
Machine Learning Pytorch Tutorial

TA : 蕭淇元

2023.02.20

mlta-2023-spring@googlegroups.com

Outline

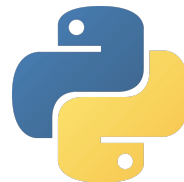
- Background: Prerequisites & What is Pytorch?
- Training & Testing Neural Networks in Pytorch
- Dataset & Dataloader
- Tensors
- torch.nn: Models, Loss Functions
- torch.optim: Optimization
- Save/load models

Prerequisites

- We assume you are already familiar with...

1. Python3

- if-else, loop, function, file IO, class, ...
- refs: [link1](#), [link2](#), [link3](#)



2. Deep Learning Basics

- Prof. Lee's 1st & 2nd lecture videos from last year
- ref: [link1](#), [link2](#)



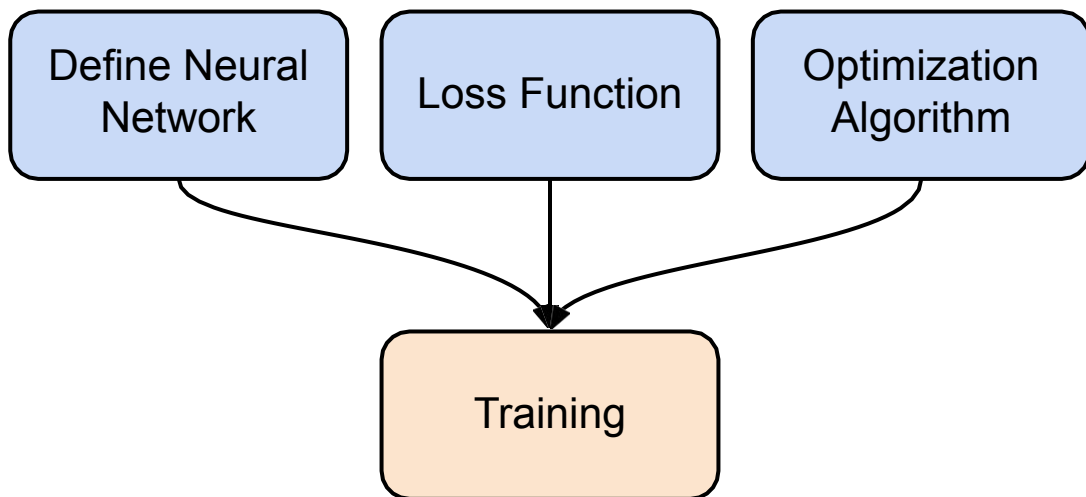
Some knowledge of **NumPy** will also be useful!

What is PyTorch?

- An **machine learning framework** in Python.
- Two main features:
 - N-dimensional **Tensor** computation (like NumPy) on **GPUs**
 - **Automatic differentiation** for training deep neural networks

