



# LAB 2 (Part 1): PYTORCH INTRODUCTION

University of Washington, Seattle

Spring 2025



# OUTLINE

## Part 1: Python as Deep Learning Platform

- Deep Learning Libraries in Python
- Installing PyTorch on Your Computer

## Part 2: Neural Net Workflow In PyTorch

- Example task: Simple Linear Regression

## Part 3: Python Concepts for PyTorch

- Python Classes
- PyTorch Tensors

## Supplementary: Additional Platforms

- Google Colab
- Google Cloud



# PYTHON AS DEEP LEARNING PLATFORM

Deep Learning Libraries in Python  
Installing PyTorch on Your Computer



# Deep Learning Libraries in Python

# Deep Learning Libraries for Python



Developed by Facebook (Meta)

- Provides modules easy to combine
- Easy to edit network
- Many pre-trained models
- Seamless integration into Python/Numpy framework



Developed by Google

- Provides Tensorboard for visualization
- Supports multiple languages (C++, Java, R)
- Slightly less intuitive to use than PyTorch
- Great community support
- Tensorflow Lite can run models on mobile devices



Developed by Apache

- Supported by Amazon Web Service
- Supports many languages
- Fast and flexible for running DL algorithms
- Features advanced GPU support
- Popular among industrial projects

# Deep Learning Libraries for Python



Developed by Facebook (Meta)

- Provides modules easy to combine
- Easy to edit network
- Many pre-trained models
- Seamless integration into Python/Numpy framework

Framework for this class



TensorFlow

Developed by Google

- Provides Tensorboard for visualization
- Supports multiple languages (C++, Java, R)
- Slightly less intuitive to use than PyTorch
- Great community support
- Tensorflow Lite can run models on mobile devices



Developed by Apache

- Supported by Amazon Web Service
- Supports many languages
- Fast and flexible for running DL algorithms
- Features advanced GPU support
- Popular among industrial projects



# Installing PyTorch on Your Computer

# Installing PyTorch on Your Computer

Windows - Anaconda Prompt

Mac and Linux - Terminal

If your computer has one of [these GPUs](#) (CUDA-Enabled GeForce and TITAN Products):

```
Anaconda Prompt (anaconda3)
(base) C:\Users\Jimin>conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

else (uses CPU instead):

```
Anaconda Prompt (anaconda3)
(base) C:\Users\Jimin>conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

Note: Make sure you have the latest graphics driver installed

Note: For non-local options, see slides 43-45





# Verify PyTorch Installation

1.

```
1 import torch
```

Import PyTorch

```
1 x = torch.rand(5, 3)
2 print(x)
```

Generate randomly initialized [torch tensor](#)

```
tensor([[0.9278, 0.0797, 0.3936],
        [0.5075, 0.4611, 0.5649],
        [0.3202, 0.1863, 0.5423],
        [0.3864, 0.6935, 0.5528],
        [0.5596, 0.8721, 0.6153]])
```

2.

```
1 torch.cuda.is_available()
```

Check if your GPU driver, CUDA is enabled and accessible by PyTorch

True



# NEURAL NETWORK WORKFLOW IN PYTORCH

Example Task: Simple Linear Regression



# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

Identify Tracked Values

Train Model

Visualization and Evaluation



# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

Identify Tracked Values

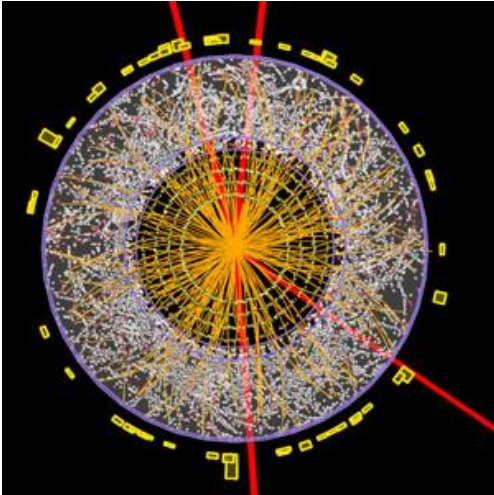
Train Model

Visualization and Evaluation



# Prepare Data

## Raw data



Example 1)

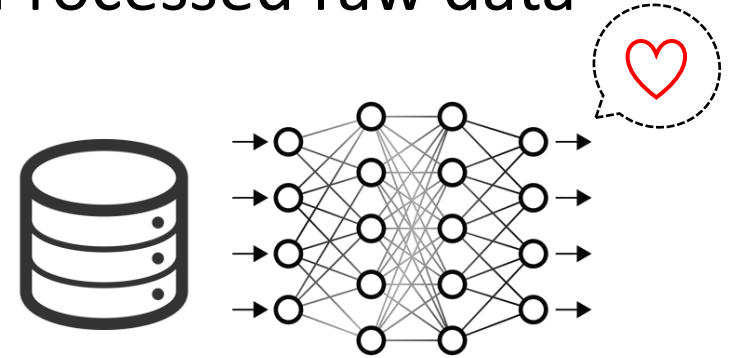
Particle feature data from  
ATLAS detector @ LHC



