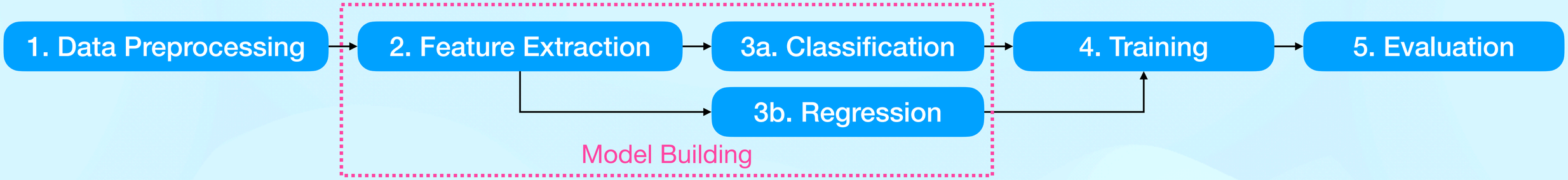
0νββ AI Summer School:
Intro to Machine Learning

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

0νββ AI Summer School, 06/20/2026

Course Outline



Theory

Important equations/theoretical derivations you should understand

Concept

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

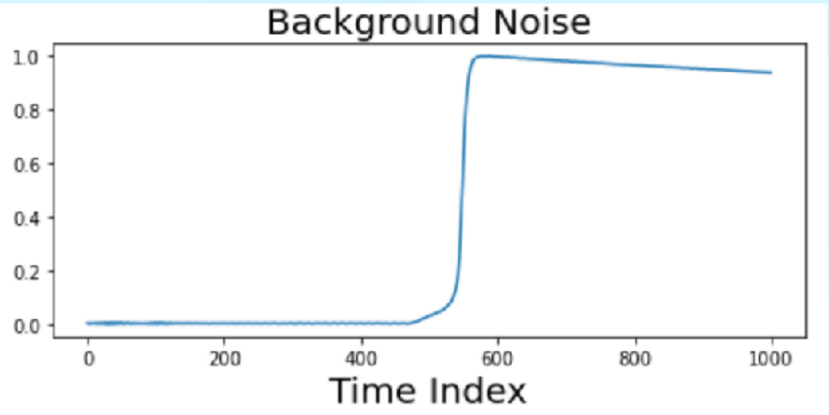
Code

The actual PyTorch implementation, useful for building your own models

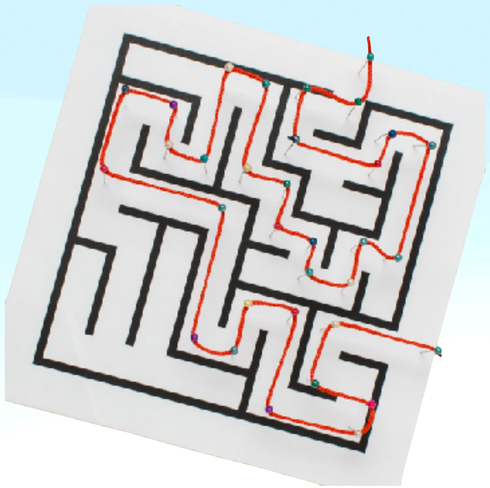
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

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

Time Series Data
(BATCH_SIZE, 1000)



Model
Map input to output



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

psd_label_low_avse
(batch_size, 1)

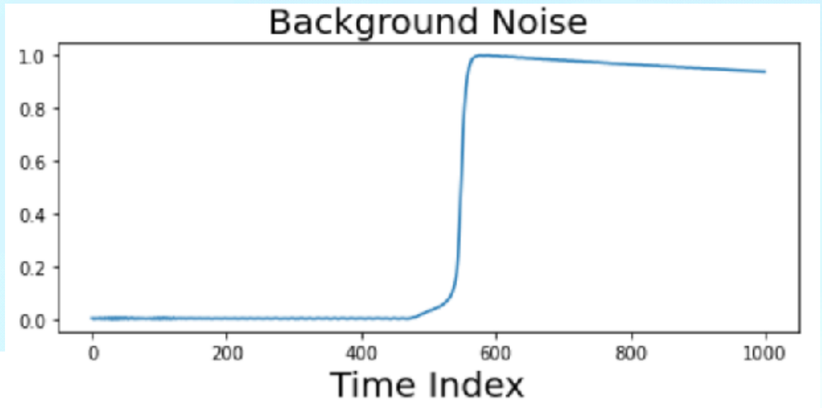
→ **Event Energy**
(batch_size, 1)

energy_label
(batch_size, 1)

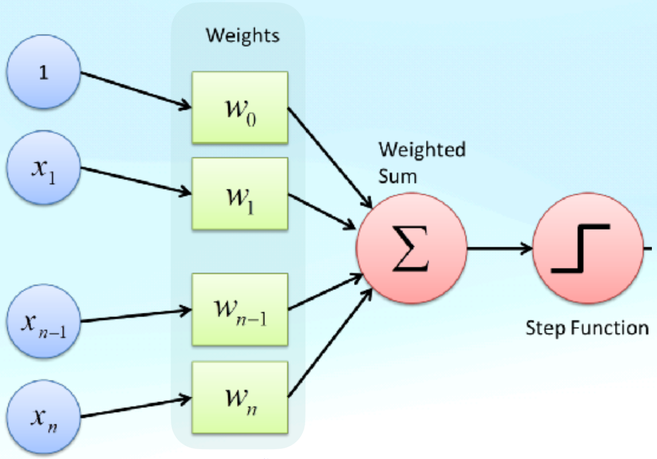
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

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

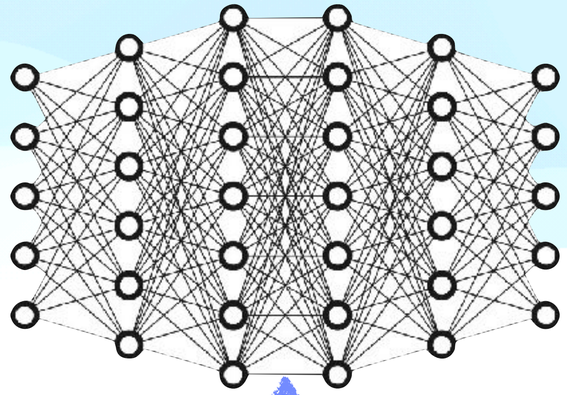
Time Series Data
(batch_size, 1000)



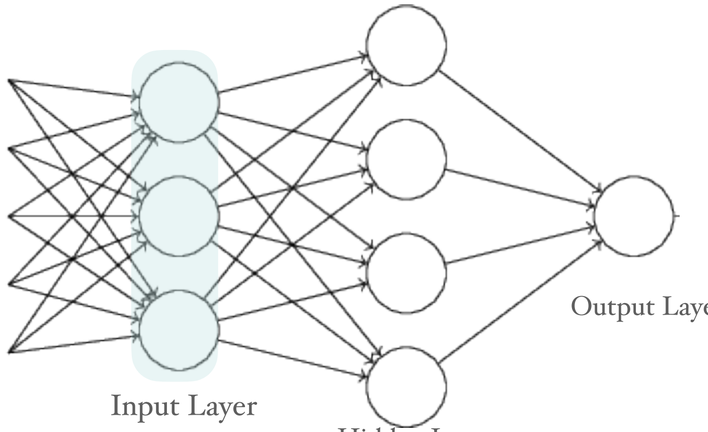
Perceptron Learning
(Linear Classifier)

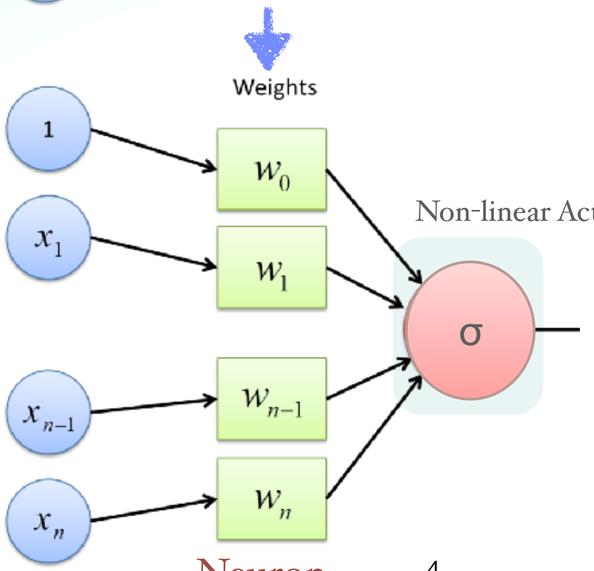


Deep Neural Networks
Adding more layers!



