Deep Learning with PyTorch

NSF Workshop – Deep Learning on GPU

Tony T. Luo

Associate Professor of Computer Science, Missouri S&T

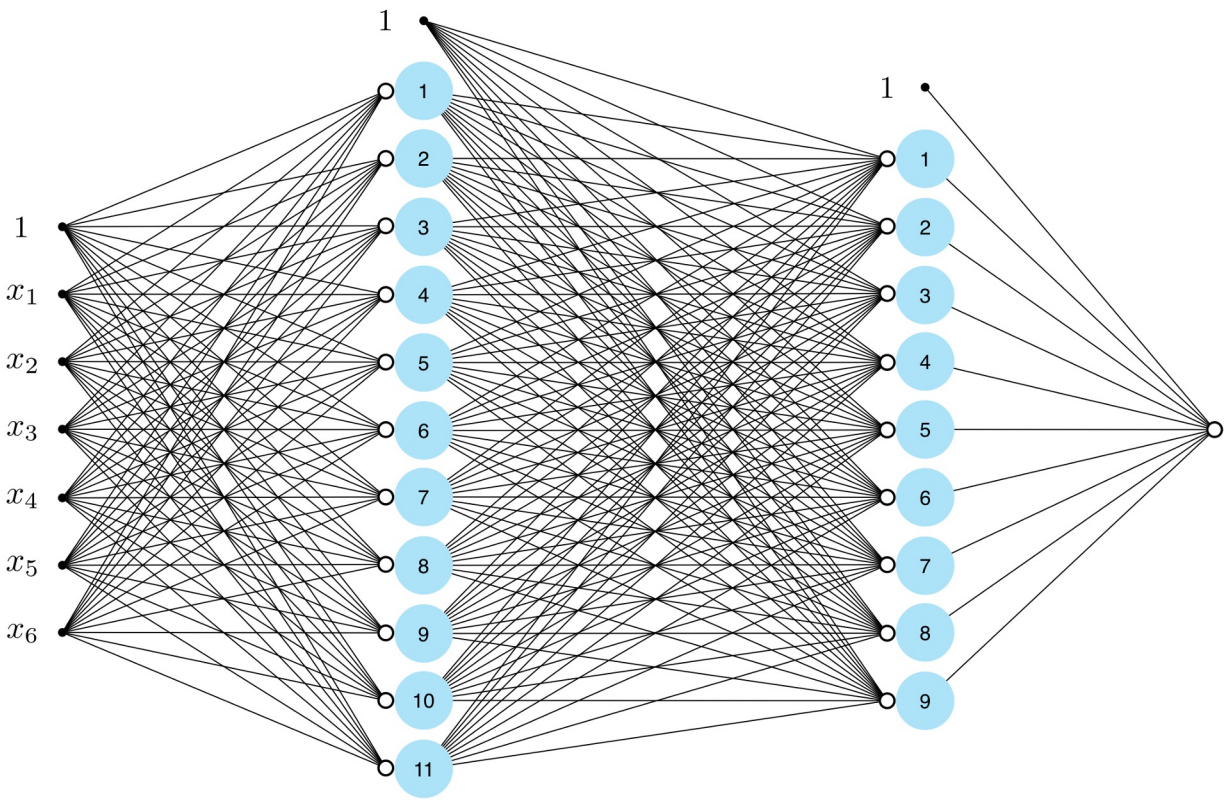
May 15, 2024

Agenda

1. Multi-Layer Perceptron (MLP)
2. PyTorch

Multi-Layer Perceptron:
fully connected neural networks

A two-layer MLP network

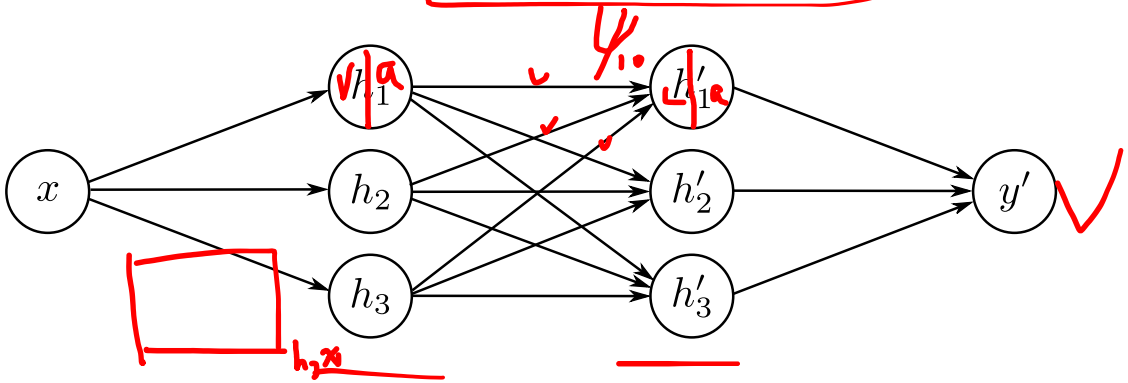


Formulation

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$h'_1 = \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]$$
$$h'_2 = \mathrm{a}[\psi_{20} + \psi_{21}h_2 + \psi_{22}h_2 + \psi_{23}h_3]$$
$$h'_3 = \mathrm{a}[\psi_{30} + \psi_{31}h_2 + \psi_{32}h_2 + \psi_{33}h_3]$$

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

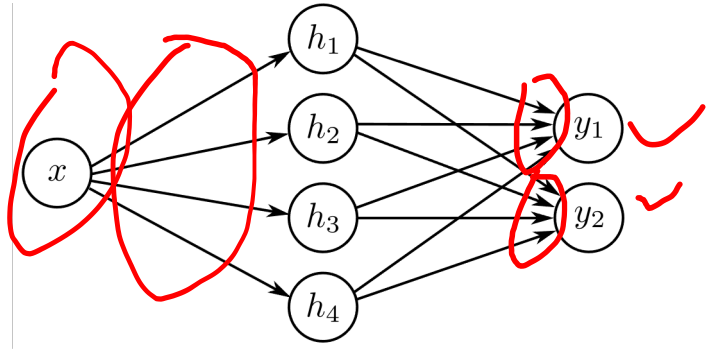


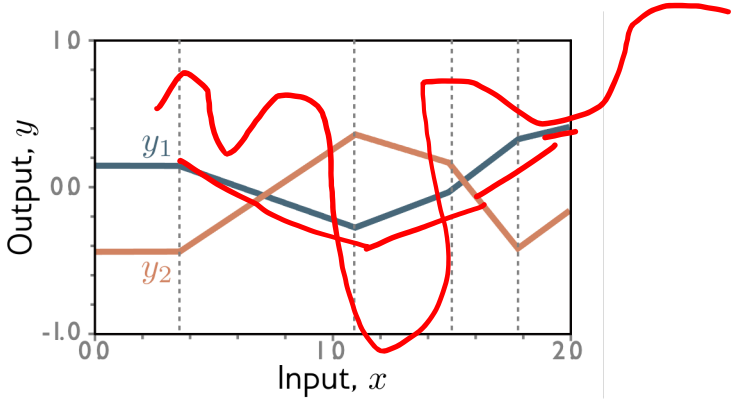
Two outputs

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$
$$h_4 = \mathrm{a}[\theta_{40} + \theta_{41}x]$$

$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$
$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4$$