Example 2)

Neural recordings from  
the brain

## Processed raw data



- Remove outliers
- Normalization
- Split train, validation, test sets
- etc

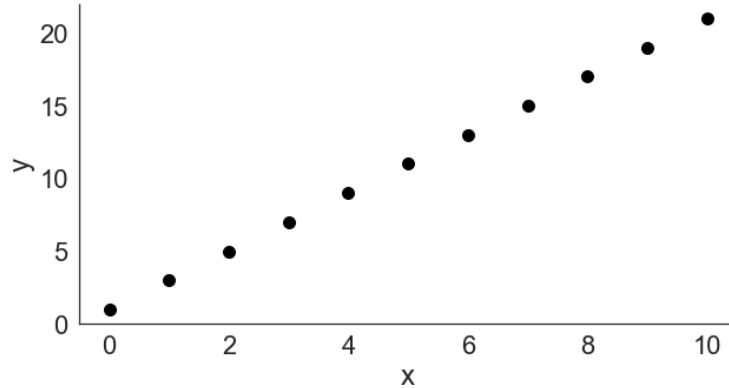


For a successful neural net model, dataset should be **Large**, **Clean** and **Diverse**  
-Andrej Karpathy (Director of Tesla Autopilot AI)



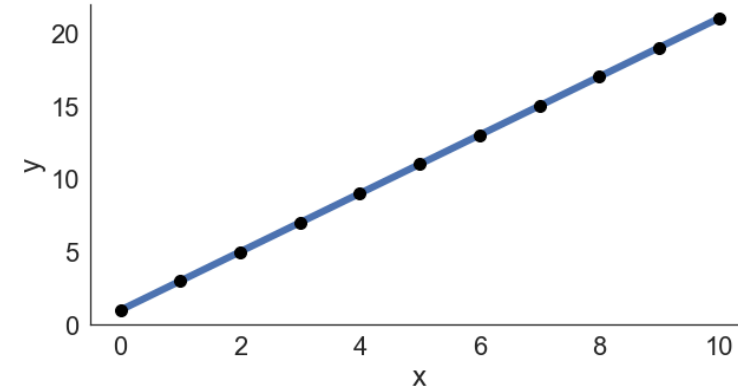
# Example Task: Linear Regression

Dataset

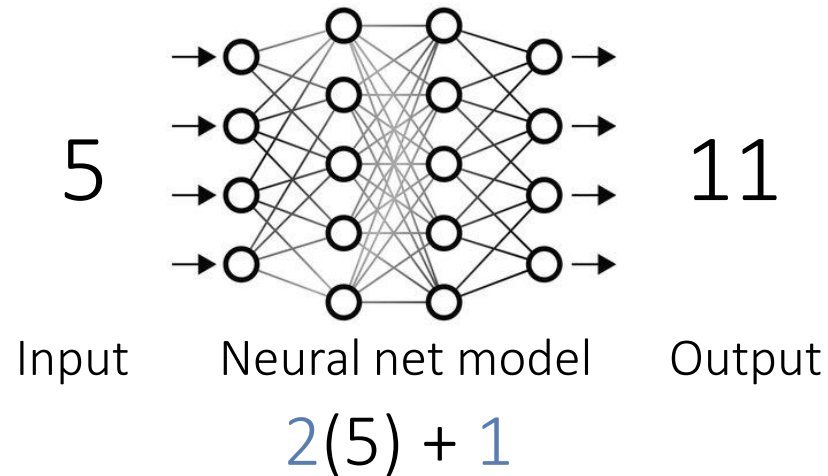


$x$  = input  
 $y$  = output

Rule to be learned



$$y = 2x + 1$$





# Prepare Data (example task)

```
1 %matplotlib inline
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import torch
```

Import necessary libraries

```
1 x_train = np.arange(11, dtype = np.float32)
2 x_train = x_train[:, np.newaxis]
3
4 y_train = (2 * x_train) + 1
```

Generate training data for x and y

```
1 print(x_train)
```

```
[[ 0.]
 [ 1.]
 [ 2.]
 [ 3.]
 [ 4.]
 [ 5.]
 [ 6.]
 [ 7.]
 [ 8.]
 [ 9.]
[10.]]
```

$x$

Inputs (features)

```
1 print(y_train)
```

```
[[ 1.]
 [ 3.]
 [ 5.]
 [ 7.]
 [ 9.]
[11.]
[13.]
[15.]
[17.]
[19.]
[21.]]
```

$y$

Output targets

Print the training data (x\_train, y\_train)