Layers
Many parallel neurons





Neuron

Concept

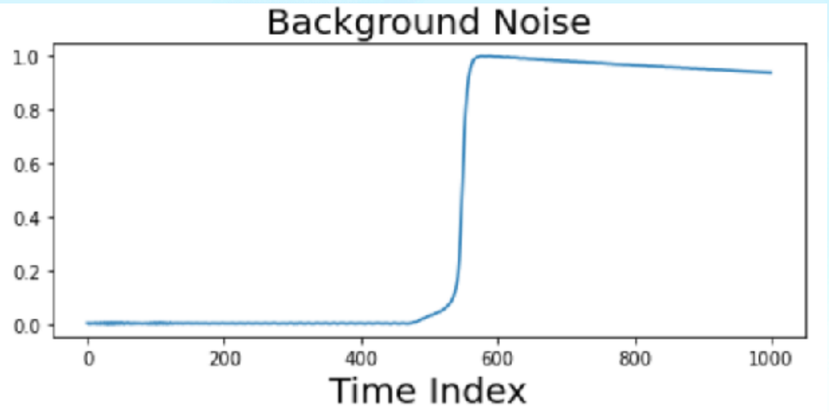
Elements of NN Layers:

- Input dimension
- Output dimension
- Activation function

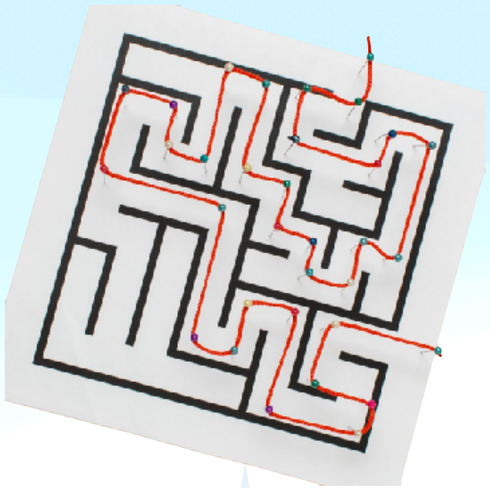
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

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

$$NNLayer(1000,512) \rightarrow \sigma$$
$$\rightarrow NNLayer(512, 256) \rightarrow \sigma$$
$$\rightarrow NNLayer(256, 32) \rightarrow \sigma$$
$$\rightarrow NNLayer(32, 1) \rightarrow \sigma$$

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

Linear Layer:
- Defined with input & output dim

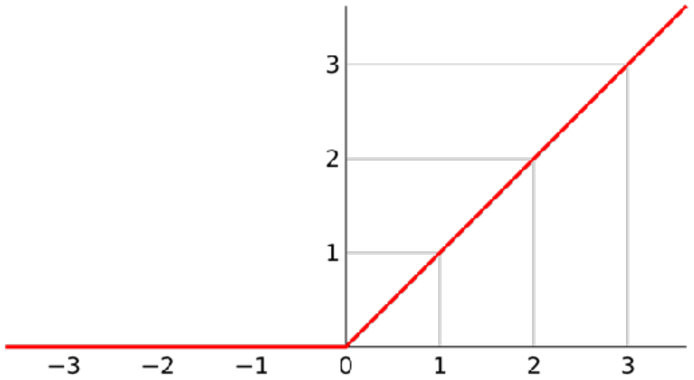
Forward Pass:
- Feed data through the neural network to obtain desired outputs
  - (BS, 1000,)→(BS, 1)
- **Called multiple times** during training and validation

Backward Pass will be discussed in Sec. 4!

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
```

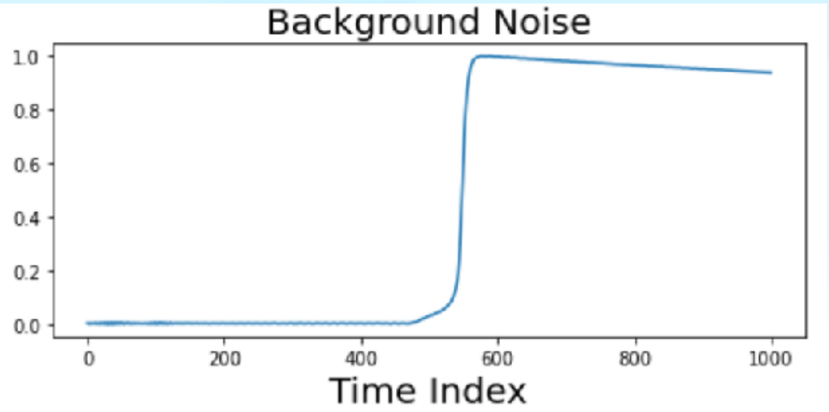
Activation Function:
- Adding non-linearity to NN
- ReLU is most commonly used



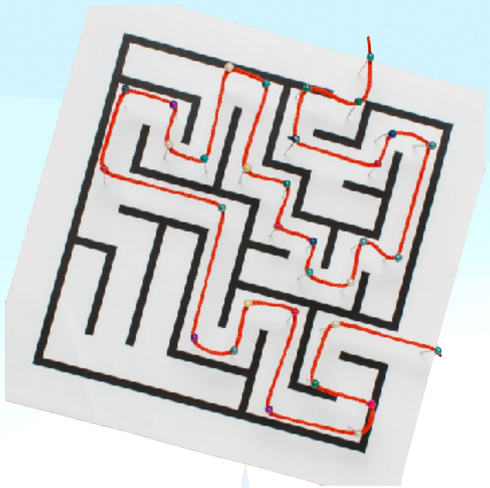
1. Data Preprocessing → 2. Feature Extraction → 3a. Classification → 4. Training → 5. Evaluation

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

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

Concept

Task Module: Task Layer + Loss Function

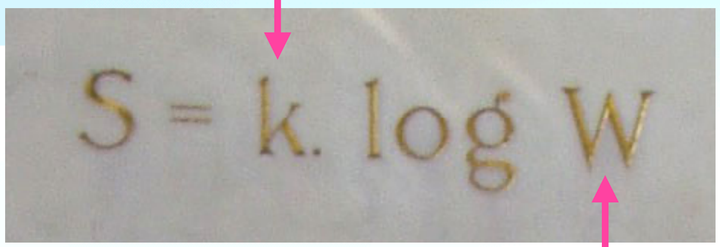
- **Task Layer:** Take low-dimensional vector as input and produce a value/vector as **output**
- **Loss Function:** produce a quantitative measure evaluating how well your algorithm models your dataset
- Minimizing the loss function means that the **model output** reproduces the **label** better

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

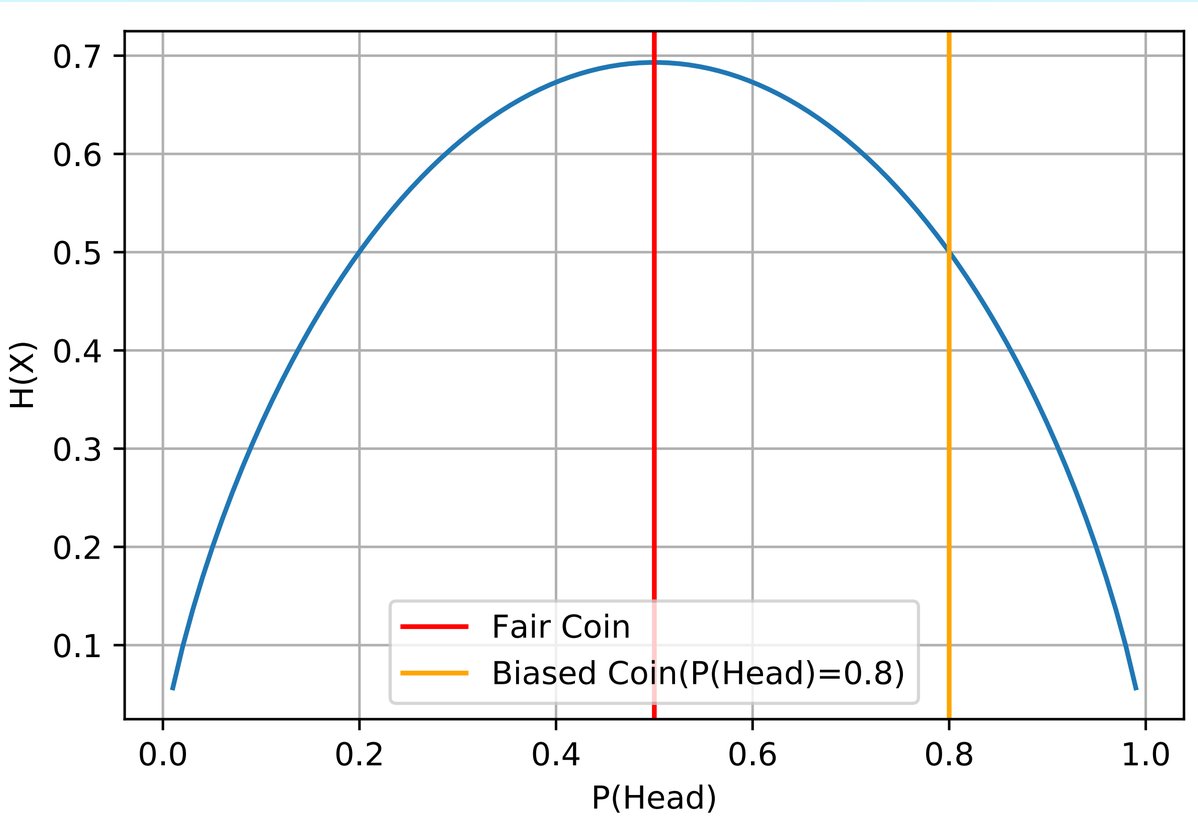
Binary entropy: $H(x) = p \log \frac{1}{p} - (1-p) log \frac{1}{1-p}$

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

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

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

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



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** $q(X)$

Concept

Binary Cross Entropy: label is either 1 or 0

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

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

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

NNLayer σ L

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

NNLayer σ L

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, n) → 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 *n* with two number between [-inf, inf]
- Target: the **array indices** of correct choice

NNLayer σ L

Concept

Regression 1

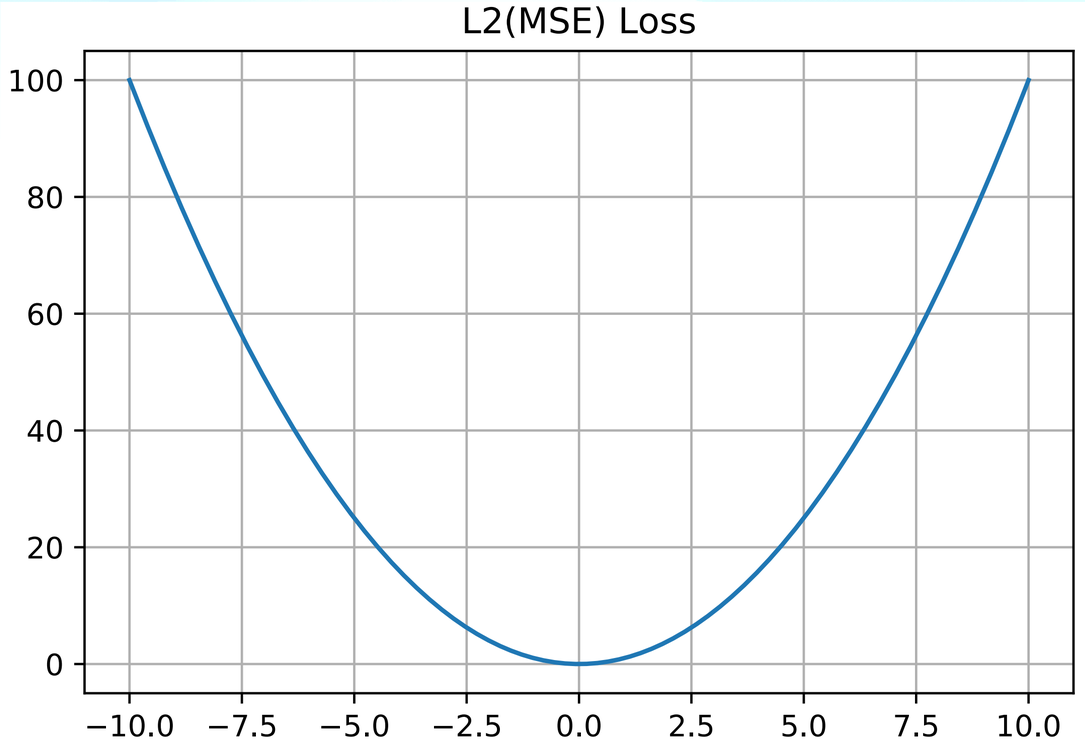
Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

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



NNLayer σ L

Concept

Regression 2

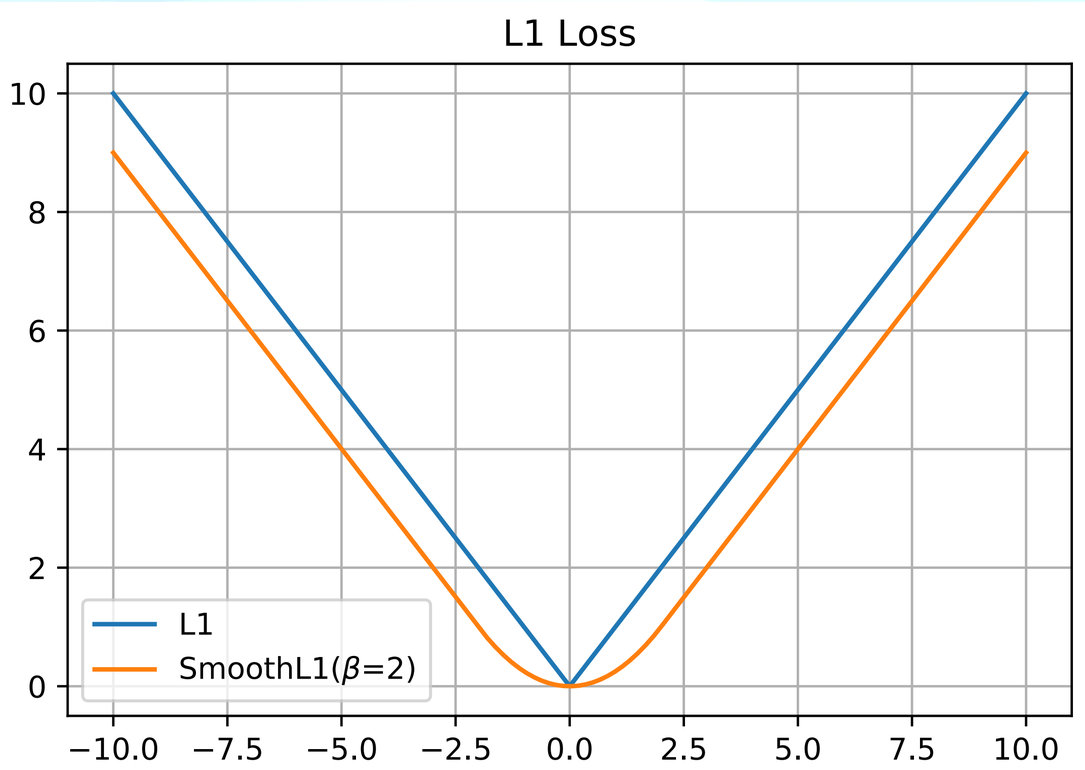
Task Layer: *NNLayer(32, 1) → One float point No. between [-inf, inf]*

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)
```

