

Introduction to PyTorch

STA326

Department of Statistics and Data Science

Feb. 25, 2026

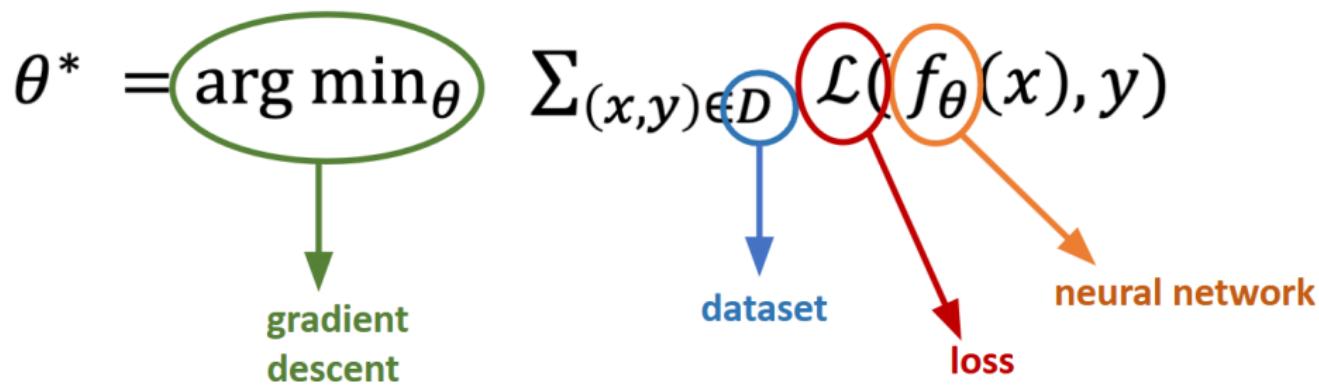
What We will cover?

- Introduction to PyTorch
- Tensor Basics
- Data Manipulation
- Data Preprocessing

Introduction to PyTorch

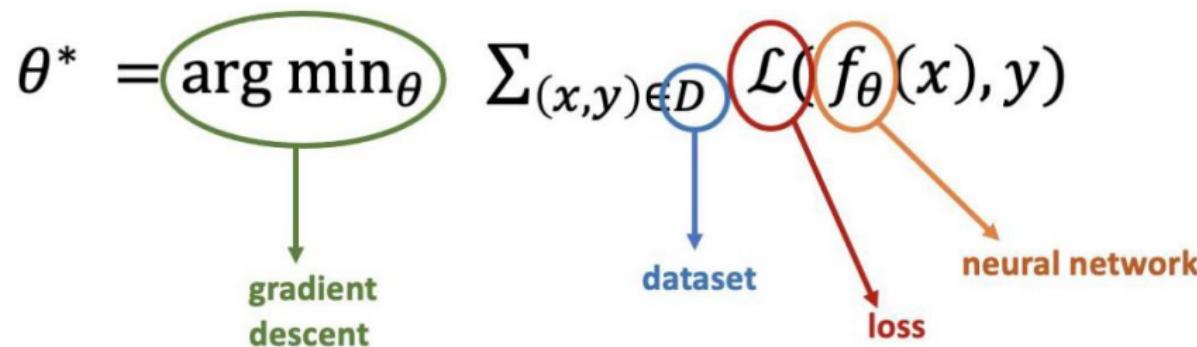
How do train a model?

- PyTorch does all of these!