# Neural Net Workflow Steps

Prepare Data

Define Model

Select Hyperparameters

Identify Tracked Values

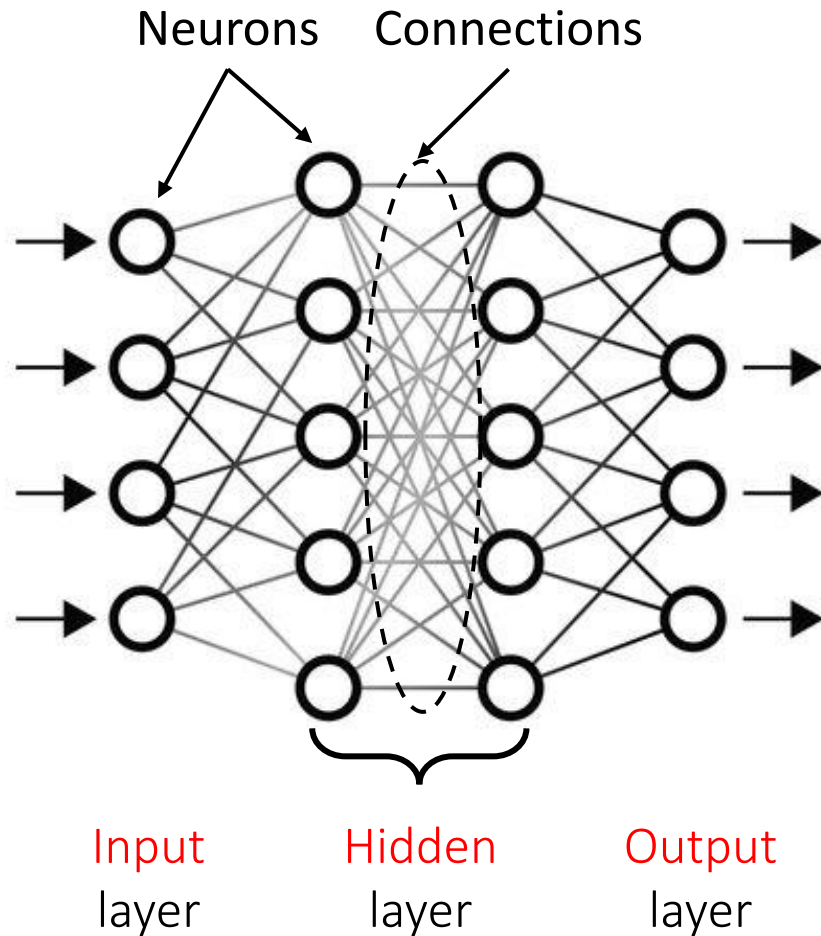
Train Model

Visualization and Evaluation





# Define Model



Dimensions (number of neurons) of **Input** and **output** layers

Number of **hidden** layers

Number of neurons for each **hidden** layer

Other network features



# Define Model (example task)

```
1 class linearRegression(torch.nn.Module):
2
3     def __init__(self, input_dim, output_dim):
4
5         super(linearRegression, self).__init__()
6
7         self.linear = torch.nn.Linear(input_dim, output_dim)
8
9     def forward(self, x):
10
11         out = self.linear(x)
12
13         return out
```

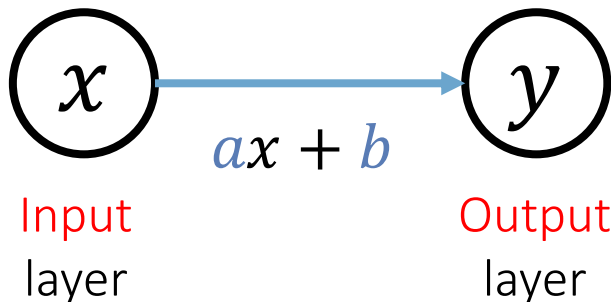
Define a model [class](#)

Initialize the model with a linear layer  
with input/output dimension

Define a feed forward function describing  
the information flow within the network

Neural Network Diagram

Training goal



$a \cong 2$  (weight)  
 $b \cong 1$  (bias)