3



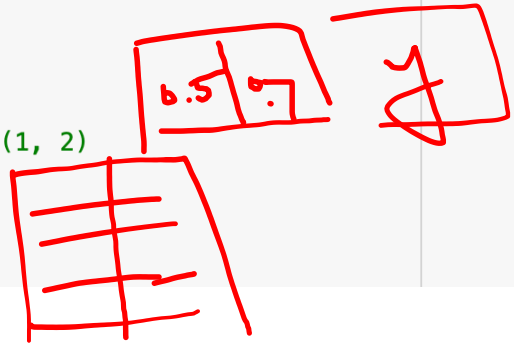


MLP implementation using Numpy

```python
import numpy as np

class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_to_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.weights_hidden_to_output = np.random.randn(self.hidden_size, self.output_size)

        self.biases_hidden = np.zeros((1, self.hidden_size))
        self.biases_output = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)
```

```python
    def forward(self, X):
        # Forward pass for the input layer to hidden layer
        self.hidden_layer_input = np.dot(X, self.weights_input_to_hidden) + self.biases_hidden
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_input)

        # Forward pass for the hidden layer to output layer
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_to_output) + self.biases_output
        self.output_layer_output = self.sigmoid(self.output_layer_input)

        return self.output_layer_output

# Example usage:
input_size = 2
hidden_size = 3
output_size = 1

mlp = MLP(input_size, hidden_size, output_size)

# Assuming we have a single input sample with shape (1, 2)
X = np.array([[0.5, 0.7]])
output = mlp.forward(X)
print("Output:", output)
```

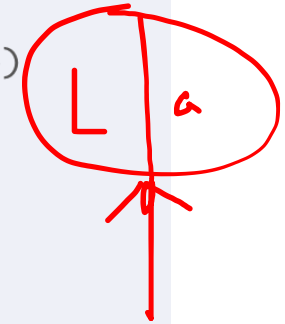
```
Output: [[0.58129955]]
```



MLP implementation using PyTorch

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        return x

```

sequential



```python
# Example usage:
input_size = 2
hidden_size = 3
output_size = 1

mlp = MLP(input_size, hidden_size, output_size)

# Assuming we have a single input sample with shape (1, 2)
input_sample = torch.tensor([[0.5, 0.7]], dtype=torch.float32)
output = mlp(input_sample)
print("Output:", output)
```

```
Output: tensor([[0.4785]], grad_fn=<SigmoidBackward0>)
```

A Crash Course of PyTorch

- What is Pytorch?
- Installation
- Tensors
- Dataset & Dataloader
- Training & Testing Neural Networks in Pytorch
- torch.nn: Models, Loss Functions
- torch.optim: Optimization algorithms
- Save/load models

What is PyTorch?

- A machine learning framework in Python.
- Two main features:
  - N-dimensional **Tensor** operations on GPUs
    - Think of it as a parallel version of NumPy
  - Automatic differentiation (**autograd**) for gradient computation
    - Used in neural network training

PyTorch

PyTorch Installation

- Install PyTorch using pip (the Python package manager) or Anaconda/Miniconda (convenient to manage Python packages and environments)

```bash
pip install torch torchvision
```

  - PyTorch.org provides installation instructions for various operating systems

- Install PyTorch with CUDA support if you have NVIDIA GPUs:
  - Download and install CUDA Toolkit: https://developer.nvidia.com/cuda-toolkit-archive
  - Download and install cuDNN: https://developer.nvidia.com/rdp/cudnn-download
  - Install PyTorch: https://pytorch.org

```
conda install pytorch torchvision torchaudio cudatoolkit=x.x -c pytorch
```

- Sometimes you may want to create a virtual environment if you work on multiple projects (so you can install PyTorch and other dependencies without affecting your system-wide Python installation or other projects):

```bash
conda create -n myenv python=3.8
conda activate myenv
conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

After you are done, simply run “PyTorch” to enter PyTorch command line interface

Verification

```
(pytorch) $ python
>>> import torch

>>> x = torch.rand(5, 3)
>>> print(x)

>>> torch.cuda.is_available()
```

The easiest way

In this course, however, you do not have to bother with this process:

- Google Colab: http://colab.research.google.com/
- UNT GPU server (via VPN): http://129.120.60.203:8000

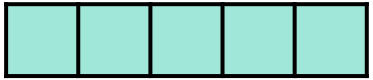
So the previous instructions are for your future use or if you prefer to use your own local machine this time

A Crash Course of PyTorch

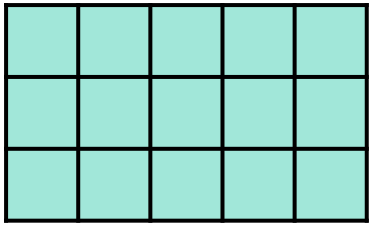
- What is Pytorch?
- Installation
- Tensors
- Dataset & Dataloader
- Training & Testing Neural Networks in Pytorch
- torch.nn: Models, Loss Functions
- torch.optim: Optimization algorithms
- Save/load models

Tensors

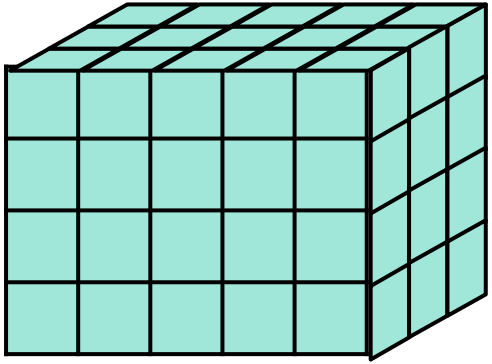
- High-dimensional matrices (arrays)



1-D tensor
e.g. audio



2-D tensor
e.g. black & white images



3-D tensor
e.g. RGB images

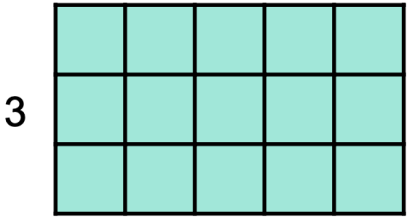
Tensors – Shape of Tensors

- Check with .shape

5

(5,)

dim 0

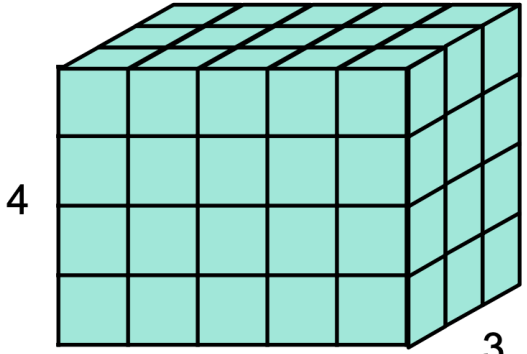


3

5

(3, 5)

dim 0 dim 1



4

5

3

(4, 5, 3)

dim 0 dim 1 dim 2

Note: **dim** in PyTorch == **axis** in NumPy

Tensors – Creating Tensors

- Directly from data (list or numpy array)

```
x = torch.tensor([[1, -1], [-1, 1]])

x = torch.from_numpy(np.array([[1, -1], [-1, 1]]))
```

```
tensor([[1., -1.],
        [-1., 1.]])
```

- Tensor of constant zeros & ones

```
x = torch.zeros([2, 2])

x = torch.ones([1, 2, 5])
```

shape

```
tensor([[0., 0.],
        [0., 0.]])
```

```
tensor([[[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]]])
```

Tensors – Common Operations

Common arithmetic functions are supported, such as:

- Addition

```
z = x + y
```

- Subtraction

```
z = x - y
```

- Power

```
y = x.pow(2)
```

- Summation

```
y = x.sum()
```

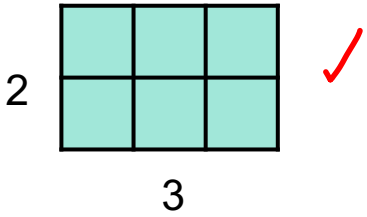
- Mean

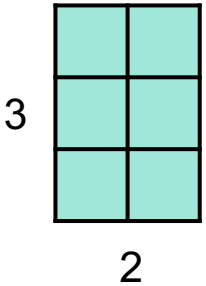
```
y = x.mean()
```

Tensors – Common Operations

- **Transpose**: transpose two specified dimensions

```
>>> x = torch.zeros([2, 3])

>>> x.shape
torch.Size([2, 3])

>>> x = x.transpose(0, 1)

>>> x.shape

torch.Size([3, 2])
```

