INTRODUCTION TO PYTORCH

VENKATAKRISHNAN V K



WHAT IS MACHINE LEARNING?

ANYONE?

MACHINE LEARNING

INTRODUCTION

Algorithms capable of **learning from data** and make inferences from unseen data

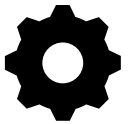
The pre-ML era approach



Problem Statement



Solve for the logic



Implement algorithm

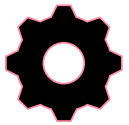
The ML approach



Problem Statement



Learn from data



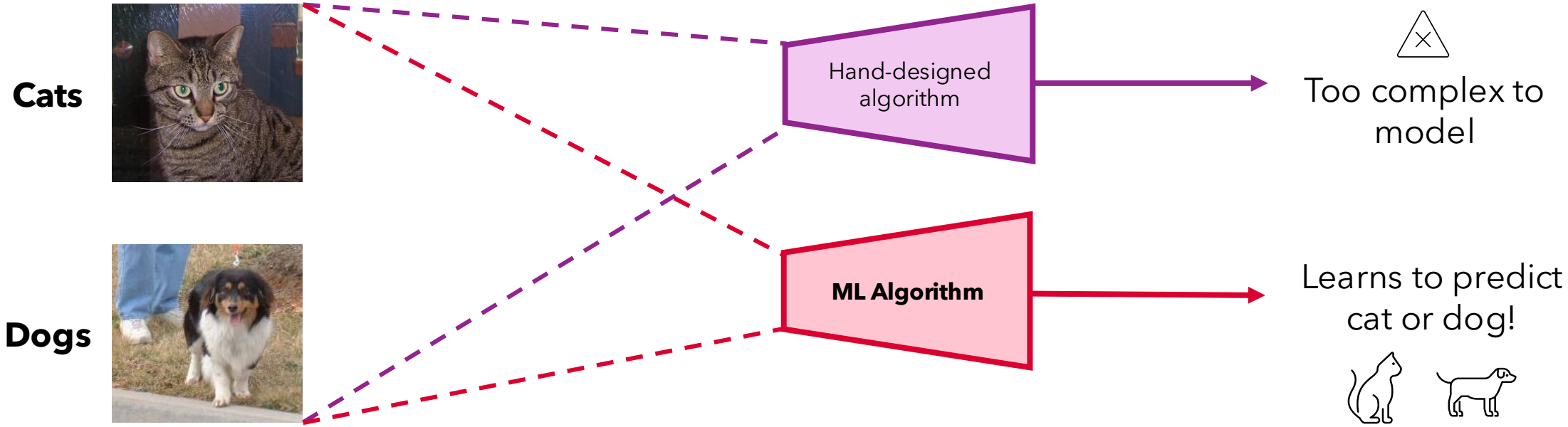
Trained model!

MACHINE LEARNING

WHY DO WE NEED ML?

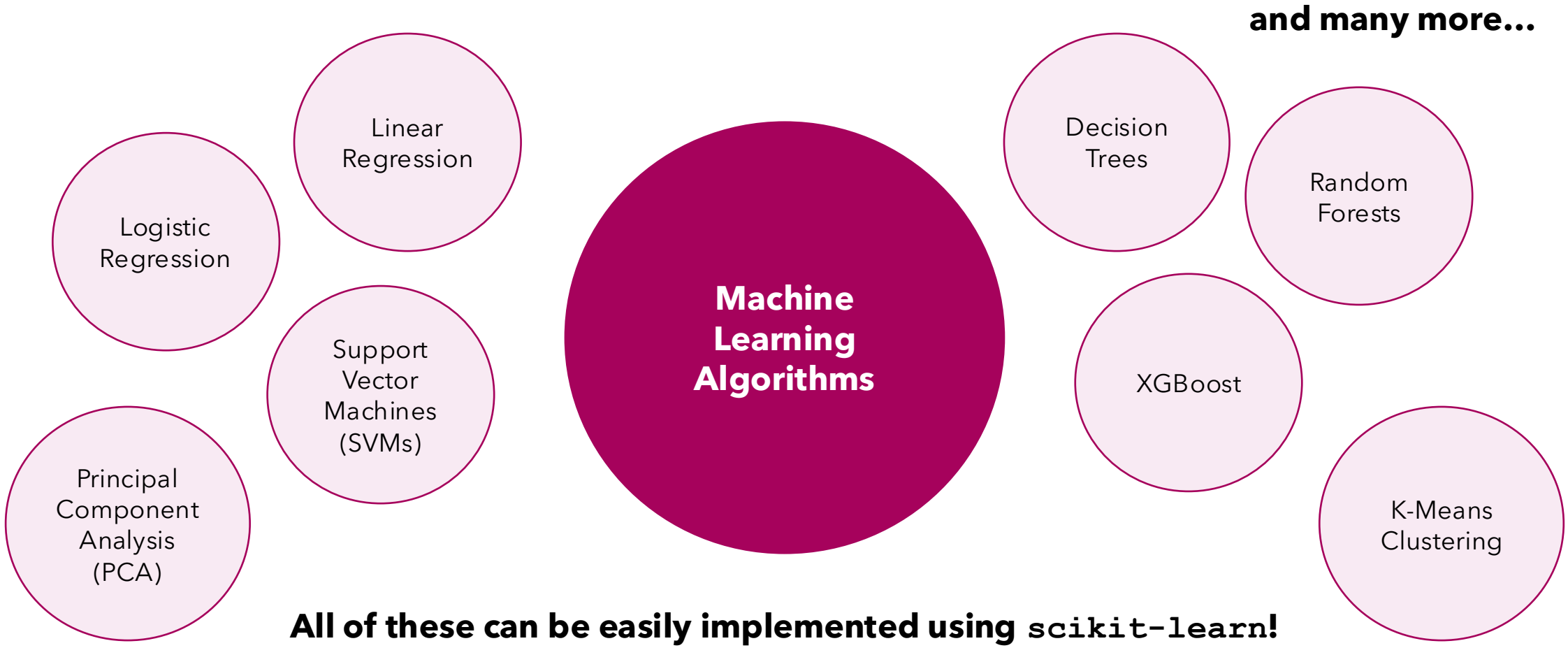
Complex problems get too messy with hand-designed algorithms

Consider a simple classification problem: **cat – dog image classification**



MACHINE LEARNING

ALGORITHMS



All of these can be easily implemented using `scikit-learn`!

MACHINE LEARNING

LIMITATIONS

Requires careful feature engineering

Difficulty in handling raw unstructured data

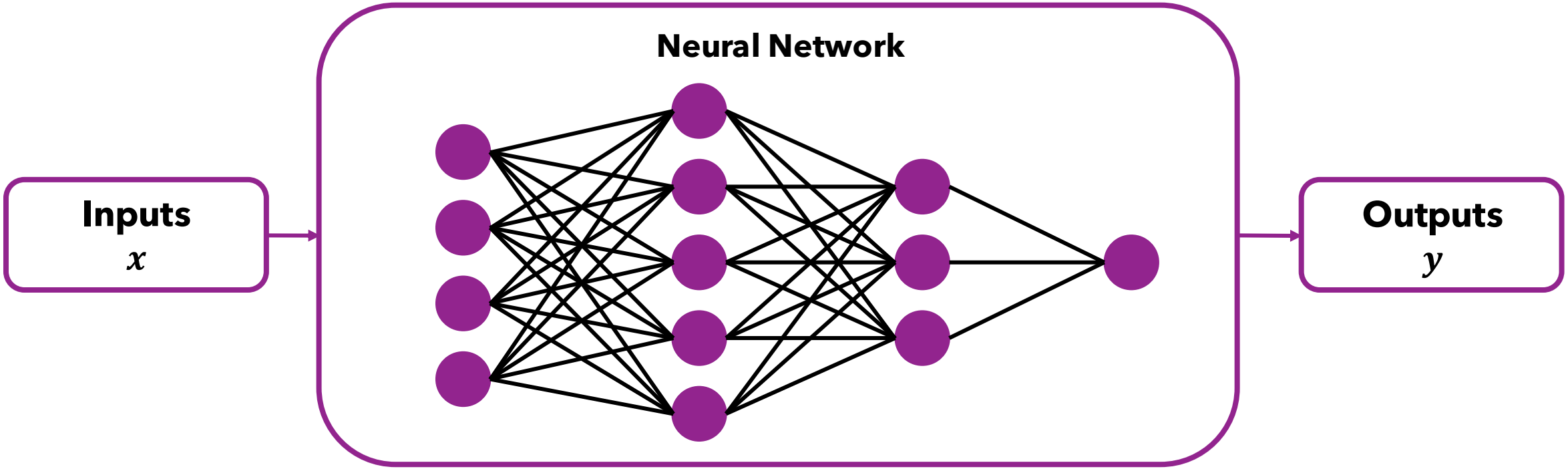
Performance plateaus for high dimensional data

Solution
Deep Learning!

DEEP LEARNING

INTRODUCTION

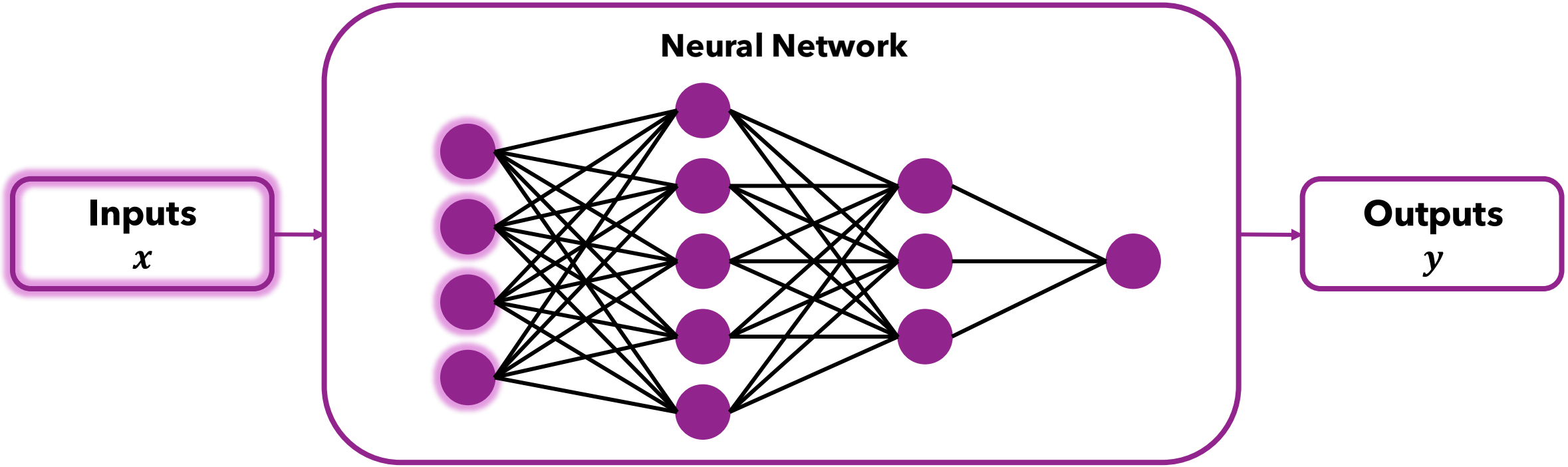
Modelling nonlinear patterns in data using multi-layer neural networks!



DEEP LEARNING

INTRODUCTION

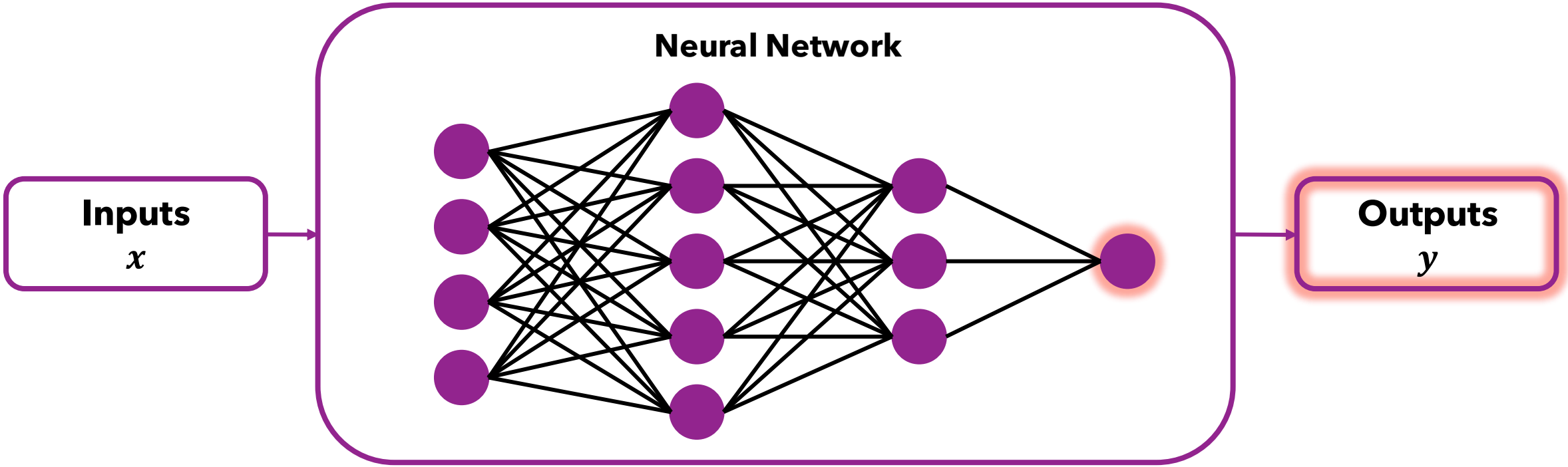
Modelling nonlinear patterns in data using multi-layer neural networks!



DEEP LEARNING

INTRODUCTION

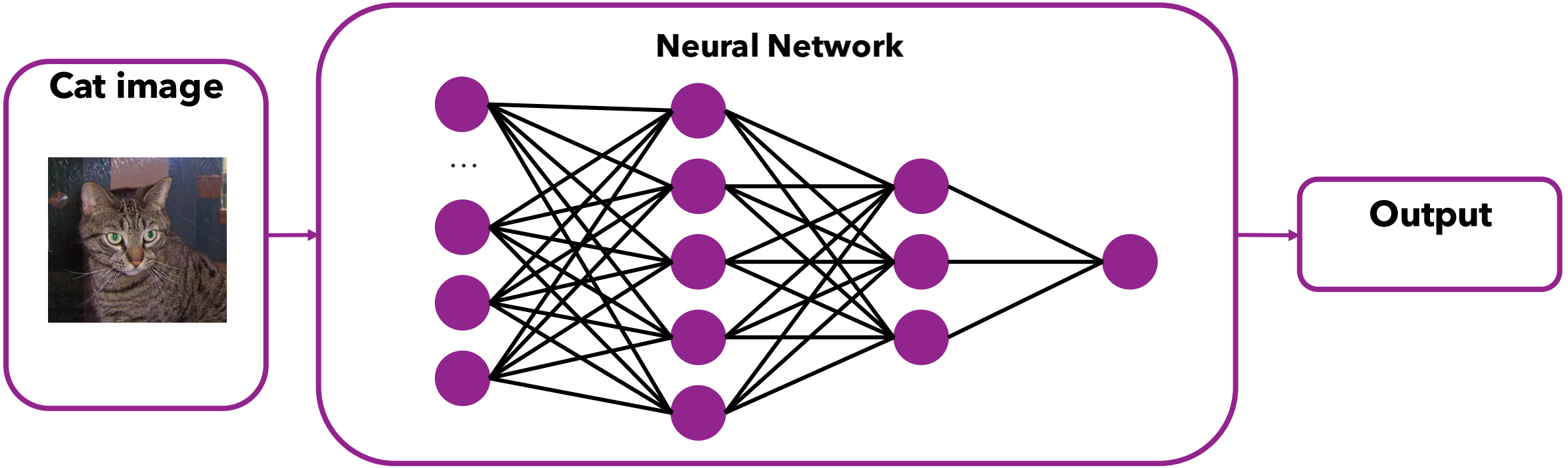
Modelling nonlinear patterns in data using multi-layer neural networks!