Input layer dimension = 1

Output layer dimension = 1

No hidden layers



# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

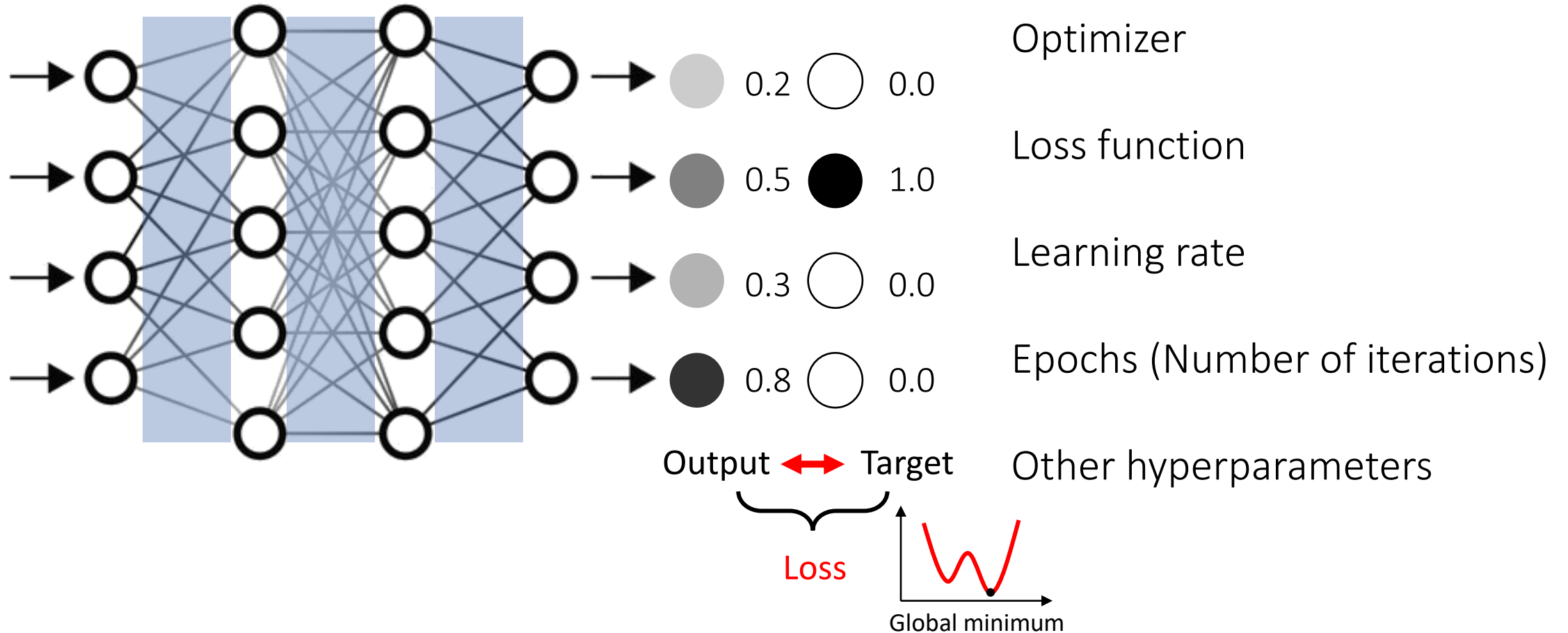
Identify Tracked Values

Train Model

Visualization and Evaluation



# Select Hyperparameters



Optimizer *minimizes* the **loss** throughout **epochs** by changing the **connection weights/biases** at the pace of **learning rate**

# Select Hyperparameters (example task)

```
1 model = linearRegression(input_dim = 1, output_dim = 1)
2
3 learning_rate = 0.01
4 epochs = 100
5
6 loss_func = torch.nn.MSELoss()
7 optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
8
9 if torch.cuda.is_available():
10     model.cuda()
```

Define the model with input/output layer dimensions

Define learning rate, epochs (# of iterations)

Define loss function (MSE) and optimizer (Gradient Descent)