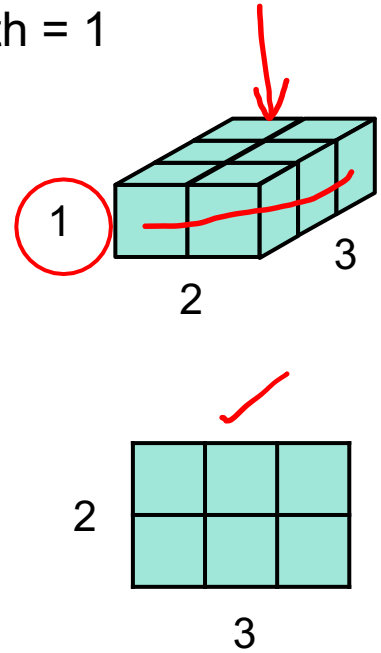




Tensors – Common Operations

- **Squeeze**: remove the specified dimension with length = 1

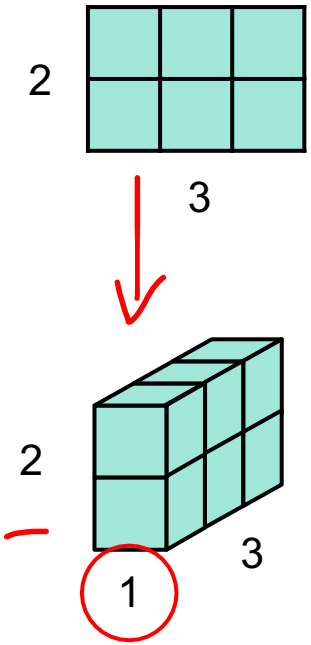
```
>>> x = torch.zeros([1, 2, 3])
>>> x.shape
torch.size([1, 2, 3])
>>> x = x.squeeze(0)
                (dim = 0)
>>> x.shape
torch.Size([2, 3])
```



Tensors – Common Operations

- **Unsqueeze**: expand a new dimension

```
>>> x = torch.zeros([2, 3])
>>> x.shape
torch.size([2, 3])
>>> x = x.unsqueeze(1)      (dim = 1)
>>> x.shape
torch.Size([2, 1, 3])
```



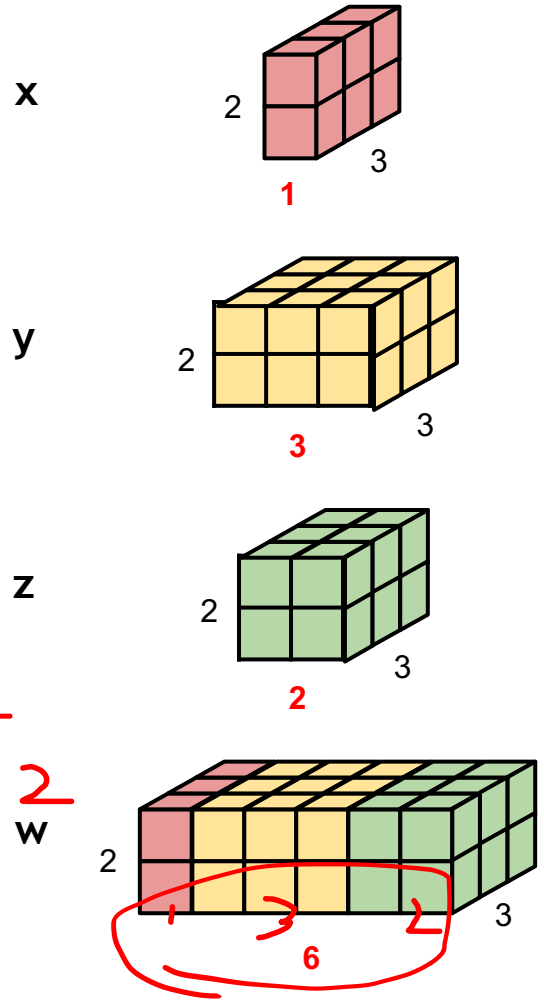
Tensors – Common Operations

- **Cat**: concatenate multiple tensors

```
>>> x = torch.zeros([2, 1, 3])
>>> y = torch.zeros([2, 3, 3])
>>> z = torch.zeros([2, 2, 3])
>>> w = torch.cat([x, y, z], dim=1)
>>> w.shape
torch.Size([2, 6, 3])
```

0

2



More operators: https://pytorch.org/docs/stable/tensors.html

Tensors – Data Type

- Using different data types for model and data will cause errors.

Data type	dtype	tensor
32-bit floating point	`torch.float`	`torch.FloatTensor`
64-bit integer (signed)	`torch.long`	`torch.LongTensor`

see https://pytorch.org/docs/stable/tensors.html for more information on data types.

Tensors – PyTorch v.s. NumPy

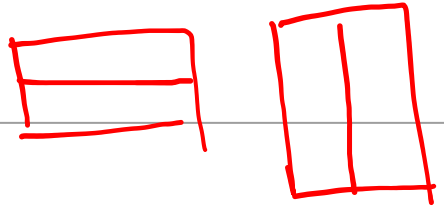
- Similar attributes

PyTorch	NumPy
`x.shape`	`x.shape`
`x.dtype`	`x.dtype`

see official documentation for more information on data types.

Tensors – PyTorch v.s. NumPy

- Many functions have the same names as well



PyTorch	NumPy
`x.reshape / x.view`	`x.reshape`
`x.squeeze()`	`x.squeeze()`
`x.unsqueeze(1)`	`np.expand_dims(x, 1)`

Tensors – Device

- Tensors & modules will be computed with **CPU** by default

  Use **.to()** to move tensors to appropriate devices.

- CPU

```
x = x.to('cpu')
```

- GPU

```
x = x.to('cuda')
```

Tensors – Device (GPU)

- Check if your computer has NVIDIA GPU

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

- Multiple GPUs: specify `'cuda:0'`, `'cuda:1'`, `'cuda:2'`, ...

- Why use GPUs?
  - Parallel computing with more cores for arithmetic calculations
  - See What is a GPU and do you need one in deep learning?