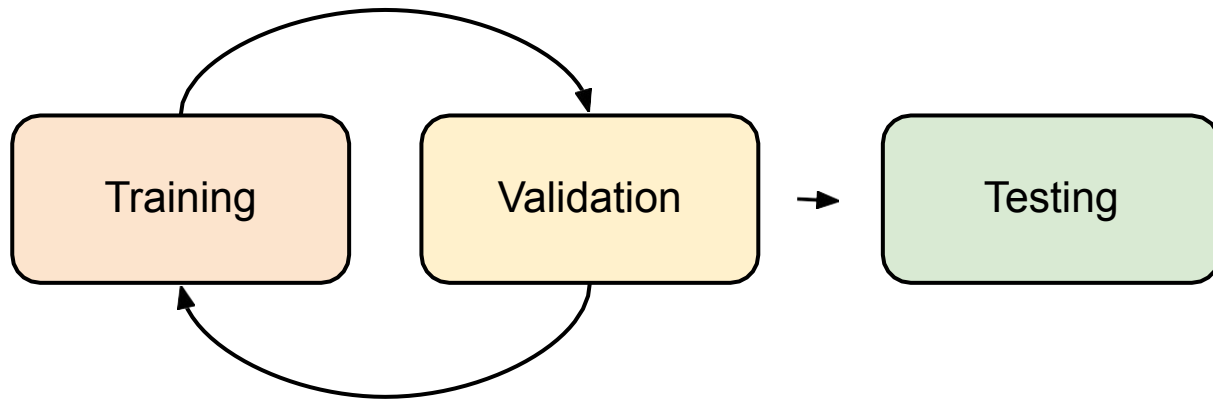


Training Neural Networks



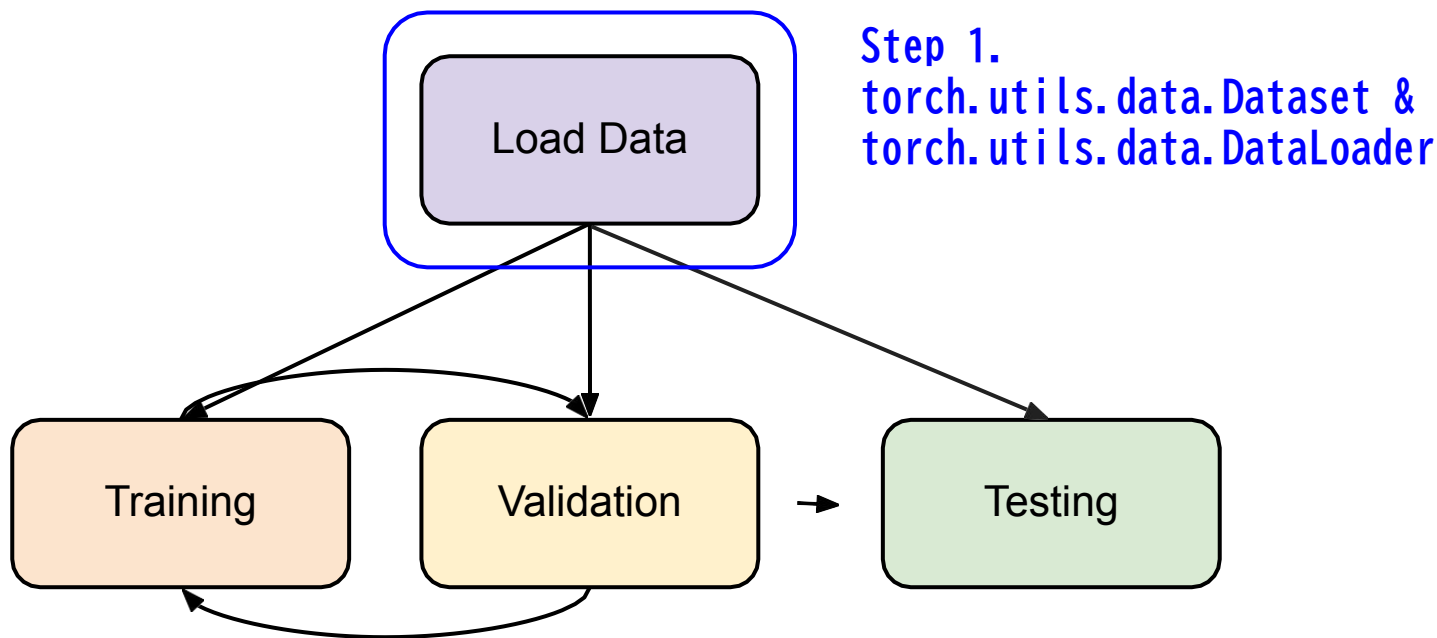
More info about the training process in [last year's lecture video](#).

Training & Testing Neural Networks



Guide for training/validation/testing can be found [here](#).

Training & Testing Neural Networks - in Pytorch



Dataset & Dataloader

- Dataset: stores data samples and expected values
- Dataloader: groups data in batches, enables multiprocessing
- `dataset = MyDataset(file)`
- `dataloader = DataLoader(dataset, batch_size, shuffle=True)`



Training: True
Testing: False

More info about batches and shuffling [here](#).