If using GPU, transfer the model to GPU memory



# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

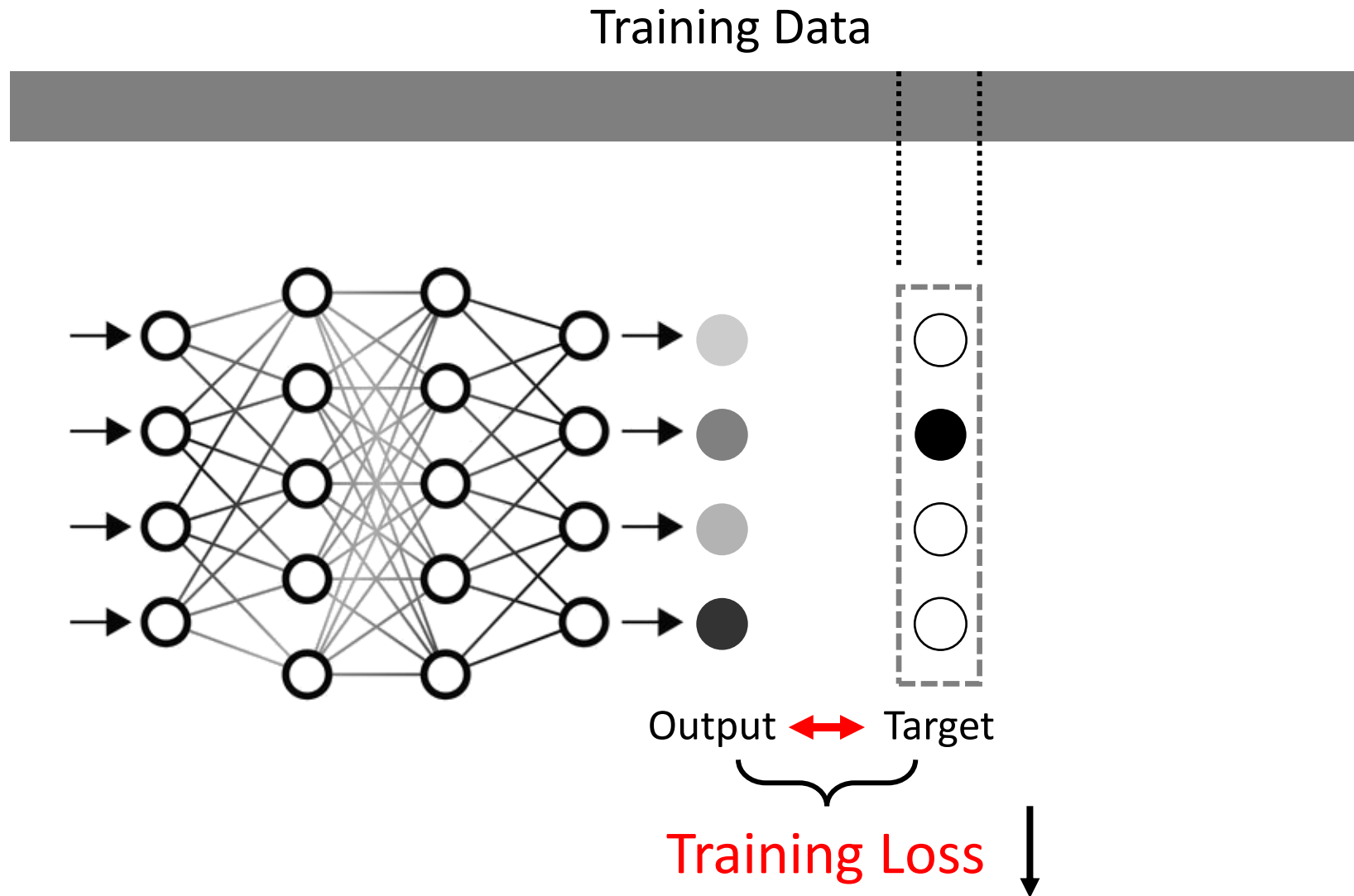
Identify Tracked Values

Train Model

Visualization and Evaluation



# Identify Tracked Values



# Identify Tracked Values (example task)

```
1 train_loss_list = []
```

Create an **empty list** or **arrays** to contain training loss





# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

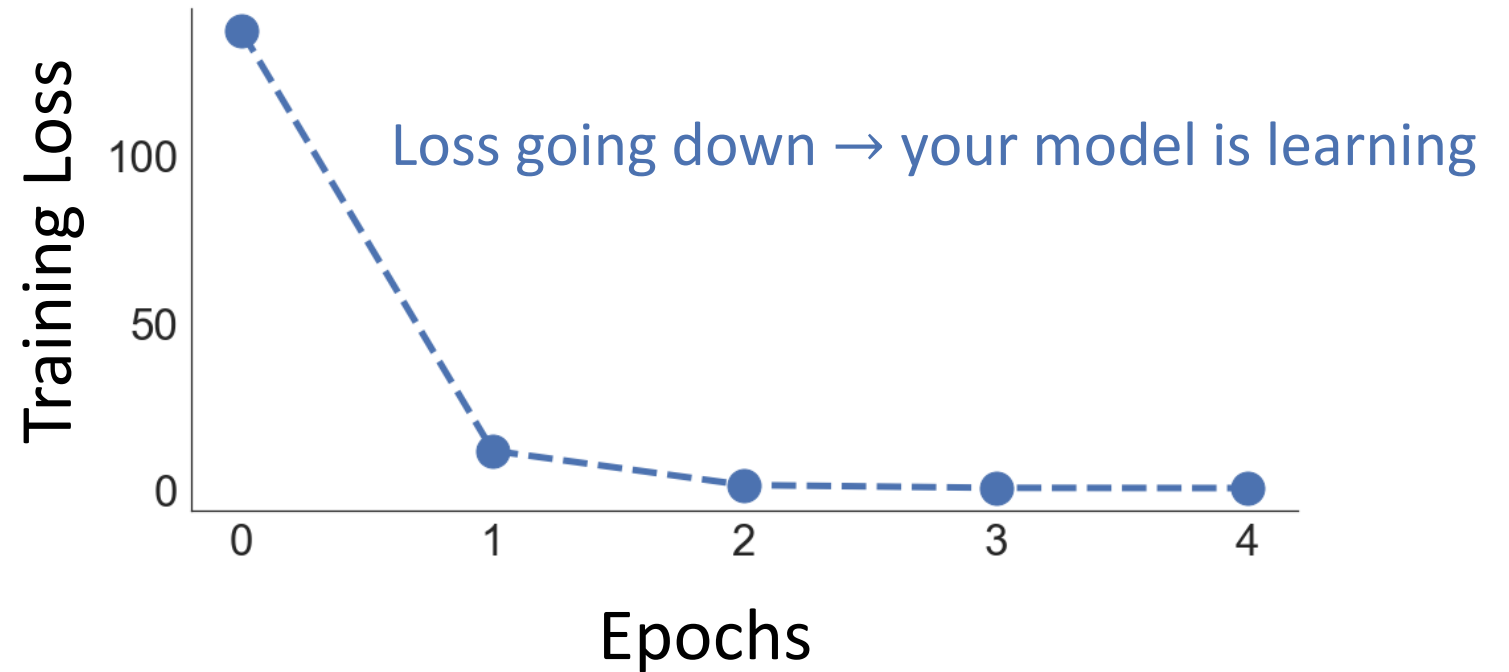
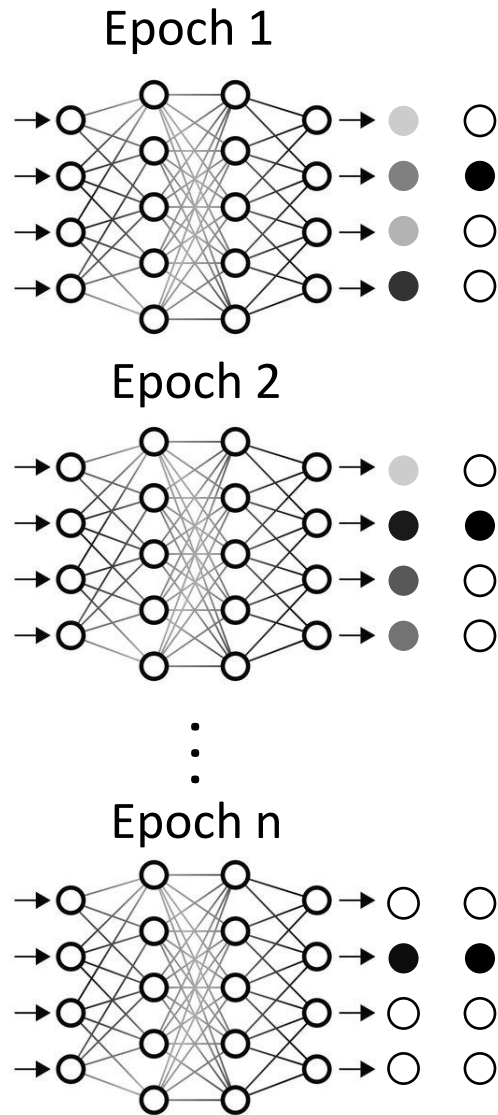
Identify Tracked Values