Tensors – Gradient Calculation

1
```
>>> x = torch.tensor([[1., 0.], [-1., 1.]], requires_grad=True)
```

2
```
>>> z = x.pow(2).sum()
```

3
```
>>> z.backward()
```

4
```
>>> x.grad
tensor([[ 2., 0.],
        [-2.,  2.]])
```

1
$$x = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

2
$$z = \sum_i \sum_j x_{i,j}^2$$

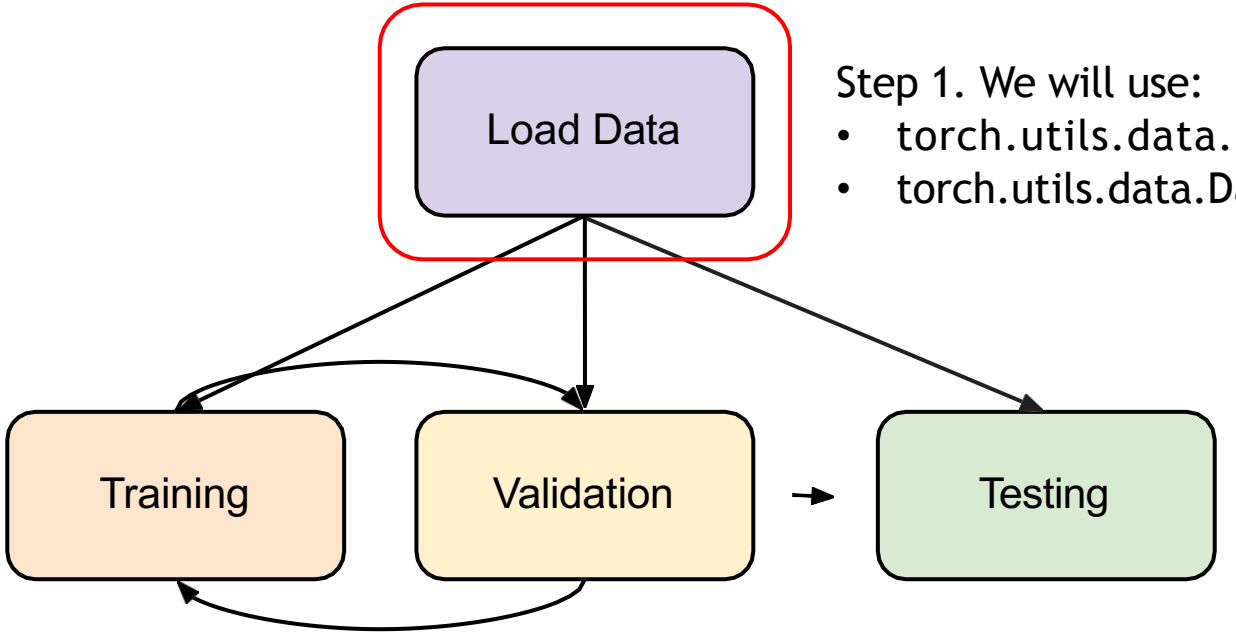
3
$$\frac{\partial z}{\partial x_{i,j}} = 2x_{i,j}$$

4
$$\frac{\partial z}{\partial x} = \begin{bmatrix} 2 & 0 \\ -2 & 2 \end{bmatrix}$$

A Crash Course of PyTorch

- What is Pytorch?
- Installation
- Tensors
- Dataset & Dataloader
- Training & Testing Neural Networks in Pytorch
- torch.nn: Models, Loss Functions
- torch.optim: Optimization algorithms
- Save/load models

Training & Testing in Pytorch



Step 1. We will use:

- torch.utils.data.Dataset
- torch.utils.data.DataLoader

Dataset & Dataloader

- Dataset: stores data samples
- Dataloader: load data in batches

- **dataset** = MyDataset (<file or url>)
- dataloader = **DataLoader**(**dataset**, batch_size, **shuffle**=True)

Dataset & Dataloader

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

class MyDataset(Dataset):
    def __init__ (self, file):
        self.data = ...

    def __getitem__(self, index):
        return self.data[index]

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

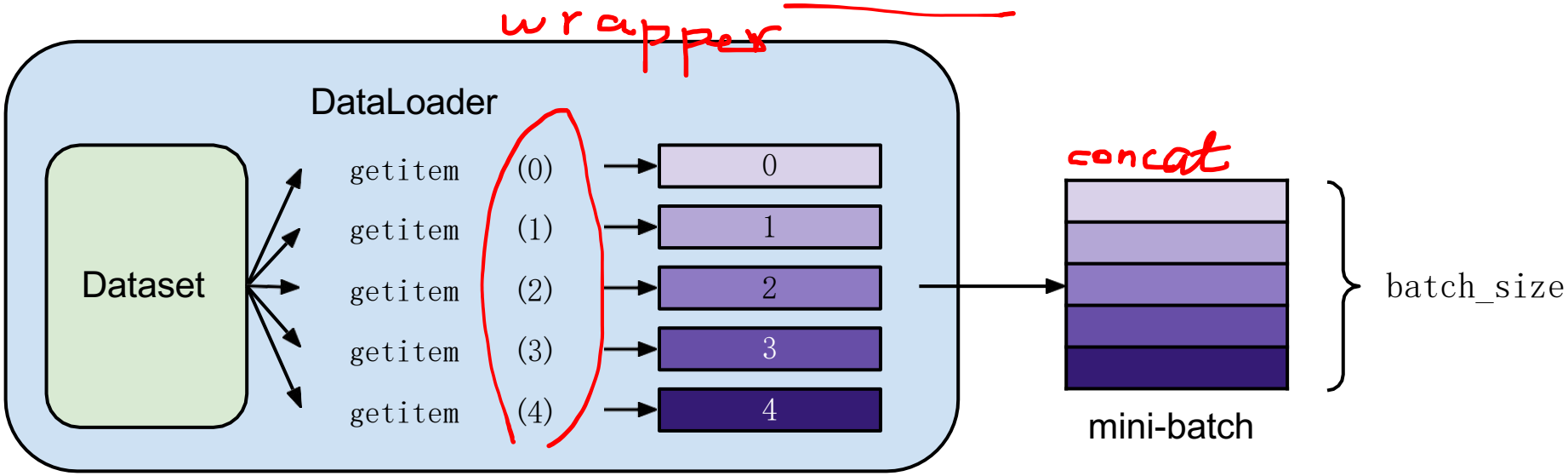
Read data & preprocess

Returns one sample at a time

Returns the size of the dataset

Dataset & Dataloader

```
dataset  = MyDataset(file)

dataloader = DataLoader(dataset, batch_size=5, shuffle=False)
```



Dataset

For simplicity, assume the CSV file has two columns: "input" and "target".

“transform” is optional.

```python
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader

class CSVDataset(Dataset):
    def __init__(self, csv_file, transform=None):
        self.data = pd.read_csv(csv_file)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = self.data.iloc[idx]
        input_data = torch.tensor(sample['input'].values, dtype=torch.float32)
        target = torch.tensor(sample['target'].values, dtype=torch.float32)

        if self.transform:
            input_data = self.transform(input_data)

        return input_data, target
```

numpy

Dataloader

```python
# Assuming we have a CSV file named 'data.csv' with 'input' and 'target' columns
csv_file = 'data.csv'

# Create an instance of the CSVDataset
dataset = CSVDataset(csv_file)

# Define the transformations (if any)
# For simplicity, let's assume we don't need any transformations
# transform = None

# Create a DataLoader
batch_size = 32
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Iterate over the DataLoader
for batch_idx, (inputs, targets) in enumerate(data_loader):
    # Here, inputs and targets are PyTorch tensors
    # You can perform your training/evaluation steps here
    # ...

    # For demonstration purposes, let's just print the batch size
    print(f"Batch {batch_idx+1}, Batch Size: {inputs.size(0)}")
```

targets

Create a csv file

bash

```bash
# Using Bash (Linux/macOS)
echo -e "input,target\n1.0,2.0\n2.0,3.0\n3.0,4.0\n4.0,5.0" > data.csv
```

powershell

```powershell
# Using PowerShell
echo "input,target`n1.0,2.0`n2.0,3.0`n3.0,4.0`n4.0,5.0" | Out-File -FilePath data.csv -Encoding UTF8
```

