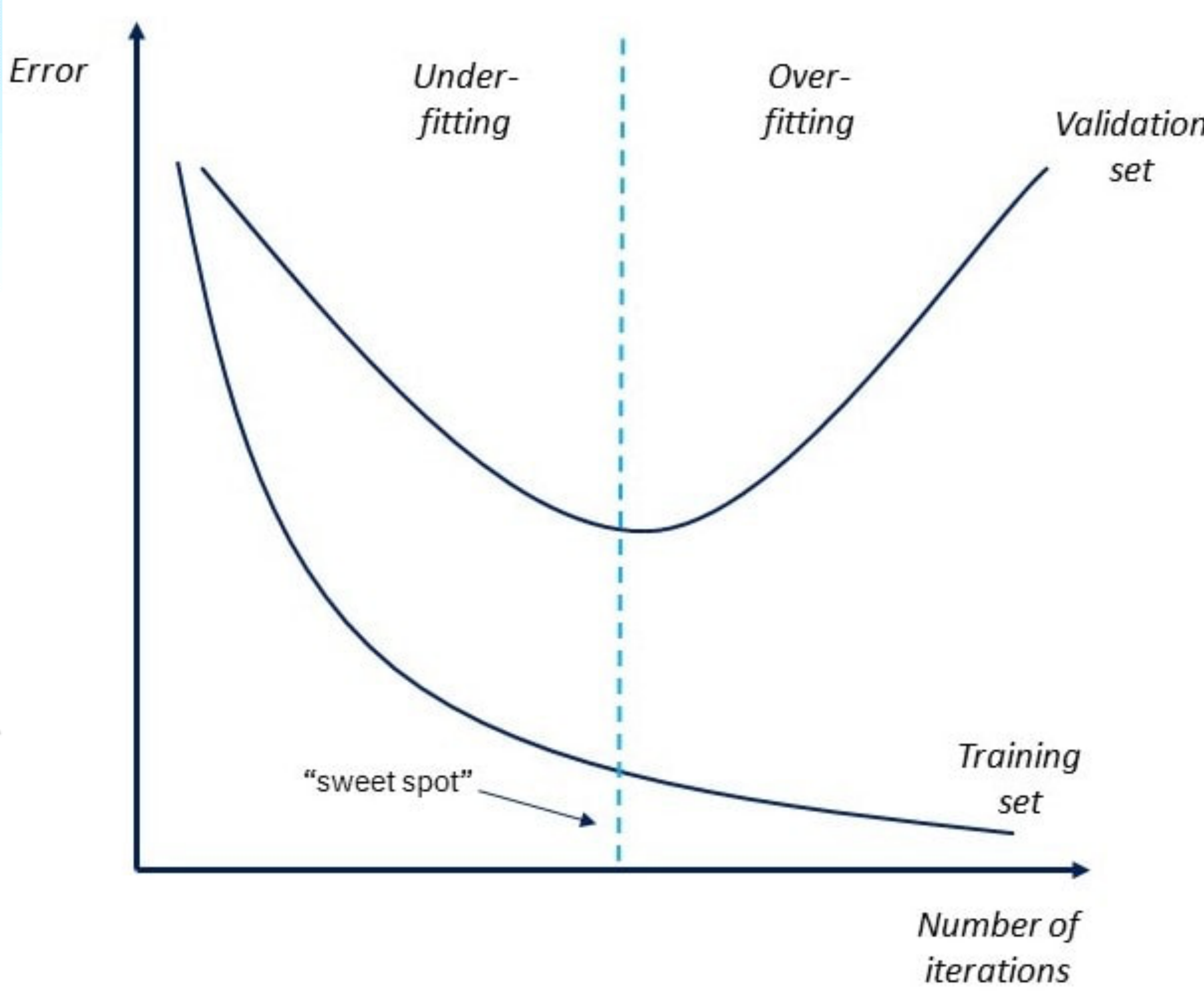
This line updates model parameters according to calculated gradients

Concept

Underfitting: occurs when model cannot adequately capture the underlying structure of the data

Overfitting: model that corresponds too closely/exactly to the training data and fail to generalize to validation data



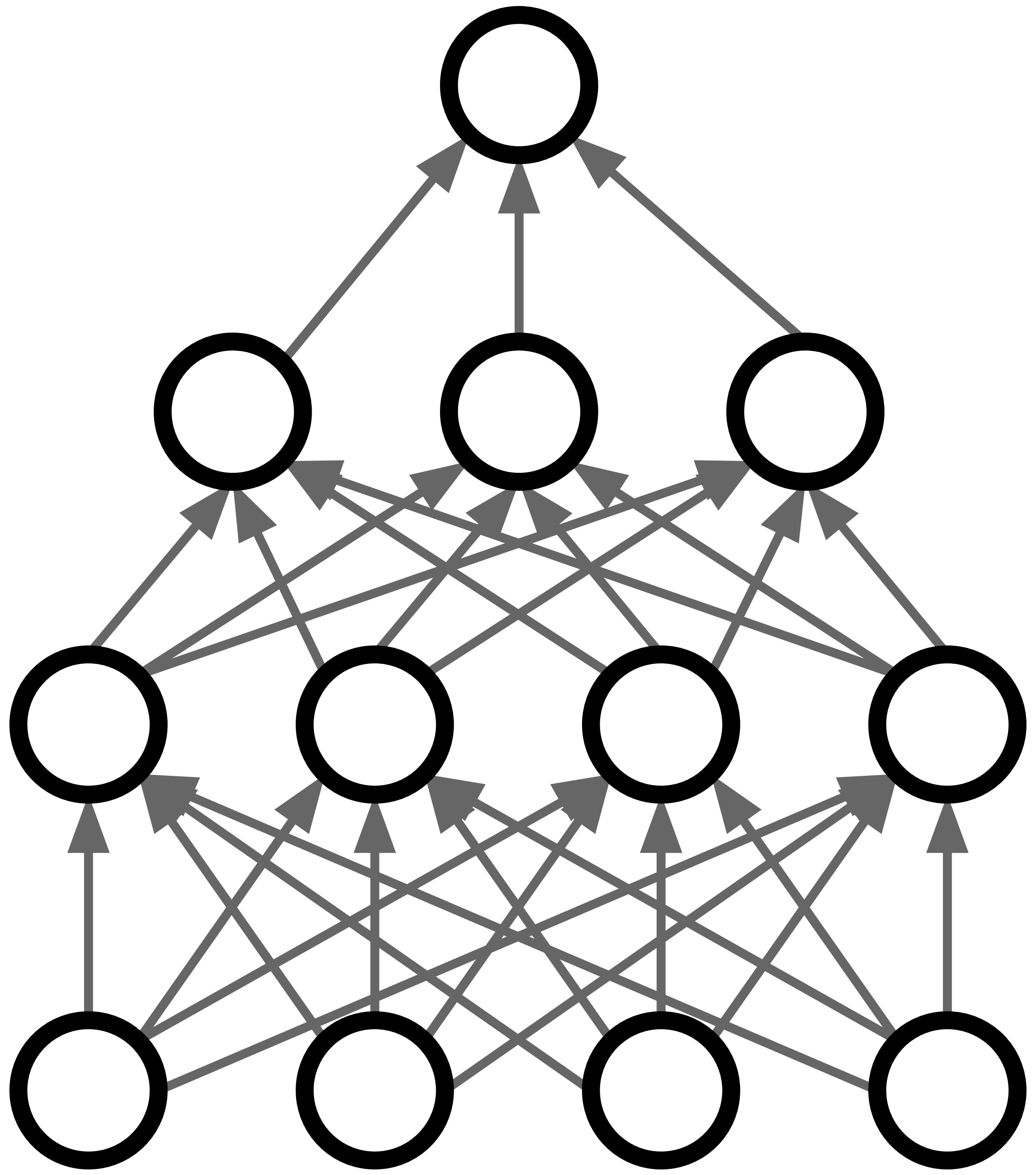


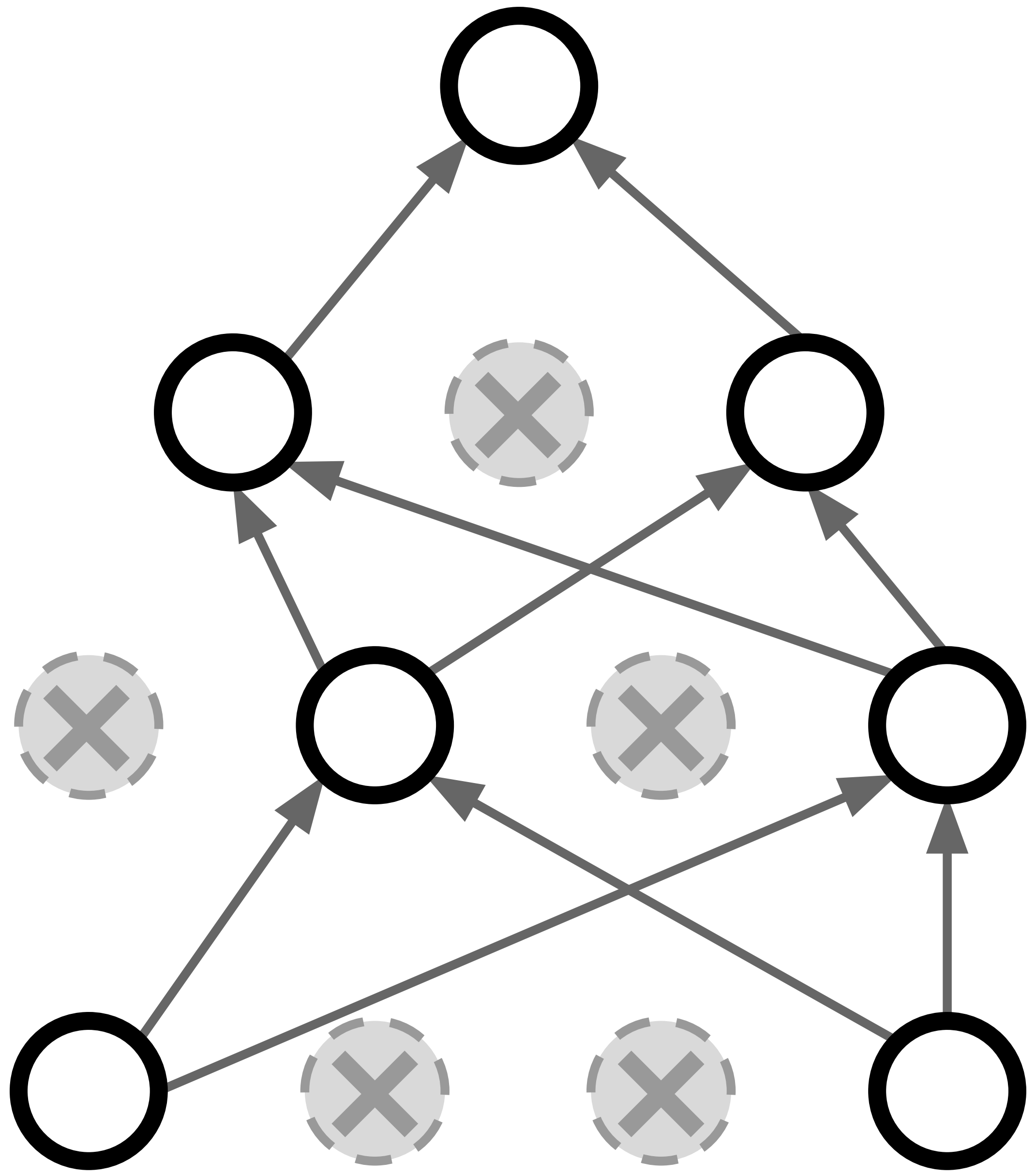
Concept

Dropout: during training, randomly set some neurons to zero which forces the NN not to become solely dependent on one neuron.

Code

```
torch.nn.Linear(1000,512),
torch.nn.ReLU(),
torch.nn.Dropout(),
```





1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Concept

Binary Classification 1

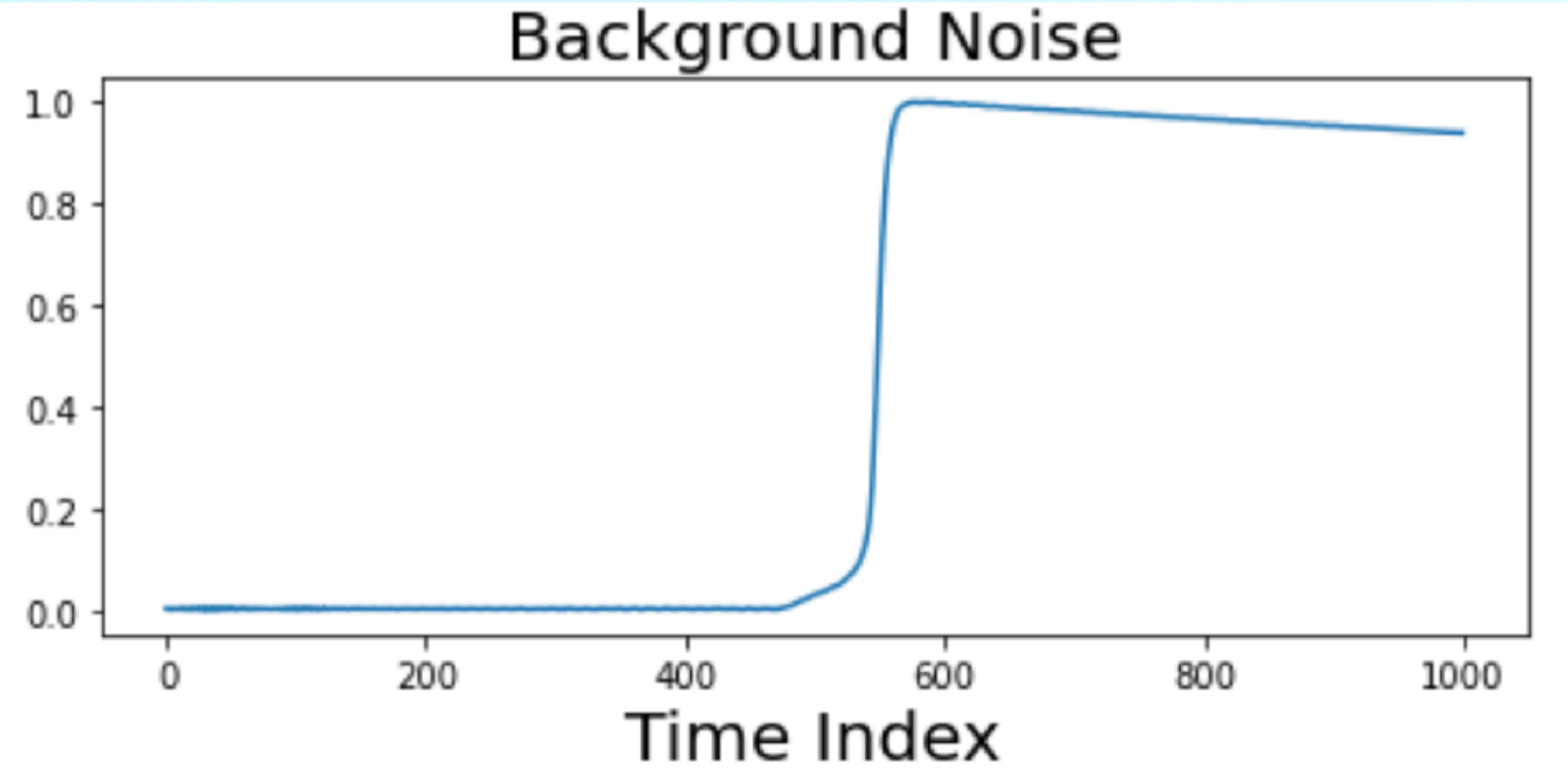
Task Layer: *NNLayer(32, 1) → One float point No.*