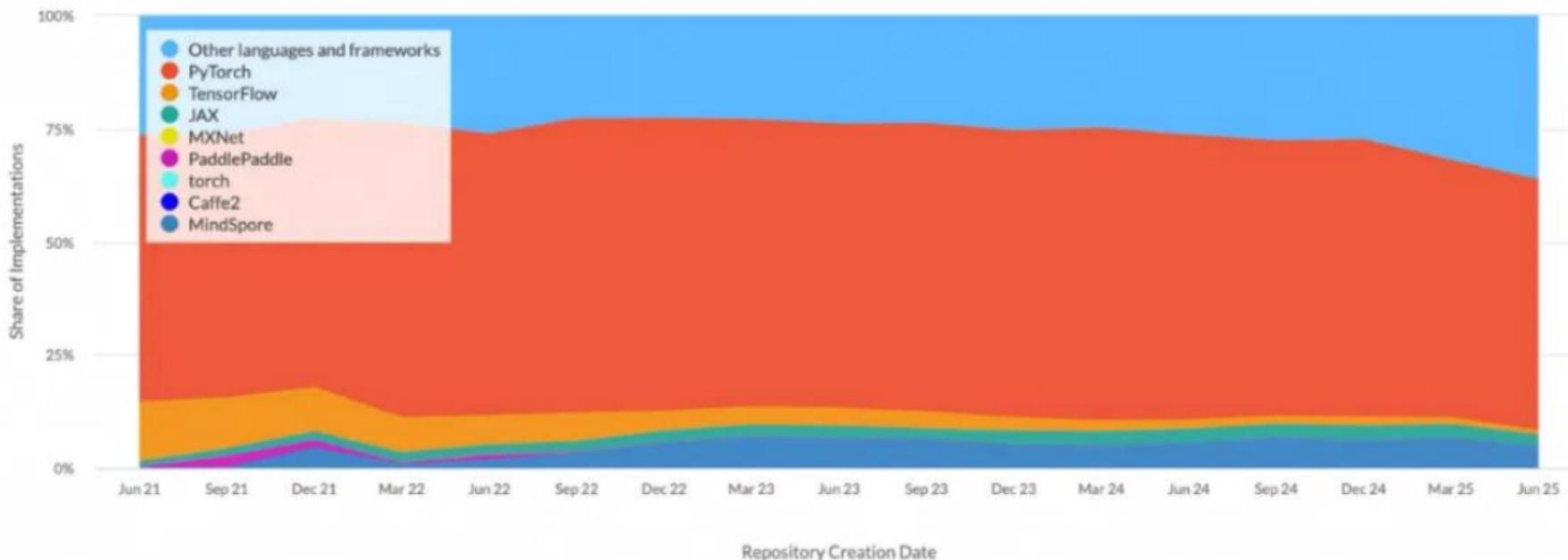
What is PyTorch?

- A open source machine learning library
- Defining neural networks
- Automating computing gradients
- And more! (datasets, optimizers, GPUs, etc.)



Why PyTorch?

Paper Implementations grouped by framework



Why PyTorch?



- Fast CPU implementations
- **CPU-only**
- **No autodiff**
- **Imperative**



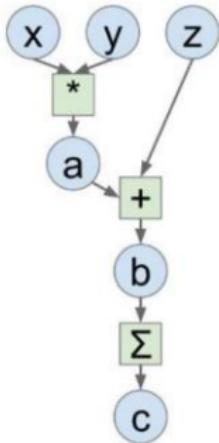
- Fast CPU implementations
- **Allows GPU**
- **Supports autodiff**
- **Imperative**

Other features include:

- Datasets and dataloading
- Common neural network operations
- Built-in optimizers (Adam, SGD, ...)

Why PyTorch?