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

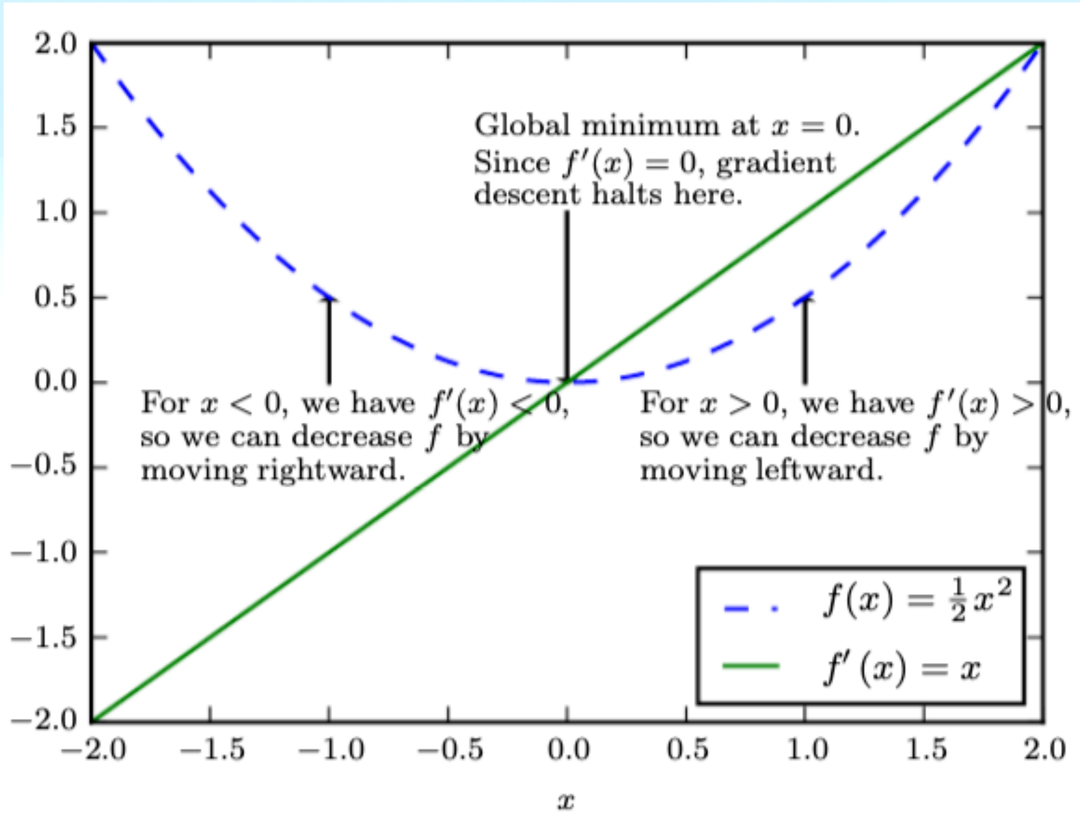
Replace **Task Layer**, **σ**, and **Loss Function** using the previously-provided concept templates to achieve different tasks!