- After training, it can be onsidered as a **“classification score”:**
  - Higher score means the answer is more likely “yes”
  - Lower score means the answer is more likely “no”
  - A **threshold** is need to distinguish “yes” from “no”

σ: *torch.sigmoid(x)*

Loss Function: *torch.nn.BCELoss()*

Time Series Data
(BATCH_SIZE, 1000)



Concept

NNLayer(1000,512) → σ

→ NNLayer(512, 256) → σ

→ NNLayer(256, 32) → σ

Concept

Binary Classification 1

Task Layer: *NNLayer(32, 1) → One float point No.*

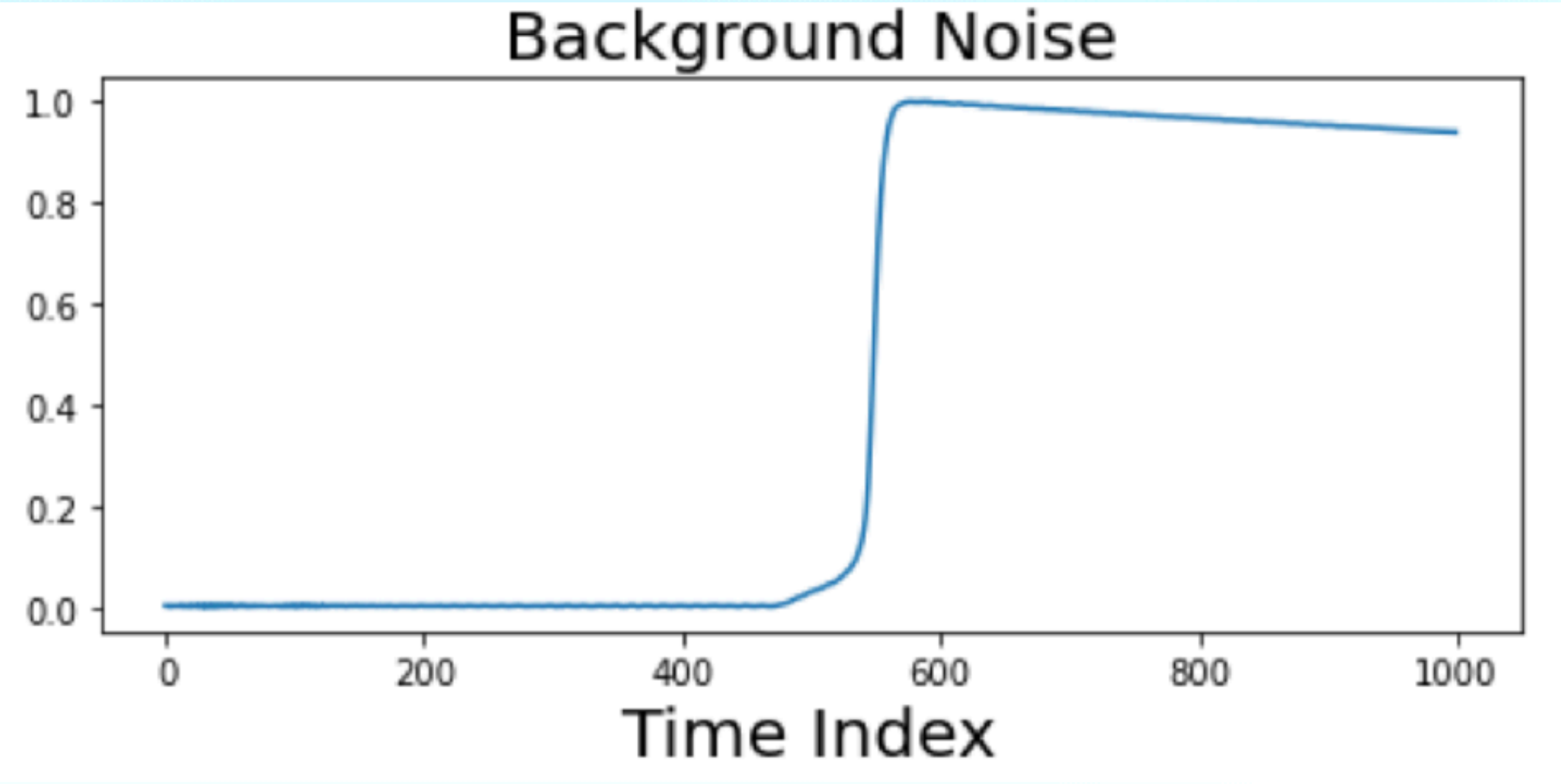
- After training, it can be onsidered as a **“classification score”:**
  - Higher score means the answer is more likely “yes”
  - Lower score means the answer is more likely “no”
  - A **threshold** is need to distinguish “yes” from “no”

σ: *torch.sigmoid(x)*

Loss Function: *torch.nn.BCELoss()*

Time Series Data

(BATCH_SIZE, 1000)



Concept

Feature Extractor Network

Take raw data as the input and output a low-dimensional vector

NNLayer(1000,512) → σ

→ NNLayer(512, 256) → σ

→ NNLayer(256, 32) → σ

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	**psd_label_avse: True Background**
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

____________ means this event is _______ly classified as __________.

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	psd_label_avse: True Background
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

True Positive means this event is Tru ly classified as Positive.

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → **5. Evaluation**

2. Feature Extraction → 3b. Regression → 4. Training

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
  - $\lambda > 0.5$: event is a signal
  - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	psd_label_avse: True Background
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

True Positive means this event is **Tru**ly classified as **Positive**.

False Positive **False** **Positive**

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	psd_label_avse: True Background
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

True Positive means this event is **Tru**ly classified as **Positive**.

False Positive — **False** — **Positive**

True Negative — **Tru** — **Negative**

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
  - $\lambda > 0.5$: event is a signal
  - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	psd_label_avse: True Background
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

True Positive	means this event is	**Tru**	ly classified as	**Positive**	.
False Positive		**False**		**Positive**	
True Negative		**Tru**		**Negative**	
False Negative		**False**		**Negative**	

Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
  - $\lambda > 0.5$: event is a signal
  - $\lambda \leq 0.5$: event is a background

	psd_label_avse: True Signal	psd_label_avse: True Background
Classified As Signal	True Positive (TP)	False Positive (FP)
Classified as bkg	False Negative (FN)	True Negative (TN)

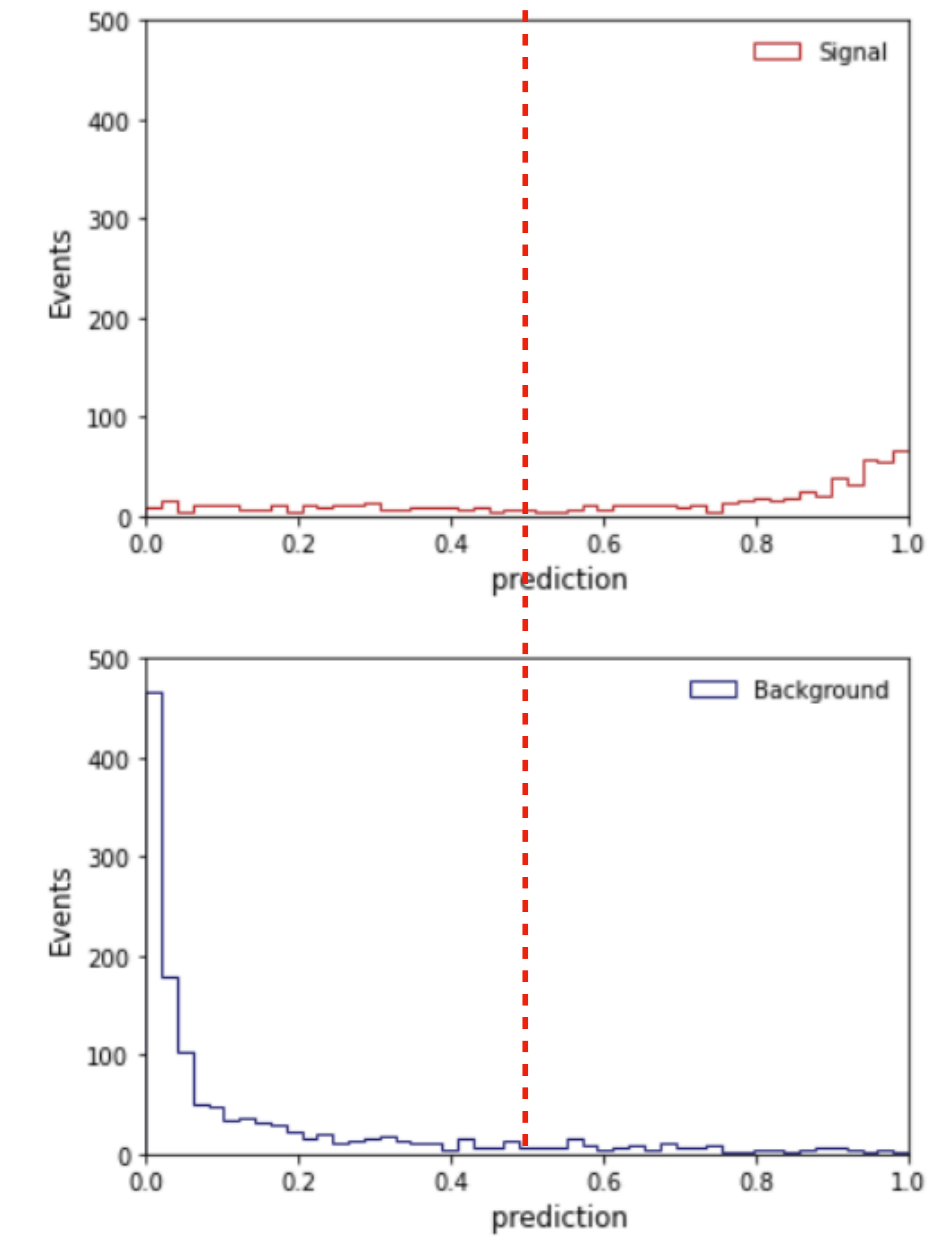
Concept

True Positive Rate (TPR) $= \dfrac{TP}{TP + FN}$ [TP+FN is the total number of signal in the dataset]

False Positive Rate (FPR) $= \dfrac{FP}{FP + TN}$ [FP+TN is the total number of backgrounds in the dataset]

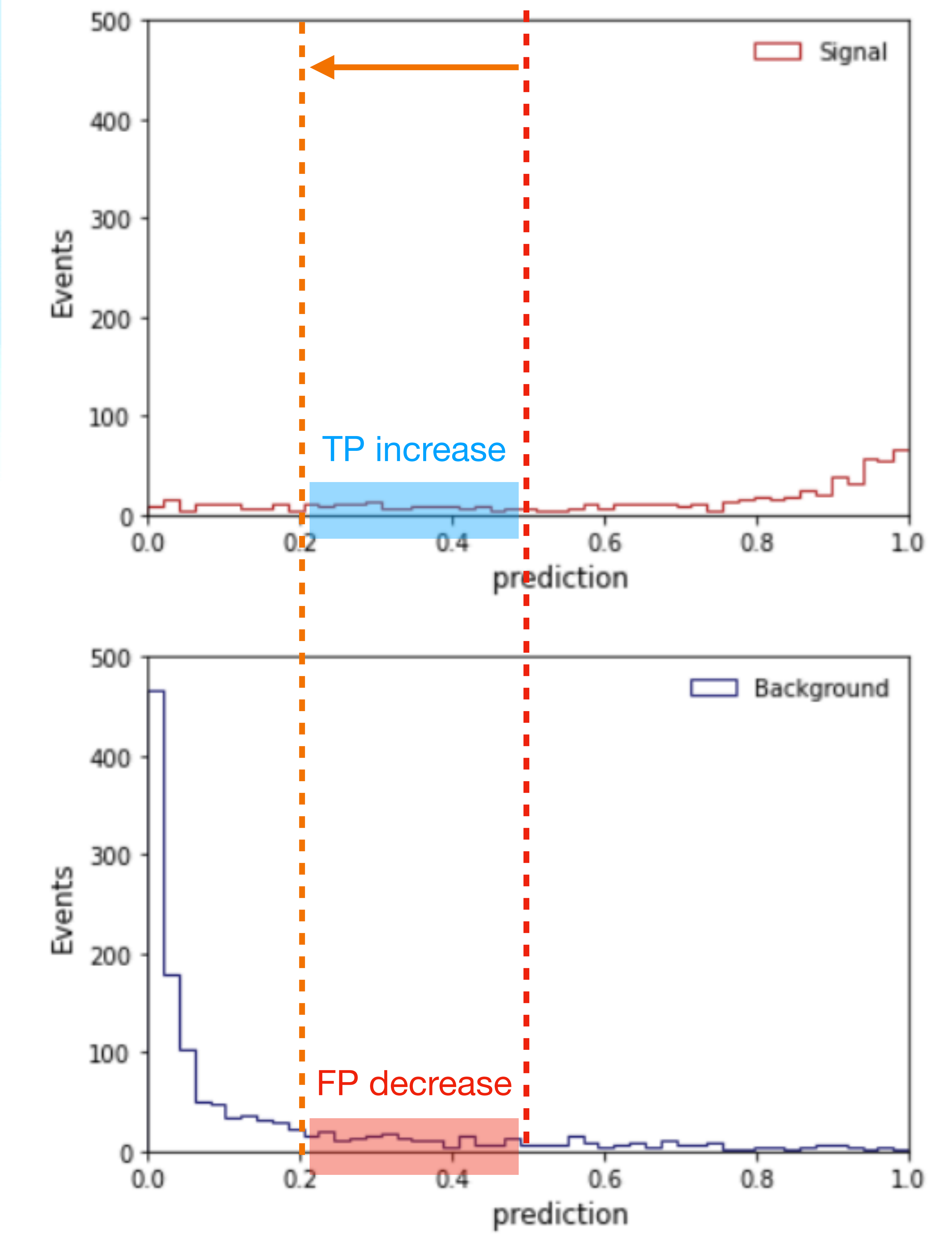
Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background