csv

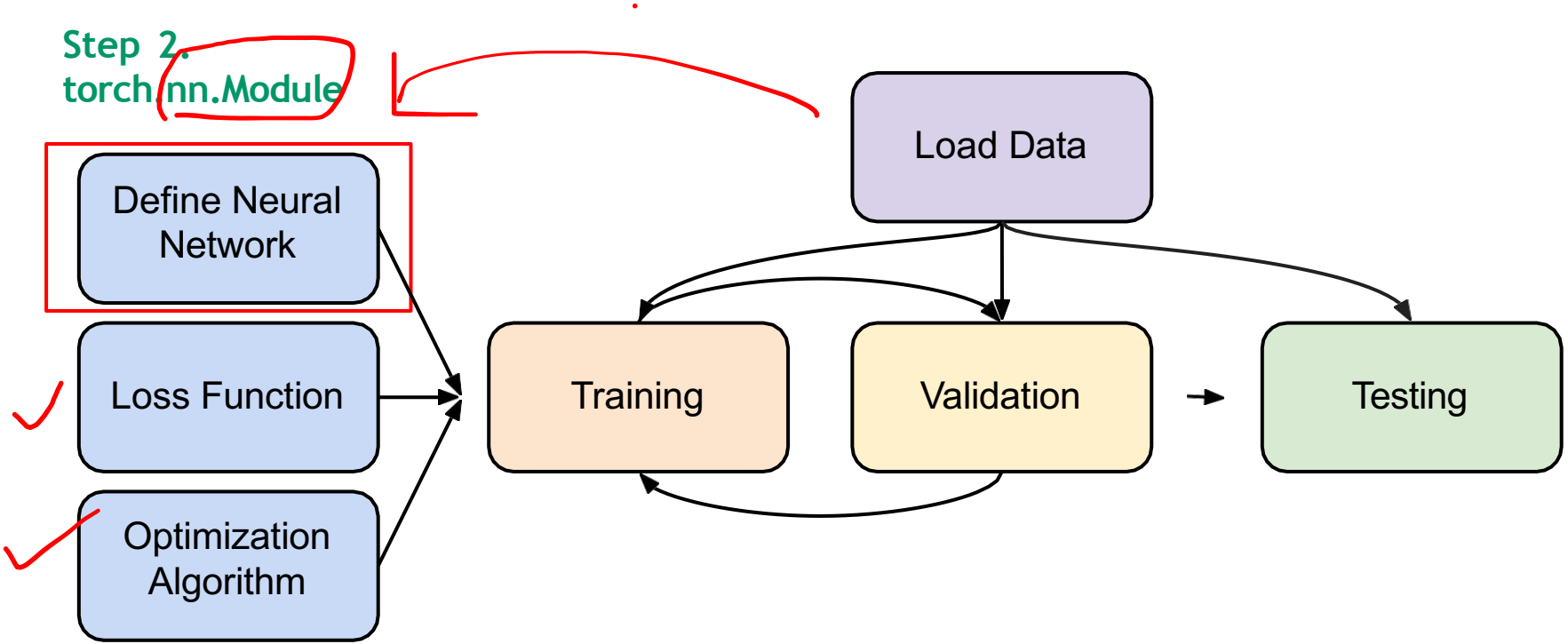
```csv
input,target
1.0,2.0
2.0,3.0
3.0,4.0
4.0,5.0
```

```
# Using Bash (Linux/macOS)
echo -e "input,target\n1.0,2.0\n2.0,3.0\n3.0,4.0\n4.0,5.0" > data.csv
# Using PowerShell
echo "input,target`n1.0,2.0`n2.0,3.0`n3.0,4.0`n4.0,5.0" | Out-File -FilePath data.csv -Encoding UTF8
# Or just a text editor!
input,target
1,2
3,4
5,6
7,8
9,10
```

A Crash Course of PyTorch

- What is Pytorch?
- Installation
- Tensors
- Dataset & Dataloader
- Training & Testing Neural Networks in Pytorch
- torch.nn: Models, Loss Functions
- torch.optim: Optimization algorithms
- Save/load models

Training & Testing Neural Networks – in Pytorch



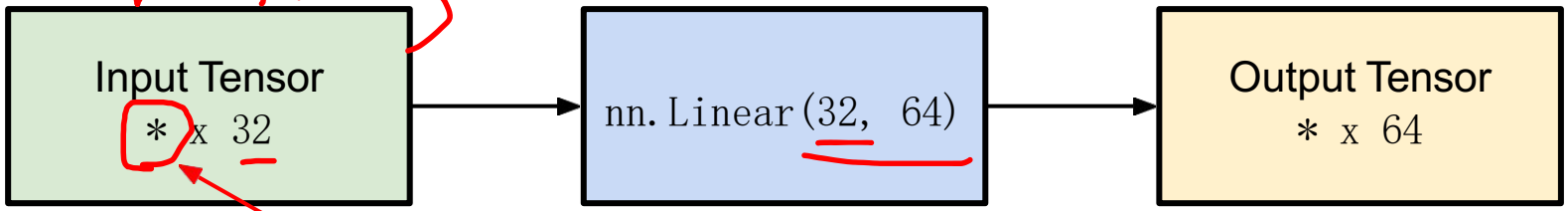
torch.nn — Network Layers

MLP

- Linear Layer (**Fully-connected** Layer)

```
nn.Linear(in_features, out_features)
```

H, W, D=3

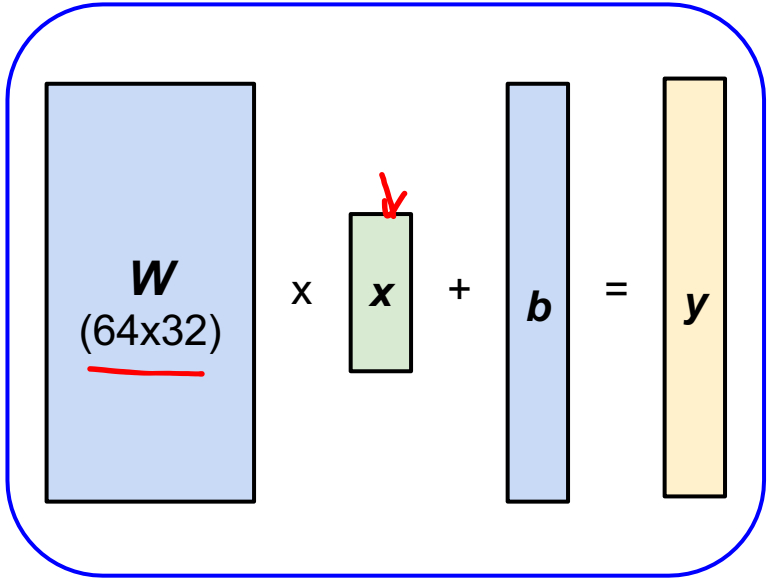


can be any shape (but last dimension must be 32)
e.g. (10, 32), (10, 5, 32), (1, 1, 3, 32), ...

torch.nn – Network Layers

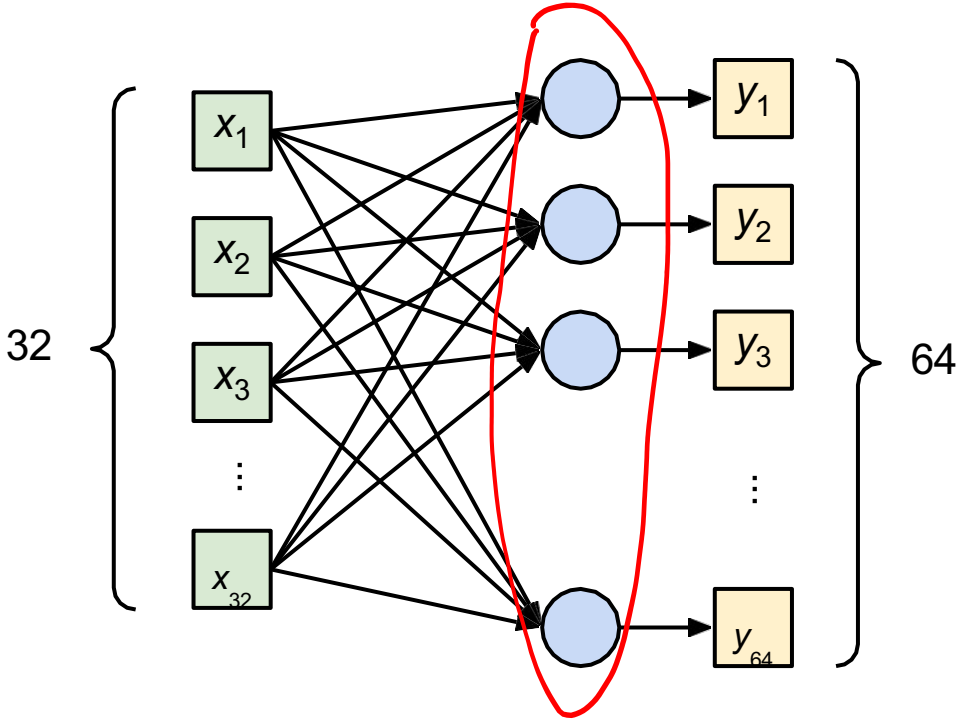
- Linear Layer (**Fully-connected** Layer)