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

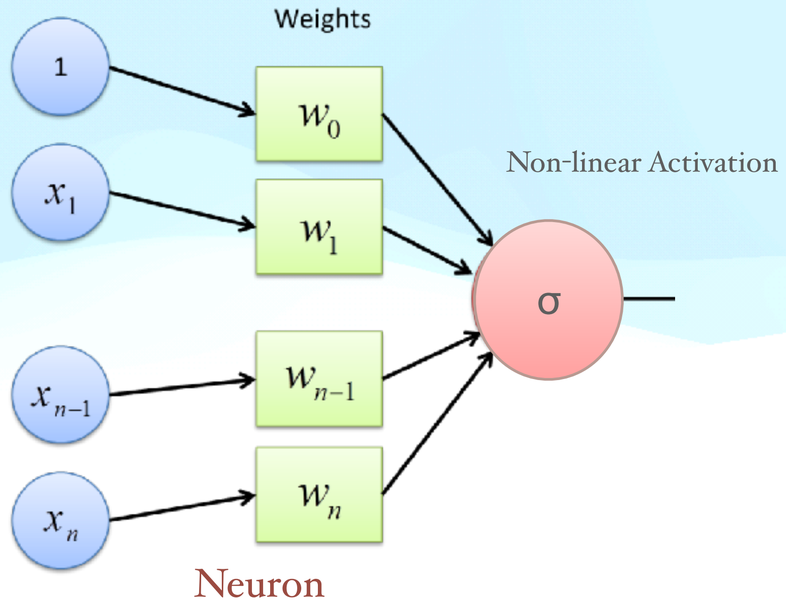


- Mainstream machine learning optimizers Adam, AdaGrad, RMSprop are first order optimization algorithms that only utilize gradient
- There are second order optimization algorithms which utilizes the Hessian matrix to avoid poorly conditioned probability space
  - $f(x^{(0)} - \gamma g) = f(x^{(0)}) -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

Theory

Expression of Neural Network:

- One Layer: $\hat{y} = \sigma(\alpha_m^T \vec{x}^i + \alpha_{0m})$
- Two Layer: $\hat{y} = \sigma_2(\beta_k^T \sigma(\alpha_m^T \vec{x}^i + \alpha_{0m}) + \beta_{0k})$
- (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 $\nabla_{\alpha,\beta} L(x, y \mid \alpha, \beta)$ is the key steps

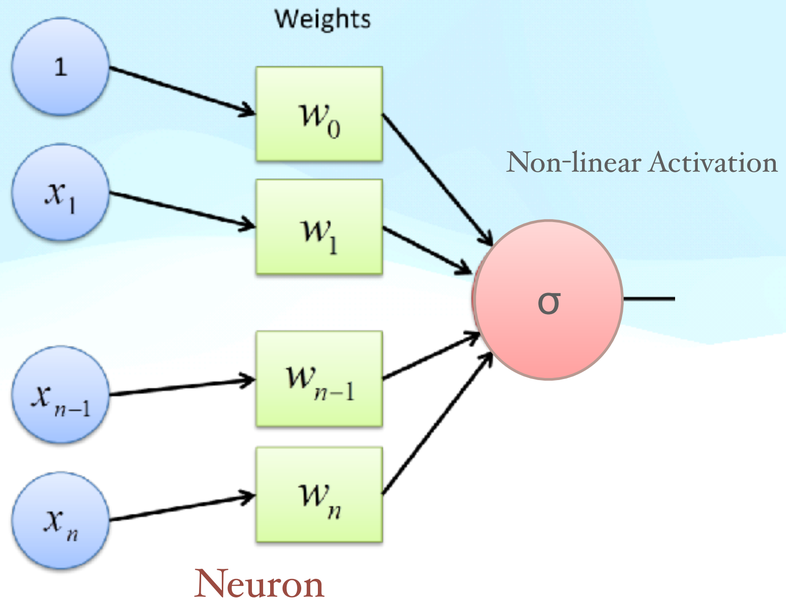
However, the gradient is supposed to be calculated on all input data simultaneously

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

Theory

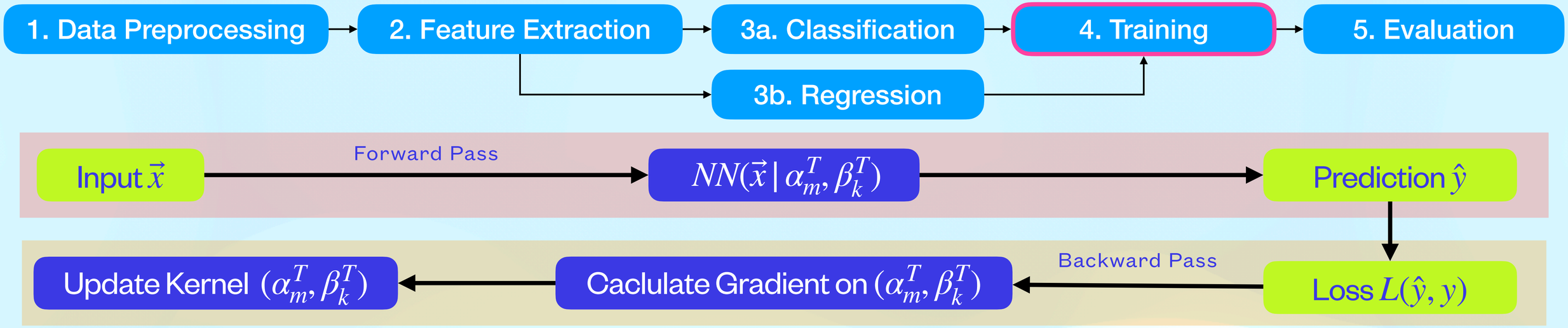
Expression of Neural Network:

- One Layer: $\hat{y} = \sigma(\alpha_m^T \vec{x}^i + \alpha_{0m})$
- Two Layer: $\hat{y} = \sigma_2(\beta_k^T \sigma(\alpha_m^T \vec{x}^i + \alpha_{0m}) + \beta_{0k})$
- (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:

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

$$\downarrow$$

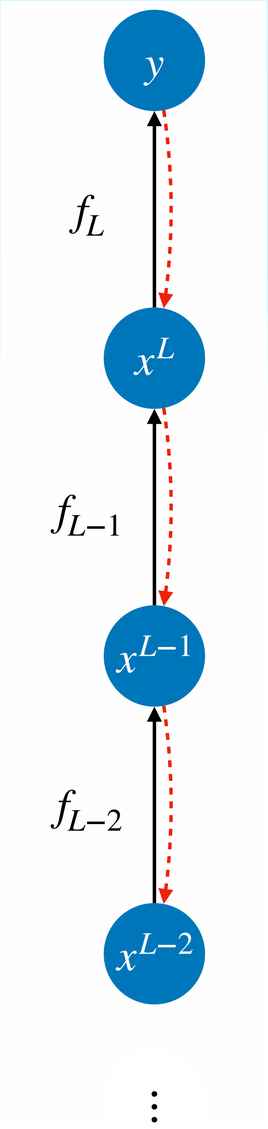
$$y = f_L(x_L)$$

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

$$\vdots$$

$$X_1 = f_1(x_1)$$

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



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

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

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

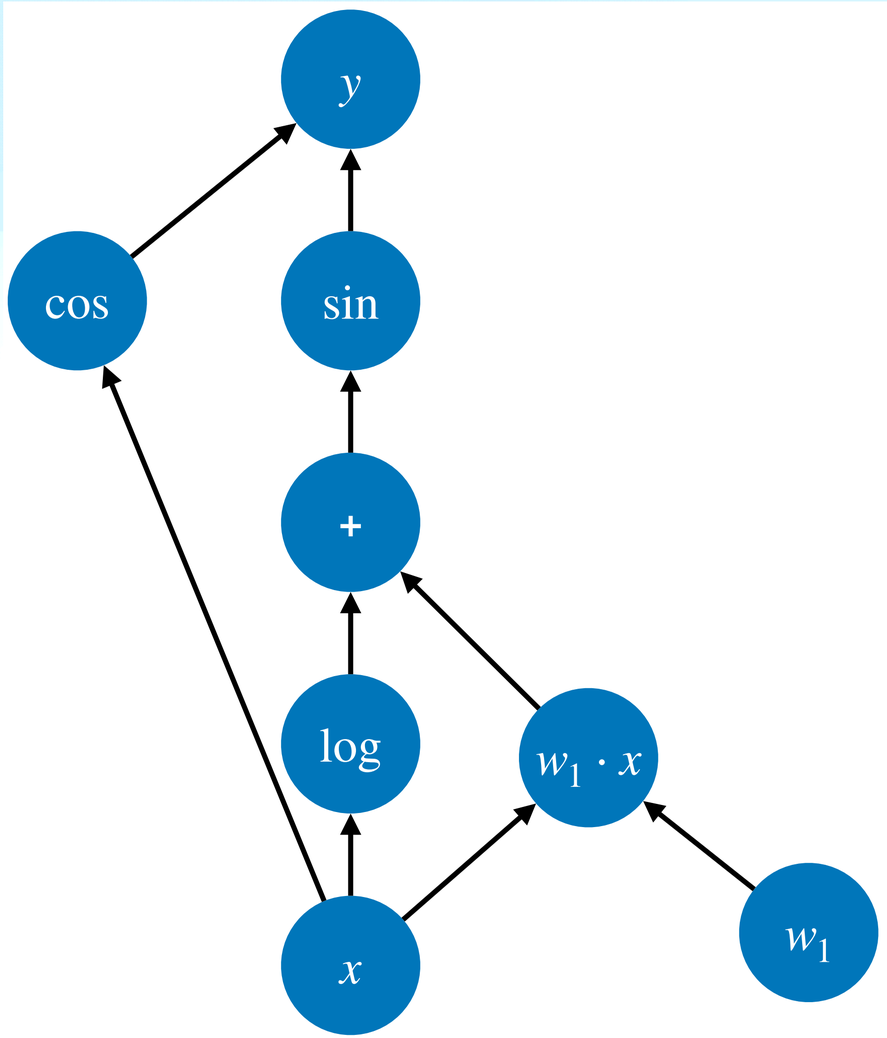
$$\frac{\partial y}{\partial x_{L-3}} = \frac{\partial y}{\partial x_{L-2}}\frac{\partial x_{L-2}}{\partial x^{L-3}}$$