**Train Model**

Visualization and Evaluation



# Train the Model



1 Epoch : Model goes through a full training dataset



# Train the Model (example task)

```
1 if torch.cuda.is_available():
2     inputs = torch.from_numpy(x_train).cuda()
3     targets = torch.from_numpy(y_train).cuda()
4 else:
5     inputs = torch.from_numpy(x_train)
6     targets = torch.from_numpy(y_train)
7
8 for epoch in range(epochs):
9
10     optimizer.zero_grad()
11
12     outputs = model(inputs)
13
14     loss = loss_func(outputs, targets)
15
16     train_loss_list.append(loss.item())
17
18     loss.backward()
19
20     optimizer.step()
21
22     print('epoch {}, loss {}'.format(epoch, loss.item()))
```

Convert inputs and targets into [PyTorch tensors](#)  
(GPU)

(CPU)

This ensures learning from each epoch is separate

Forward pass the inputs through the network to produce outputs

Compute the loss and append to tracking list

Compute how much changes to be made to weights/biases

Update the weights/biases

Print the epoch # and loss value



# Neural Net Workflow

Prepare Data

Define Model

Select Hyperparameters

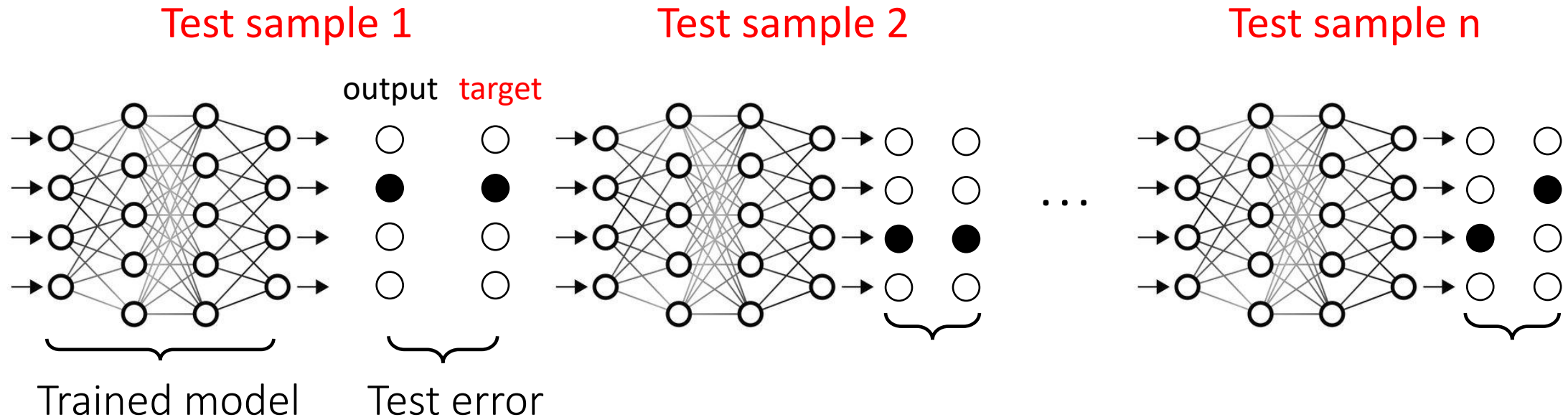
Identify Tracked Values

Train Model

Visualization and Evaluation



# Visualization and Evaluation



Commonly used evaluation metrics:

Mean squared error (MSE), Classification accuracy, etc

Typically, your trained model is tested on an **unknown** dataset outside of training data  
(More on this in Lab 3)