DEEP LEARNING

INTRODUCTION

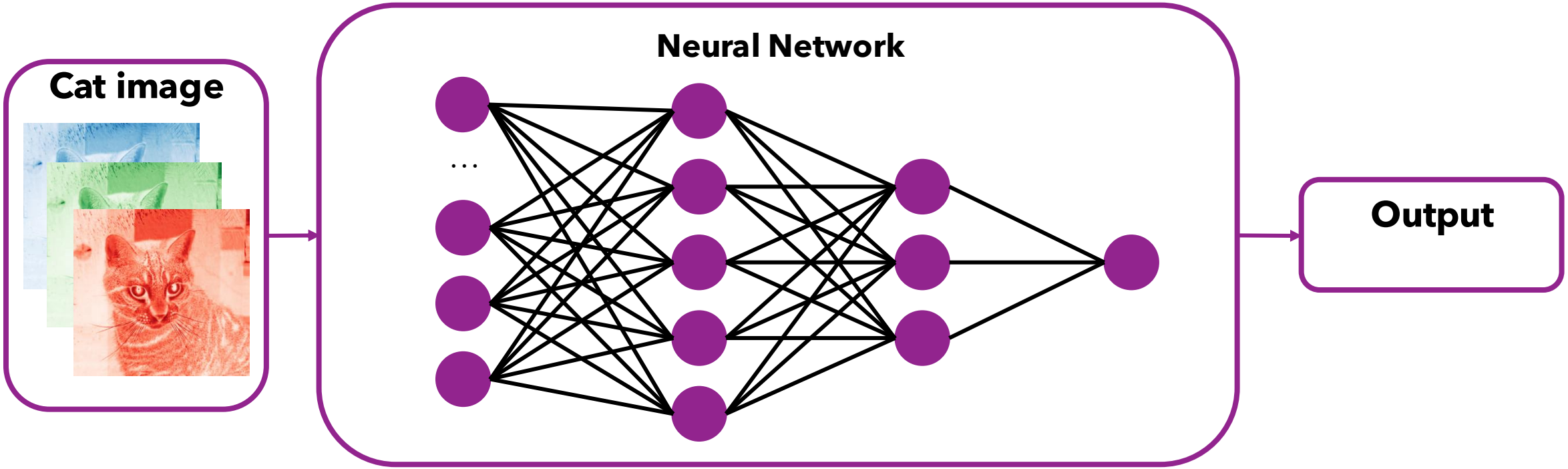
Modelling nonlinear patterns in data using multi-layer neural networks!



DEEP LEARNING

INTRODUCTION

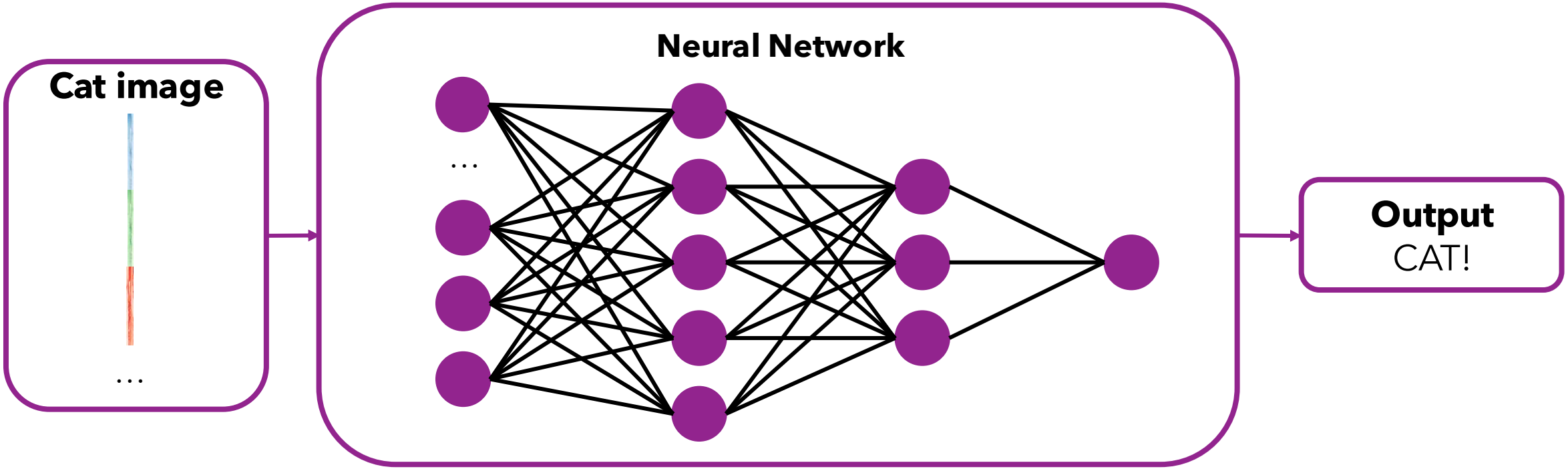
Modelling nonlinear patterns in data using multi-layer neural networks!



DEEP LEARNING

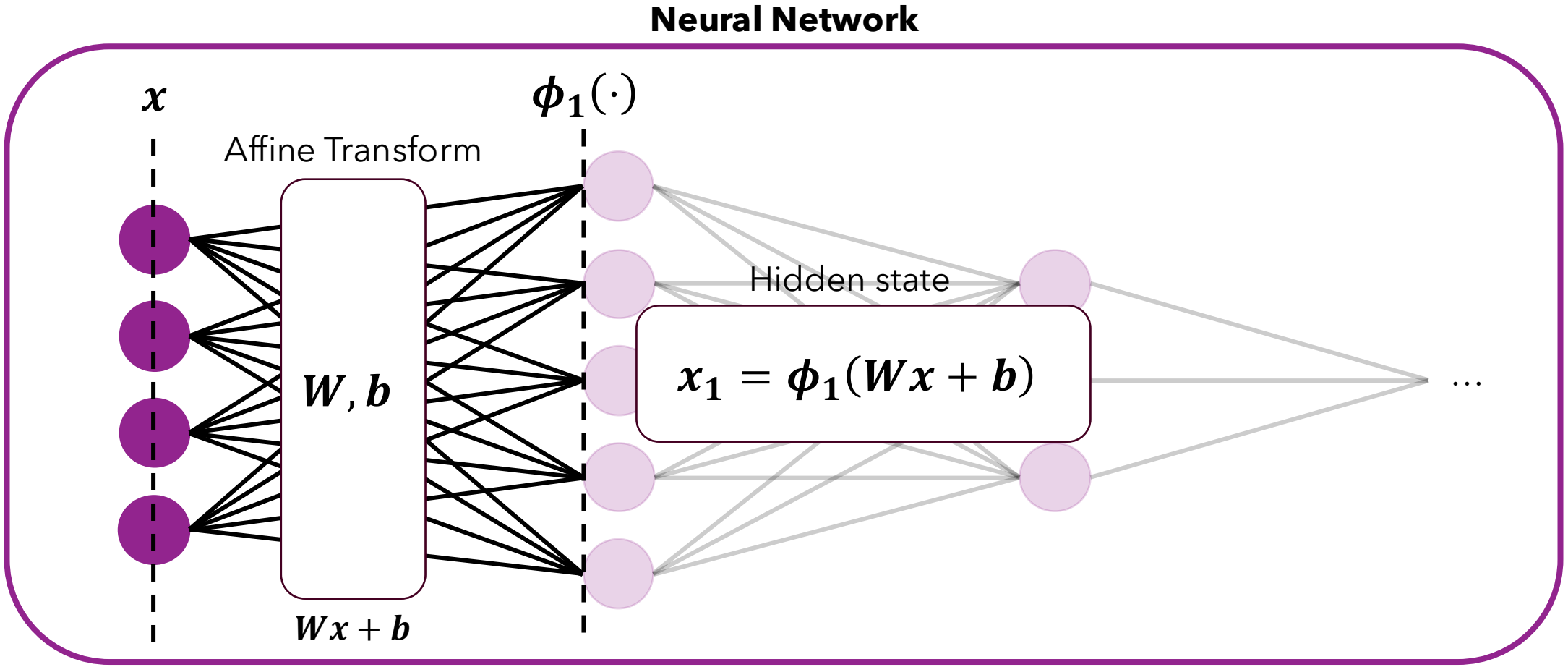
INTRODUCTION

Modelling nonlinear patterns in data using multi-layer neural networks!