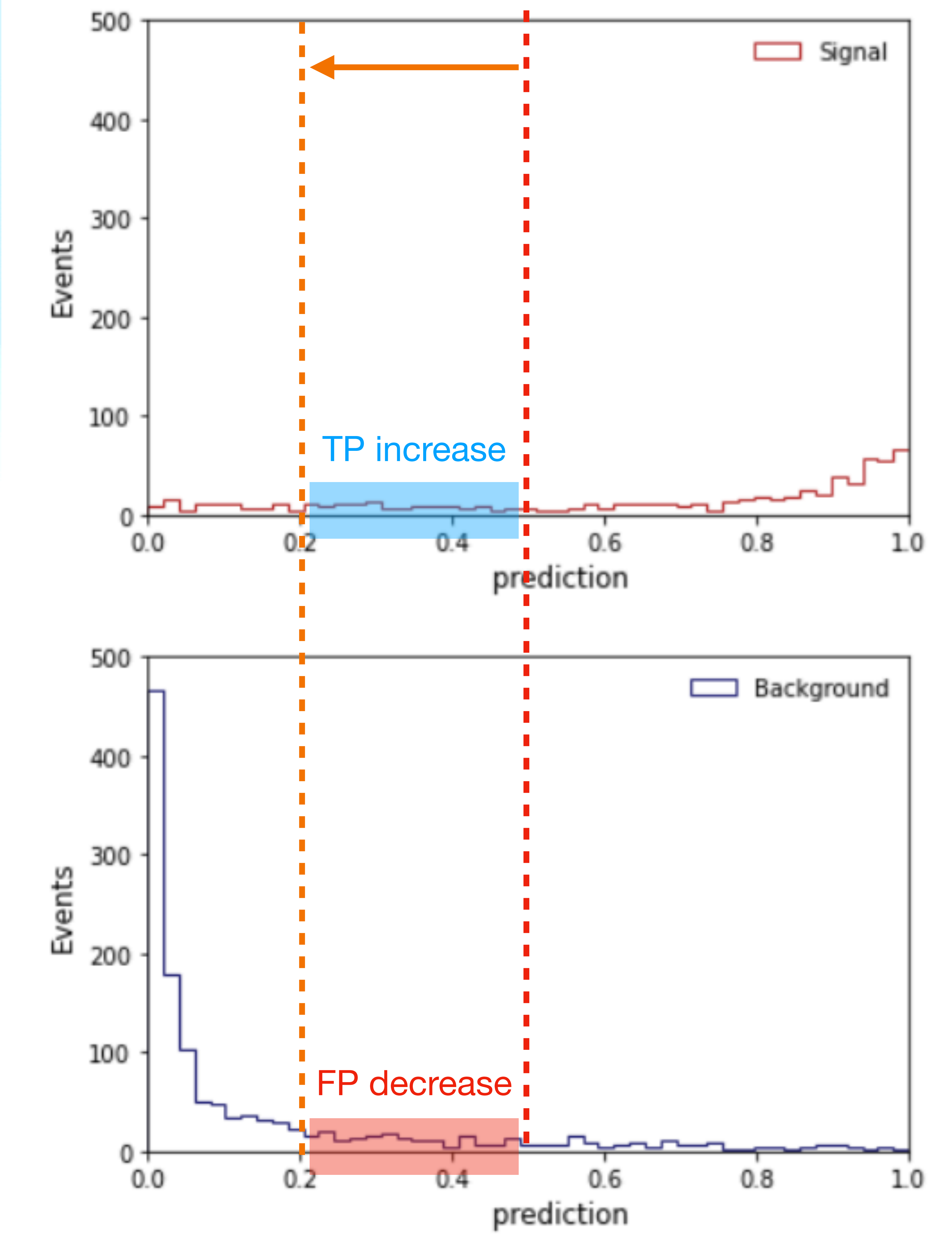
Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background



Setup: Classifying Majorana Demonstrator Waveforms

- Signal/positive: single-site waveform
- Backgrounds/negative: multi-site waveform
- λ: Classification score produced by training the NN
- Cutting Threshold: set at to at 0.5 to define our yes/no answer:
    - $\lambda > 0.5$: event is a signal
    - $\lambda \leq 0.5$: event is a background



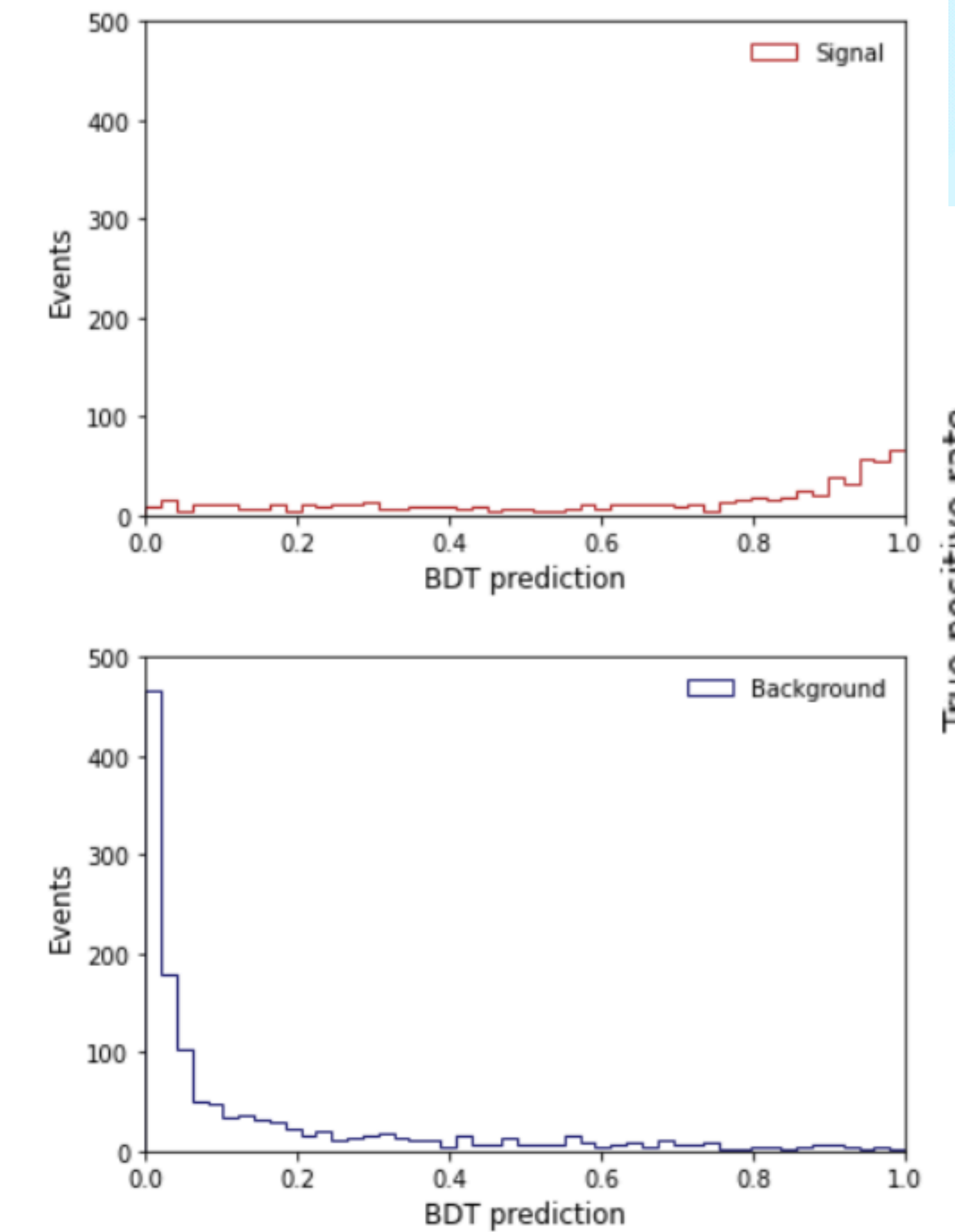
- Model did not change at all
- simply changing the cutting threshold will result in different TPR & FPR
- We need a threshold-independent metric to compare model performance!

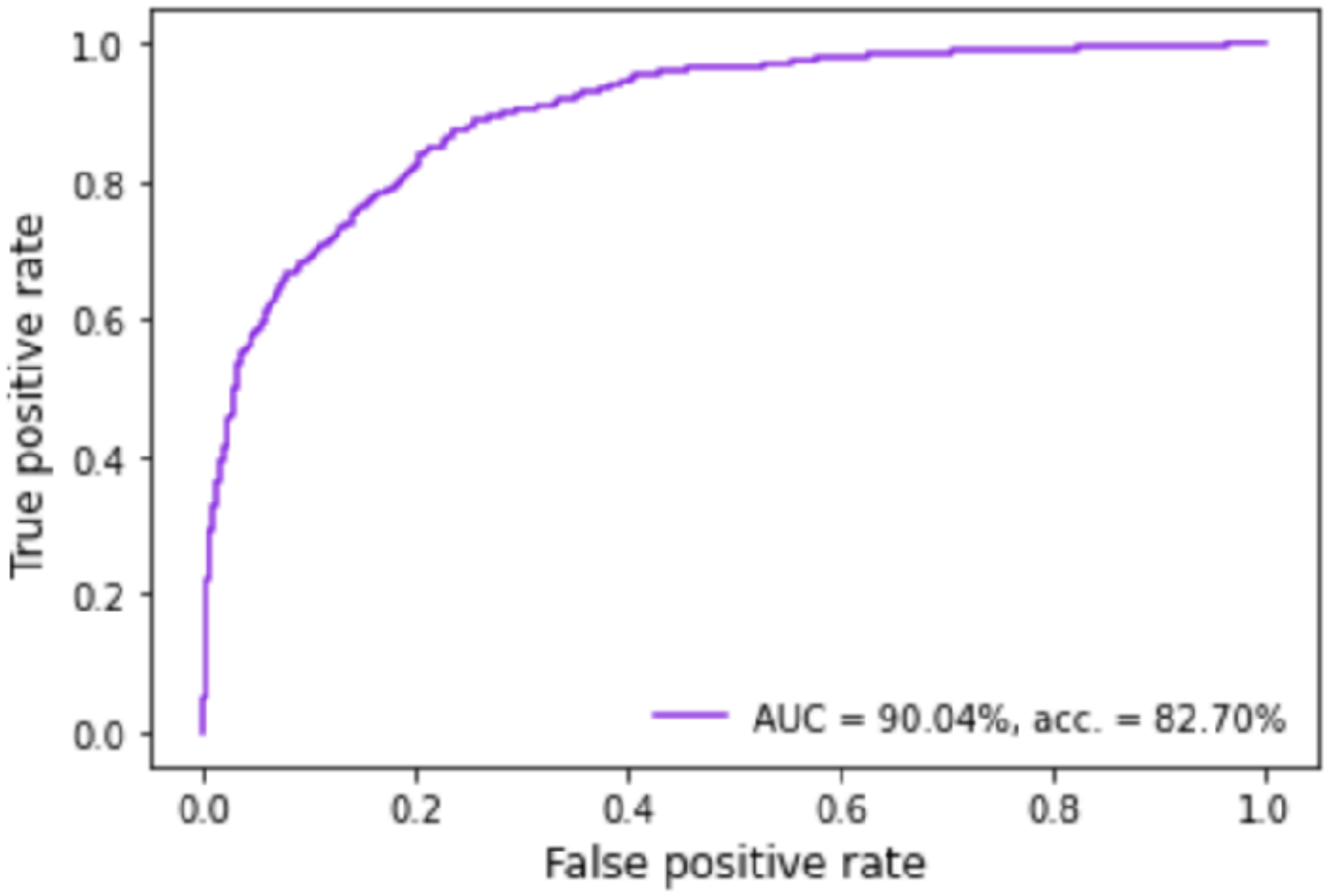
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Concept

Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible





1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

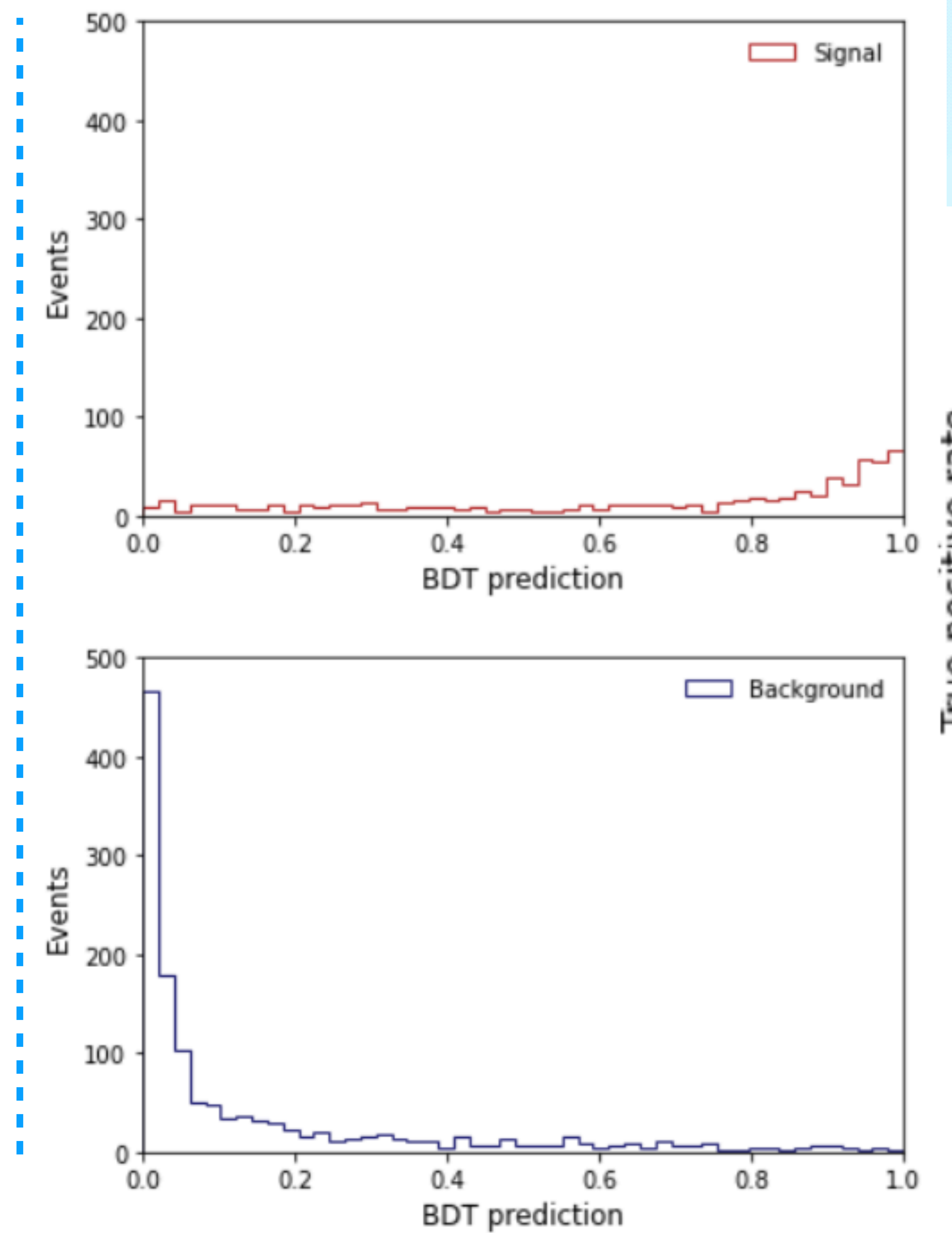
2. Feature Extraction → 3b. Regression → 4. Training