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

Concept

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

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



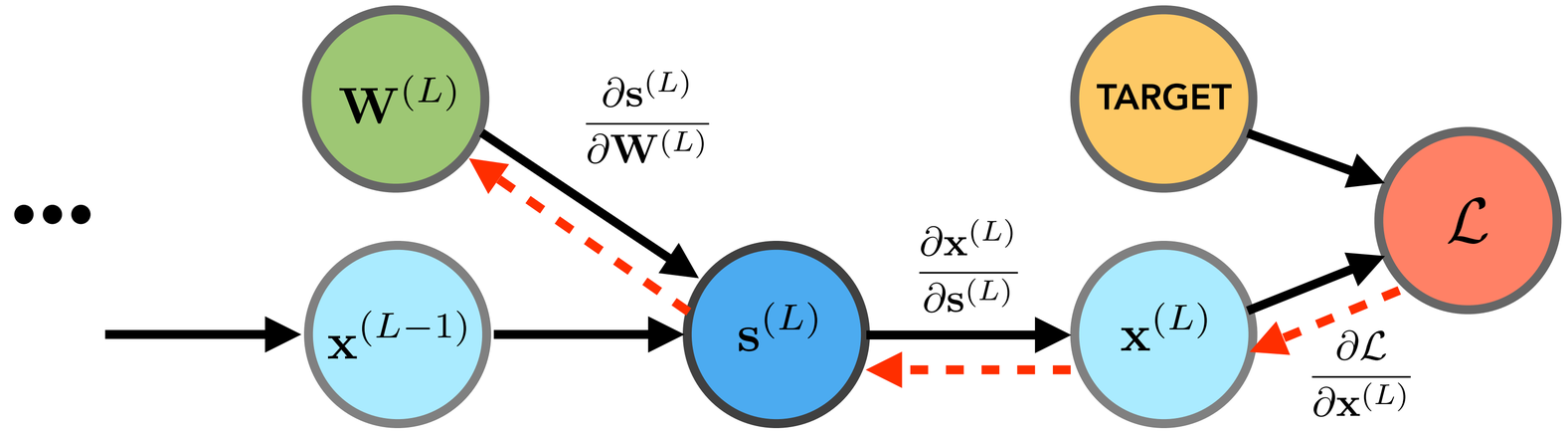
Modern deep neural networks are also computational graph! This means can calculate gradient along the graph.

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

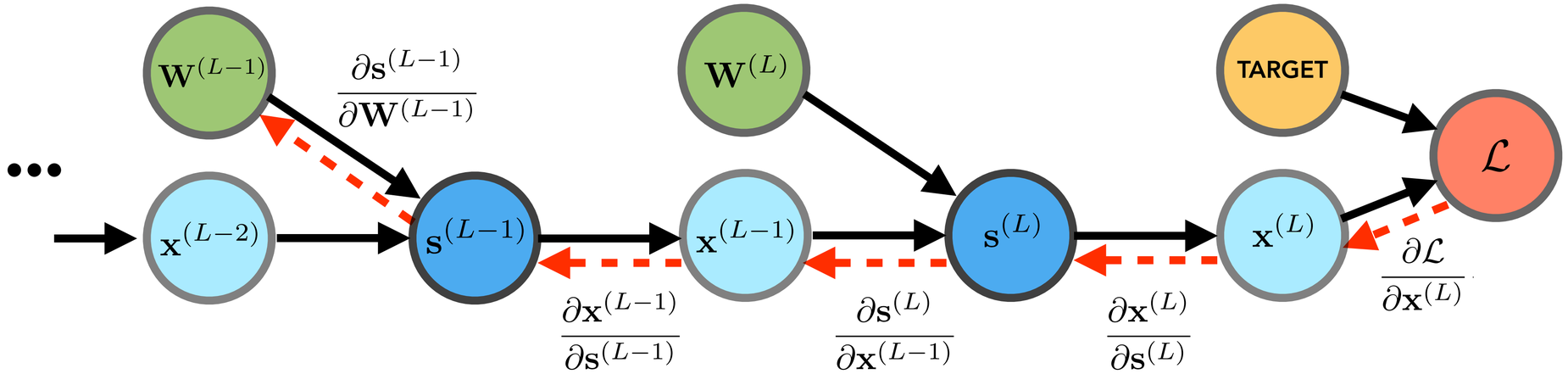
derivative of the non-linearity

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

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:



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

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)

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

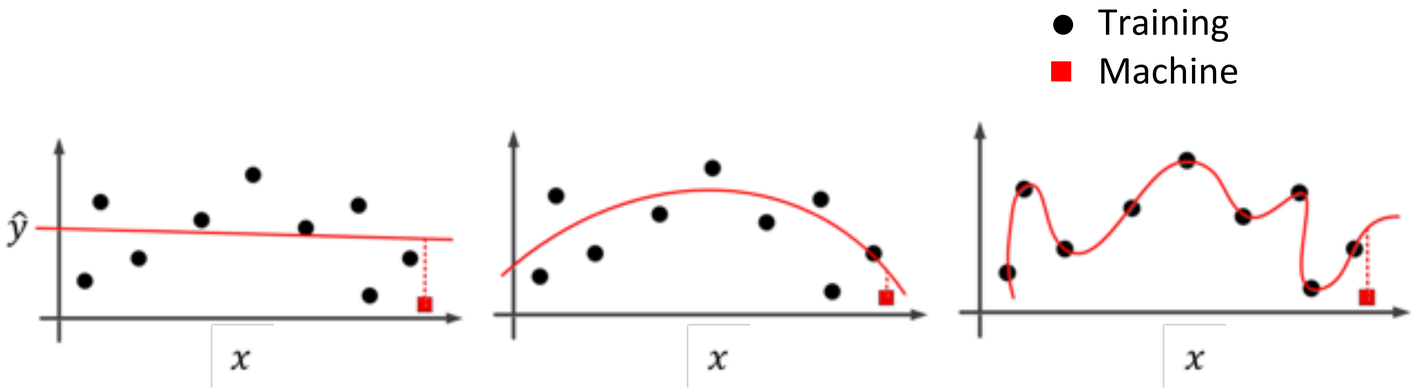
This line updates model parameters according to calculated gradients

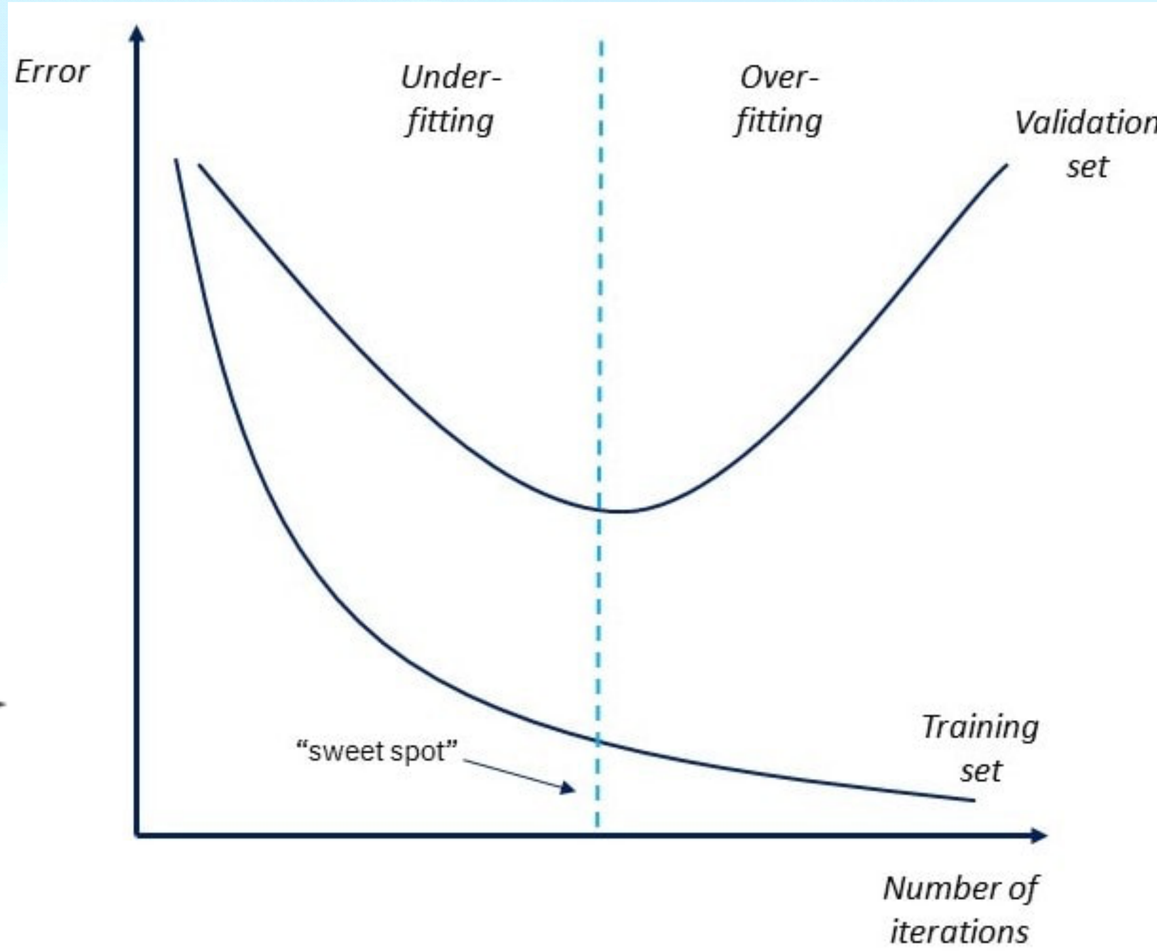
This line reset the gradient to 0 at the end of current batch, so that we are ready to train the next batch of data