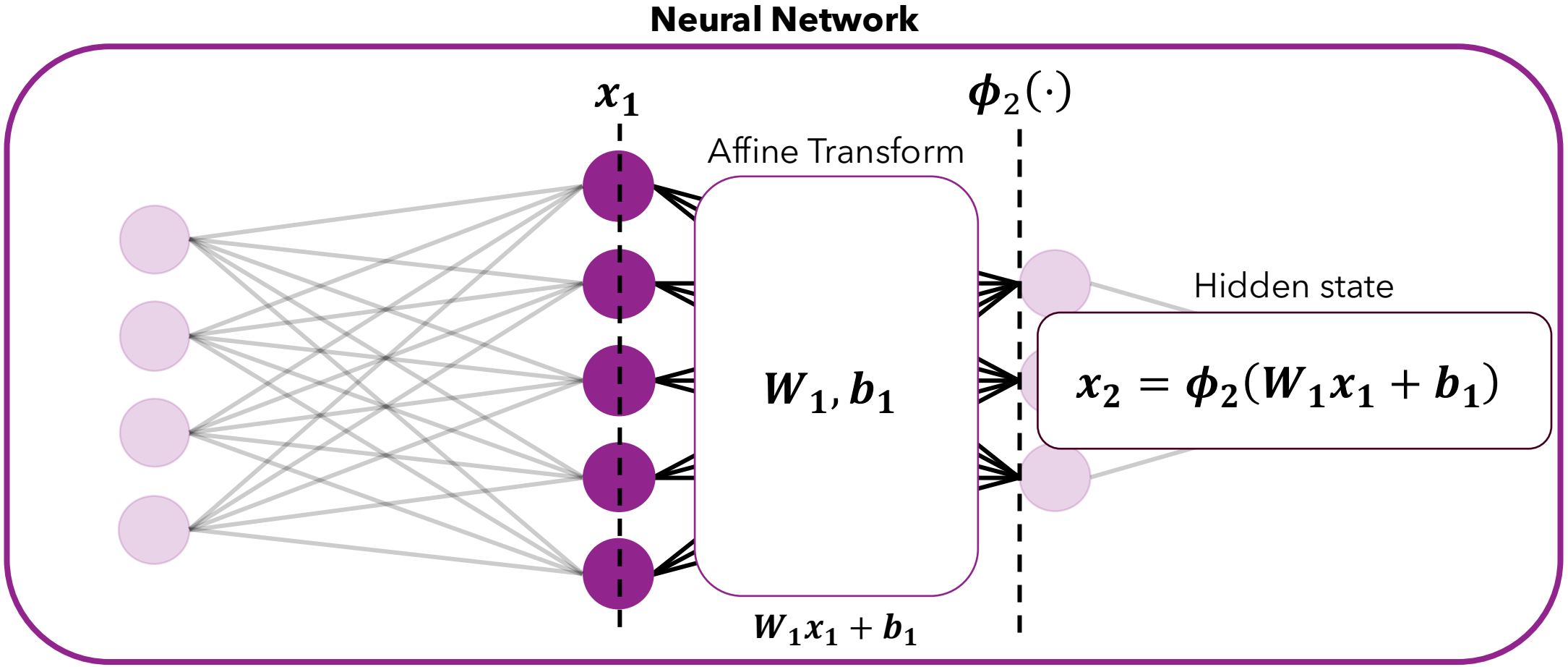
DEEP LEARNING

INTRODUCTION



DEEP LEARNING

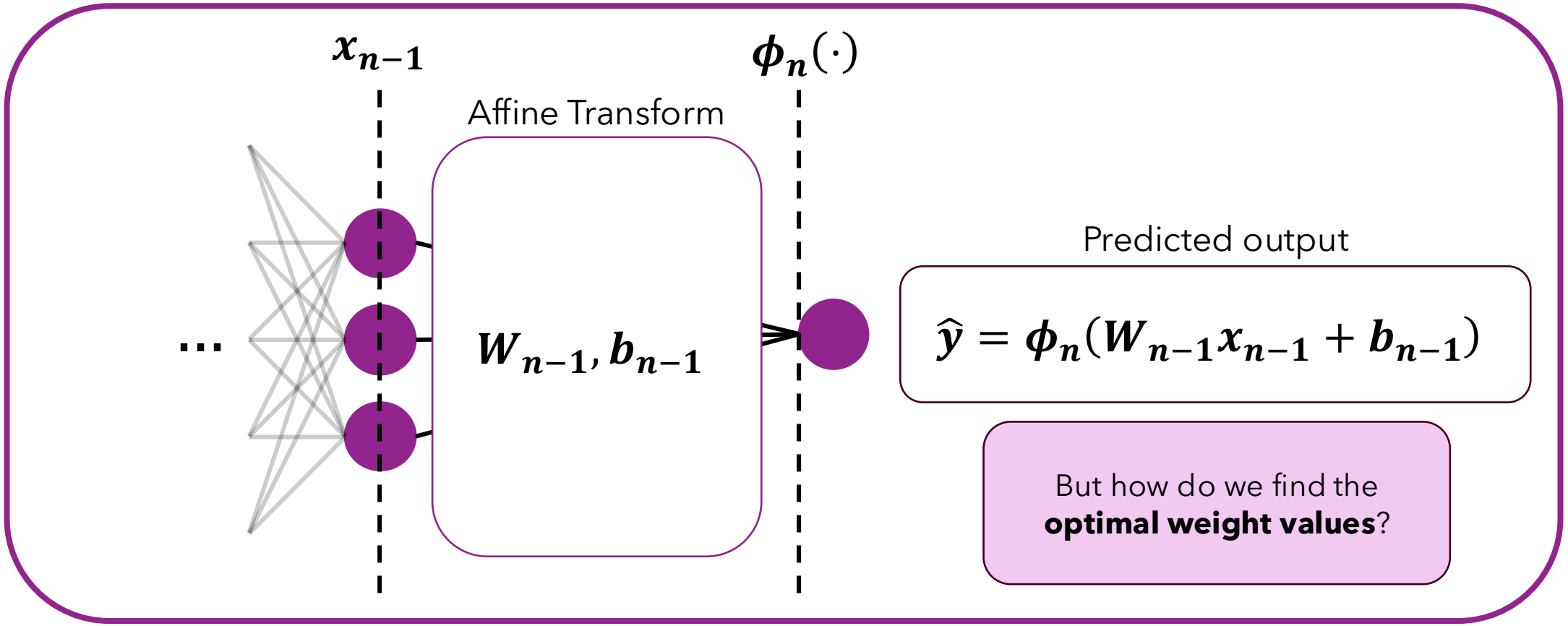
INTRODUCTION



DEEP LEARNING

INTRODUCTION

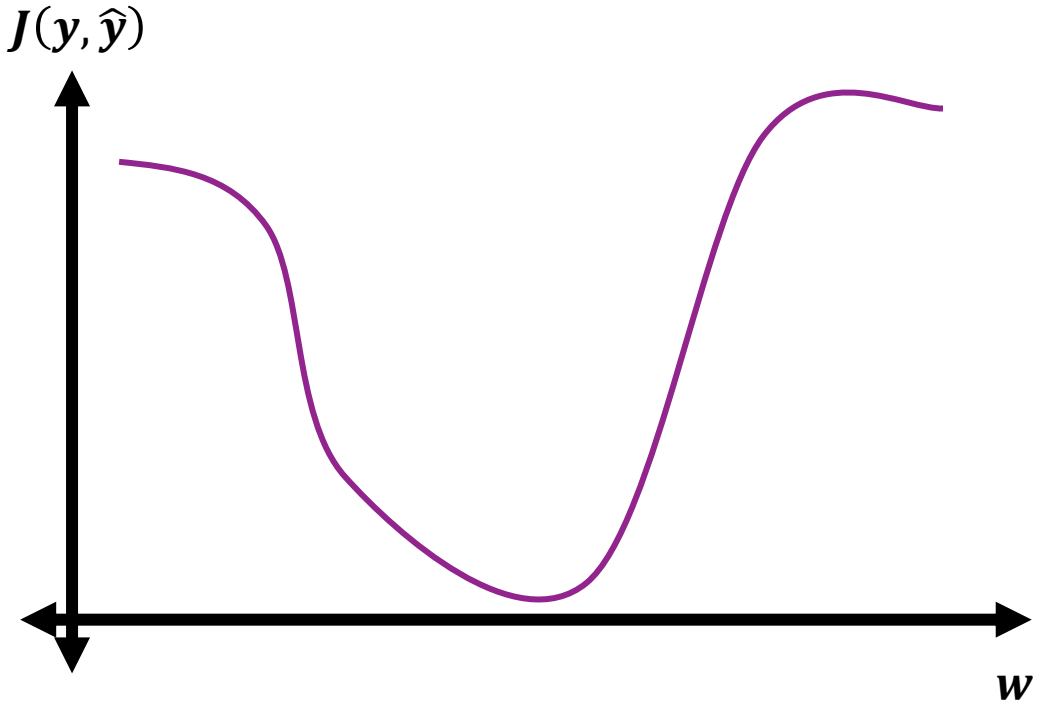
Neural Network

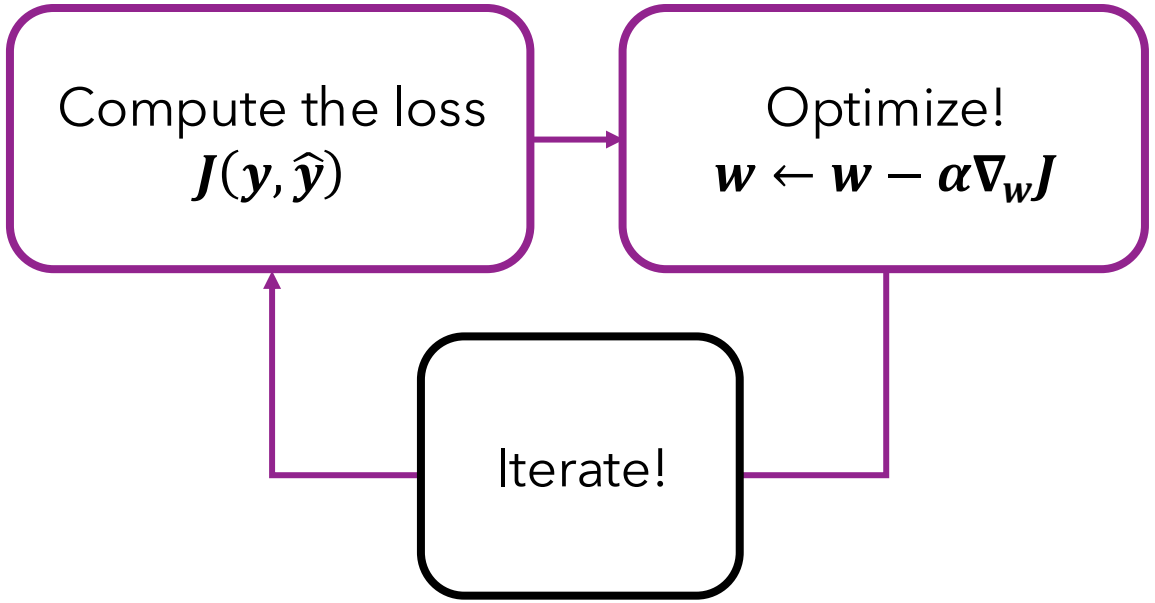


DEEP LEARNING

GRADIENT DESCENT

An iterative optimization technique to converge towards optimal parameters



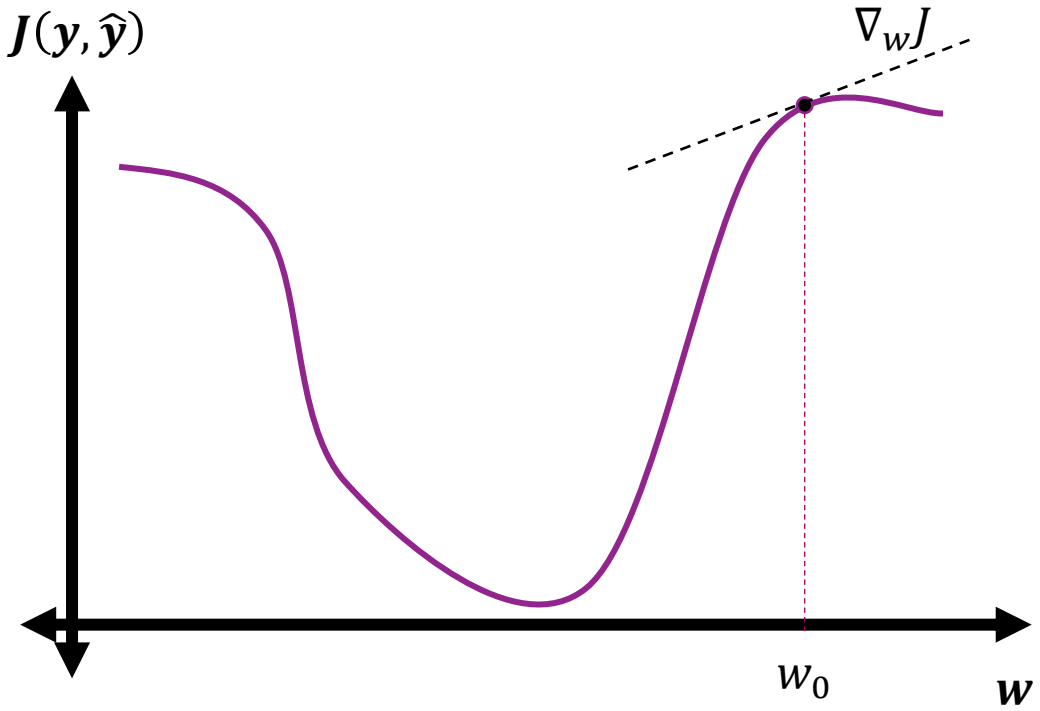


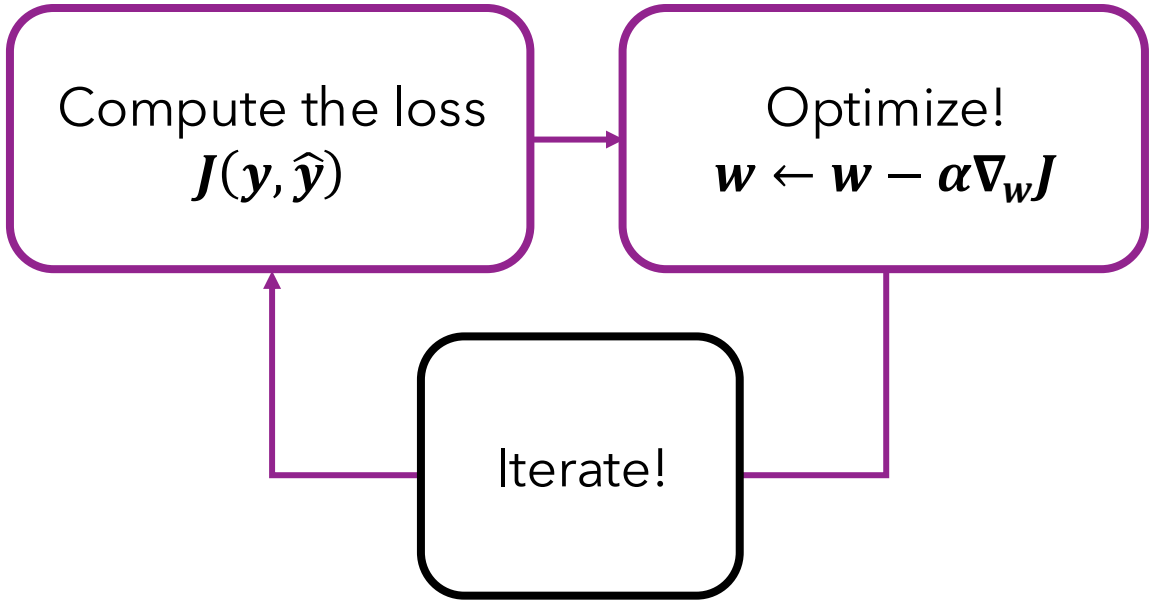
α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



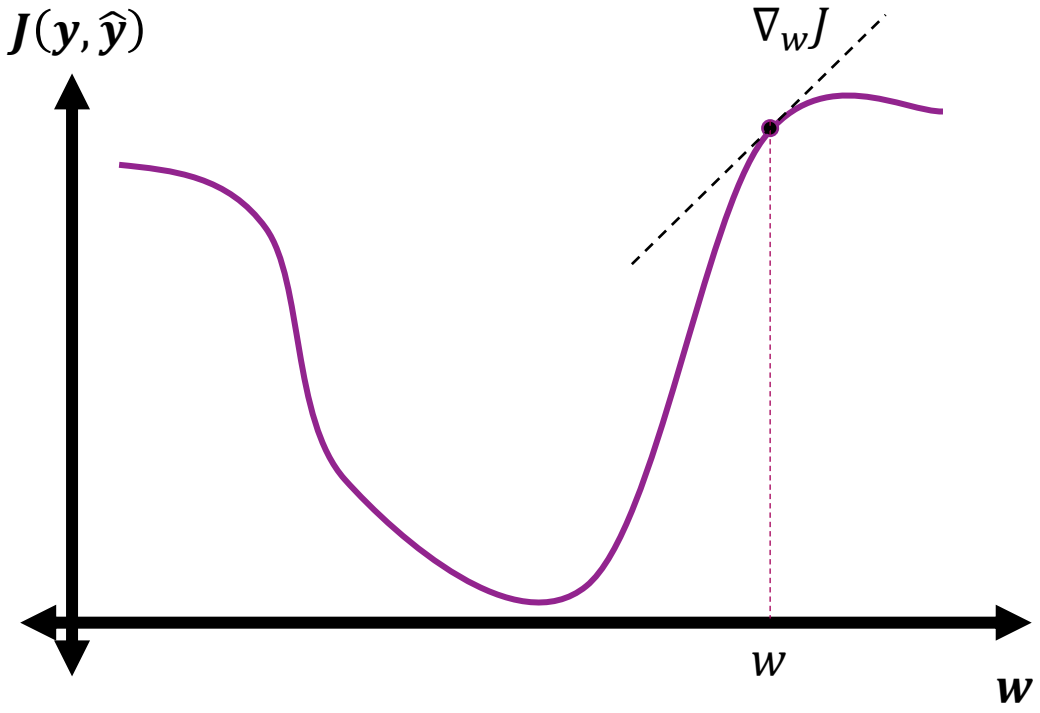


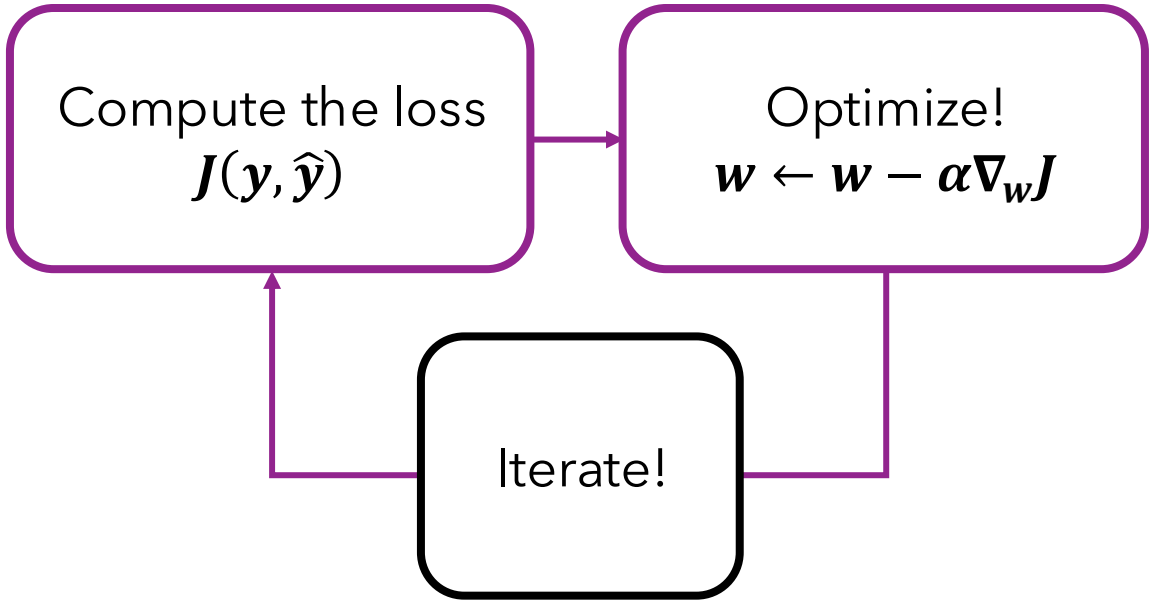
α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