Concept

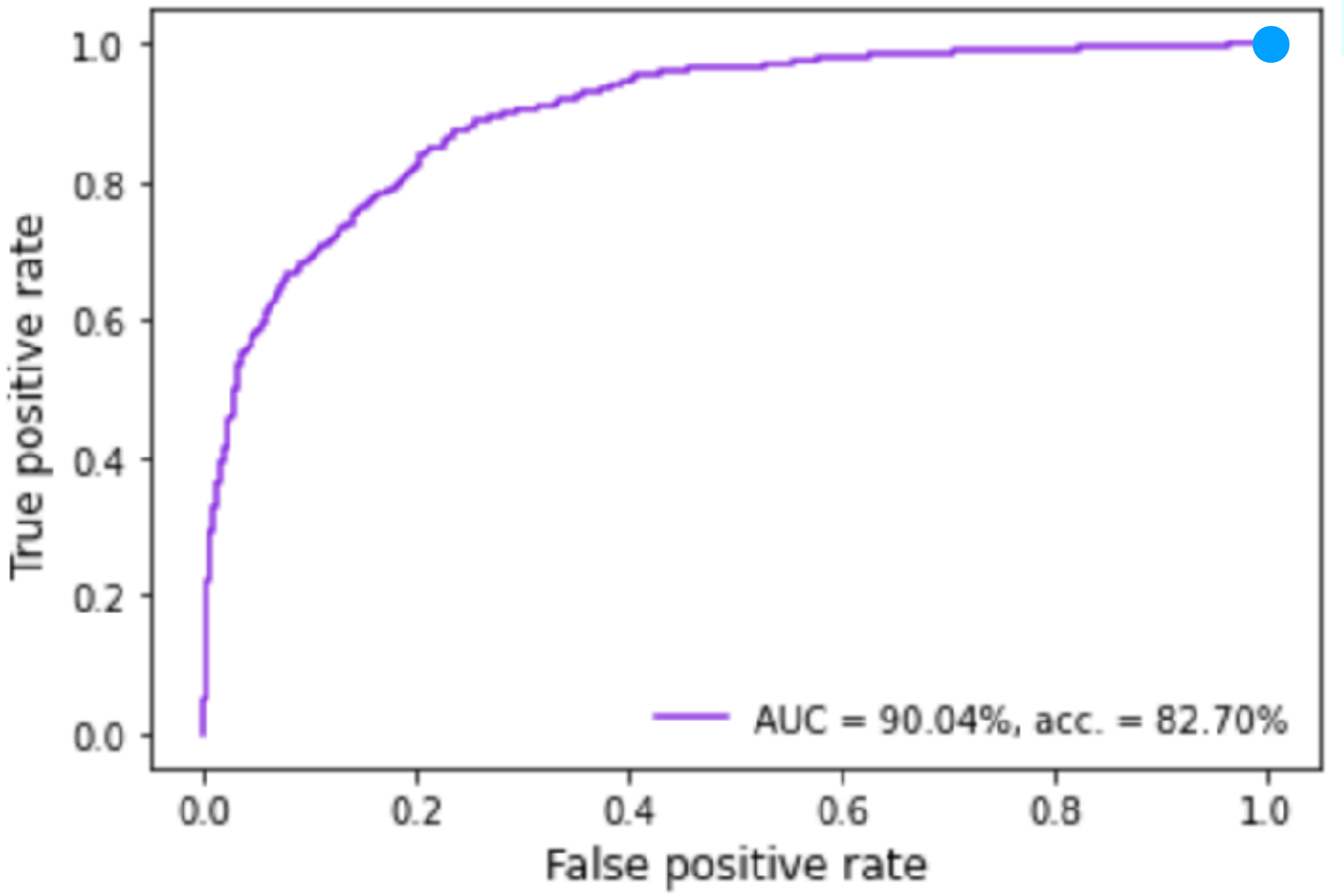
Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible

Cutting at -0.1:

- TPR = 100%
- FPR = 100%



All ROC curves start at (1,1)



Concept

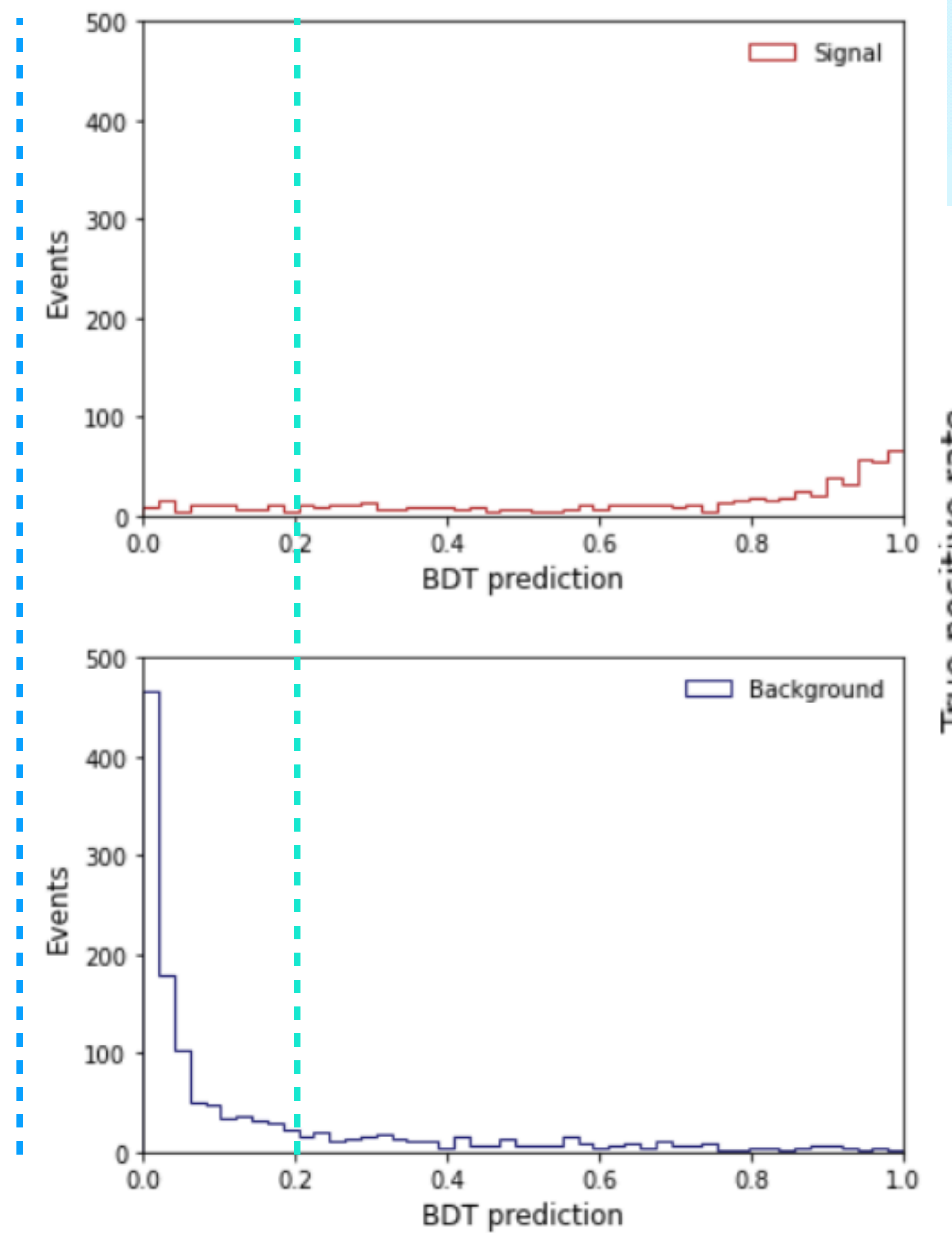
Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible

Cutting at -0.1:

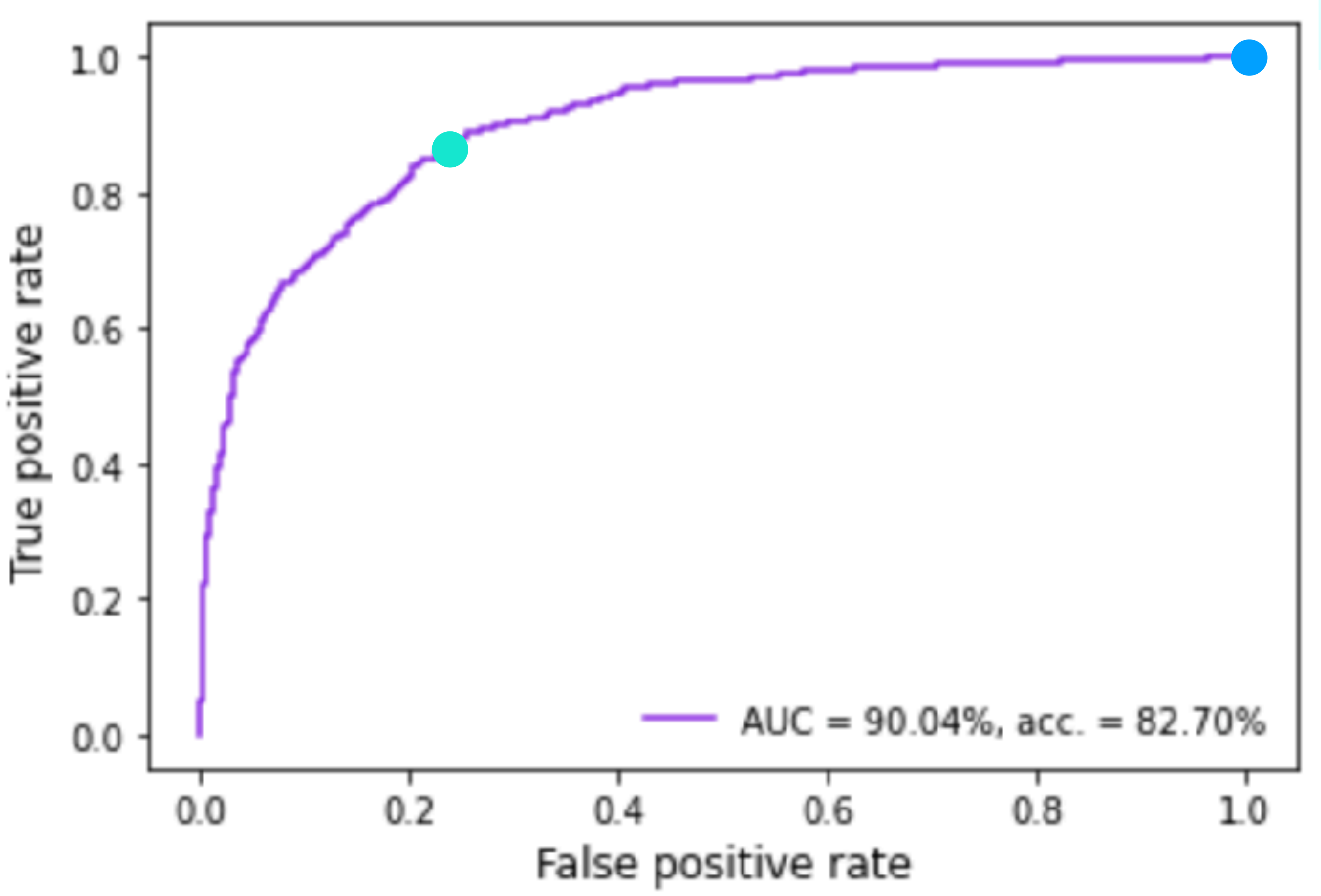
- TPR = 100%
- FPR = 100%

Cutting at 0.2:

- TPR = 87%
- FPR = 24%



All ROC curves start at (1,1)



1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Concept

Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible

Cutting at -0.1:

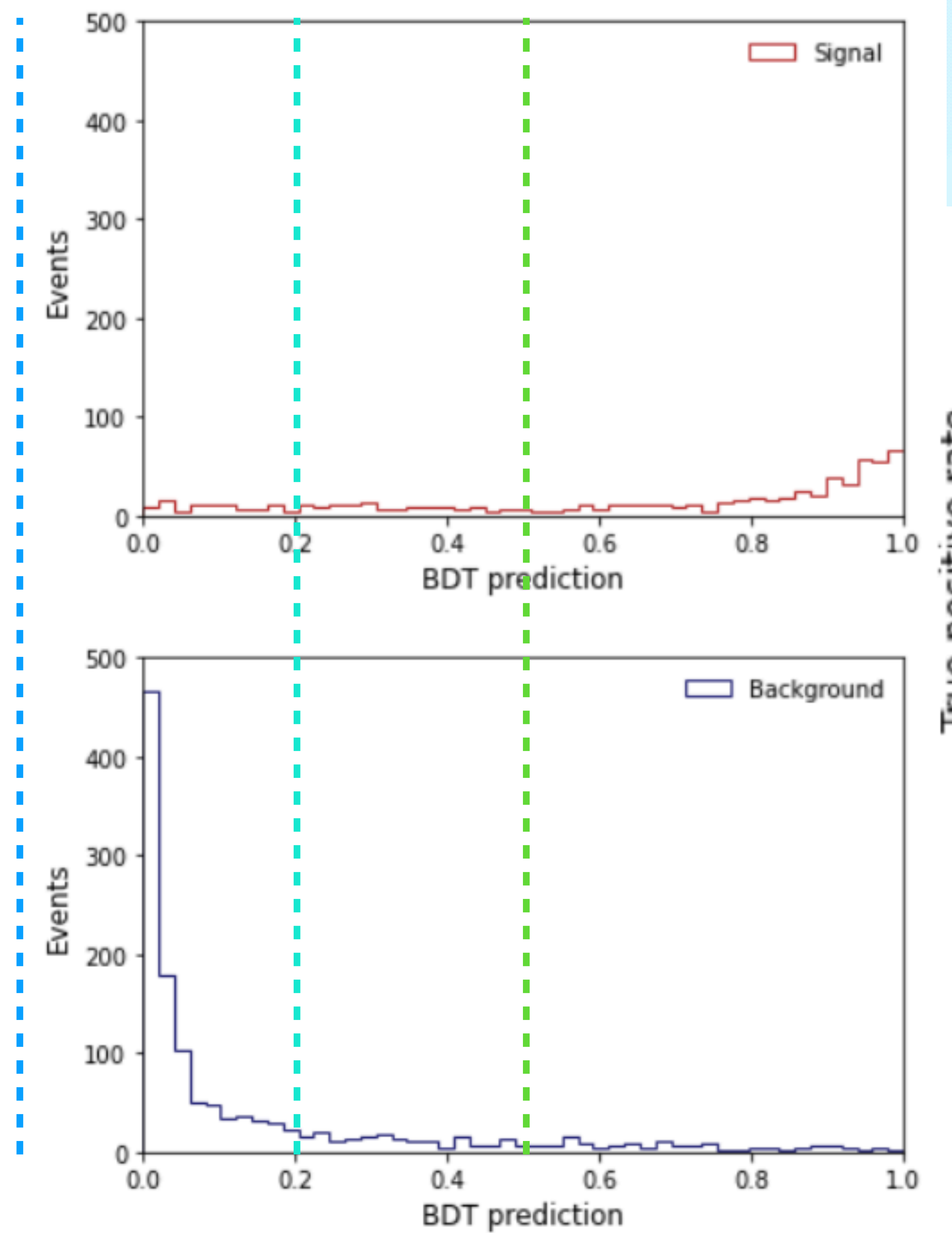
- TPR = 100%
- FPR = 100%

Cutting at 0.2:

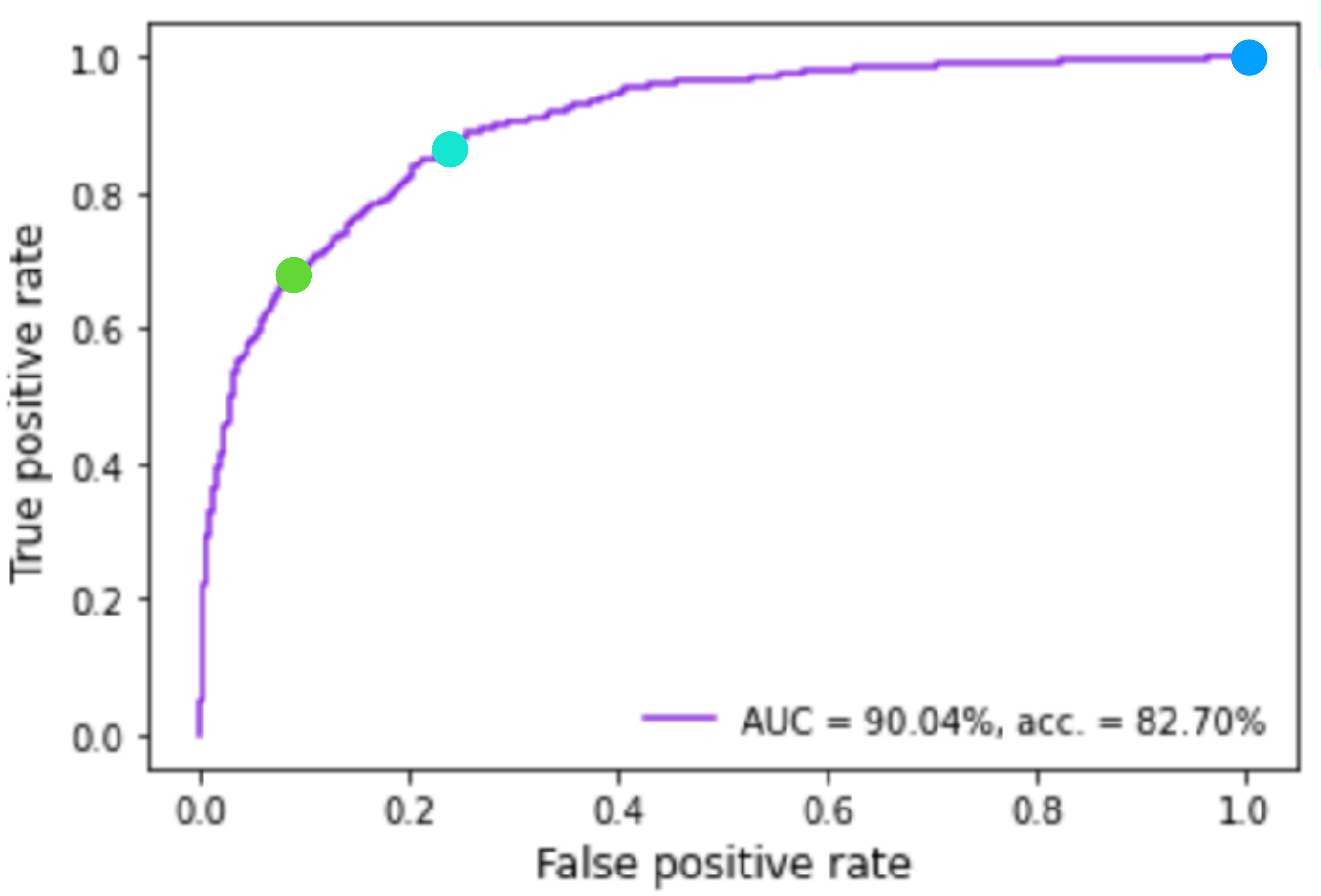
- TPR = 87%
- FPR = 24%

Cutting at 0.5:

- TPR = 70%
- FPR = 10%



All ROC curves start at (1,1)



1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → **5. Evaluation**

2. Feature Extraction → 3b. Regression → 4. Training

Concept

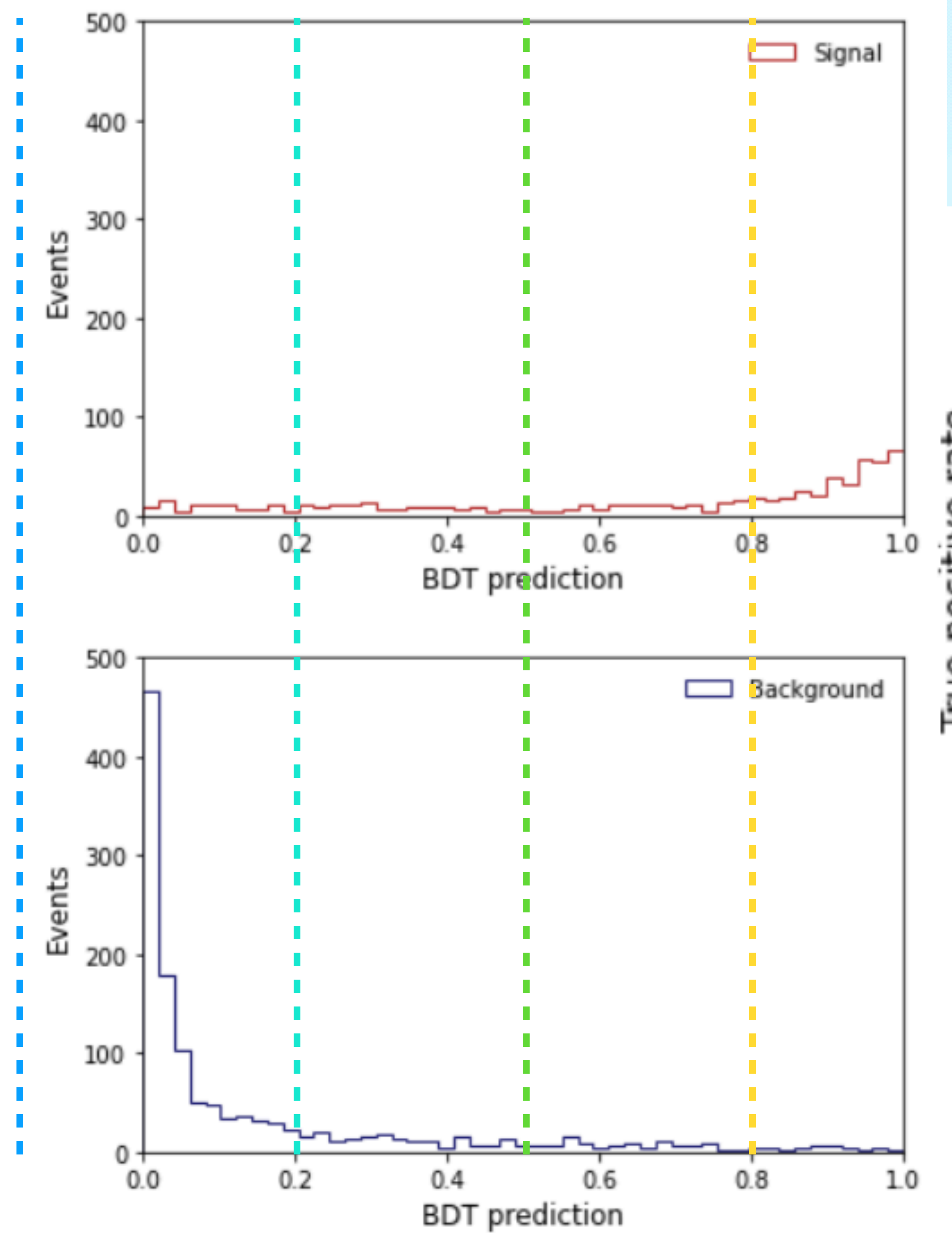
Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible

Cutting at -0.1:
- TPR = 100%
- FPR = 100%

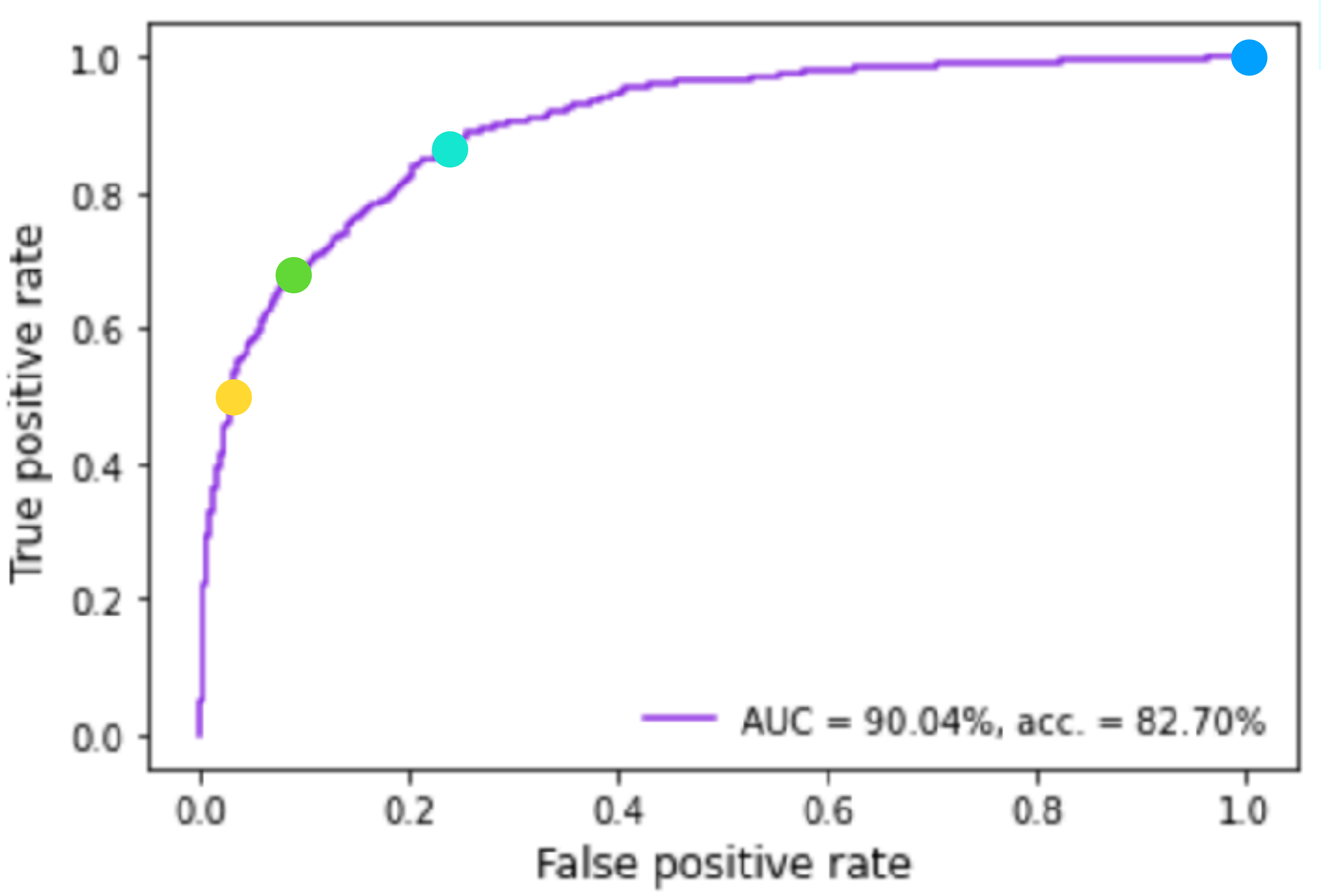
Cutting at 0.2:
- TPR = 87%
- FPR = 24%

Cutting at 0.5:
- TPR = 70%
- FPR = 10%

Cutting at 0.8:
- TPR = 50%
- FPR = 3%



All ROC curves start at (1,1)



Concept

Receiver Operating Characteristic (ROC) Curve: Evaluate our model at all thresholds possible