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



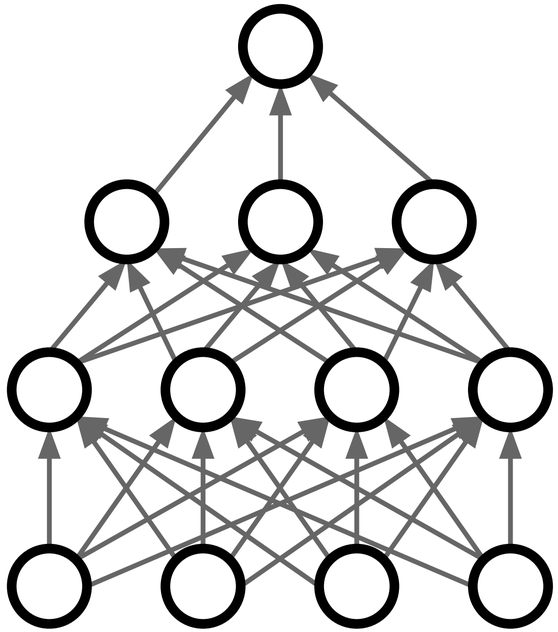


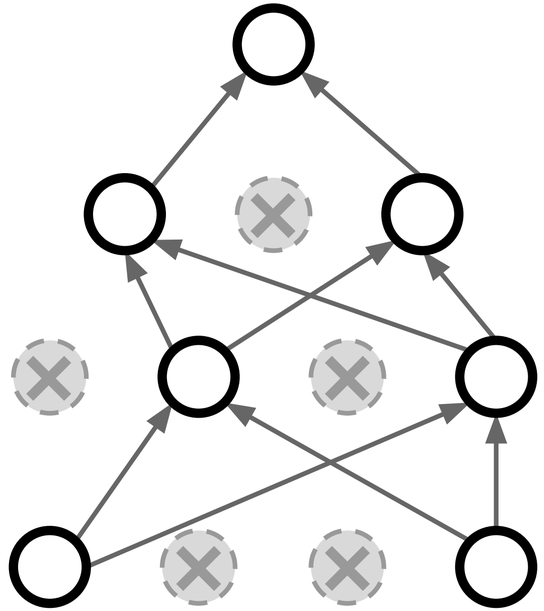
Concept

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

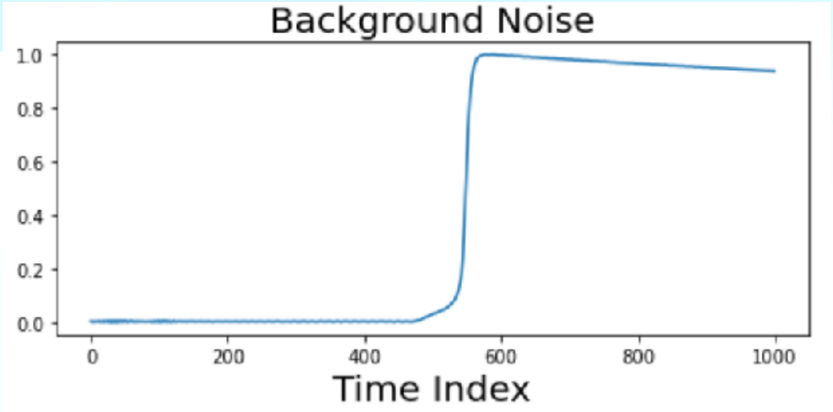
Code

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





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 Module

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()*

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	**Tru**	**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)**

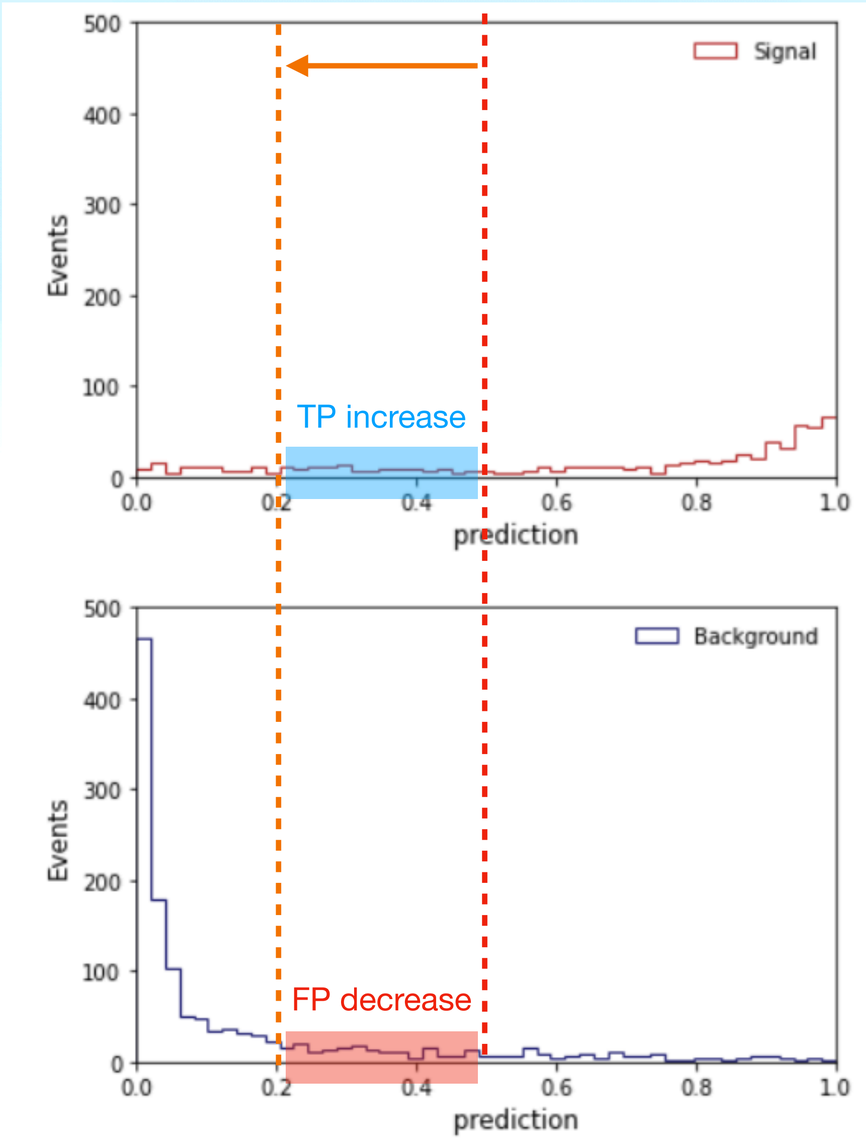
Concept

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



- 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!