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

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

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

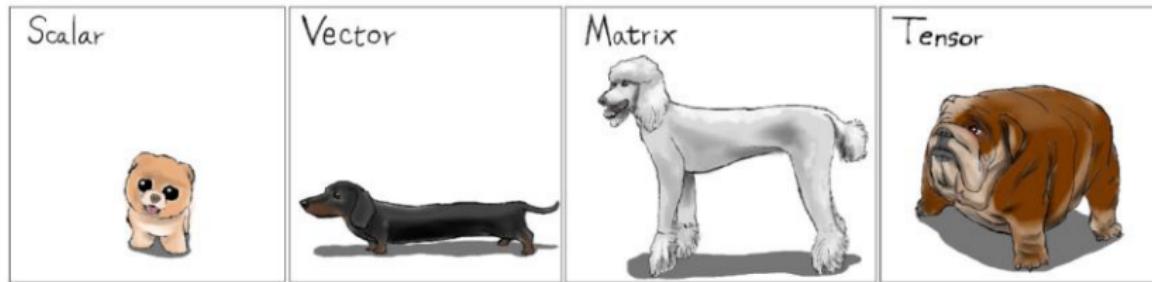
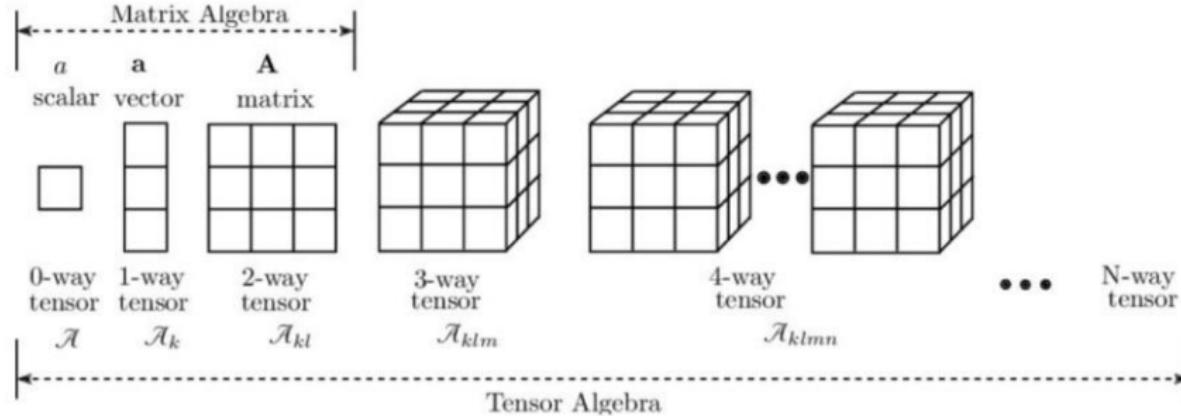
c.backward()
```

- It is pythonic-concise, close to Python conventions
- Autograd-automatic differentiation

Tensor Basics

Tensors in PyTorch

- Comparison with NumPy: PyTorch supports GPU and automatic differentiation.



Why Do We Need Data Operations (e.g., Tensor)?

- **Definition:** A multi-dimensional array (like NumPy's ndarray but with additional capabilities).
- **Example:** Output: tensor([0, 1, 2, ..., 11])

```
import torch
x = torch.arange(12)
print(x)
```

- **Access Shape and Size:**

```
print(x.shape) # torch.Size([12])
print(x.numel()) # Total elements: 12
```

Shape

A

	Axis 1				
Axis 0	32	27	5	54	1
	99	4	23	3	57
	76	42	34	82	5

`A.shape == (3, 5)`

Data Manipulation

Reshaping Tensors

- Reshape with:

```
X = x.reshape(3, 4)  
print(X)
```

- Automatic Dimension Calculation (-1)

```
X = x.reshape(-1, 4) # Auto calculates rows  
print(X.shape) # (3, 4)
```

Initializing Tensors

- Zeros, Ones, Random Values:

```
torch.zeros((2,3,4))  
torch.ones((2,3,4))  
torch.randn(3, 4) # Normal distribution
```

- Create Tensor from Others:

List: `torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4]])`

NumPy: `A = X.numpy()
B = torch.tensor(A)`

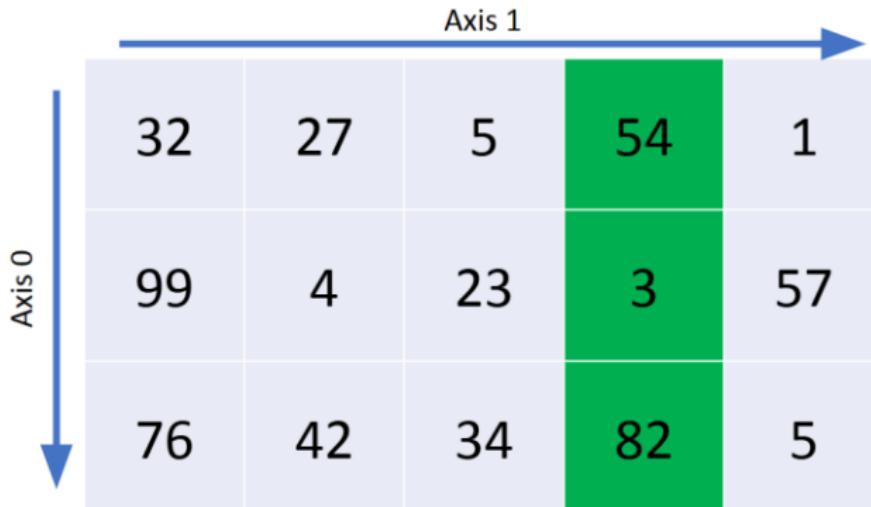
Indexing

■ Basic Indexing:

Axis 1				
Axis 0	32	27	5	54
	99	4	23	3
	76	42	34	82
				5

$A[0, 3]$

Indexing



$A[:, 3]$

Indexing

Axis 1				