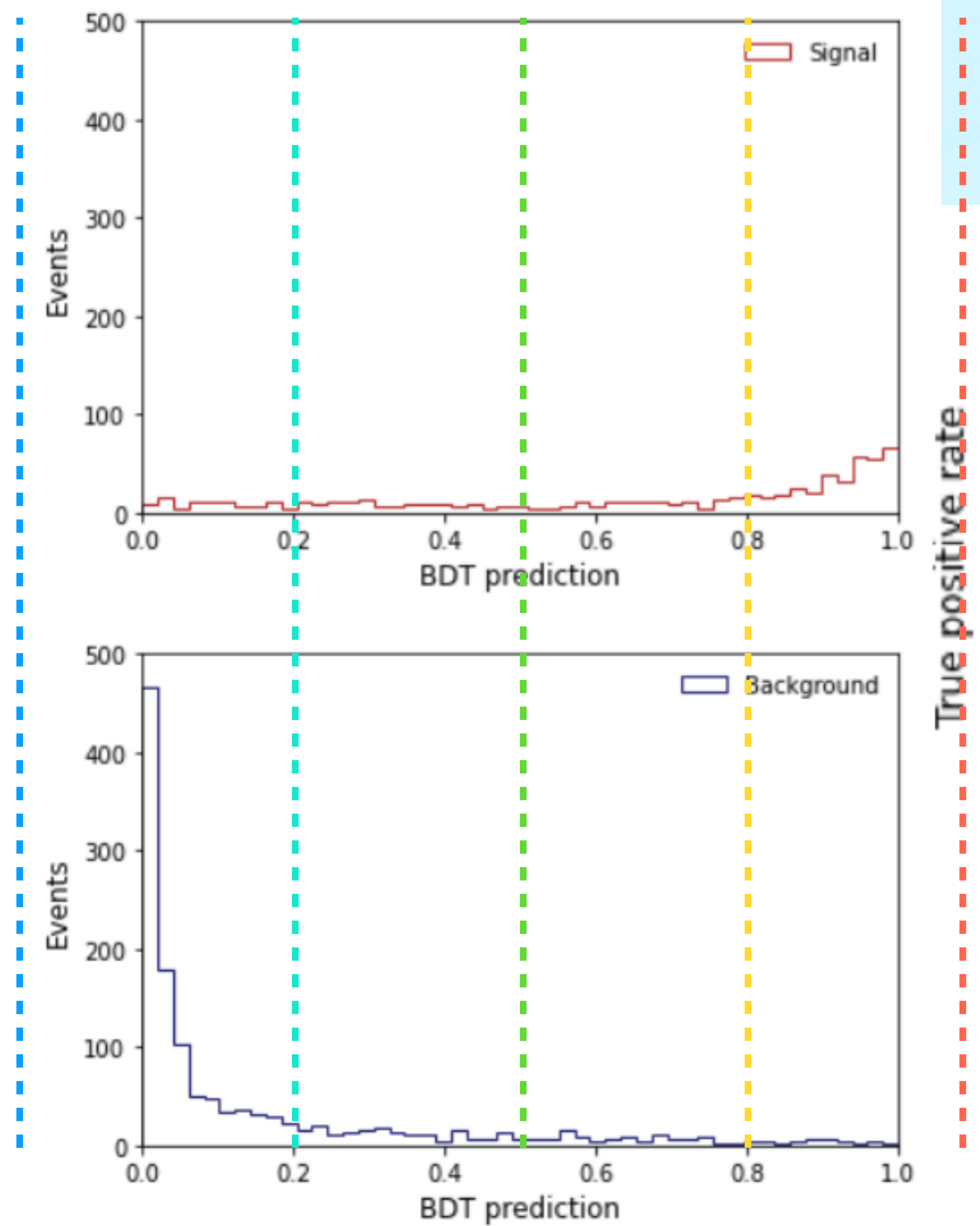
Cutting at -0.1:
- TPR = 100%
- FPR = 100%

Cutting at 0.2:
- TPR = 87%
- FPR = 24%

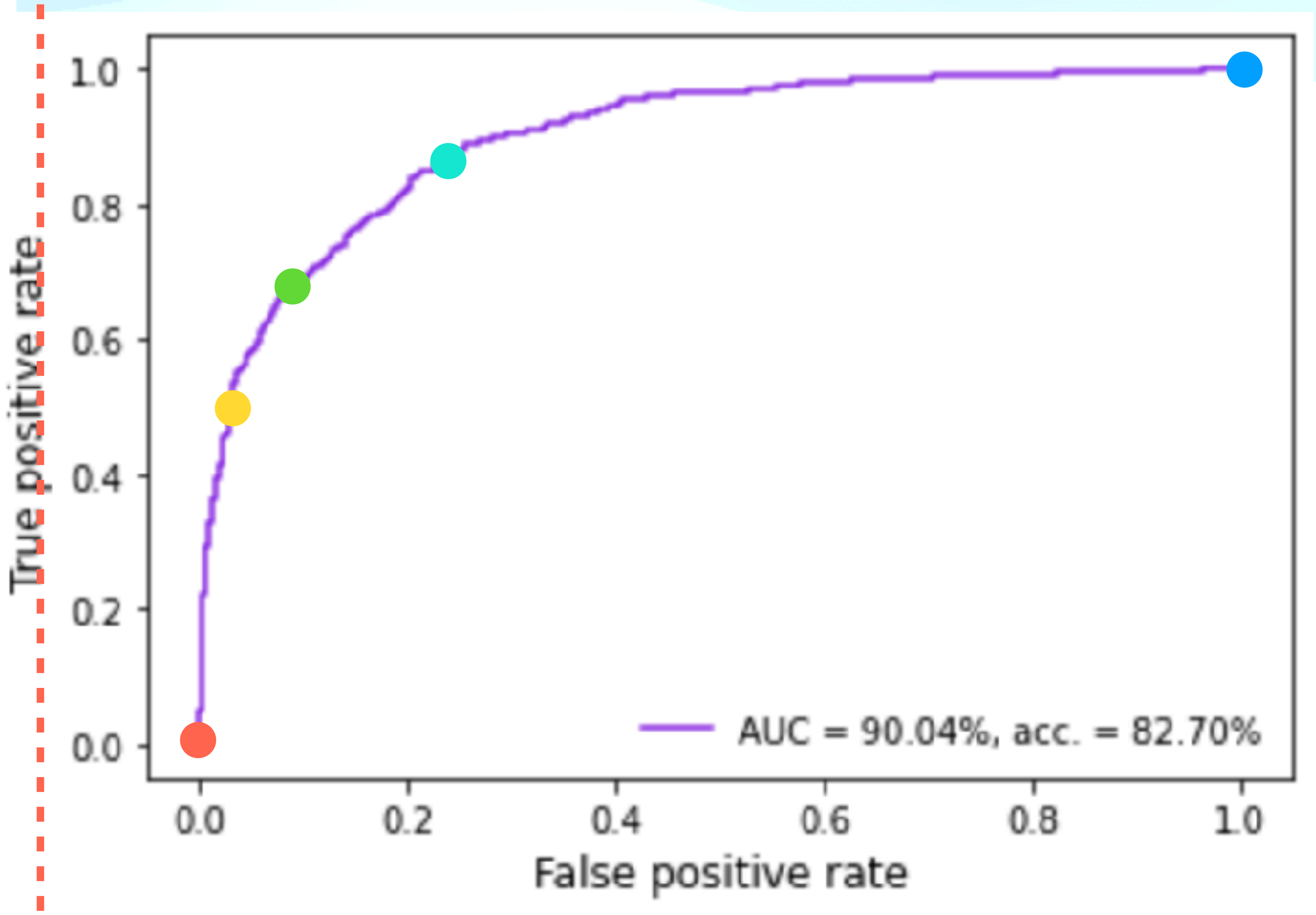
Cutting at 0.5:
- TPR = 70%
- FPR = 10%

Cutting at 0.8:
- TPR = 50%
- FPR = 3%

Cutting at 1.1:
- TPR = 0%
- FPR = 0%



All ROC curves start at (1,1)



All ROC curves ends at (0,0)

Perfect ROC Curve

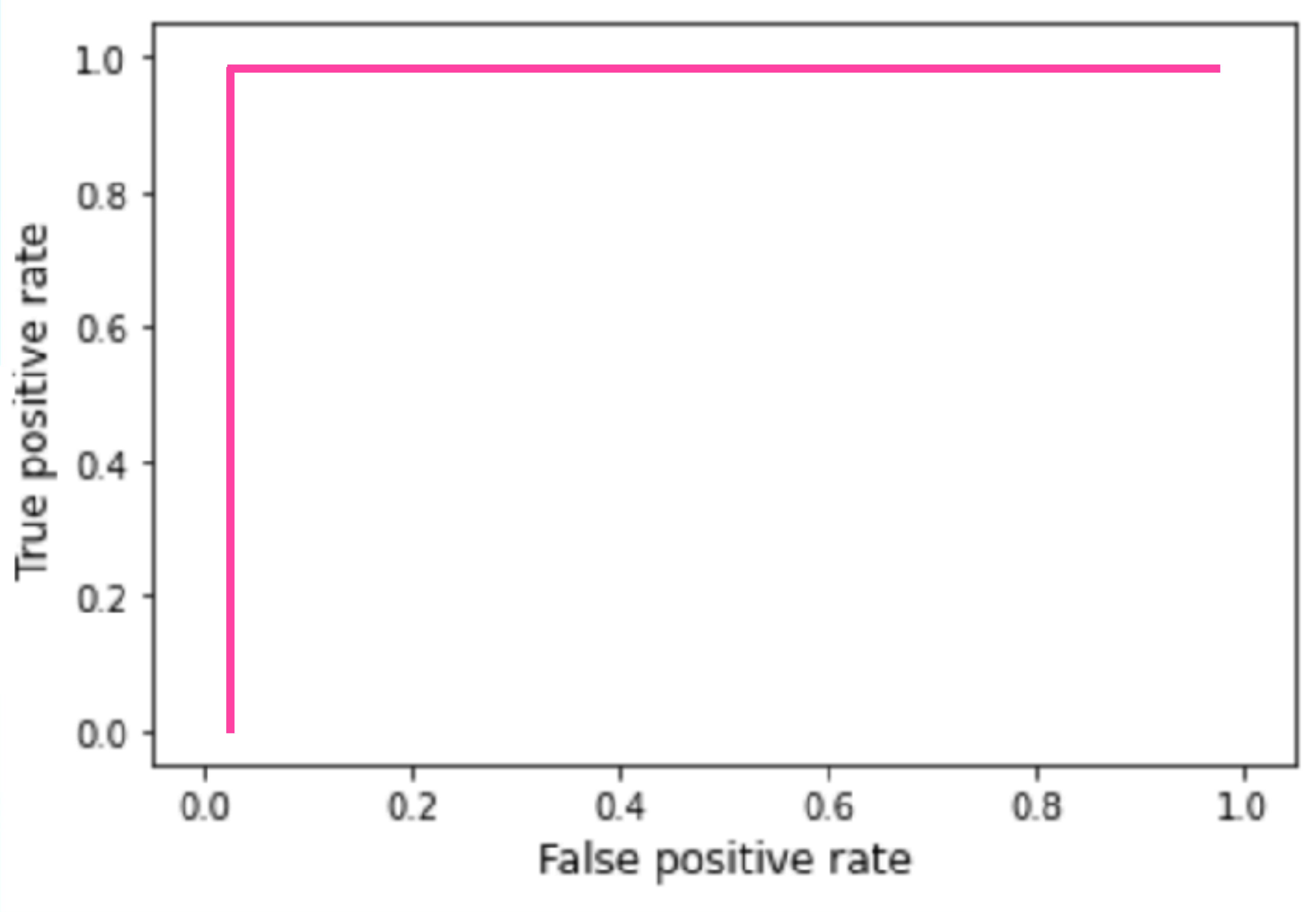
- TPR = 100% at FPR = 0%
- Area under curve = 1
- Every event is classified correctly

Worst ROC Curve

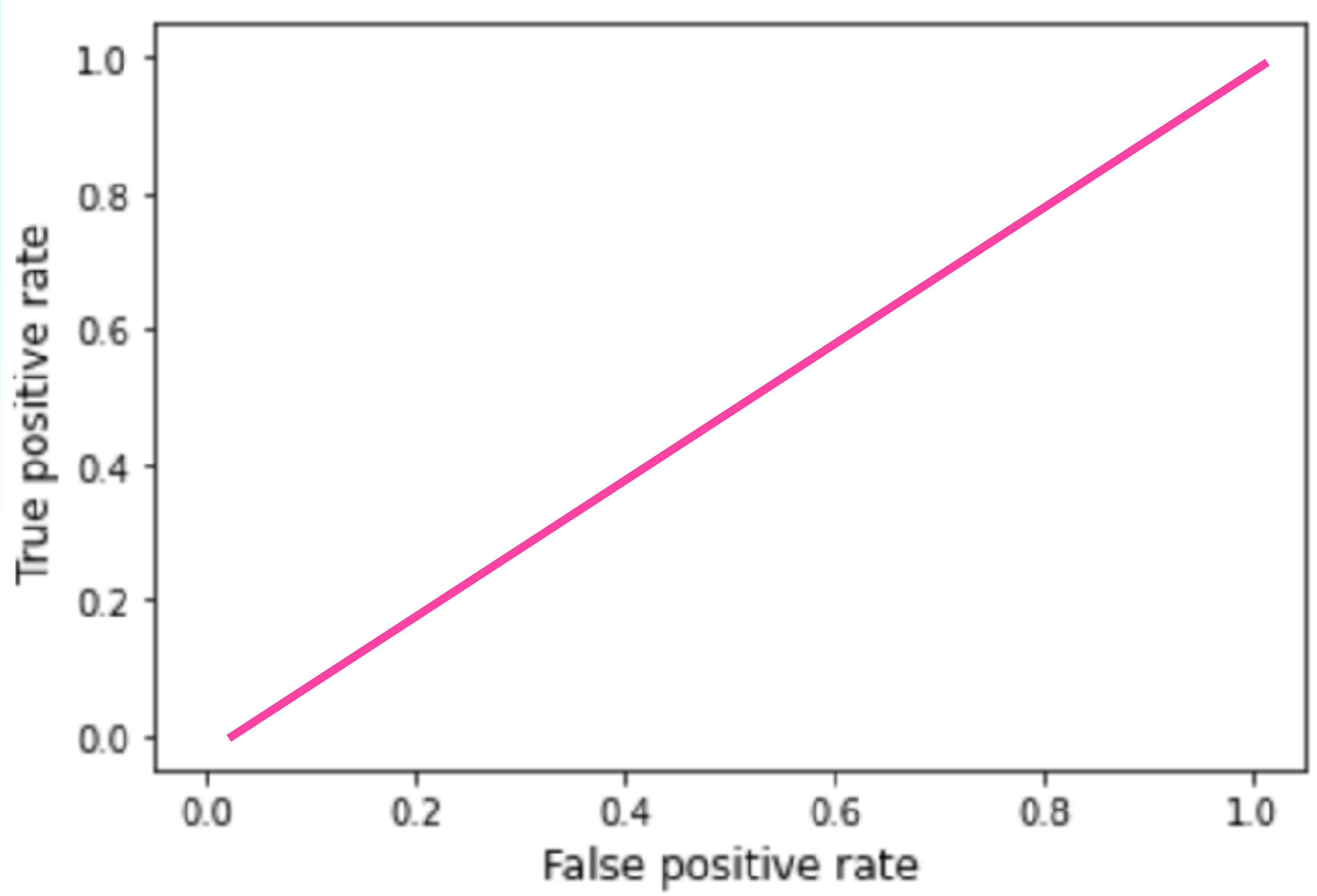
- TPR == FPR
- Area under curve = 0.5
- Classify by tossing a coin