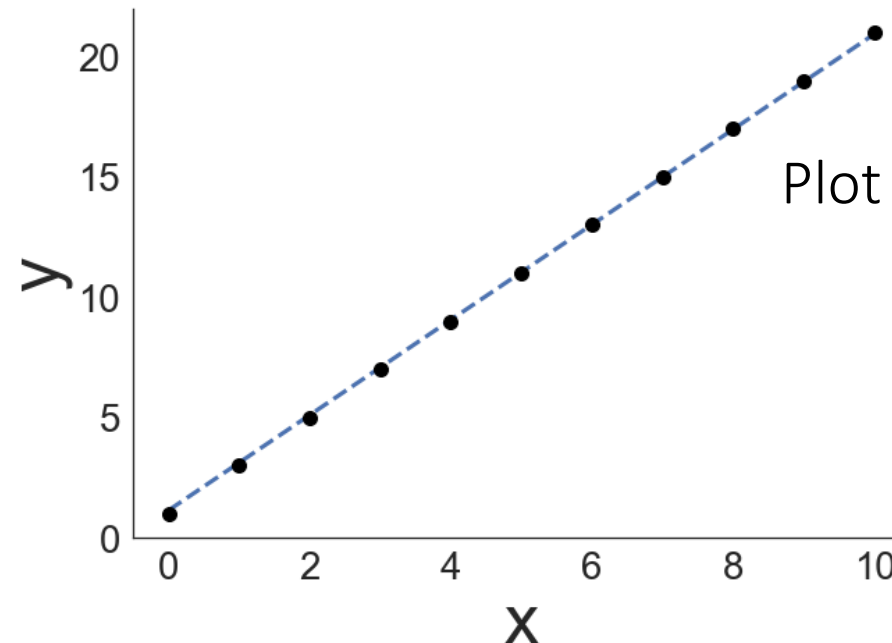
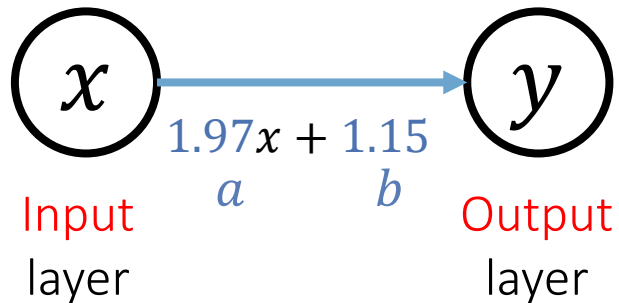
# Visualization and Evaluation (example task)

```
1 with torch.no_grad():
2
3     if torch.cuda.is_available():
4
5         predicted = model(torch.from_numpy(x_train).cuda()).cpu().numpy()
6
7     else:
8
9         predicted = model(torch.from_numpy(x_train)).numpy()
10
11 print(predicted)
12 print("a: " + str(model.linear.weight.cpu().numpy()), "b: " + str(model.linear.bias.cpu().numpy()))
```

Feed the trained model with the original  $x_{\text{train}}$  to produce  $y$  predictions

```
[[ 1.1490046]
 [ 3.12752 ]
 [ 5.1060357]
 [ 7.0845513]
 [ 9.063067 ]
 [11.041583 ]
 [13.020099 ]
 [14.998614 ]
 [16.977129 ]
 [18.955645 ]
 [20.93416  ]]
a: [[1.9782206]] b: [1.1490046]
```

Predicted  $y$



Plot the targets vs prediction



# PYTHON CONCEPTS FOR PYTORCH

Python Classes

PyTorch Tensors

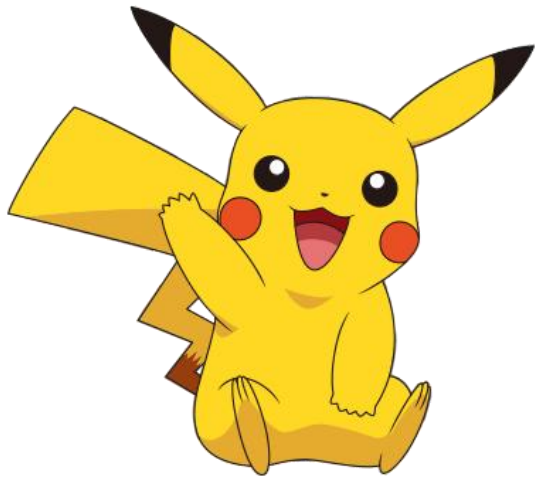


# Python Classes





# Python Classes



**Class:** Pokemon

Name: Pikachu  
Type: Electric  
Health: 70

} Attributes

Attack()  
Dodge()  
Evolve()

} Methods



**Class:** Pokemon

Name: Espeon  
Type: Psychic  
Health: 90

Attack()  
Dodge()  
Evolve()

Class: collection of objects (pokemon) of the same type (pokemon class)



# Python Classes

```
1 class Pokemon():
2     def __init__(self, Name, Type, Health):
3         self.Name = Name
4         self.Type = Type
5         self.Health = Health
6
7     def whats_your_name(self):
8         print("My name is " + self.Name + "!")
9
10    def attack(self):
11        print("Electric attack! Zap!!")
12
13    def dodge(self):
14        print("Pikachu Dodge!")
15
16    def evolve(self):
17        print("Evolving to Raichu!!")
```

```
1 pk1 = Pokemon(Name = "Pikachu", Type = "Electric", Health = 70)
```

```
1 pk1.Name
```

'Pikachu'

```
1 pk1.whats_your_name()
```

My name is Pikachu!

```
1 pk1.attack()
```

Electric attack! Zap!!