Axis 0	32	27	5	54
	99	4	23	3
	76	42	34	82
				5

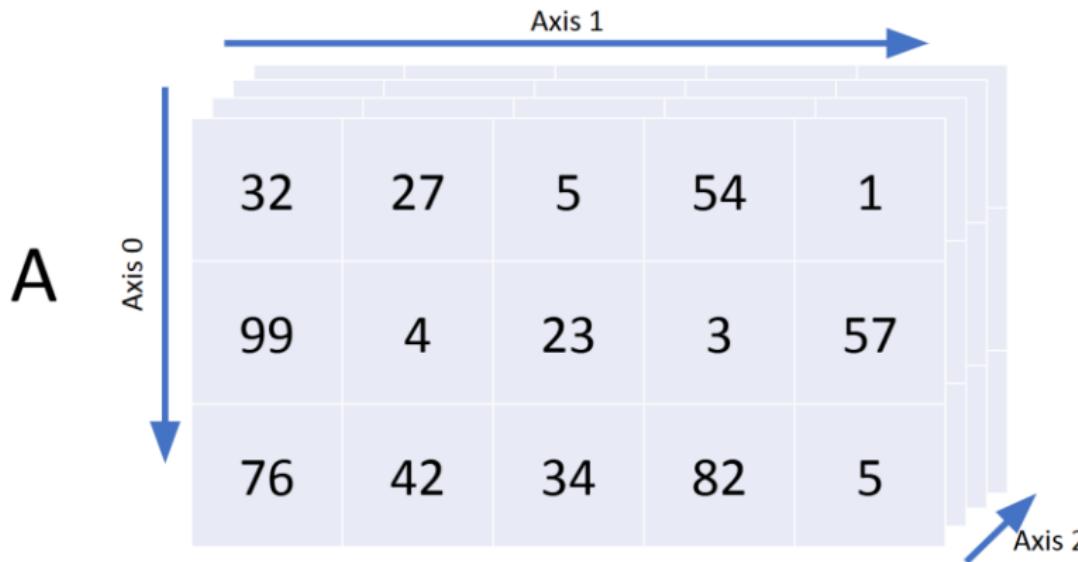
$A[0, :]$

Indexing

Axis 1				
Axis 0	32	27	5	54
	99	4	23	3
	76	42	34	82
				5

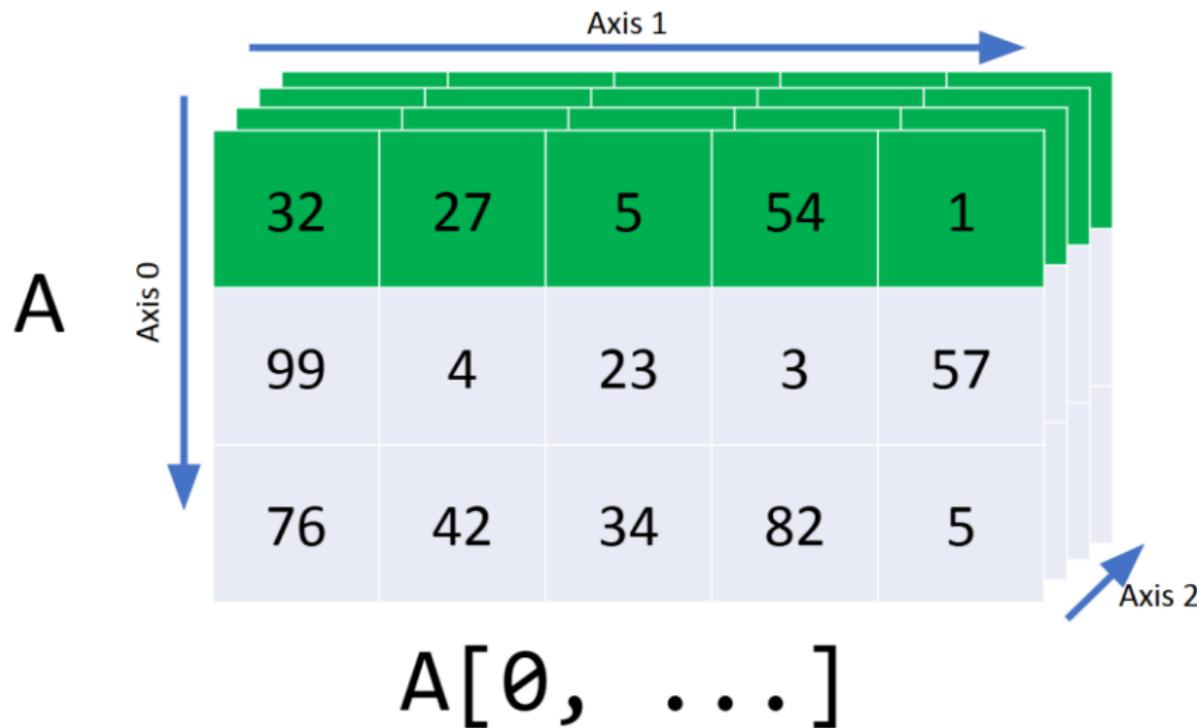
$A[0, 2:4]$

Multidimensional Indexing

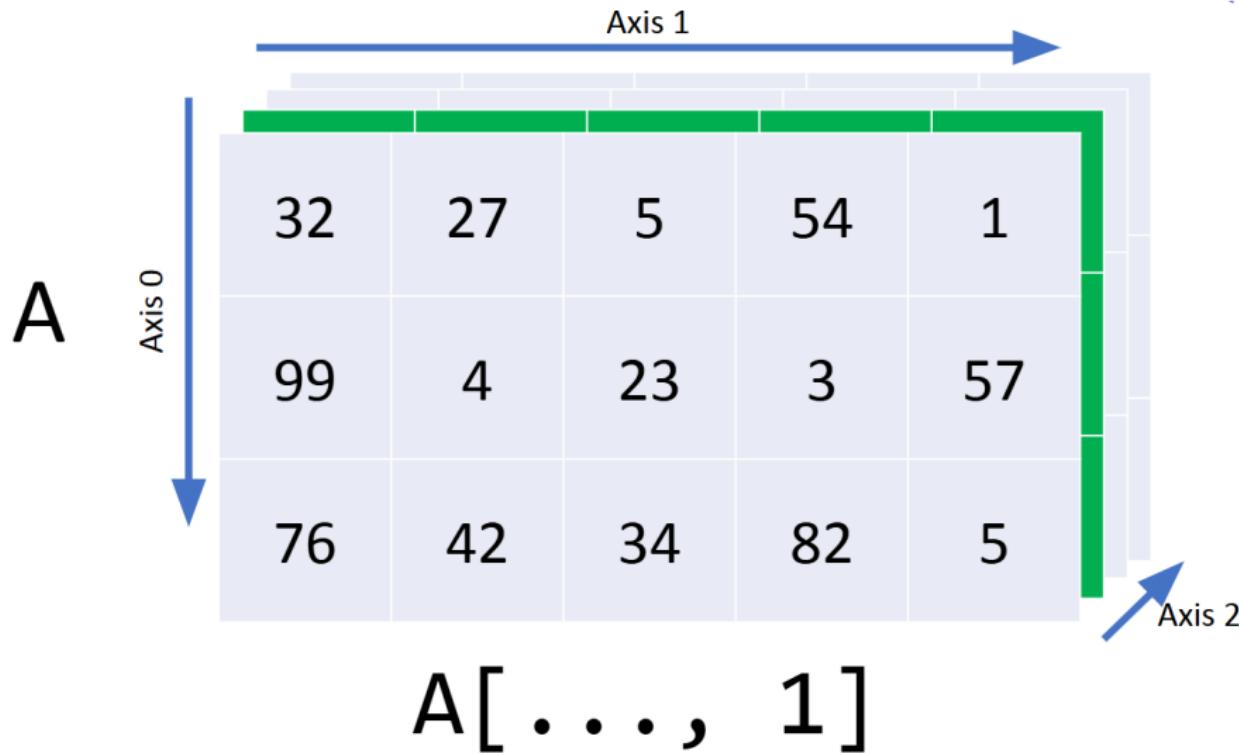


`A.shape == (3, 5, 4)`

Multidimensional Indexing



Multidimensional Indexing



Assigning Values

■ Assigning Values:

```
X[1, 2] = 9
```

■ Assigning Multiple Values:

```
X[0:2, :] = 12
```

Mathematical Operations

- Element-wise Operations:

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
print(x + y, x - y, x * y, x / y, x ** y)
```

- Exponential Function:

```
torch.exp(x)
```

Concatenation and Splitting of Tensors

- Concatenating along different dimensions:

```
A = torch.tensor([[1, 2], [3, 4]])
B = torch.tensor([[5, 6], [7, 8]])
print(torch.cat((A, B), dim=0))
print(torch.cat((A, B), dim=1))
```

- Splitting Tensors:

```
Z = torch.arange(12).reshape(3, 4)
split_result = torch.split(Z, 2, dim=1) # Split
print(split_result)
```

Broadcasting

- **Broadcasting:** Expands smaller tensors to match the larger tensor's dimensions.
- **Example:**

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
print(a + b) # Automatically expands dimensions
```

- Why Useful?
 - 1) Avoids unnecessary memory allocation.
 - 2) Simplifies mathematical operations.

Saving Memory

- Running operations can cause new memory to be allocated to host results.

```
before = id(Y)  
Y = Y + X  
id(Y) == before
```



False

```
before = id(X)  
X += Y  
id(X) == before
```



True

Conversion to Other Python Objects

- Converting to a NumPy tensor (ndarray)

- **Example:**

```
A = X.numpy()  
B = torch.from_numpy(A)  
type(A), type(B)
```

- Convert a size-1 tensor to a Python scalar

- **Example:**

```
a = torch.tensor([3.5])  
a, a.item(), float(a), int(a)
```

Data Preprocessing

Why Data Preprocessing Matters?

■ Why Data Preprocessing Matters?

- Real-world data is often messy and requires cleaning.
- Deep learning models need structured and numerical data.
- Pandas and PyTorch provide powerful tools for data manipulation.

■ Reading the Dataset:

- Reading and Loading Data
- Handling Missing Values
- Converting Categorical Data
- Transforming Data into PyTorch Tensors

Reading the Dataset

- **CSV Files:** Comma-separated values format is widely used for structured data.