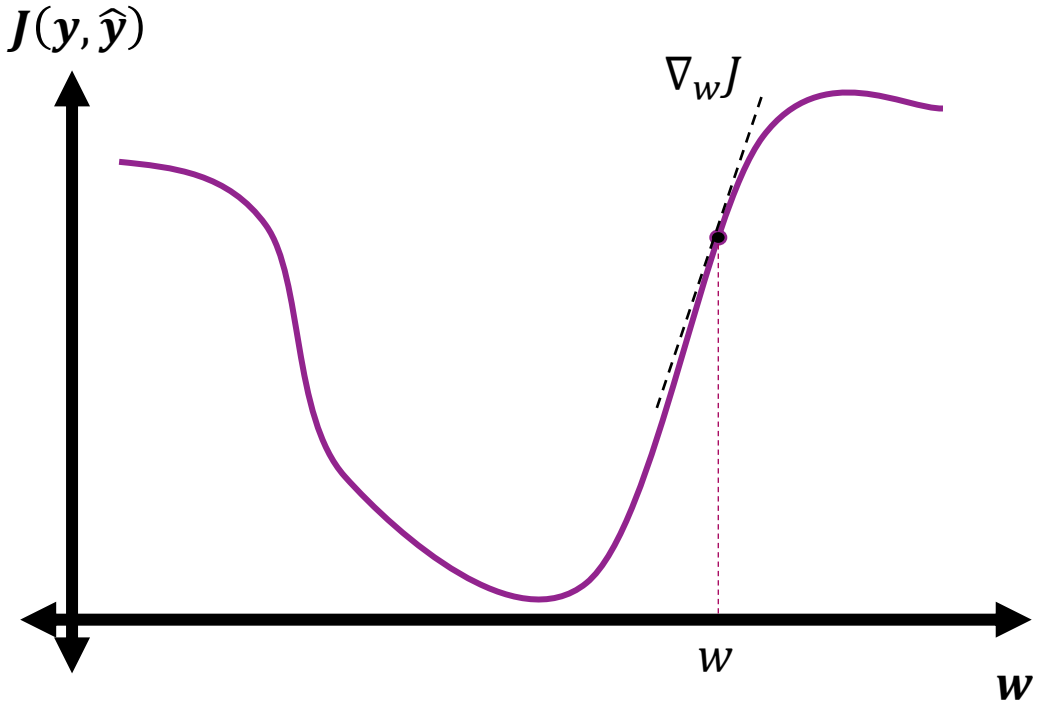


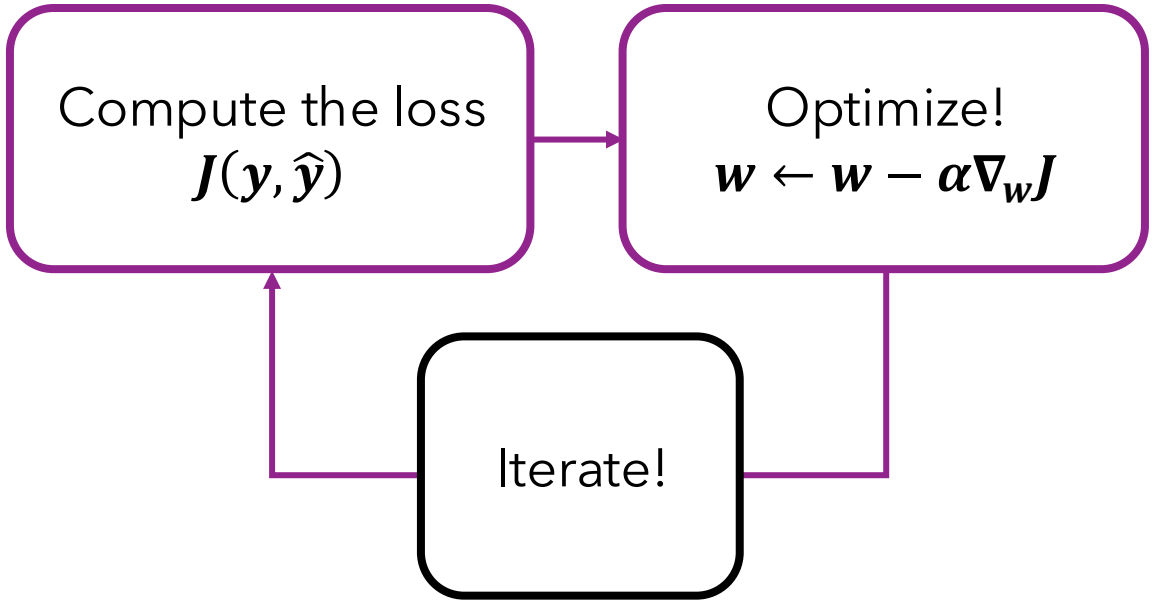
α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



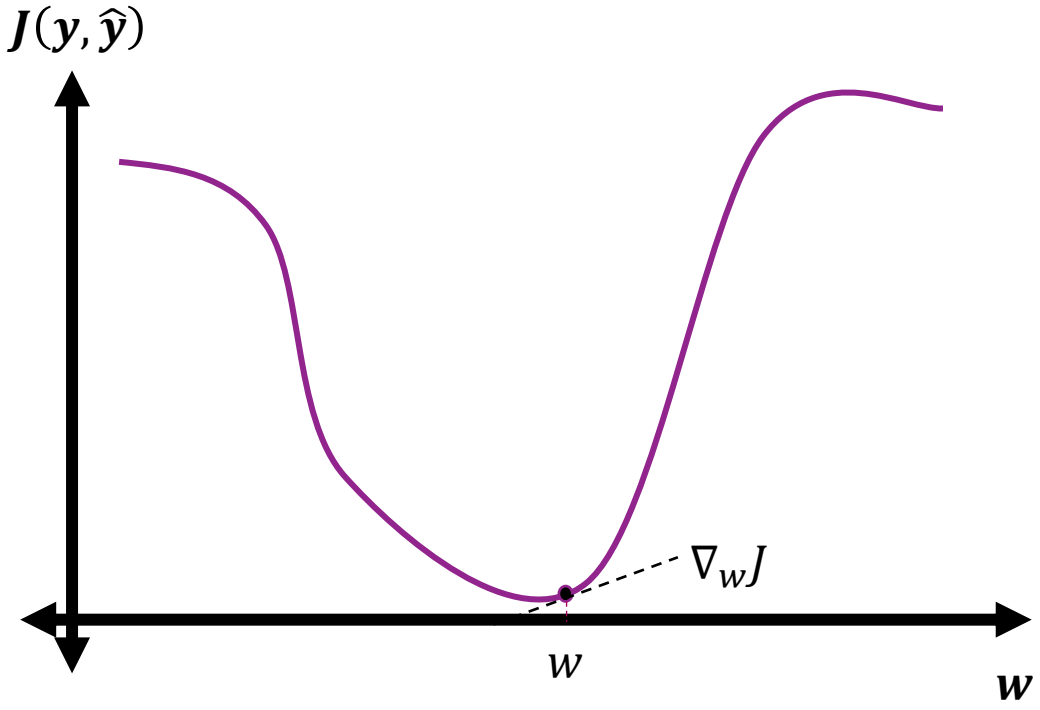


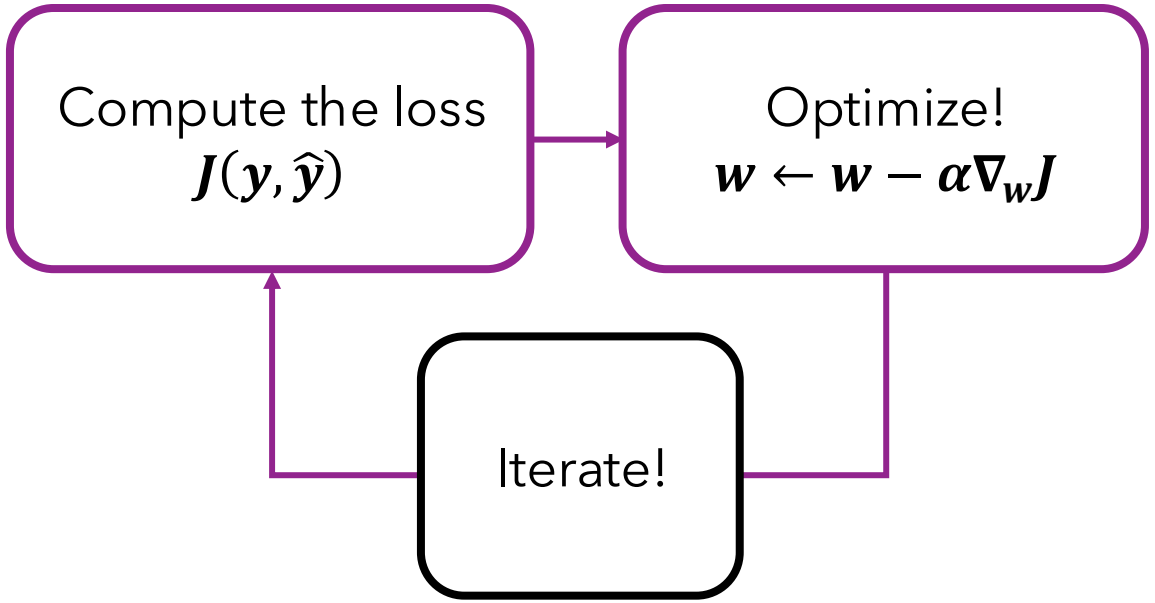
α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



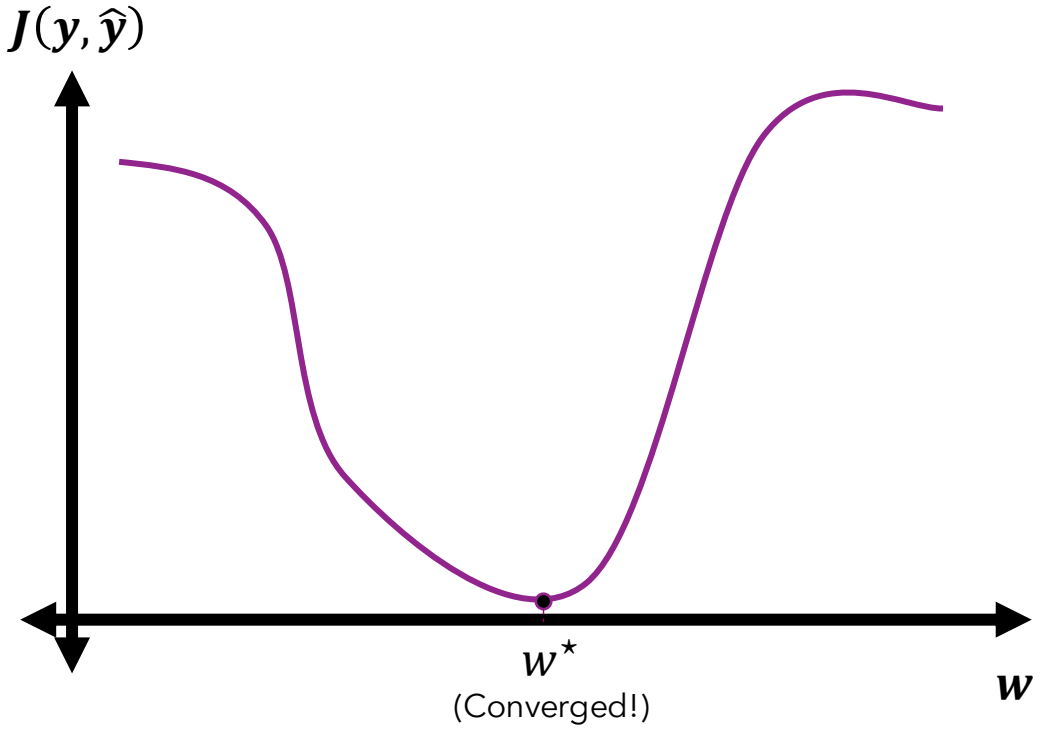


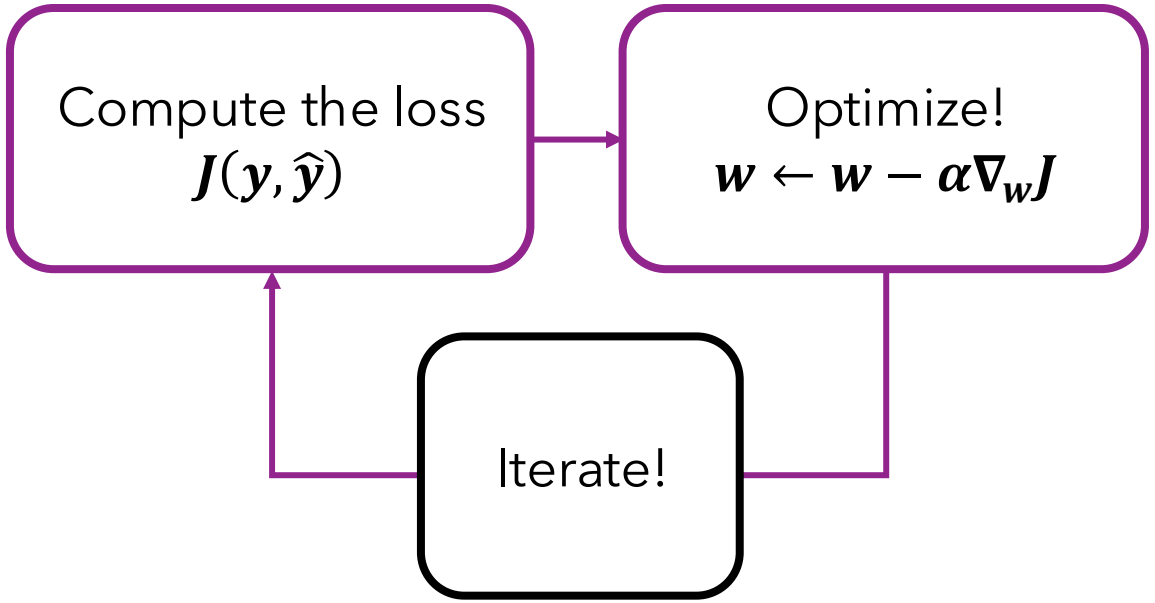
α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



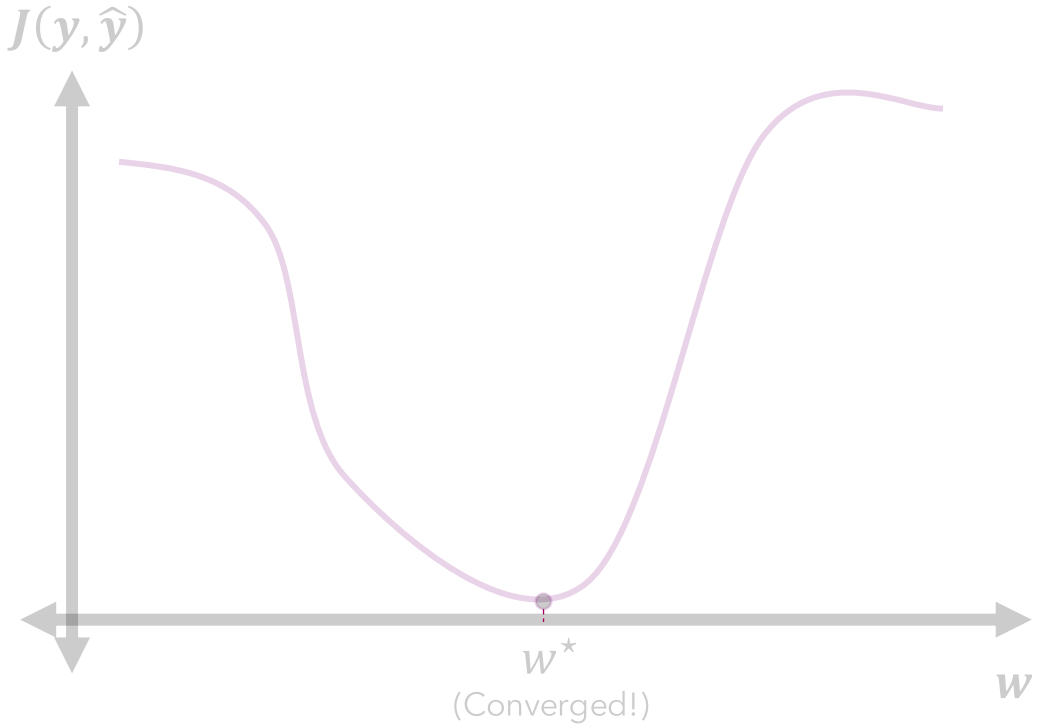


α: Learning Rate (Hyperparameter)

DEEP LEARNING

GRADIENT DESCENT: EXAMPLE

An iterative optimization technique to converge towards optimal parameters



Compute the loss
$J(y, \hat{y})$

Optimize!
$w \leftarrow w - \alpha \nabla_w J$

Iterate!