Anti-perfect ROC Curve

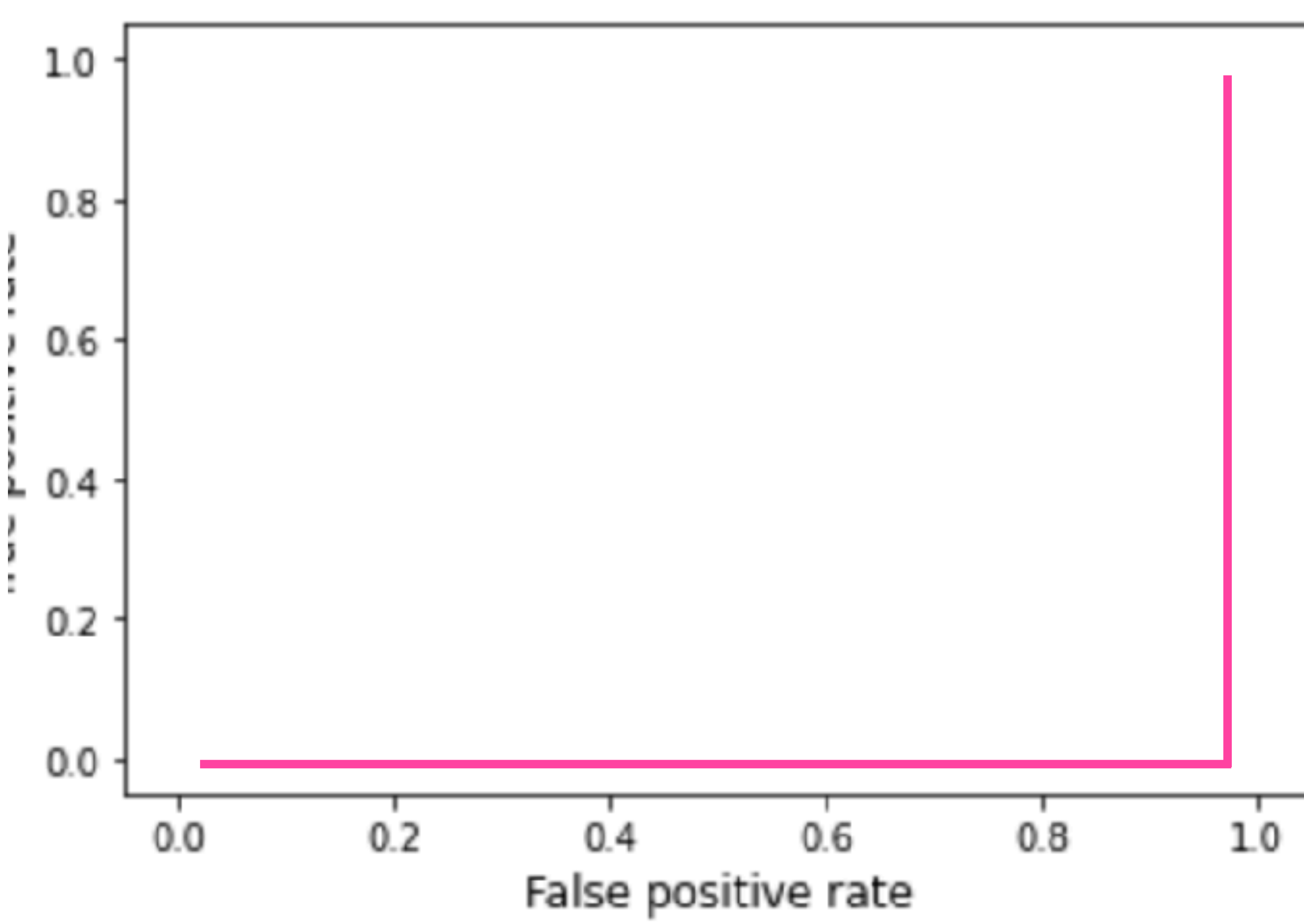
- TPR = 0% at FPR = 100%
- Area under curve = 0
- Every event is classified wrong



$\forall \lambda : \lambda_{sig} > \lambda_{bkg}$



$\lambda \sim \mathrm{Unif}(0,1)$



$\forall \lambda : \lambda_{sig} < \lambda_{bkg}$

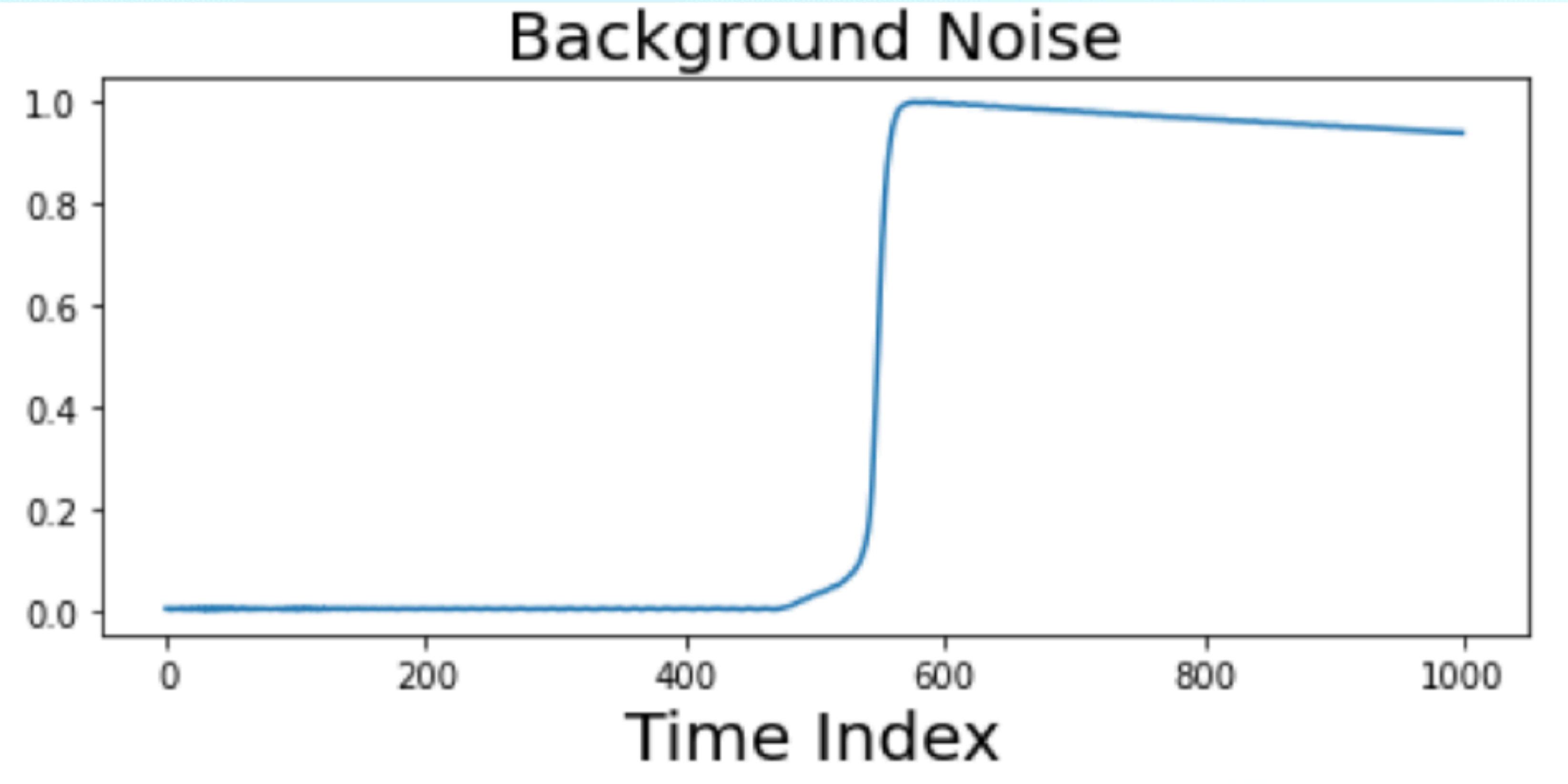
Concept

The **Area Under the Curve (AUC)** of ROC curve is equal to the probability that a model will rank a randomly chosen signal higher than a randomly chosen background

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Time Series Data
(BATCH_SIZE, 1000)



Concept

NNLayer(1000,512) → *σ*
→ *NNLayer(512, 256)* → *σ*
→ *NNLayer(256, 32)* → *σ*

Concept

Binary Classification 1

Task Layer: *NNLayer(32, 1) → One float point No.*

- After training, it can be onsidered as a **"classification score":**
  - Higher score means the answer is more likely "yes"
  - Lower score means the answer is more likely "no"
  - A **threshold** is need to distinguish "yes" from "no"

σ: *torch.sigmoid(x)*

Loss Function: *torch.nn.BCELoss()*

Concept

Regression 1

Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

σ:

- None if you want to fit a physics quantity like energy
- torch.sigmoid(x) if you want to model a percentage like efficiency

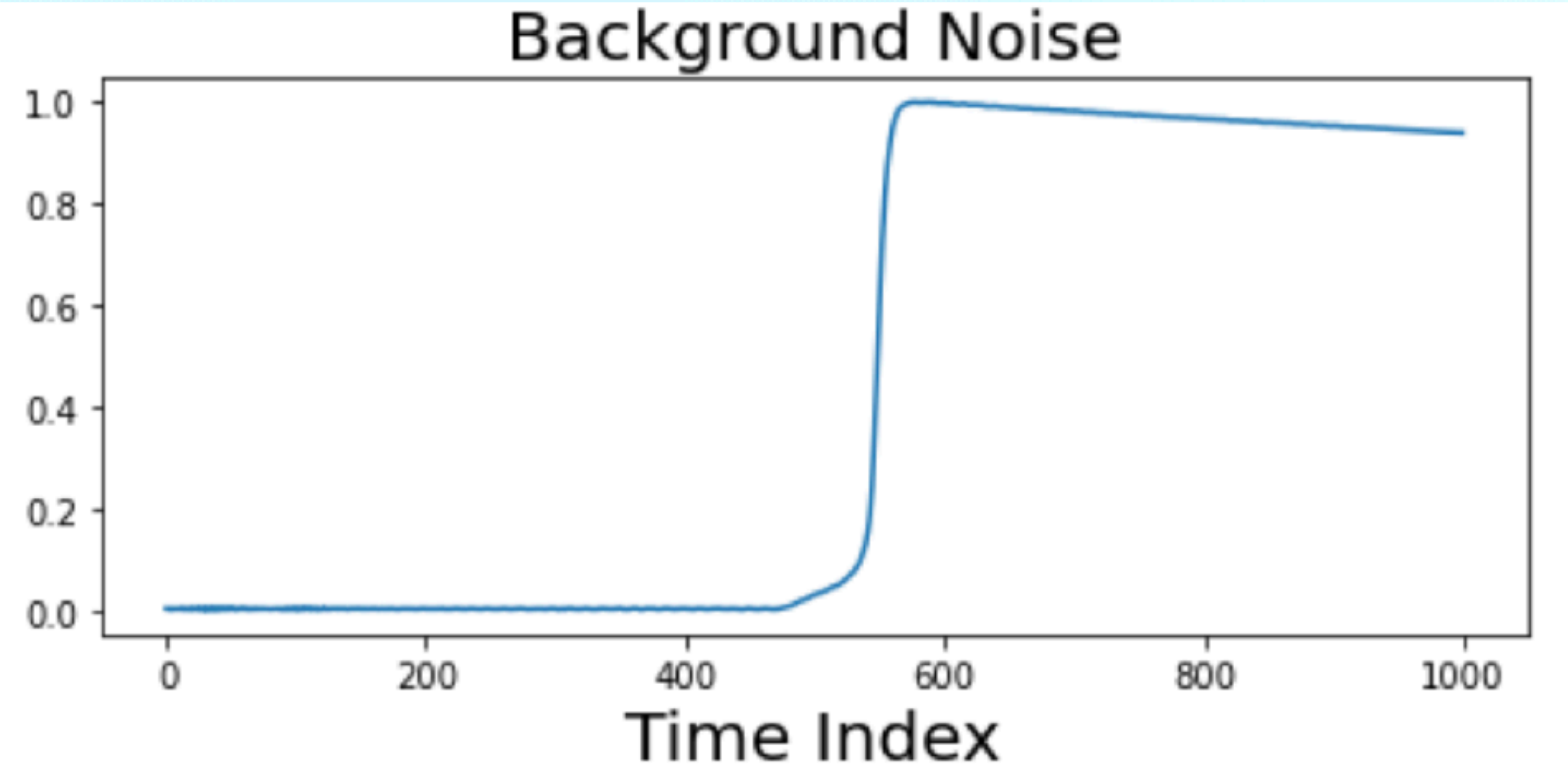
Loss Function: *torch.nn.MSELoss()*

- $L = (\mathrm{TL\_Output} - \mathrm{Energy})^2$ for energy reconstruction

1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

2. Feature Extraction → 3b. Regression → 4. Training

Time Series Data
(BATCH_SIZE, 1000)



Concept

Feature Extractor Network

Take raw data as the input and output a low-dimensional vector

NNLayer(1000,512) → σ

→ NNLayer(512, 256) → σ

→ NNLayer(256, 32) → σ

Concept

Binary Classification 1

Task Layer: *NNLayer(32, 1) → One float point No.*

- After training, it can be onsidered as a **"classification score":**
  - Higher score means the answer is more likely "yes"
  - Lower score means the answer is more likely "no"
  - A **threshold** is need to distinguish "yes" from "no"

σ: *torch.sigmoid(x)*

Loss Function: *torch.nn.BCELoss()*

Concept

Regression 1

Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

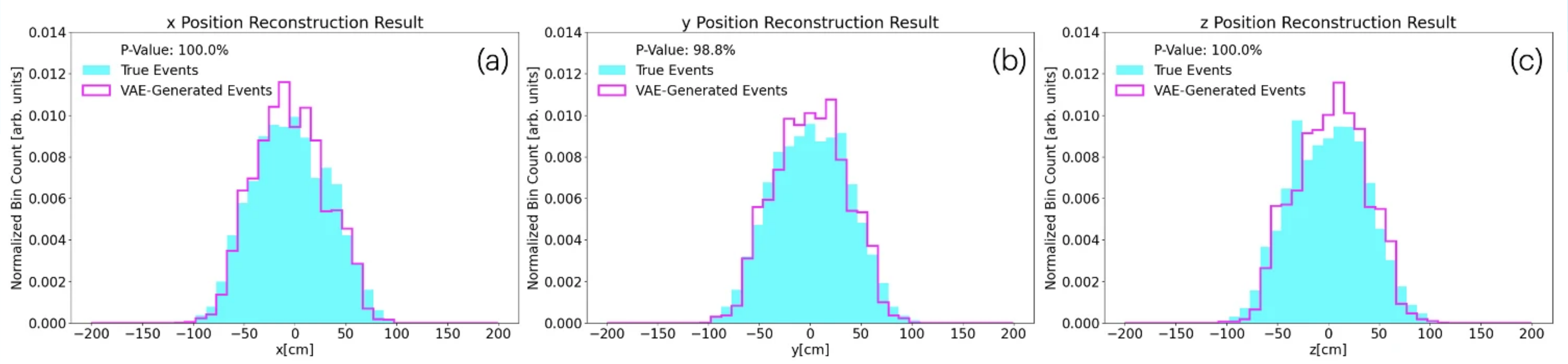
σ:

- None if you want to fit a physics quantity like energy
- torch.sigmoid(x) if you want to model a percentage like efficiency

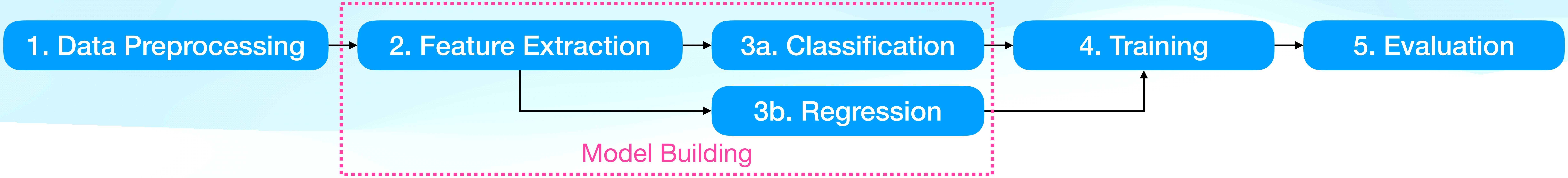
Loss Function: *torch.nn.MSELoss()*

- $L = (\text{TL\_Output} - \text{Energy})^2$ for energy reconstruction

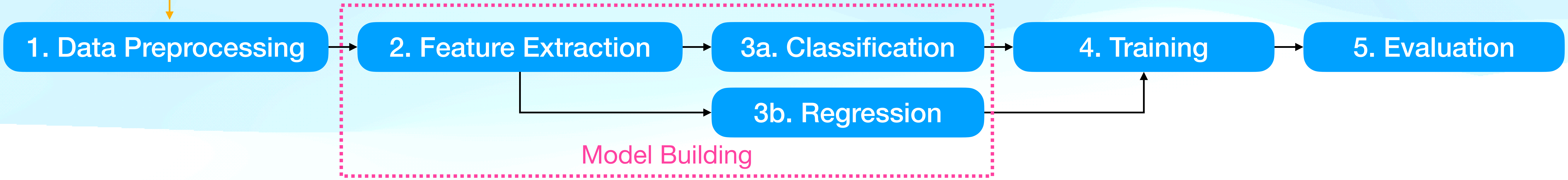
- Regression performance can be evaluated the same way as reconstruction result in physics!
- Additionally, the value of MSE loss is a good metric to understand performance.