```
import pandas as pd  
data = pd.read_csv('../data/house_tiny.csv')  
print(data)
```

- **Sample Output:**

	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000

Handling Missing Values

- **Types of Missing Data:** *NaN*, Empty or missing values detected by pandas.
- **Strategies to Handle Missing Data:**
 - **Drop missing values:** Remove incomplete rows.
 - **Remove incomplete rows:** Fill in missing values with heuristics(e.g., mean value).

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000



	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

Why Convert Categorical Data?

Converting Data to PyTorch Tensors

- PyTorch models work with tensors instead of Pandas data.
- Example:

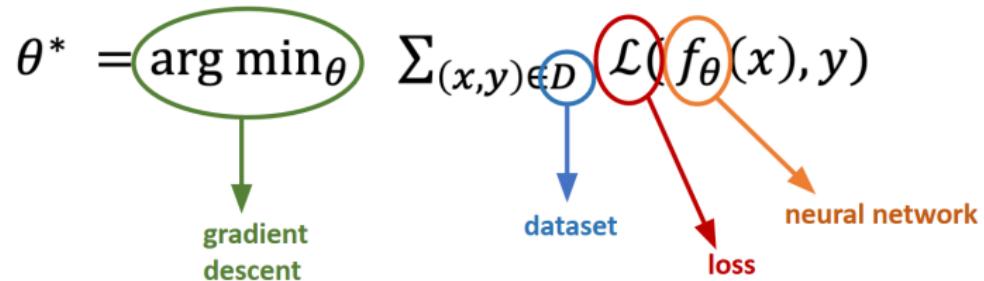
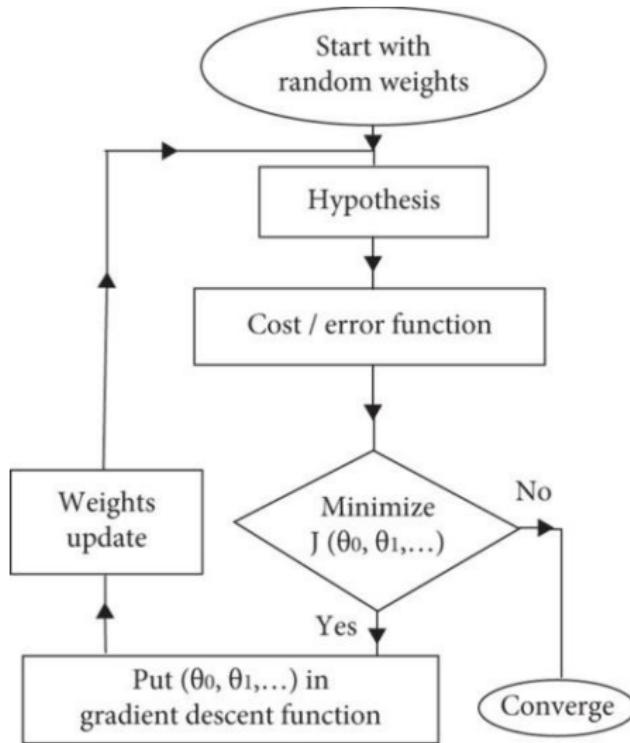
```
import torch
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
print(X, y)
```



```
(tensor([[nan, 0., 1.],
       [2., 0., 1.],
       [4., 1., 0.],
       [nan, 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

Linear Regression

■ Workflow:



- Linear Regression is very often assumed to be the *simplest* model of Machine Learning

Exercise

- **pytorch_preliminary.ipynb**: Exercise 1-4.
- **linear_regression.ipynb**: Exercise 1-6 on linear regression.