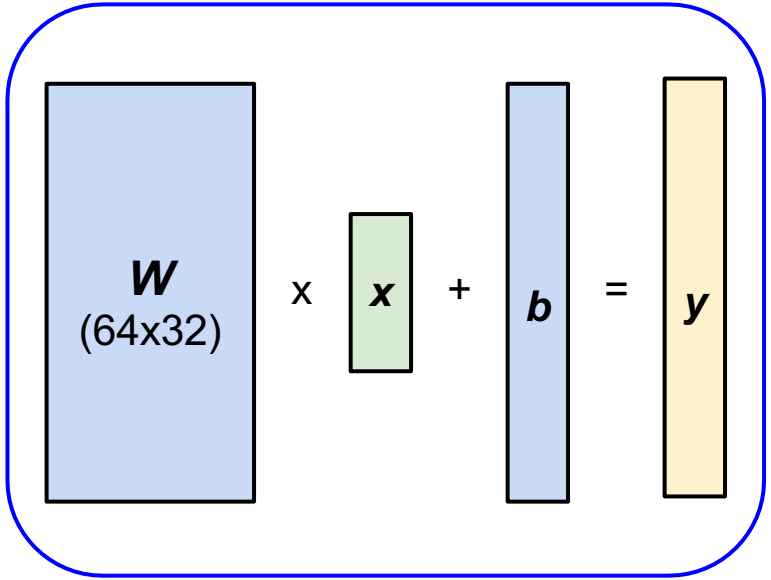
```
>>> layer = torch.nn.Linear(32, 64)
>>> layer.weight.shape
torch.Size([64, 32])
>>> layer.bias.shape
torch.Size([64])
```



torch.nn – Network Layers

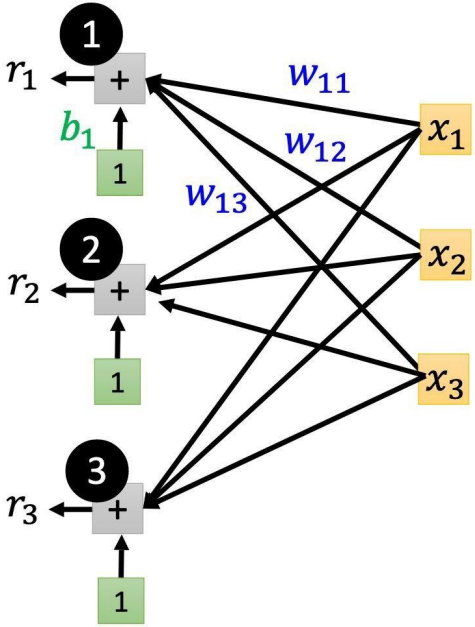
- Linear Layer (**Fully-connected** Layer)

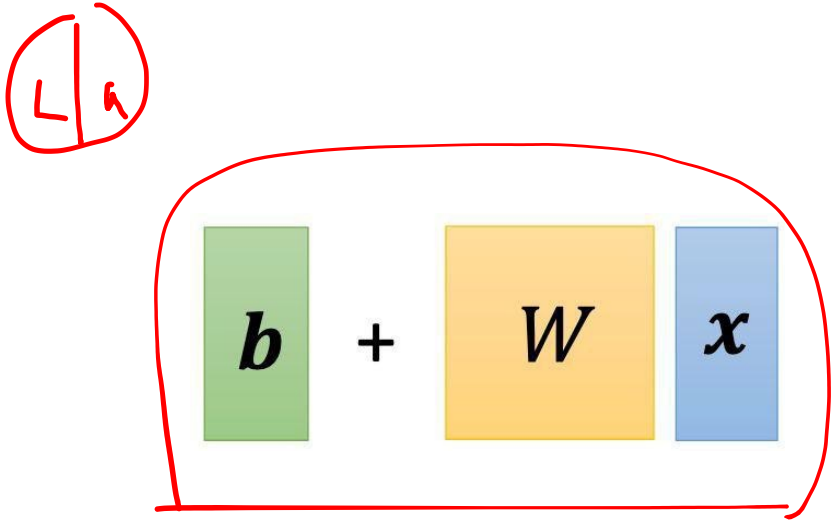




torch.nn – Network Layers

- Linear Layer (**Fully-connected** Layer)





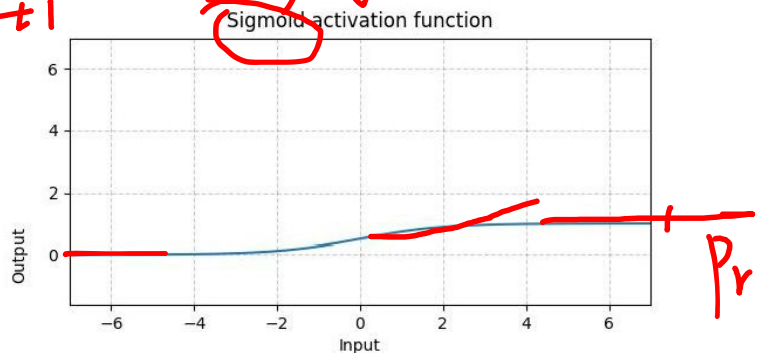
torch.nn – Non-Linear Activation Functions

- Sigmoid Activation

  `nn.Sigmoid()`

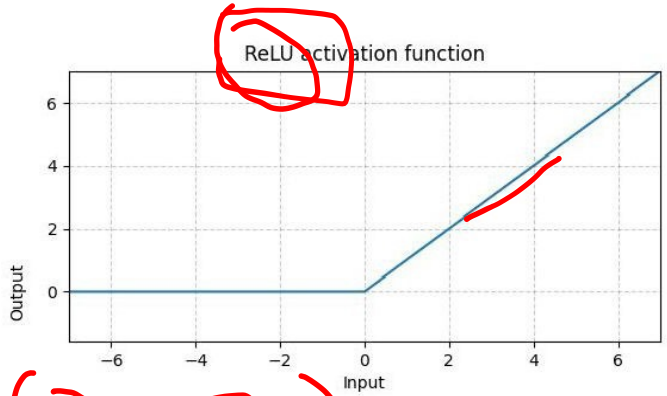
$w_{t+1} = w_t - \eta \nabla$

tanh



- ReLU Activation

  `nn.ReLU()`



max(0, x)