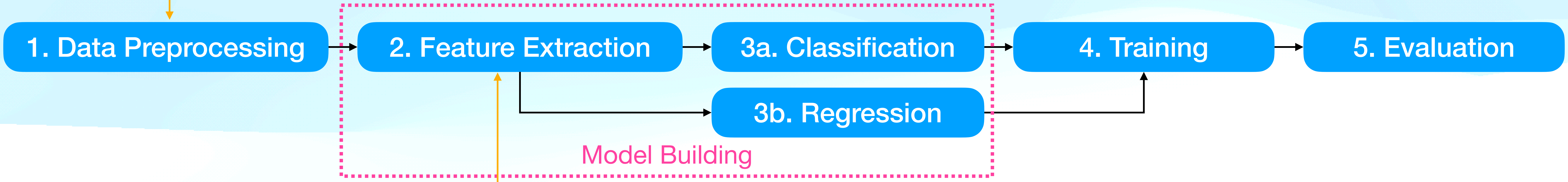
https://link.springer.com/article/10.1140/epjc/s10052-024-12980-7


1. Data Preprocessing
2. Feature Extraction
3a. Classification
3b. Regression
4. Training
5. Evaluation
Model Building

1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. **Data pre-processing:** transform your data to remove unwanted informations
4. Wrap dataset object to create a **data loader**



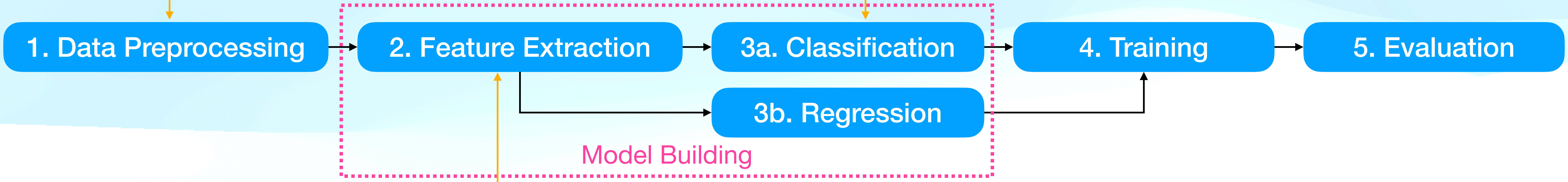
1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. **Data pre-processing:** transform your data to remove unwanted informations
4. Wrap dataset object to create a **data loader**



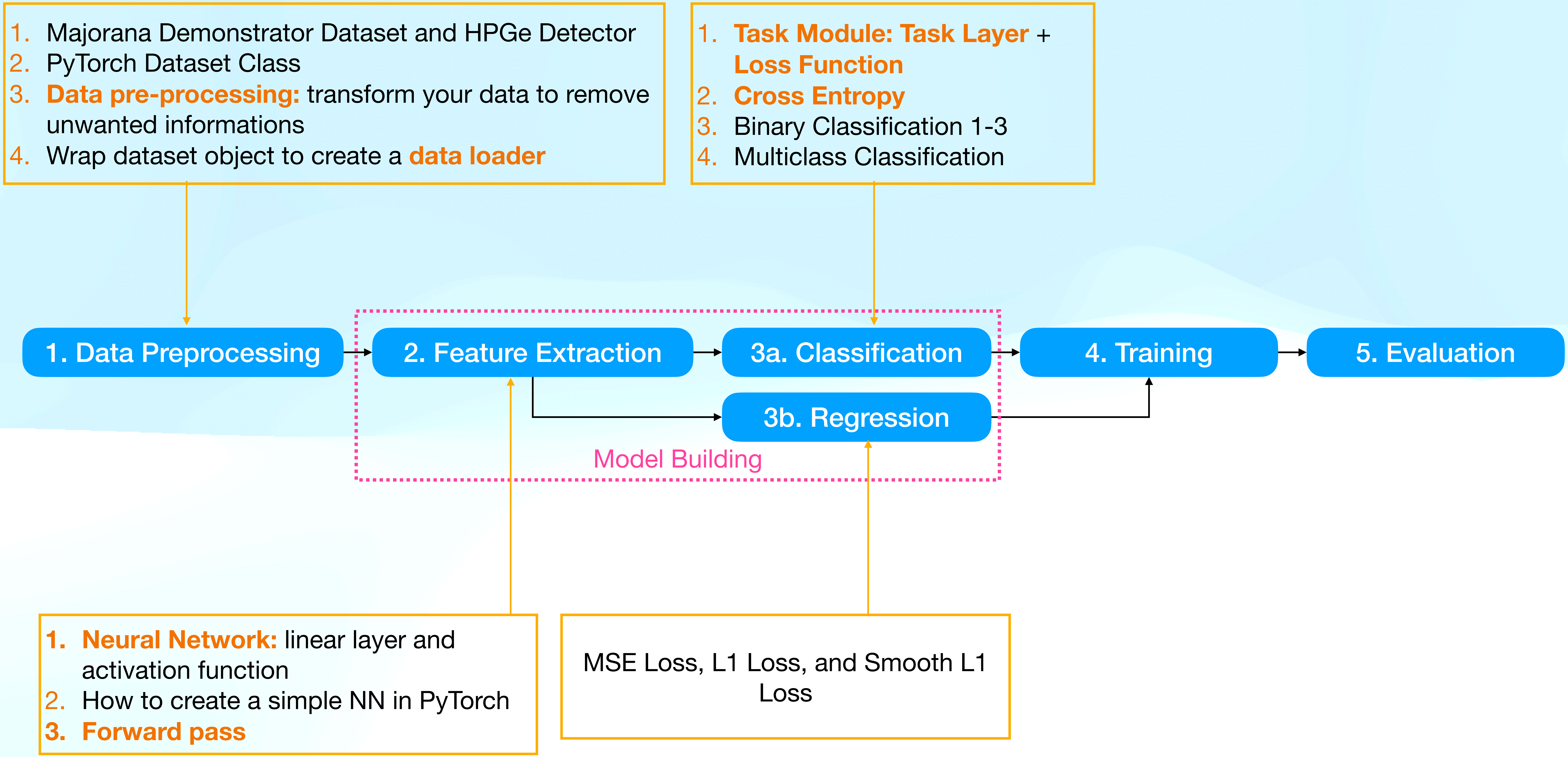
1. **Neural Network:** linear layer and activation function
2. How to create a simple NN in PyTorch
3. **Forward pass**

1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. **Data pre-processing:** transform your data to remove unwanted informations
4. Wrap dataset object to create a **data loader**

1. **Task Module: Task Layer** + **Loss Function**
2. **Cross Entropy**
3. Binary Classification 1-3
4. Multiclass Classification



1. **Neural Network:** linear layer and activation function
2. How to create a simple NN in PyTorch
3. **Forward pass**


1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. Data pre-processing: transform your data to remove unwanted informations
4. Wrap dataset object to create a data loader
1. Task Module: Task Layer + Loss Function
2. Cross Entropy
3. Binary Classification 1-3
4. Multiclass Classification
1. Data Preprocessing
2. Feature Extraction
3a. Classification
3b. Regression
4. Training
5. Evaluation
Model Building
1. Neural Network: linear layer and activation function
2. How to create a simple NN in PyTorch
3. Forward pass
MSE Loss, L1 Loss, and Smooth L1 Loss


1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. Data pre-processing: transform your data to remove unwanted informations
4. Wrap dataset object to create a data loader
1. Task Module: Task Layer + Loss Function
2. Cross Entropy
3. Binary Classification 1-3
4. Multiclass Classification
1. Stochastic Gradient Descent
2. Backward Pass: Computational graph and Backpropagation
3. PyTorch implementation: one line
1. Data Preprocessing
2. Feature Extraction
3a. Classification
3b. Regression
4. Training
5. Evaluation
Model Building
1. Neural Network: linear layer and activation function
2. How to create a simple NN in PyTorch
3. Forward pass
MSE Loss, L1 Loss, and Smooth L1 Loss


1. Majorana Demonstrator Dataset and HPGe Detector
2. PyTorch Dataset Class
3. Data pre-processing: transform your data to remove unwanted informations
4. Wrap dataset object to create a data loader
1. Task Module: Task Layer + Loss Function
2. Cross Entropy
3. Binary Classification 1-3
4. Multiclass Classification
1. Stochastic Gradient Descent
2. Backward Pass: Computational graph and Backpropagation
3. PyTorch implementation: one line
1. Data Preprocessing
2. Feature Extraction
3a. Classification
3b. Regression
4. Training
5. Evaluation
Model Building
1. Neural Network: linear layer and activation function
2. How to create a simple NN in PyTorch
3. Forward pass
MSE Loss, L1 Loss, and Smooth L1 Loss
1. Overfitting vs. Underfitting
2. Evaluate binary classication: ROC Analysis
3. Evaluate regression: reconstruction algorithm

Some Useful Links

Code

All lecture materials: Link
Jupyter Notebook Code: Link

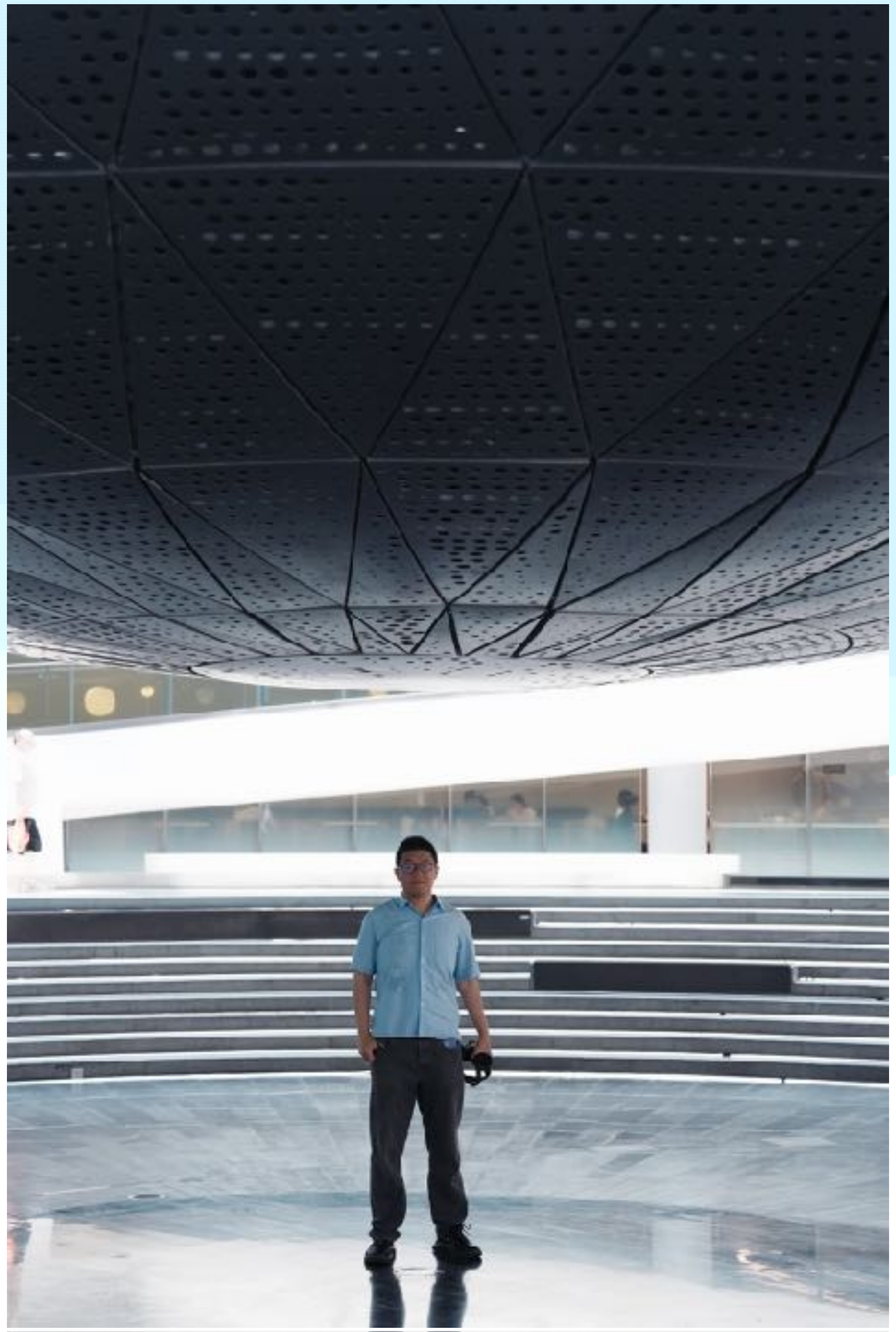
Concept

The Practical Machine Learning
https://pire.gemadarc.org/education/school21/#ai

2024 Summer Bootcamp on Deep Learning and Applications
https://ai-bootcamp2024.github.io/

MIT 6.S191 Introduction to Deep Learning
http://introtodeeplearning.com/

Andrew Ng: Deep Learning Specialization
Link



My Website:
https://aobol.github.io/AoboLi/

My Email Addresses:
aol002@ucsd.edu