Dataset & Dataloader

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

```
class MyDataset(Dataset):
```

```
    def init (self, file):
```

```
        self.data = ...
```



Read data & preprocess

```
    def __getitem__(self, index):
```

```
        return self.data[index]
```



Returns one sample at a time

```
    def __len__(self):
```

```
        return len(self.data)
```

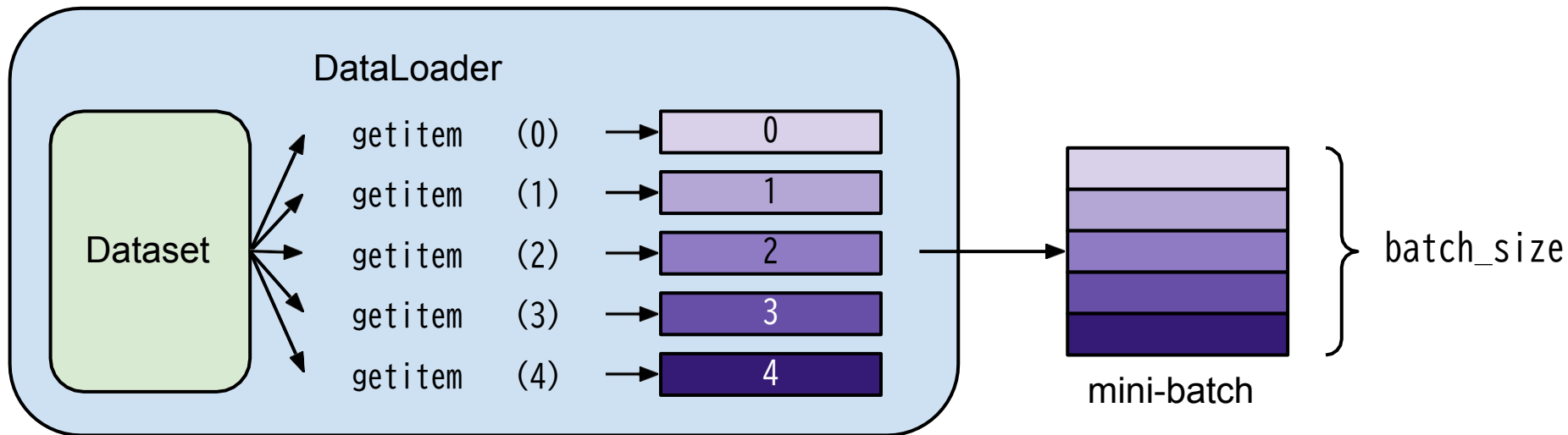


Returns the size of the dataset

Dataset & Dataloader

```
dataset = MyDataset(file)
```

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

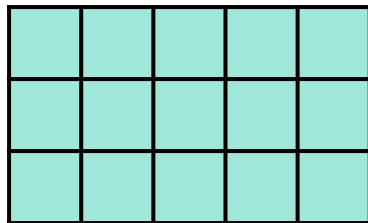


Tensors

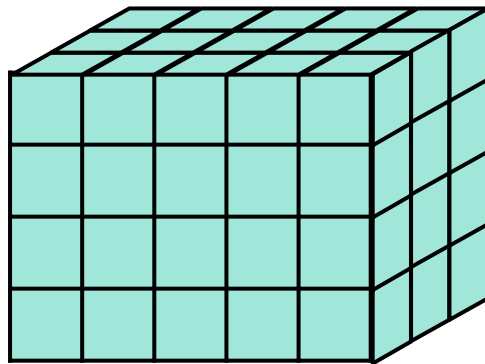
- High-dimensional matrices (arrays)



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`

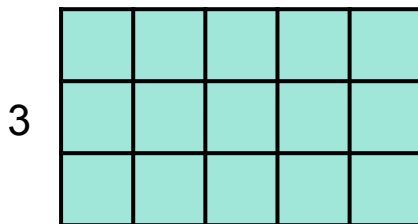


5

(5,)



dim 0



3

5

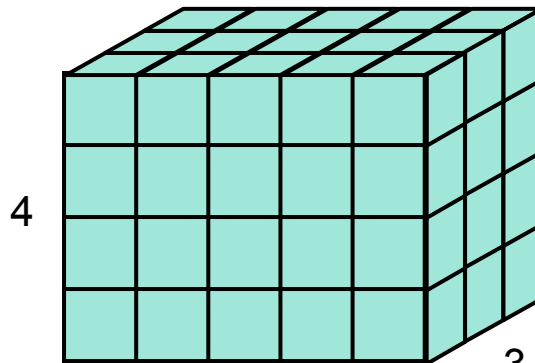
(3, 5)