Creating a class "Pokemon"

Initialize the Pokemon object with provided Name, Type and Health parameters

Add functions for each method

Create a Pokemon object named "pk1"

Name can be inferred using .Name directive

Calling each function within the class prints the intended statements



# Python Classes: super() function

Parent class

```
1 class linearRegression(torch.nn.Module):
2     def __init__(self, inputSize, outputSize):
3         super(linearRegression, self).__init__()
4         self.linear = torch.nn.Linear(inputSize, outputSize)
5
6     def forward(self, x):
7         out = self.linear(x)
8         return out
```

Initializing the parent class (`torch.nn.Module`) allows us to use attributes/methods from `nn.Module` - e.g. `nn.Linear()`

More on Python classes: <http://introtopython.org/classes.html>



# PyTorch Tensors



# PyTorch Tensors

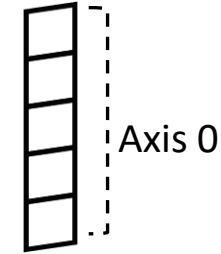
Main data structure for PyTorch

Like NumPy arrays, but optimized for machine learning

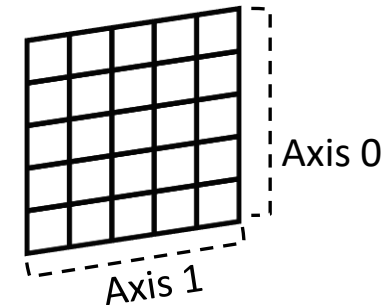
- Can be processed by both CPU and GPU
- Optimized for automatic differentiation (auto-grad)

Three main attributes

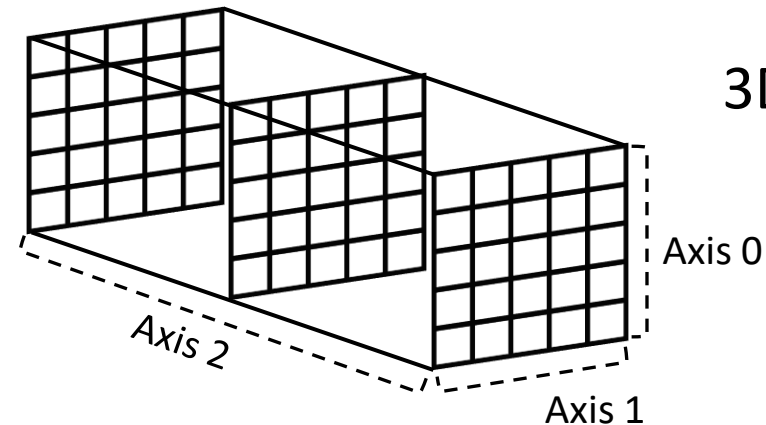
- shape – dimensions of array
- datatype – form of each entry (float, int, etc)
- device – CPU or cuda (GPU)



1D Tensor



2D Tensor



3D Tensor



# PyTorch Tensors vs NumPy Arrays

Creating a NumPy array

```
1 array1 = np.array([1,2,3,4])  
2 print(array1, type(array1))
```

```
[1 2 3 4] <class 'numpy.ndarray'>
```

Creating a torch tensor

```
1 tensor1 = torch.tensor([1,2,3,4])  
2 print(tensor1, type(tensor1))
```

```
tensor([1, 2, 3, 4]) <class 'torch.Tensor'>
```

NumPy Array -> PyTorch Tensor

```
1 array1_torch = torch.from_numpy(array1)  
2 print(array1_torch, type(array1_torch))
```

```
tensor([1, 2, 3, 4], dtype=torch.int32) <class 'torch.Tensor'>
```

PyTorch Tensor -> NumPy Array

```
1 tensor1_numpy = tensor1.numpy()  
2 print(tensor1_numpy, type(tensor1_numpy))
```

```
[1 2 3 4] <class 'numpy.ndarray'>
```

More on Numpy arrays vs torch tensors:

<https://rickwierenga.com/blog/machine%20learning/numpy-vs-pytorch-linalg.html>



# Handling Torch Tensors

Moving tensors to CPU

`.cpu()`

```
1 tensor1_cpu = tensor1.cpu()  
2 print(tensor1_cpu.device)
```

cpu

Moving tensors to GPU

`.gpu()`

```
1 tensor1_gpu = tensor1.cuda()  
2 print(tensor1_gpu.device)
```

cuda:0

Disabling gradient calculation

`torch.no_grad()`

```
1 with torch.no_grad():  
2  
3     predicted = model(torch.from_numpy(x_train)).numpy()
```

(useful when you just want to inspect the values inside tensors)