Concept

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

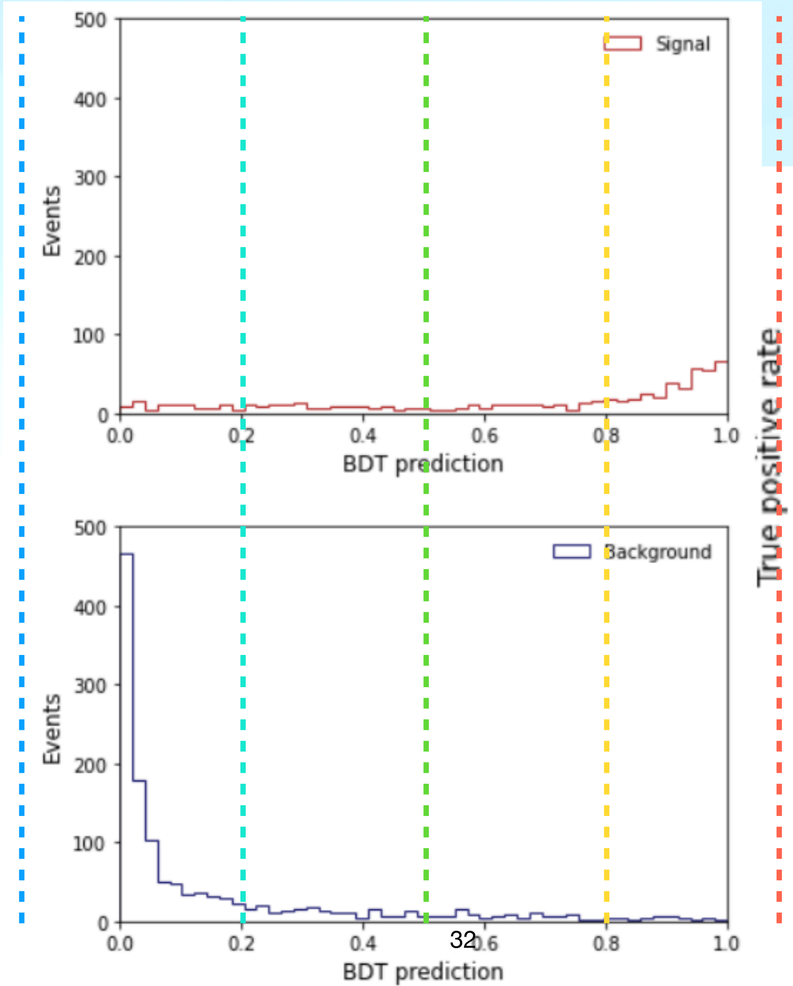
Cutting at -0.1:
- TPR = 100%
- FPR = 100%

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

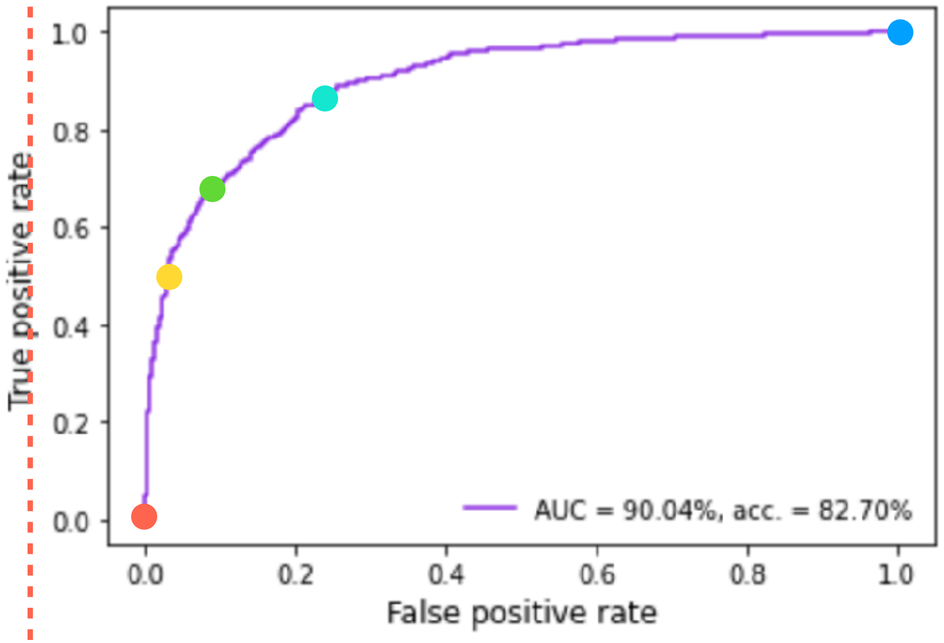
Cutting at 0.5:
- TPR = 70%
- FPR = 10%

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

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



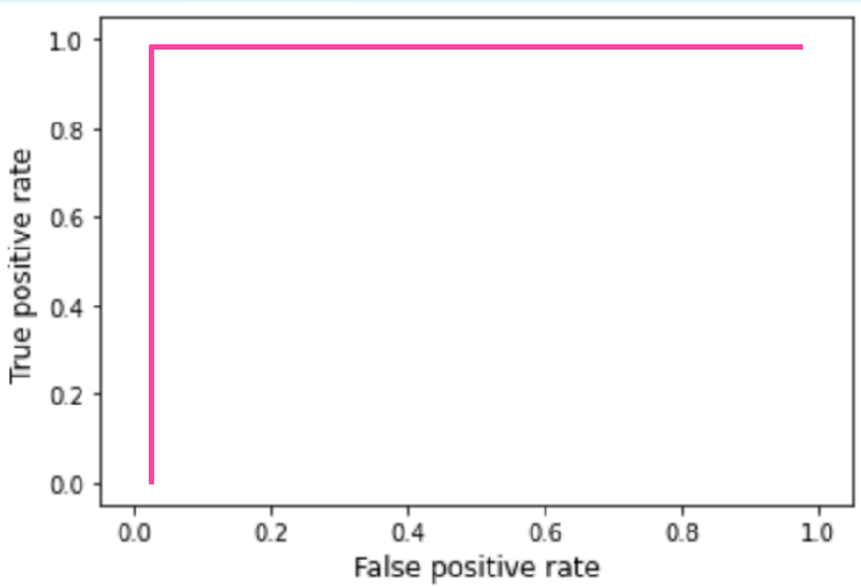
All ROC curves start at (1,1)



All ROC curves ends at (0,0)

Perfect ROC Curve

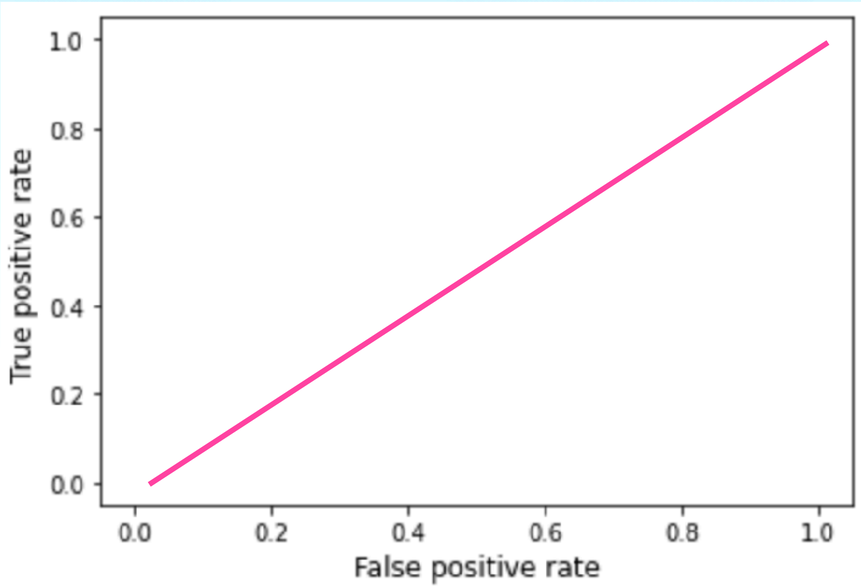
- TPR = 100% at FPR = 0%
- Area under curve = 1
- Every event is classified correctly



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

Worst ROC Curve

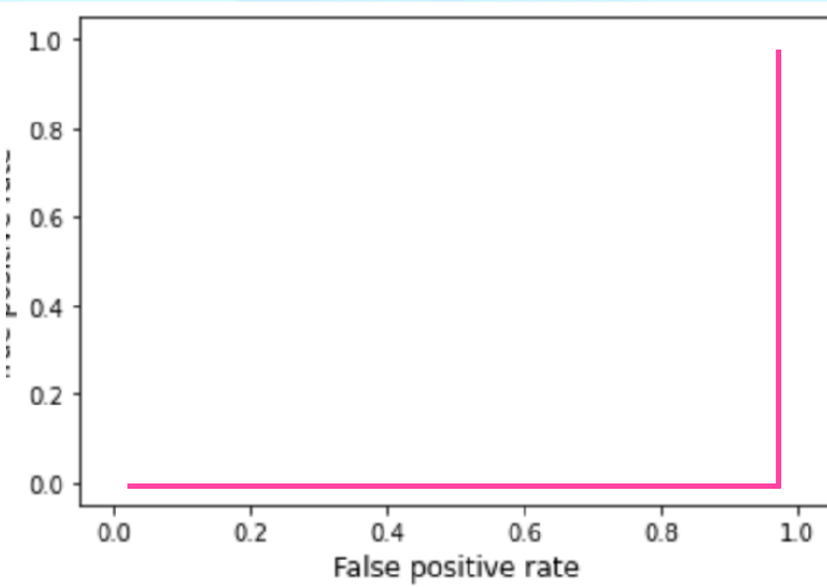
- TPR == FPR
- Area under curve = 0.5
- Classify by tossing a coin