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.ndarray)

```
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

$$z = x + y$$

- Summation

$$y = x.\text{sum}()$$

- Subtraction

$$z = x - y$$

- Mean

$$y = x.\text{mean}()$$

- Power

$$y = x.\text{pow}(2)$$

Tensors – Common Operations

- **Transpose:** transpose two specified dimensions

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

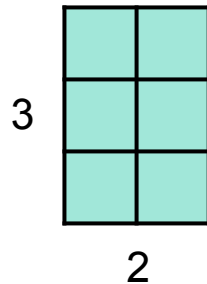
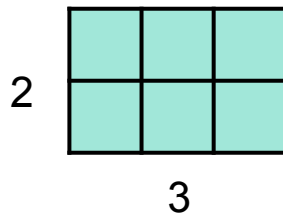
```
>>> x.shape
```

```
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
```

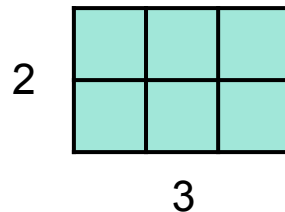
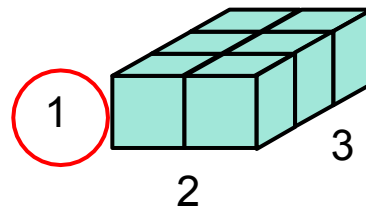
```
torch.Size([1, 2, 3])
```

```
>>> 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])
```

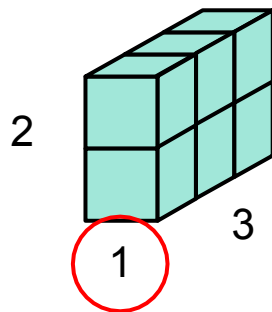
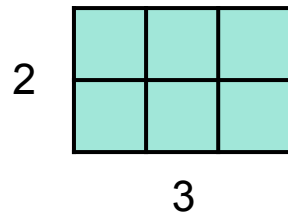
```
>>> x.shape
```

```
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])
```

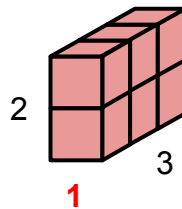
```
>>> w = torch.cat([x, y, z], dim=1)
```