How do we compute this?

DEEP LEARNING

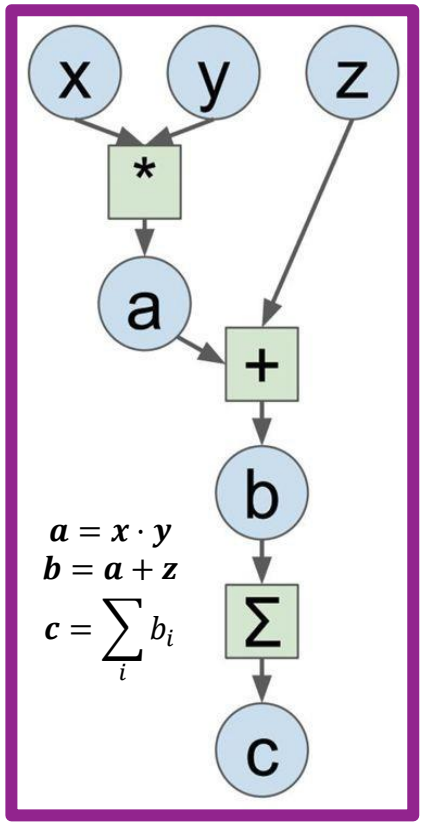
BACKPROPAGATION

Process to compute the gradients for optimizing deep learning models

Computed by traversing the constructed computational graph

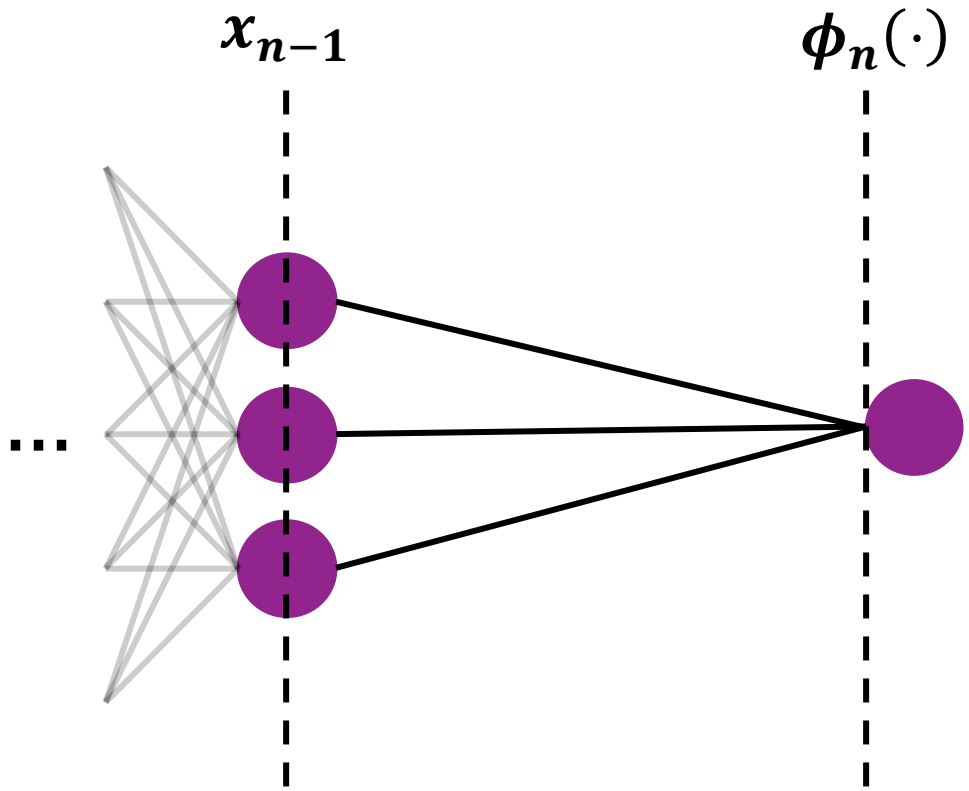
Uses chain rule

Example



DEEP LEARNING

BACKPROPAGATION



Predicted output

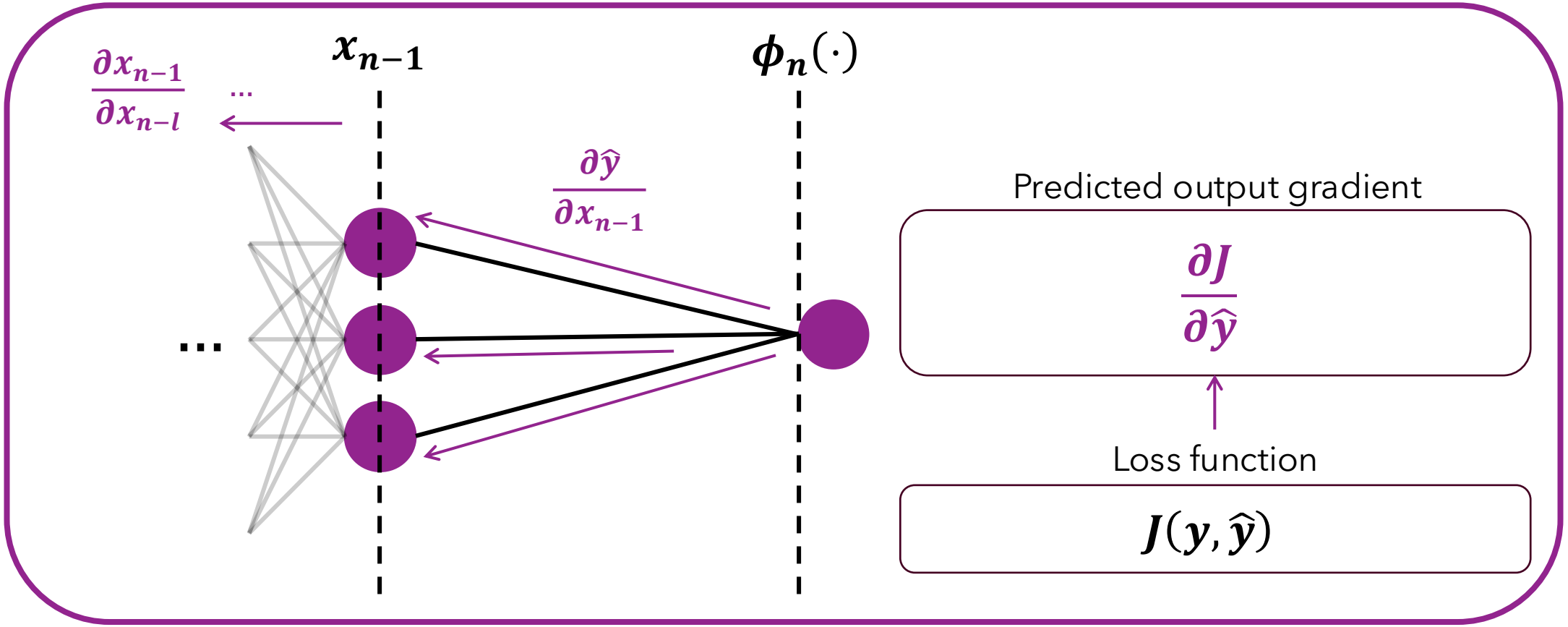
$$\hat{y} = \phi_n(W_{n-1}x_{n-1} + b_{n-1})$$

Loss function

$$J(y, \hat{y})$$

DEEP LEARNING

BACKPROPAGATION



Computed using the **chain rule**!

DEEP LEARNING

REFERENCES

Loss functions

https://www.ibm.com/think/topics/loss-function

Backprop

https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/

Optimization

Overview
https://www.ibm.com/think/topics/optimization-algorithms#18238369

Math behind DL optimization
https://www.d2l.ai/chapter_optimization/

Deep Learning

ML Specialization Course by Andrew Ng
https://www.youtube.com/playlist?list=PLkDaE6sCZn6FNC6YRfRQc_FbeQrF8BwGI

Neural Network Intro Course by Andrew Ng
https://www.youtube.com/playlist?list=PLkDaE6sCZn6Ec-XTbcX1uRg2_u4xOEky0

Questions?

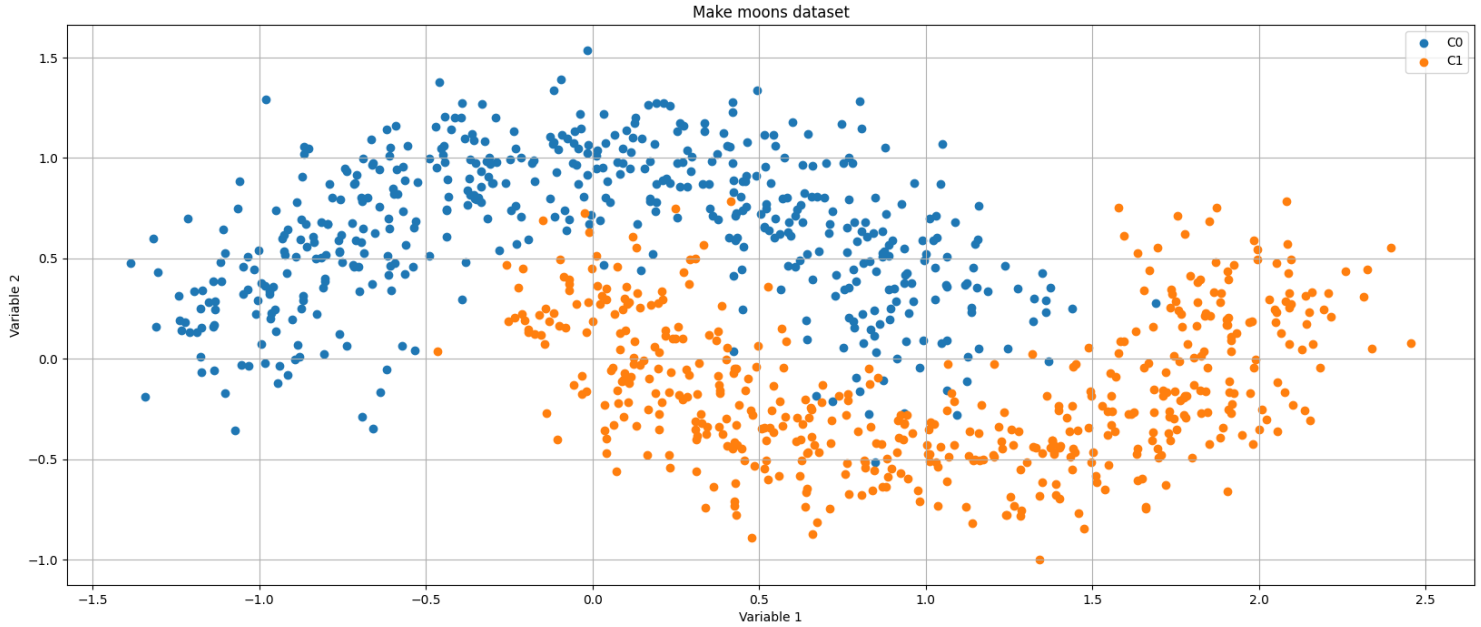
PYTORCH

WORKSHOP



PROBLEM STATEMENT

MAKE MOONS



We want the neural network to learn the *decision boundary* that **separates the two moon-shaped classes** from their (x, y) coordinates.

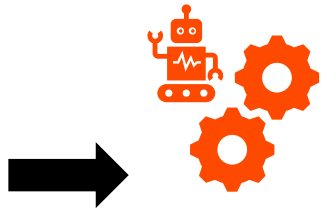
PROBLEM STATEMENT

WHY IS IT USEFUL?

Consider a financial problem:

"Bad" customer detection for credit

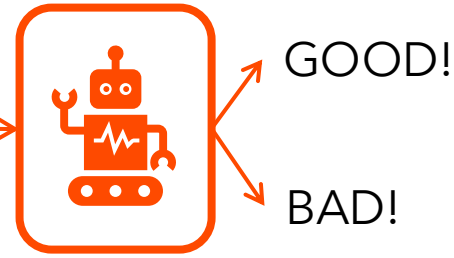
FICO (v_1)	Score (v_2)	...	Defaults (v_n)	Customer (y)
720	90	...	1	"good"
810	95	...	0	"good"
...	...	...	...	...
600	92	...	2	"bad"



Train the model

FICO (v_1)	Score (v_2)	...	Defaults (v_n)
770	92	...	1

New data



Model

GOOD!

BAD!

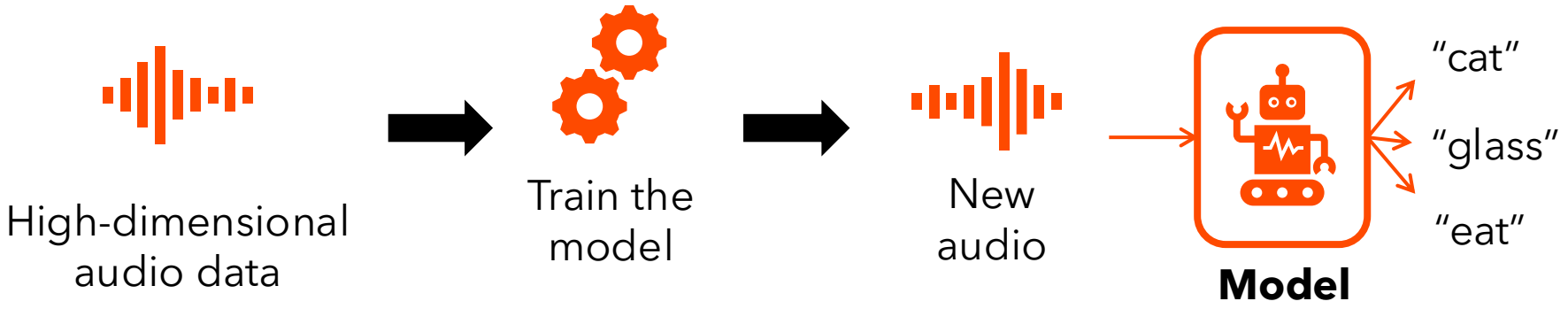
We want the neural network to learn the *decision boundary* that **separates the two types of customers** from their $(v_1, v_2, \ldots, v_n)$ coordinates.

PROBLEM STATEMENT

WHY IS IT USEFUL?