torch.nn – Build your own neural network

Functional

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

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

Initialize your model & define layers

Compute output of your NN

torch.nn – Build your own neural network

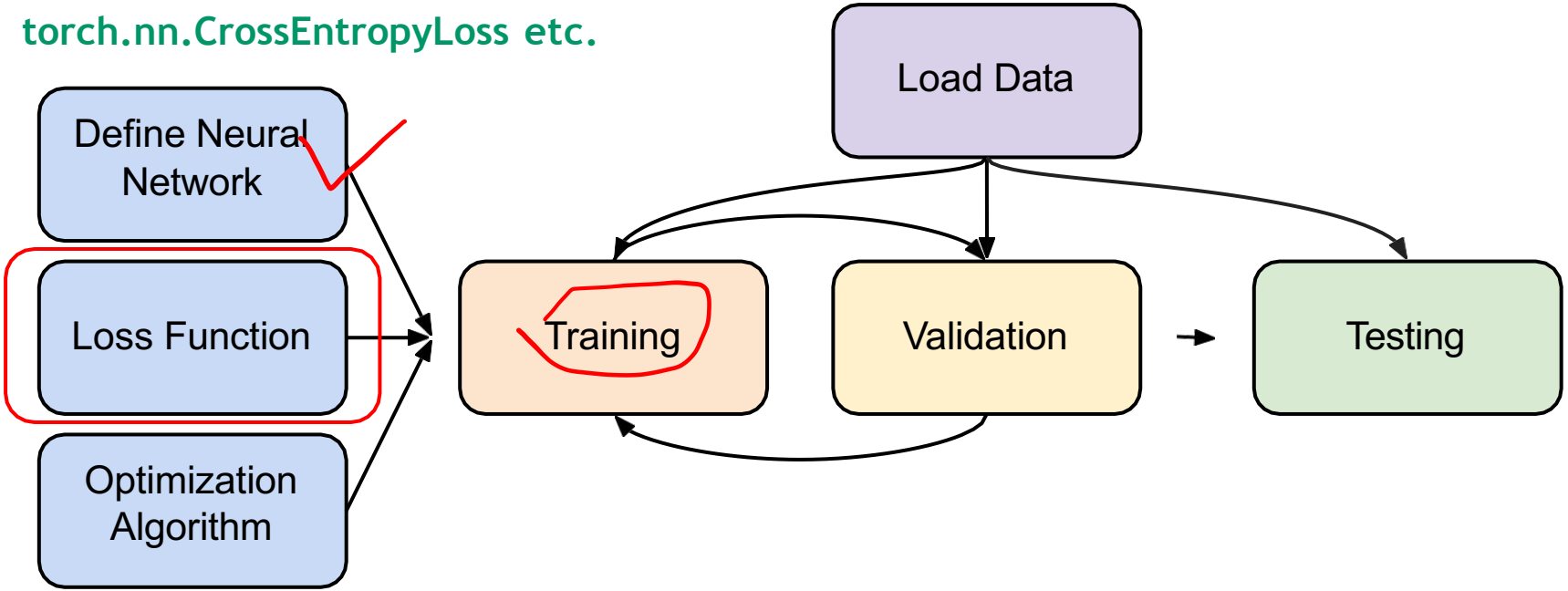
```
import torch.nn as nn

class MyModel(nn.Module):
    def init(self):
        super(MyModel,self).init ()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

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

=

```
import torch.nn as nn

class MyModel(nn.Module):
    def _init_ (self):
        super(MyModel,self).__init__()
        self.layer1 = nn.Linear(10, 32)
        self.layer2 = nn.Sigmoid()
        self.layer3 = nn.Linear(32, 1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        return out
```

Training & Testing Neural Networks in Pytorch

Step 3.
torch.nn.MSELoss
torch.nn.CrossEntropyLoss etc.



torch.nn – Loss Functions

- Mean Squared Error (for regression tasks)

  `criterion = nn.MSELoss()` $\frac{1}{B}\sum(y_i - \hat{y}_i)^2$ 3:2

- Cross Entropy (for classification tasks)

  `criterion = nn.CrossEntropyLoss()` $-\frac{1}{B}\sum_{B} y_i \log y_i$

- `loss = criterion(predicted_value, expected_value)`

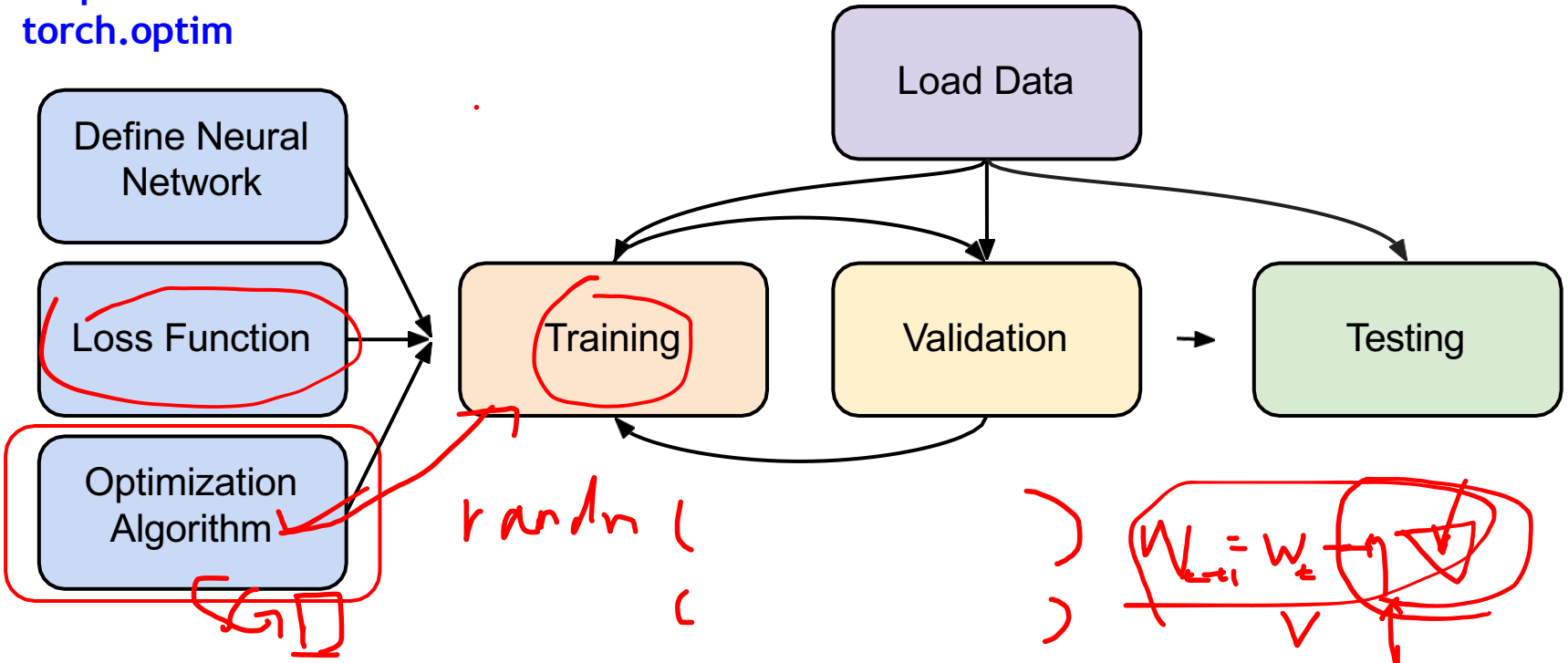
$\hat{y}$ y

$\frac{1}{B}\sum y_i \log y_i - (1-y_i)\log(1-y_i)$

$(y, \hat{y})$

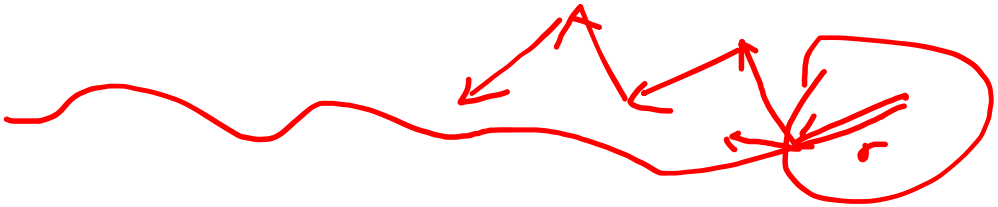
Training & Testing Neural Networks – in Pytorch

Step 4.
torch.optim



torch.optim

- Gradient-based **optimization algorithms** that adjust network parameters to reduce error.



- E.g. Stochastic Gradient Descent (SGD)