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

Anti-perfect ROC Curve

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



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

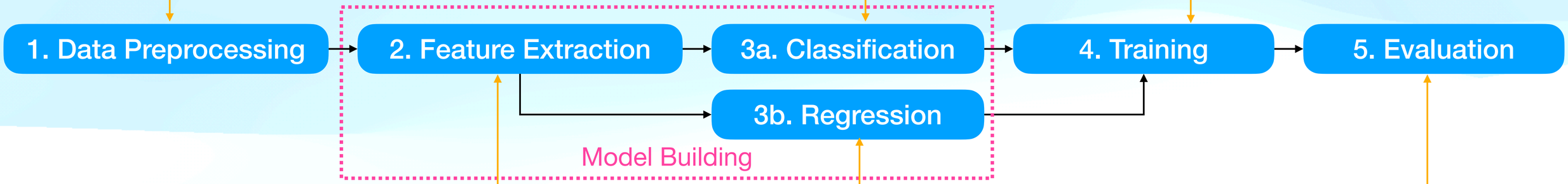
Concept

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. 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. **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

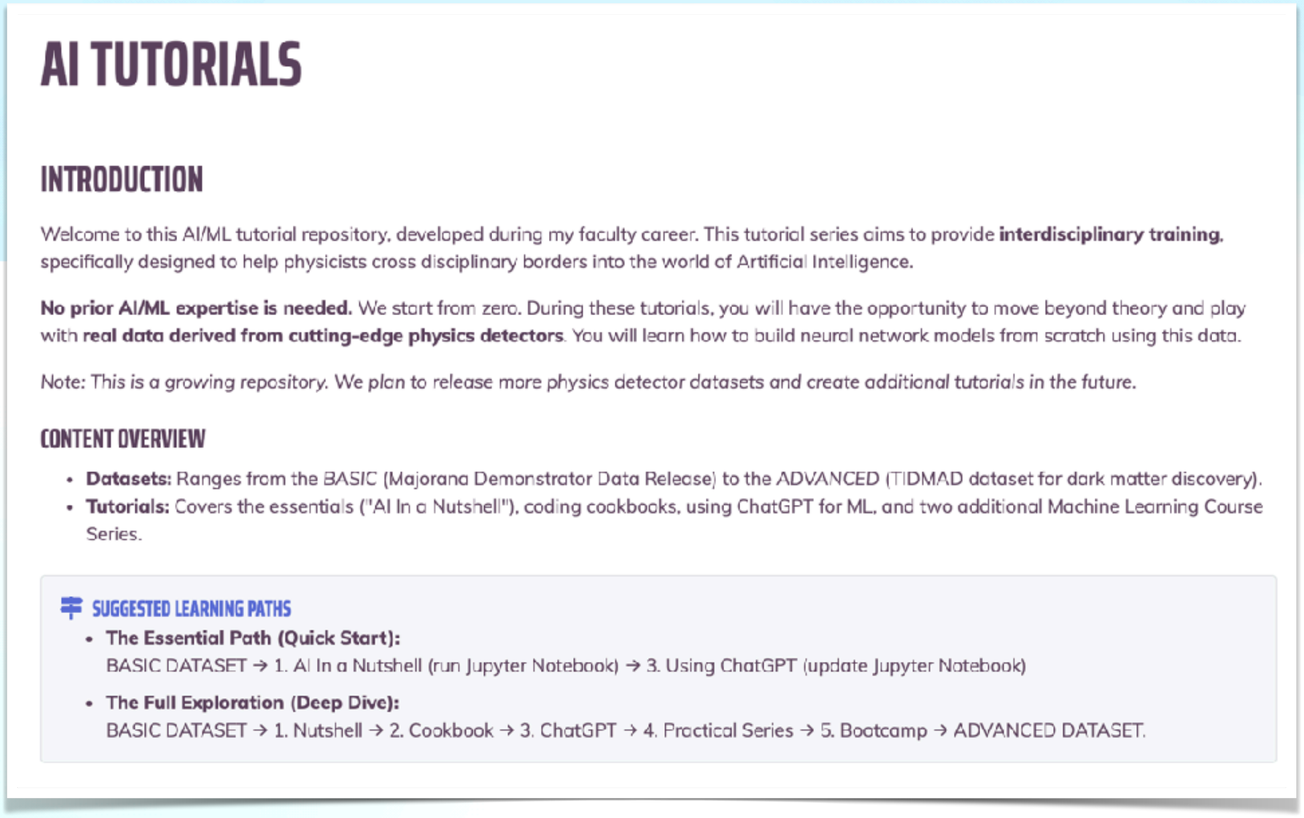
1. **Overfitting** vs. Underfitting
2. Evaluate binary classication: **ROC Analysis**
3. Evaluate regression: reconstruction algorithm

Rare AI Lab - Interdisciplinary AI Education

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

Data Science for Physicists: teach physics students to build AI models from zero background

Physics for Data Scientist: invoke data science students' interest in physics research



AI TUTORIALS

INTRODUCTION

Welcome to this AI/ML tutorial repository, developed during my faculty career. This tutorial series aims to provide **interdisciplinary training**, specifically designed to help physicists cross disciplinary borders into the world of Artificial Intelligence.

No prior AI/ML expertise is needed. We start from zero. During these tutorials, you will have the opportunity to move beyond theory and play with **real data derived from cutting-edge physics detectors**. You will learn how to build neural network models from scratch using this data.

Note: This is a growing repository. We plan to release more physics detector datasets and create additional tutorials in the future.

CONTENT OVERVIEW

- **Datasets:** Ranges from the BASIC (Majorana Demonstrator Data Release) to the ADVANCED (TIDMAD dataset for dark matter discovery).
- **Tutorials:** Covers the essentials ("AI In a Nutshell"), coding cookbooks, using ChatGPT for ML, and two additional Machine Learning Course Series.

SUGGESTED LEARNING PATHS

- **The Essential Path (Quick Start):**
  BASIC DATASET → 1. AI In a Nutshell (run Jupyter Notebook) → 3. Using ChatGPT (update Jupyter Notebook)
- **The Full Exploration (Deep Dive):**
  BASIC DATASET → 1. Nutshell → 2. Cookbook → 3. ChatGPT → 4. Practical Series → 5. Bootcamp → ADVANCED DATASET.

Online AI Tutorials

https://aobol.github.io/AoboLi/#tutorials

UCSD SMASH & NSF HDR Hackathon

https://indico.cern.ch/event/1624615/

0νββ AI Summer School

A Neutrino26 Satellite Event

https://indico.physics.ucsd.edu/event/2/

Neutrino Physics & Machine Learning

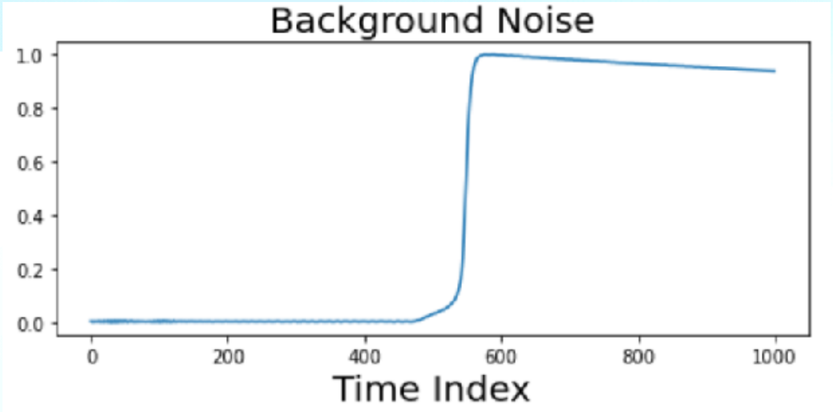
A Neutrino26 Satellite Event

https://sites.uci.edu/npml2026/

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]*

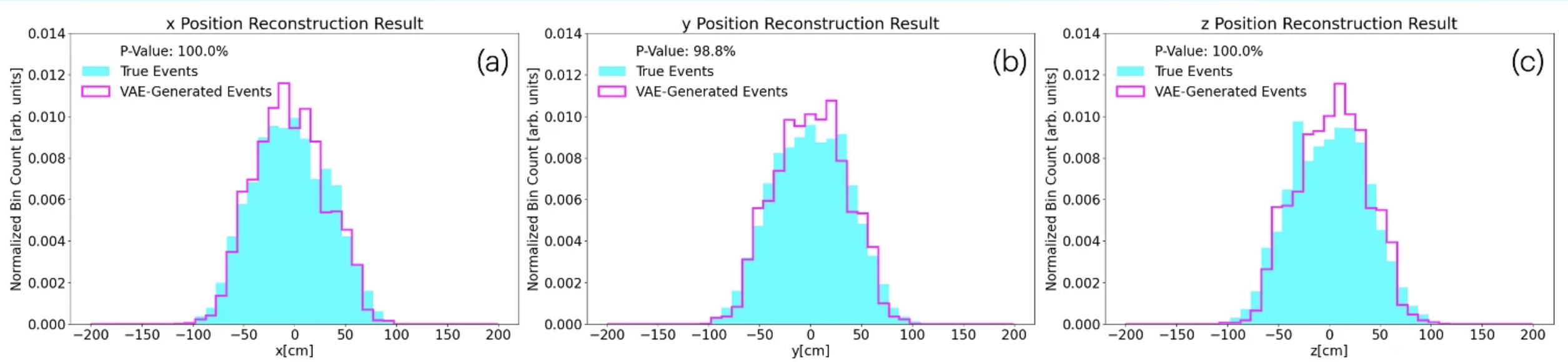
σ:

- 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