Consider a signal processing problem:

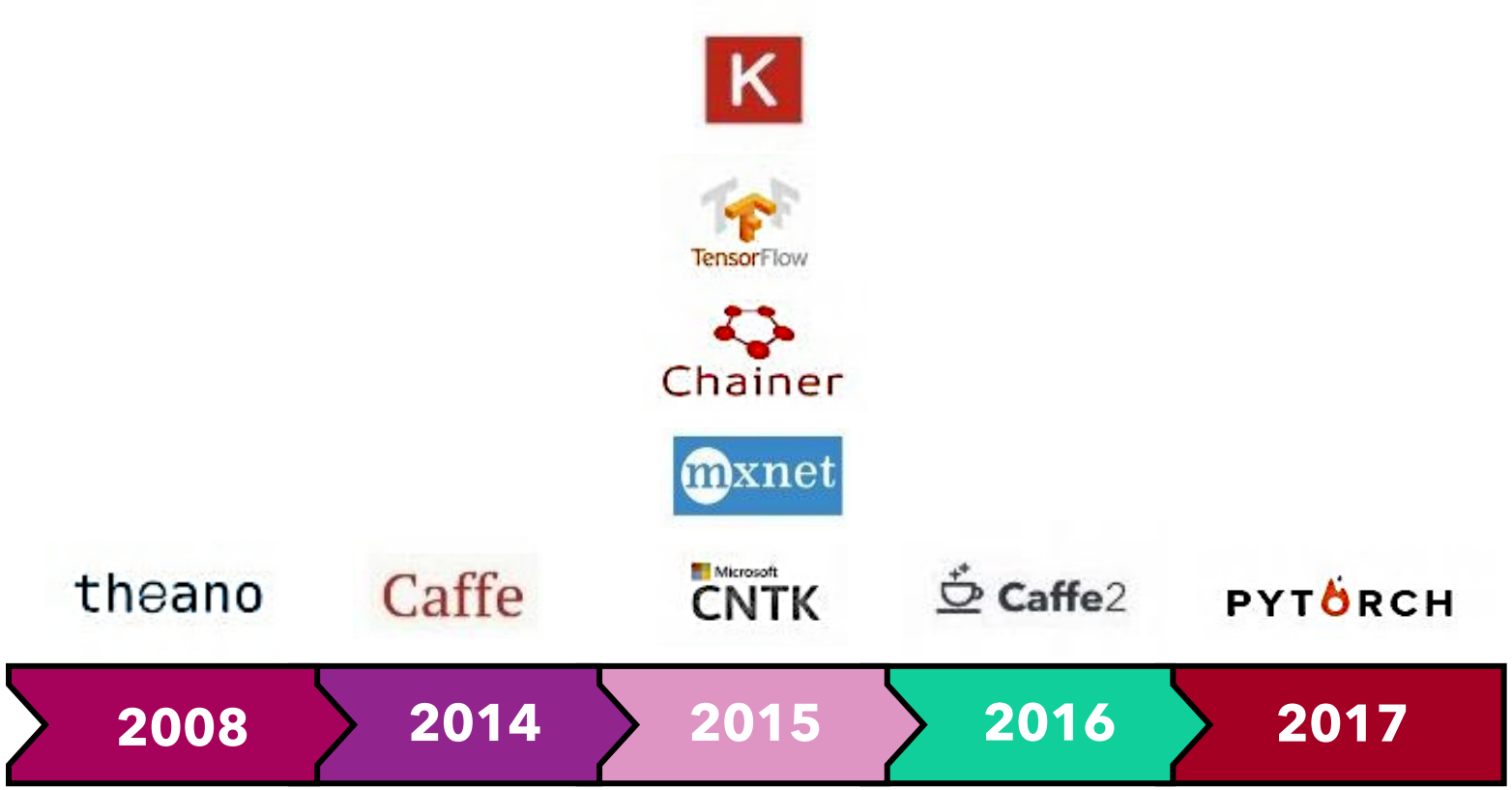
Word detection from spoken language



We want the neural network to learn the *decision boundary* that **separates the multiple word classes** from their $(\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n})$ coordinates.

DEEP LEARNING FRAMEWORKS

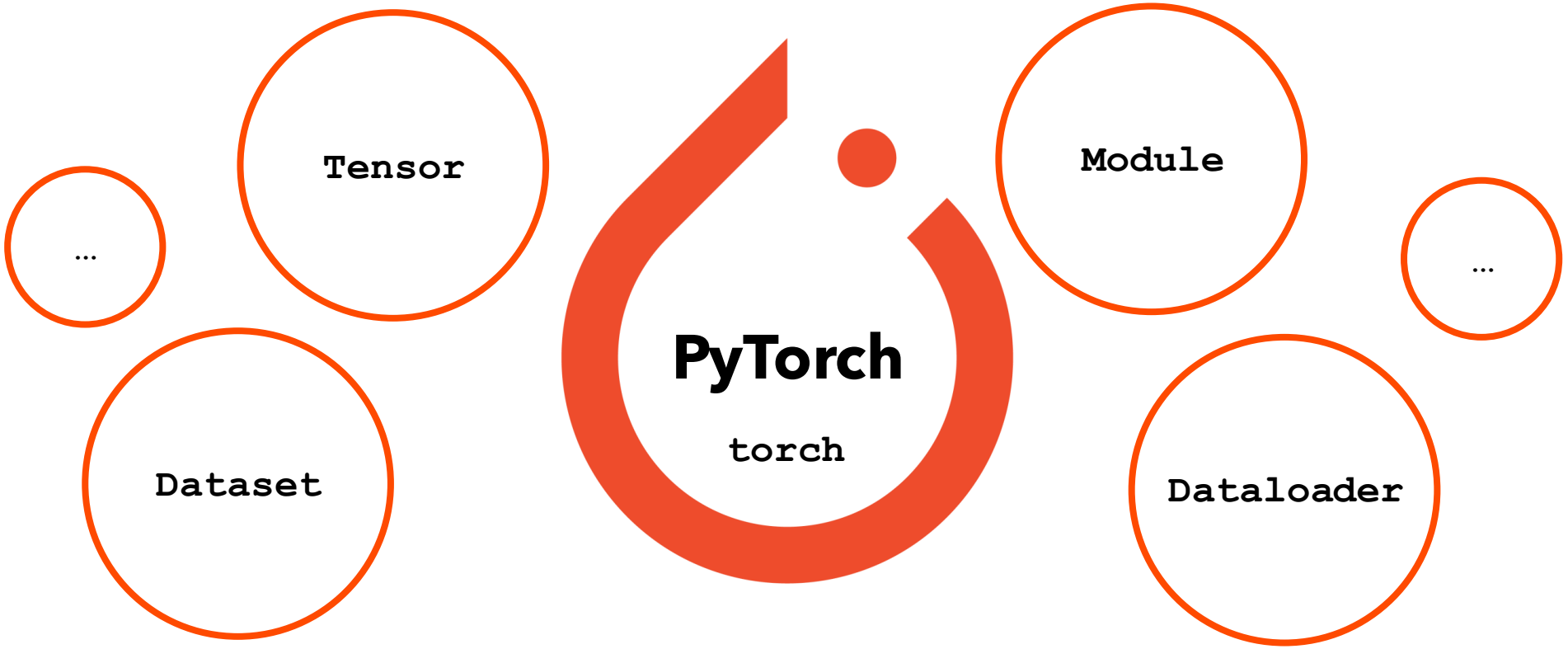
BRIEF HISTORY



PYTORCH

INTRODUCTION

Popular framework for **building** and **training** deep learning models



TENSORS

INTRODUCTION

A **multi-dimensional array** of a single type, while providing the option of **GPU acceleration** for high performance and **automatic differentiation**

`torch.tensor(...)`
Initializes a torch tensor from a Python list or a `numpy` array.

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad = True)
y = torch.randn(N, D, requires_grad = True)
z = torch.randn(N, D, requires_grad = True)
```

```
a = x * y
b = a + z
c = torch.sum(b)
```

```
c.backward()

print(x.grad)
print(y.grad)
print(z.grad)
```

TENSORS

INITIALIZATION

`randn(N, D)`
Creates a Gaussian random matrix.

Shape: $N \times D$
Mean = **0**, Variance = **1**

`requires_grad`
Specifies whether gradients need to be computed for this tensor

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad = True)
y = torch.randn(N, D, requires_grad = True)
z = torch.randn(N, D, requires_grad = True)
```

```
a = x * y
b = a + z
c = torch.sum(b)
```

```
c.backward()

print(x.grad)
print(y.grad)
print(z.grad)
```

TENSORS

FORWARD PASS

Each operation adds nodes to the computational graph

PyTorch functions such as **`torch.sum`** are vectorized

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad = True)
y = torch.randn(N, D, requires_grad = True)
z = torch.randn(N, D, requires_grad = True)
```

```
a = x * y
b = a + z
c = torch.sum(b)
```

```
c.backward()

print(x.grad)
print(y.grad)
print(z.grad)
```

TENSORS

BACK PROPAGATION

PyTorch allows automatic differentiation for back propagation!

`backward()`

1. Calculates the derivatives of **`c`** with respect to the inputs **`x`**, **`y`** and **`z`**.
2. Writes the gradient to the `grad` attribute of **`x`**, **`y`** and **`z`**.

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad = True)
y = torch.randn(N, D, requires_grad = True)
z = torch.randn(N, D, requires_grad = True)
```

```
a = x * y
b = a + z
c = torch.sum(b)
```

```
c.backward()

print(x.grad)
print(y.grad)
print(z.grad)
```

Try predicting the result of the print statements

TENSORS

BACK PROPAGATION

PyTorch allows automatic differentiation for back propagation!

`backward()`

1. Calculates the derivatives of **`c`** with respect to the inputs **`x`**, **`y`** and **`z`**.
2. Writes the gradient to the `grad` attribute of **`x`**, **`y`** and **`z`**.

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad = True)
y = torch.randn(N, D, requires_grad = True)
z = torch.randn(N, D, requires_grad = True)
```

```
a = x * y
b = a + z
c = torch.sum(b)
```

```
c.backward()

print(x.grad)     # y
print(y.grad)     # x
print(z.grad)     # torch.ones(N, D)
```

Try predicting the result of the print statements

MODULE

INTRODUCTION

Contains the building blocks for creating **any** neural network!

Base Class
`torch.nn.Module`

Initialization Method
`__init__()`
Initializes the weights and other attributes

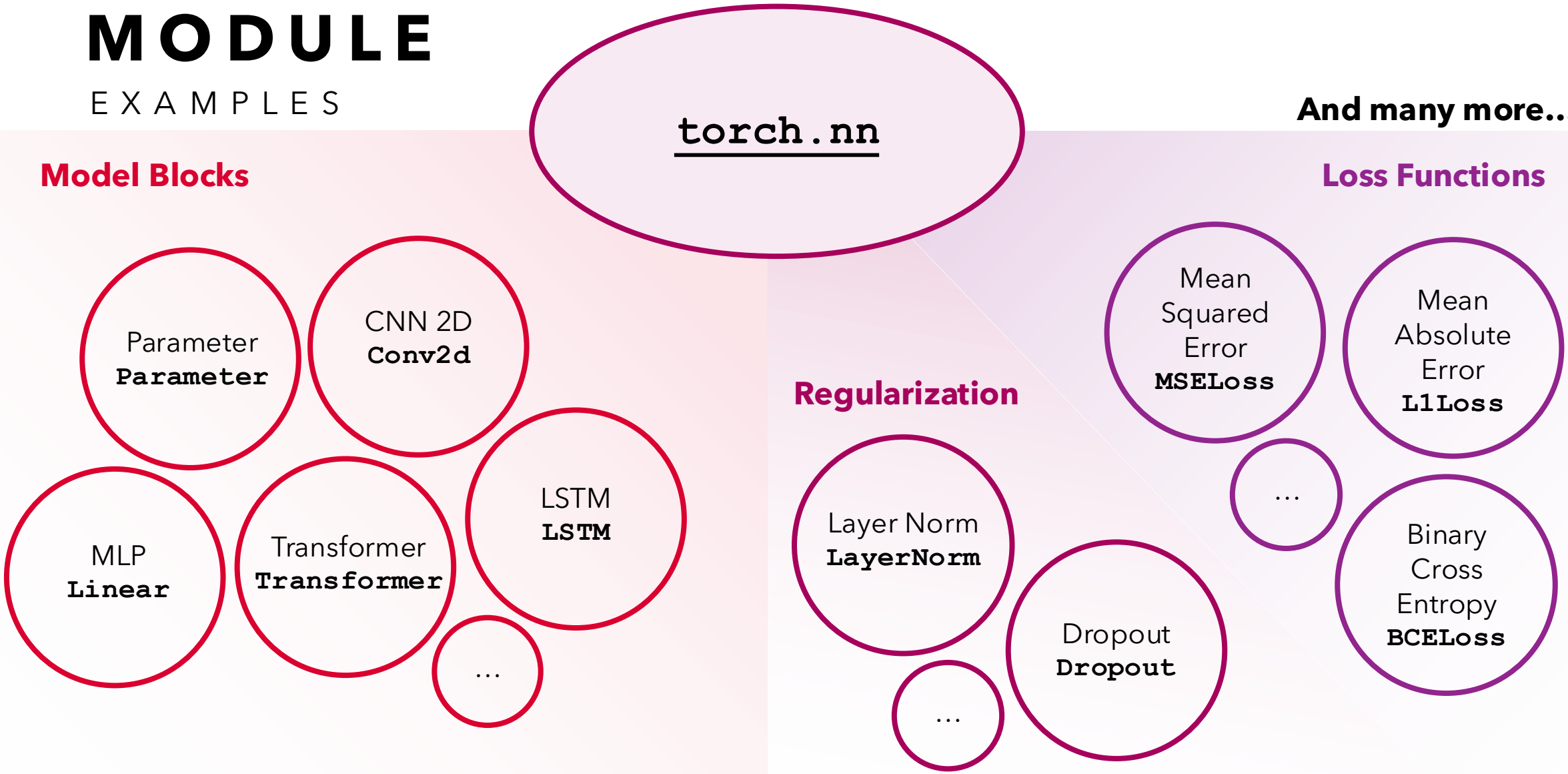
Forward Method
`forward()`
The computations required during forward pass

Backward method
`backward()`
Hidden; computes the gradients using backprop

Parameters method
`parameters()`
Hidden; returns the parameters in the model as a `dict`

MODULE

EXAMPLES



MODULE

DEFINING A NEURAL NETWORK

`__init__`

The constructor function

The layers and parameters are defined in this function

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
```

`forward`

The forward pass function

Here, we define the forward pass computations, based on the model architectures and layers defined in **`__init__`**.

```
def forward(self, x):
    return self.layers(x)
```

Creating the model

We create the model by creating an object of the class.

The forward pass is callable directly

```
model = NeuralNetwork()
out = model(x)
```

MODULE

DEFINING A NEURAL NETWORK

`__init__`

The constructor function

The layers and parameters are defined in this function

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
```

`forward`

The forward pass function

Here, we define the forward pass computations, based on the model architectures and layers defined in `__init__`.

```
def forward(self, x):
    return self.layers(x)
```

Creating the model

We create the model by creating an object of the class.

The forward pass is callable directly

```
model = NeuralNetwork()
out = model(x)
```

MODULE

DEFINING A NEURAL NETWORK

`__init__`

The constructor function

The layers and parameters are defined in this function

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
```

`forward`

The forward pass function

Here, we define the forward pass computations, based on the model architectures and layers defined in **`__init__`**.

```
def forward(self, x):
    return self.layers(x)
```

Creating the model

We create the model by creating an object of the class.