```
torch.optim.SGD(model.parameters(), lr, momentum = 0)
```

$$v = \beta v - (1-\beta)\nabla$$

0.9 0.1

torch.optim

RMSProp

Adam → β, γ

```
optimizer = torch.optim.SGD(model.parameters(), lr)
```

- For every batch of data: — Data L

1. Call `optimizer.zero_grad()` to reset gradients of model parameters.
2. Call `loss.backward()` to backpropagate gradients of prediction loss.
3. Call `optimizer.step()` to adjust model parameters.

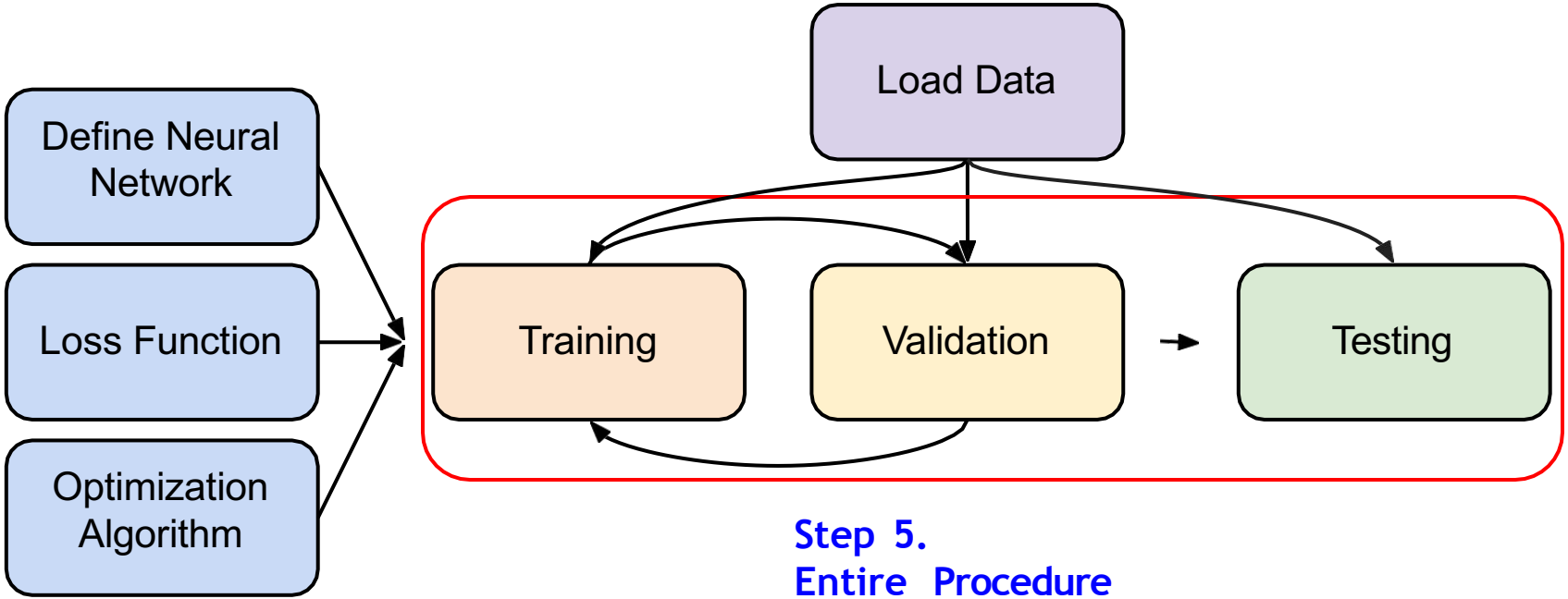
$W = W - \eta \nabla$

→ z.bwd

x.grad

See official documentation for more optimization algorithms.

Training & Testing Neural Networks – in Pytorch



Step 5.
Entire Procedure

Neural Network Training Setup

```
dataset = MyDataset(file)
tr_set = DataLoader(dataset, 16, shuffle=True)
model = MyModel().to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), 0.1)
```

read data via MyDataset

put dataset into Dataloader

construct model and move to device

set loss function

set optimizer

$\mathrm{crit}(\hat{y}, y)$

.step()

w/θ

$\hat{y} = \mathrm{model}(x)$

Neural Network Training Loop

```
for epoch in range(n_epochs):

    model.train()

    for x, y in tr_set:

      optimizer.zero_grad()

      x, y = x.to(device), y.to(device)

      pred = model(x)

      loss = criterion(pred, y)

      loss.backward()

      optimizer.step()
```

iterate n_epochs

set model to train mode

Iterate over data

set gradient to zero

move data to device (cpu/cuda)

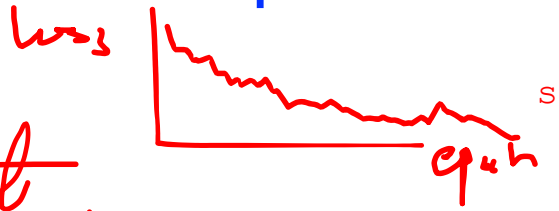
forward pass (compute output)

compute loss

compute gradient (backpropagation)

update model with optimizer

Neural Network Validation Loop




```
model.eval()                                    set model to evaluation mode
total_loss = 0
for x, y in val_set:                            iterate through the dataloader
    x, y = x.to(device), y.to(device)           move data to device (cpu/cuda)
    with torch.no_grad():                       disable gradient calculation
        pred = model(x)                         forward pass (compute output)
        loss = criterion(pred, y)               compute loss
        total_loss += loss.cpu().item() * len(x)    accumulate loss
    avg_loss = total_loss / len(val_set.dataset) compute averaged loss
```

Neural Network Testing Loop

```
model.eval()
preds = []
for x, y in test_set:
    x = x.to(device)
    with torch.no_grad():
        pred = model(x)
        preds.append(pred.cpu())
```

set model to evaluation mode

iterate through the dataloader

move data to device (cpu/cuda)

disable gradient calculation

forward pass (compute output)

collect prediction

Note: model.eval(), torch.no_grad()

- **model.eval()**
  Changes behaviour of some model layers, such as dropout and batch normalization.

  Random

- **with torch.no_grad()**
  Prevents calculation of gradients. Set during validation/testing.

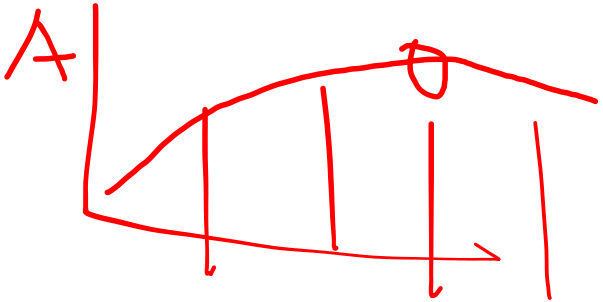
Save/Load Trained Models

- Save

```
torch.save(model.state_dict(), path)
```

- Load

```
ckpt = torch.load(path)

model.load_state_dict(ckpt)
```



More About PyTorch

- torchvision
  - computer vision
- torchaudio
  - speech/audio processing
- torchtext
  - natural language processing
- skorch
  - scikit-learn + pyTorch

References

- Official Pytorch Tutorials
- https://numpy.org/