The forward pass is callable directly

```
model = NeuralNetwork()
out = model(x)
```

MODULE

DEFINING A NEURAL NETWORK

`__init__`

The constructor function

The layers and parameters are defined in this function

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
```

`forward`

The forward pass function

Here, we define the forward pass computations, based on the model architectures and layers defined in `__init__`.

```
def forward(self, x):
    return self.layers(x)
```

Creating the model

We create the model by creating an object of the class.

The forward pass is callable directly

```
model = NeuralNetwork()
out = model(x)
```

MODULE

TASK 1: DEFINE YOUR NEURAL NETWORK!

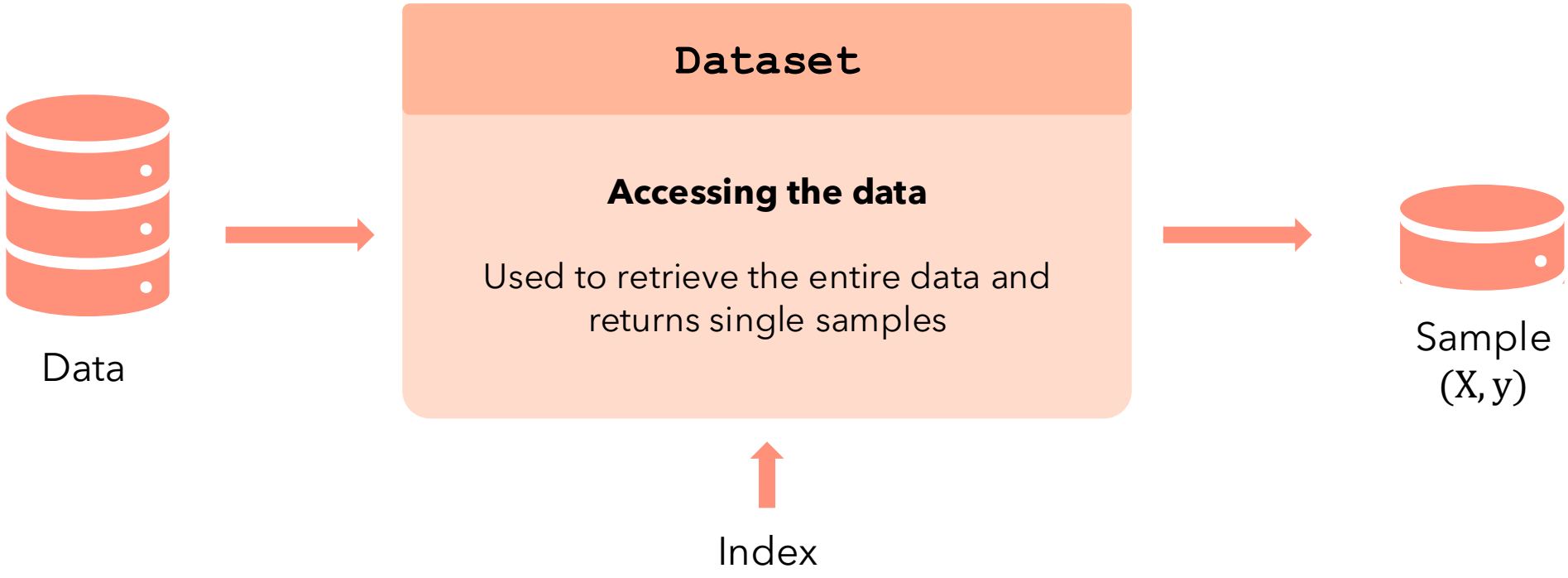


DEFINING THE MODEL

DATASET AND DATALOADER

INTRODUCTION

Deep Learning often needs **small batches** of data for iterative training

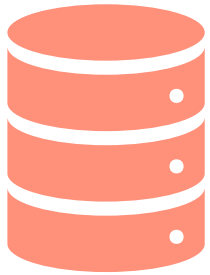


DATASET AND DATALOADER

INTRODUCTION

Deep Learning often needs **small batches** of data for iterative training

Cat-dog image classification



Data

Dataset

Accessing the data

Used to retrieve the entire data and returns single samples

Index



Sample



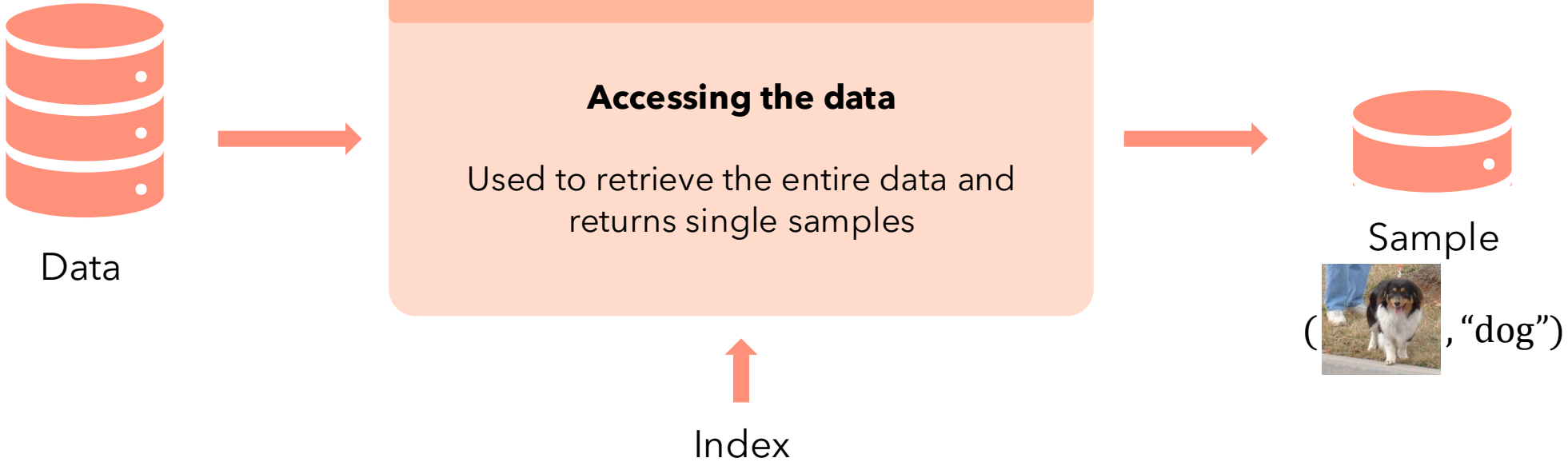
(, “cat”)

DATASET AND DATALOADER

INTRODUCTION

Deep Learning often needs **small batches** of data for iterative training

Cat-dog image classification



DATASET AND DATALOADER

INTRODUCTION

Deep Learning often needs **small batches** of data for iterative training

`Dataset`

Accessing the data

Used to retrieve the entire data and returns single samples

`DataLoader`

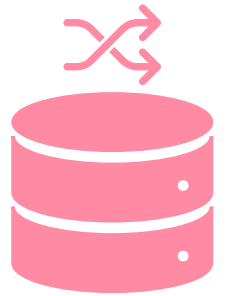
Batching/Loading data

Wraps an iterable around the **`Dataset`** object, and handles:

1. Batching
2. Shuffling
3. Parallel loading

Index

Shuffling is possible!



Batch

$([X_1, X_2, \dots, X_b], [y_1, y_2, \dots, y_b])$

DATASET AND DATALOADER

DEFINING A CUSTOM DATASET

Step 1

Define a class with a suitable name

Step 2

Must inherit from **`Dataset`**

Step 3

Define **three** functions

`__init__`

`__len__`

`__getitem__`

DATASET AND DATALOADER

DEFINING A CUSTOM DATASET

`__init__`

The constructor function

The data files or the data itself is initialized here

```
import torch
import os
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, data_dir = None, *args):
        self.data_dir = data_dir
        self.data_files = os.listdir(self.data_dir)
        ...
```

`__len__`

The length function

Here, we return the total length or size of the dataset (number of samples)

```
    def __len__(self):
        return len(self.data_files)
```

`__getitem__`

The iteration function

Used to access individual samples in the dataset. Inputs the *index*; Outputs the corresponding *data sample*

```
    def __getitem__(self, index):
        numpy_file = self.data_files[index]
        data = torch.from_numpy(np.load(numpy_file))
        return data
```

DATASET AND DATALOADER

DEFINING A CUSTOM DATASET

`__init__`

The constructor function

The data files or the data itself is initialized here

```python
import torch
import os
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, data_dir = None, *args):
        self.data_dir = data_dir
        self.data_files = os.listdir(self.data_dir)
        ...
```

`__len__`

The length function

Here, we return the total length or size of the dataset (number of samples)

```python
def __len__(self):
    return len(self.data_files)
```

`__getitem__`

The iteration function

Used to access individual samples in the dataset. Inputs the *index*; Outputs the corresponding *data sample*

```python
def __getitem__(self, index):
    numpy_file = self.data_files[index]
    data = torch.from_numpy(np.load(numpy_file))
    return data
```

DATASET AND DATALOADER

DEFINING A CUSTOM DATASET

`__init__`

The constructor function

The data files or the data itself is initialized here

```
import torch
import os
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, data_dir = None, *args):
        self.data_dir = data_dir
        self.data_files = os.listdir(self.data_dir)
        ...
```

`__len__`

The length function

Here, we return the total length or size of the dataset (number of samples)

```
def __len__(self):
    return len(self.data_files)
```

`__getitem__`

The iteration function

Used to access individual samples in the dataset. Inputs the *index*; Outputs the corresponding *data sample*

```
def __getitem__(self, index):
    numpy_file = self.data_files[index]
    data = torch.from_numpy(np.load(numpy_file))
    return data
```

DATASET AND DATALOADER

DEFINING A CUSTOM DATASET

`__init__`

The constructor function

The data files or the data itself is initialized here

```
import torch
import os
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, data_dir = None, *args):
        self.data_dir = data_dir
        self.data_files = os.listdir(self.data_dir)
        ...
```

`__len__`

The length function

Here, we return the total length or size of the dataset (number of samples)

```
def __len__(self):
    return len(self.data_files)
```

`__getitem__`

The iteration function

Used to access individual samples in the dataset. Inputs the *index*; Outputs the corresponding *data sample*

```
def __getitem__(self, index):
    numpy_file = self.data_files[index]
    data = torch.from_numpy(np.load(numpy_file))
    return data
```

DATASET AND DATALOADER

DEFINING THE DATALOADER

```
from torch.utils.data import DataLoader

train_data = CustomDataset(
    data_dir="data_directory”
)

train_dataloader = DataLoader(
    train_data,
    batch_size=32,
    shuffle=True
)
```

Import the DataLoader class

Create the dataset object

DATASET AND DATALOADER

DEFINING THE DATALOADER

```
from torch.utils.data import DataLoader

train_data = CustomDataset(
    data_dir="data_directory"
)

train_dataloader = DataLoader(
    train_data,
    batch_size=32,
    shuffle=True
)
```

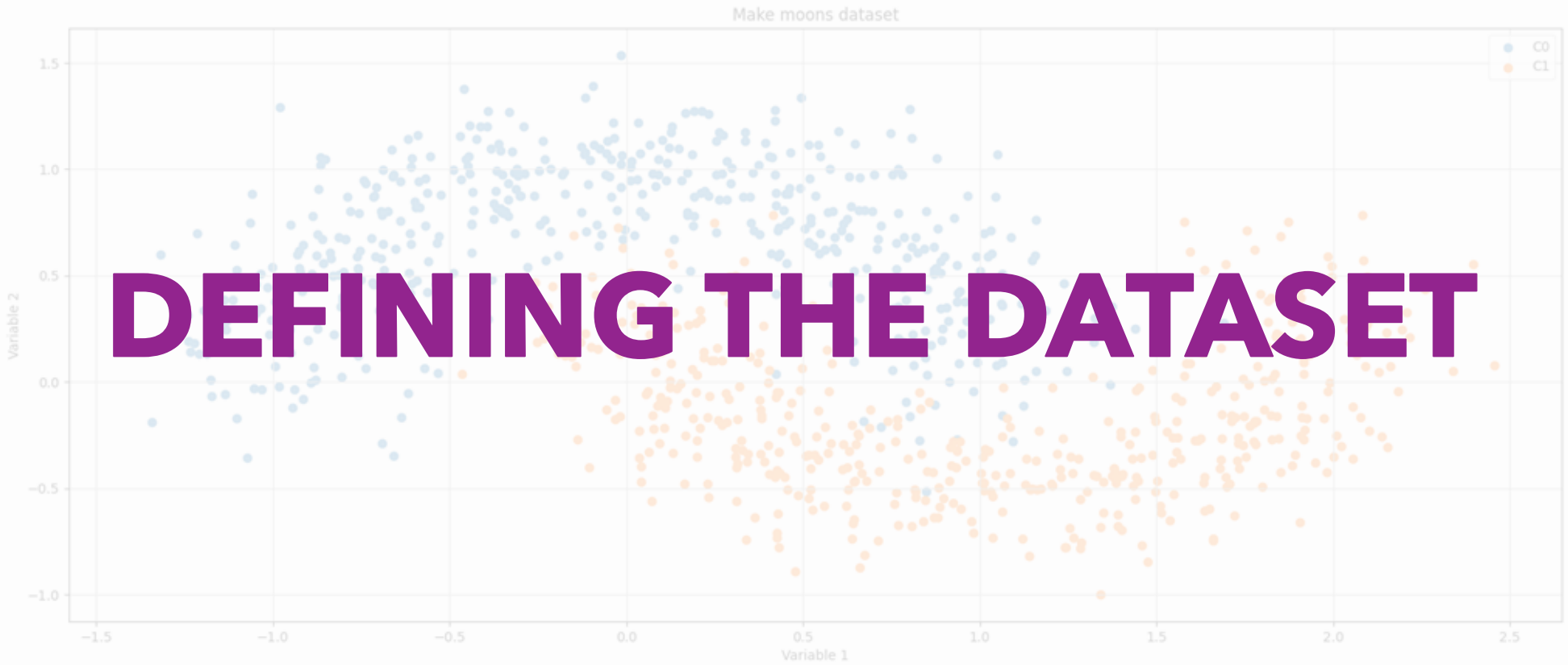
Create the **DataLoader**

1. **`train_data`**: The Dataset object
2. **`batch_size`**: The batch size for batching the data samples
3. **`shuffle`**: If `true`, then it shuffles the mini-batches for each iteration for stochasticity.

PyTorch Tutorial

DATASET AND DATALOADER

TASK 2: DEFINE THE MAKE MOONS DATASET



DEFINING THE DATASET

TRAINING

LOSS, OPTIMIZER AND SCHEDULER

Loss functions[1]
(torch.nn)

Regression
nn.MSELoss, …

Binary Classification
nn.BCELoss, …

Multi-class Classification
nn.CrossEntropyLoss, …

Possible to create custom loss functions!

Optimizer[2]
(torch.optim)

Stochastic Gradient Descent
optim.SGD

RMSProp
optim.RMSprop

Adam (Recommended)
optim.Adam

Possible to create custom optimizers!

Scheduler[3]
(torch.optim.lr_scheduler)

Scheduling learning rates is key to convergence!

Example schedulers –
StepLR, LinearLR, ExponentialLR, CosineAnnealingLR, …

[1,2,3] Click to redirect to PyTorch website for more details

TRAINING

LOSS, OPTIMIZER AND SCHEDULER

Loss functions[1]
(torch.nn)

Regression
nn.MSELoss, …

Binary Classification
nn.BCELoss, …

Multi-class Classification
nn.CrossEntropyLoss, …

Possible to create custom loss functions!

Optimizer[2]
(torch.optim)

Stochastic Gradient Descent
optim.SGD

RMSProp
optim.RMSprop

Adam (Recommended)
optim.Adam

Possible to create custom optimizers!

Scheduler[3]
(torch.optim.lr_scheduler)

Scheduling learning rates is key to convergence!

Example schedulers –
StepLR, LinearLR, ExponentialLR, CosineAnnealingLR, …

[1,2,3] Click to redirect to PyTorch website for more details

TRAINING

LOSS, OPTIMIZER AND SCHEDULER

Loss functions[1]
(torch.nn)

Regression
nn.MSELoss, …

Binary Classification
nn.BCELoss, …

Multi-class Classification
nn.CrossEntropyLoss, …

Possible to create custom loss functions!

Optimizer[2]
(torch.optim)

Stochastic Gradient Descent
optim.SGD

RMSProp
optim.RMSprop

Adam (Recommended)
optim.Adam

Possible to create custom optimizers!

Scheduler[3]
(torch.optim.lr_scheduler)

Scheduling learning rates is key to convergence!

Example schedulers –
StepLR, LinearLR, ExponentialLR, CosineAnnealingLR, …

[1,2,3] Click to redirect to PyTorch website for more details

TRAINING

LOSS, OPTIMIZER AND SCHEDULER

Loss functions[1]
(torch.nn)

Regression
nn.MSELoss, …

Binary Classification
nn.BCELoss, …

Multi-class Classification
nn.CrossEntropyLoss, …

Possible to create custom loss functions!

Optimizer[2]
(torch.optim)

Stochastic Gradient Descent
optim.SGD

RMSProp
optim.RMSprop

Adam (Recommended)
optim.Adam

Possible to create custom optimizers!

Scheduler[3]
(torch.optim.lr_scheduler)

Scheduling learning rates is key to convergence!

Example schedulers –
StepLR, LinearLR, ExponentialLR, CosineAnnealingLR, …

[1,2,3] Click to redirect to PyTorch website for more details

TRAINING

LOSS, OPTIMIZER AND SCHEDULER EXAMPLE

Loss functions[1]
(torch.nn)

```
model = CustomMLP()
criterion = nn.MSELoss()
optimizer = optim.Adam(
    model.parameters(),
    lr=0.001
)
scheduler = ExponentialLR(
    optimizer,
    gamma=0.99
)
```

Optimizer[2]
(torch.optim)

```
model = CustomMLP()
criterion = nn.MSELoss()
optimizer = optim.Adam(
    model.parameters(),
    lr=0.001
)
scheduler = ExponentialLR(
    optimizer,
    gamma=0.99
)
```

Scheduler[3]
(torch.optim.lr_scheduler)

```
model = CustomMLP()
criterion = nn.MSELoss()
optimizer = optim.Adam(
    model.parameters(),
    lr=0.001
)
scheduler = ExponentialLR(
    optimizer,
    gamma=0.99
)
```

[1,2,3] Click to redirect to PyTorch website for more details

TRAINING

PIPELINE

Defining the train function for one epoch!

1. **`dataloader`**: The dataloader object for iterating through the dataset in batches
2. **`model`**: The neural network model object inheriting from **`nn.Module`**
3. **`loss_fn`**: The loss function
4. **`optimizer`**: The optimizer object
5. **`scheduler`**: The LR scheduler object
6. **`device:`** The device: CPU or GPU

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()      # Or outside this loop
```

TRAINING

PIPELINE

Setting the model to **train** mode

Required to activate certain training specific layers

E.g. **`Dropout`**, **`BatchNorm2d`** etc.

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()      # Or outside this loop
```

TRAINING

PIPELINE

`for batch, (X, y_true) in enumerate(dataloader):`
Iterating through the dataloader to get *mini-batches* of data

`<tensor>.to(device)`
Shift the tensors to the set device (CPU or GPU)

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()      # Or outside this loop
```

TRAINING

PIPELINE

`model(X)`
Equivalent to calling `model.forward(X)`.

Constructs the computational graph for backprop

`loss_fn(y_true, y_pred)`
Computes the loss using the defined loss function

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()      # Or outside this loop
```

TRAINING

PIPELINE

`optimizer.zero_grad()`
Set the gradients to zero before backprop

`loss.backward()`
Calculates the gradients by traversing the constructed computational graph

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()      # Or outside this loop
```

TRAINING

PIPELINE

`optimizer.step()`
Updates the parameter values

`scheduler.step()`
Updates the learning rate

```
def train_epoch(
    dataloader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    device
):
    model.train()

    for batch, (X, y_true) in enumerate(dataloader):
        X = X.to(device)
        y_true = y_true.to(device)

        y_pred = model(X)
        loss = loss_fn(y_true, y_pred)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
        scheduler.step()       # Or outside this loop
```

TRAINING

SAVING AND LOADING

Trained models should be saved for future use, and loaded when needed

Saving

Saving the model (`.pt`)

```
torch.save(<model>, <save_loc>)
```

Saving the model weights (`.pth`)

```
torch.save(<model>.state_dict(),
<save_loc>)
```

Loading

Loading the model (`.pt`)

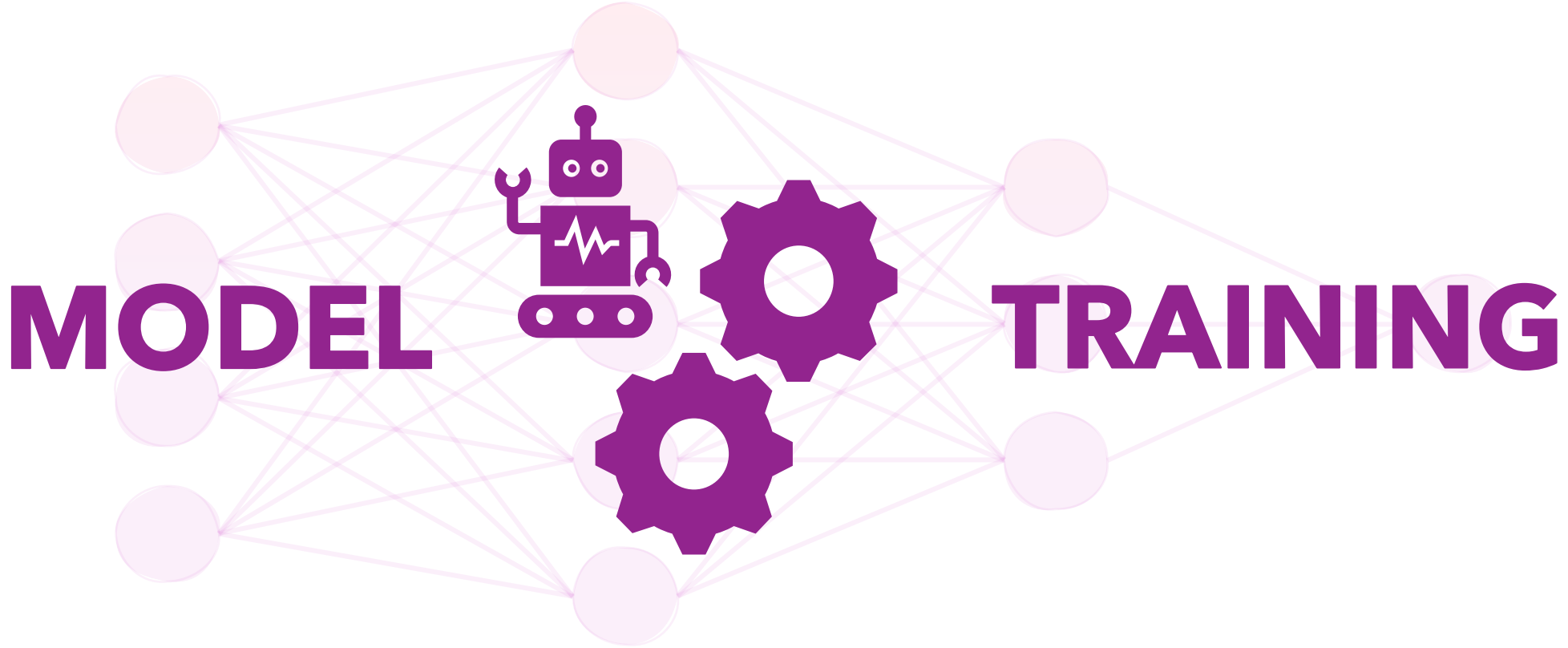
```
model = torch.load(<save_loc>)
```

Loading the model weights (`.pth`)

```
model = NeuralNetwork()
model.load_state_dict(<save_loc>)
```

TRAINING

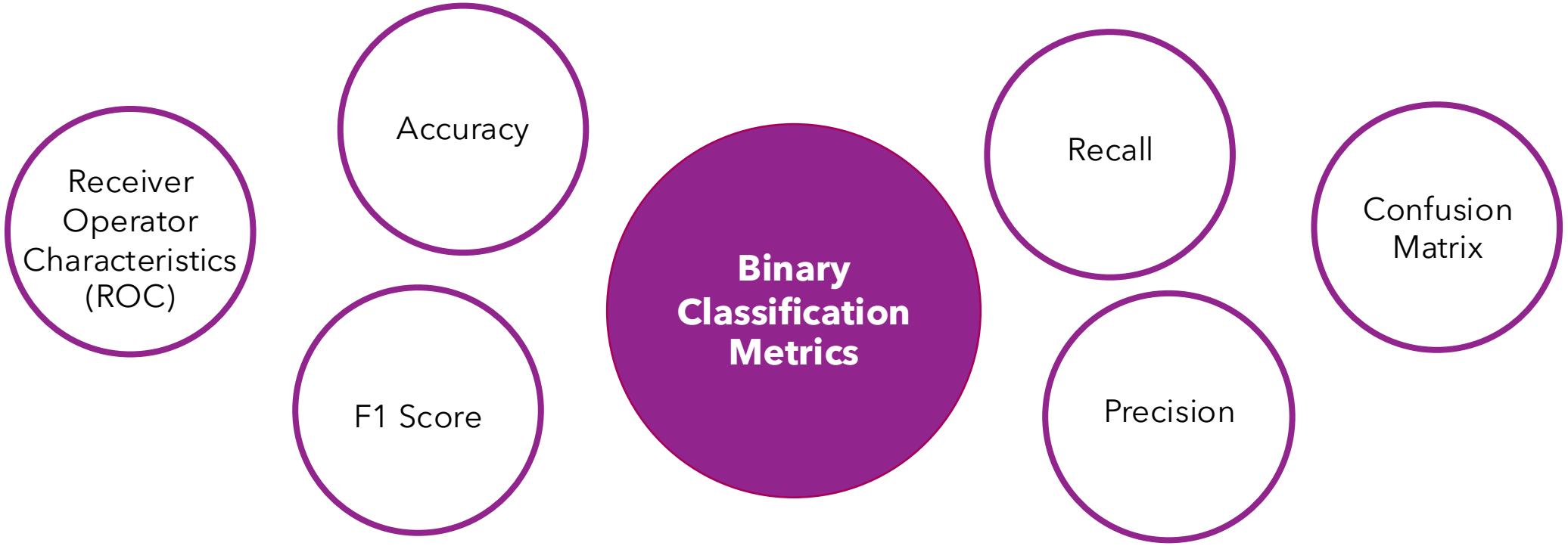
TASK 3: TRAIN YOUR MODEL!



EVALUATION

METRICS

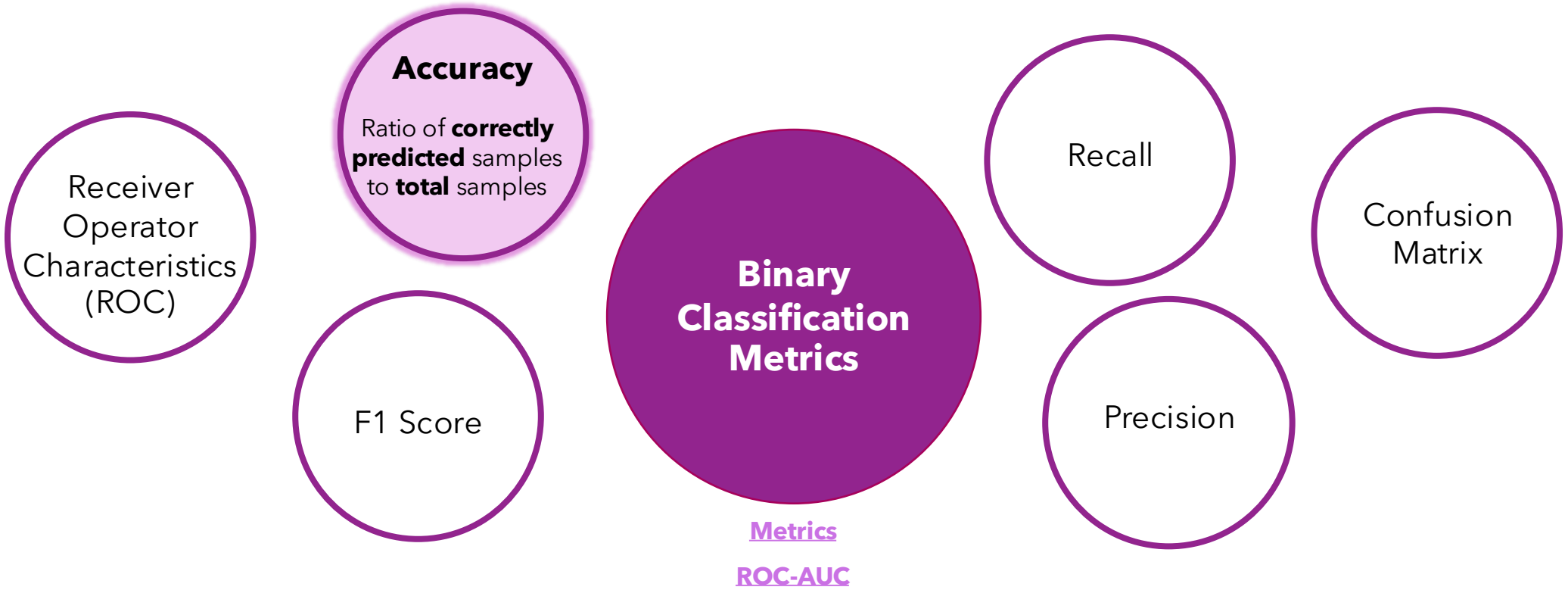
Trained models need to evaluated by measure performance on an unseen dataset



EVALUATION

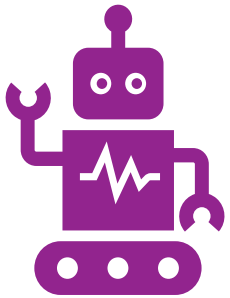
METRICS

Trained models need to evaluated by measure performance on an unseen dataset



EVALUATION

TASK 4: EVALUATE YOUR MODEL!

MODEL  EVALUATION

CONGRATULATIONS!

You have a working model!

WRAPUP

Machine Learning

- What is ML?
- Why ML?
- Algorithms
- Limitations

Deep Learning

- Neural Networks
- The Math
- Gradient Descent
- Backpropagation

PyTorch Workshop

- Tensors
- Module
- Dataset
- DataLoader
- Training
- Evaluation

THANK YOU!