```
>>> w.shape
```

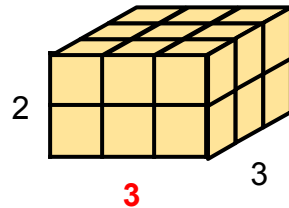
```
torch.Size([2, 6, 3])
```

more operators: <https://pytorch.org/docs/stable/tensors.html>

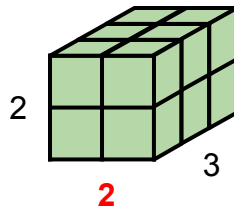
x



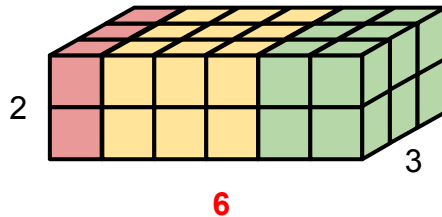
y



z



w



Tensors – Data Type

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

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

see [official documentation](#) 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.

ref: <https://github.com/wkentaro/pytorch-for-numpy-users>

Tensors – PyTorch v.s. NumPy

- Many functions have the same names as well

PyTorch	NumPy
<code>x.reshape / x.view</code>	<code>x.reshape</code>
<code>x.squeeze()</code>	<code>x.squeeze()</code>
<code>x.unsqueeze(1)</code>	<code>np.expand_dims(x, 1)</code>

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

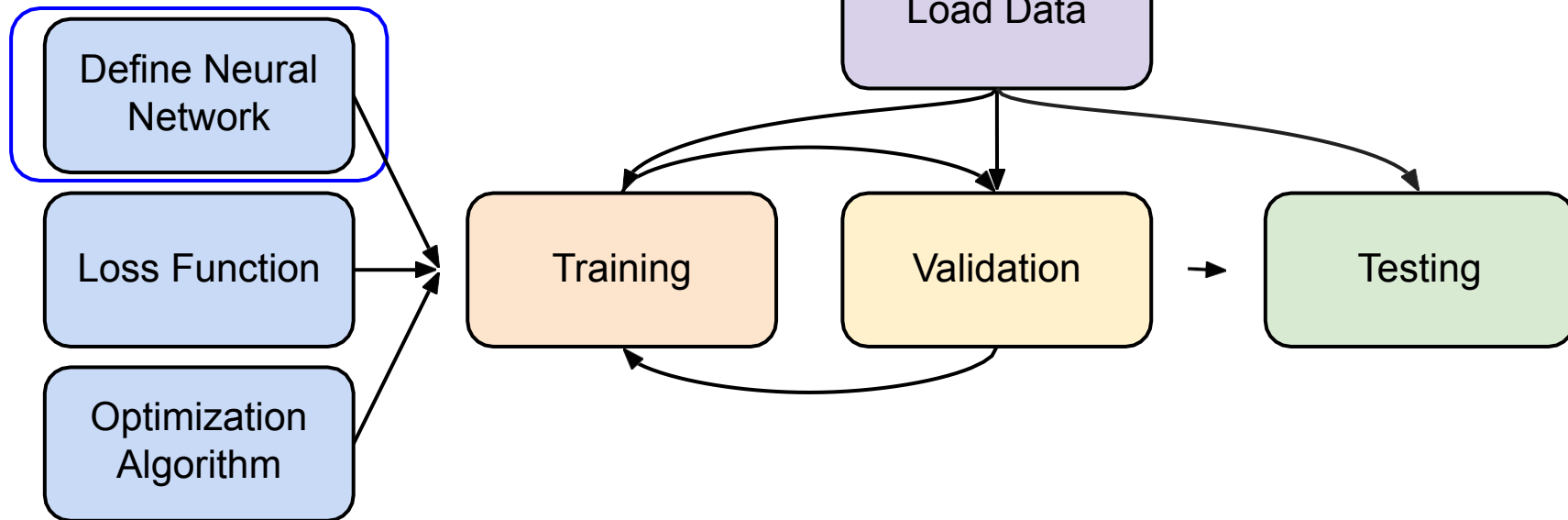
3
$$\frac{\partial z}{\partial x_{i,j}} = 2x_{i,j}$$

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

See [here](#) to learn about gradient calculation.

Training & Testing Neural Networks – in Pytorch

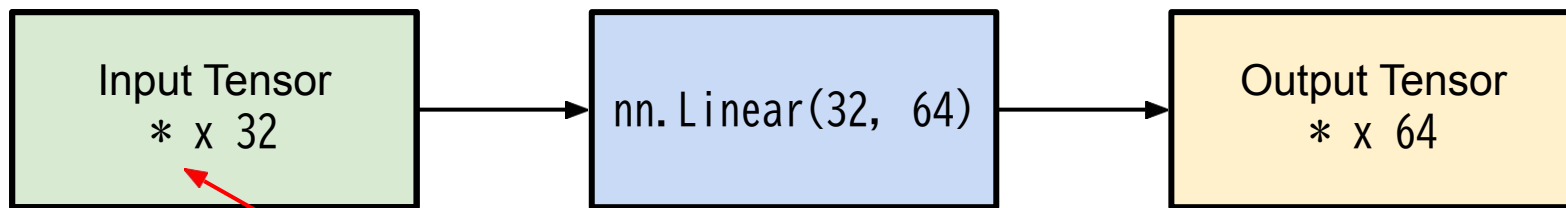
Step 2.
`torch.nn.Module`



torch.nn – Network Layers

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

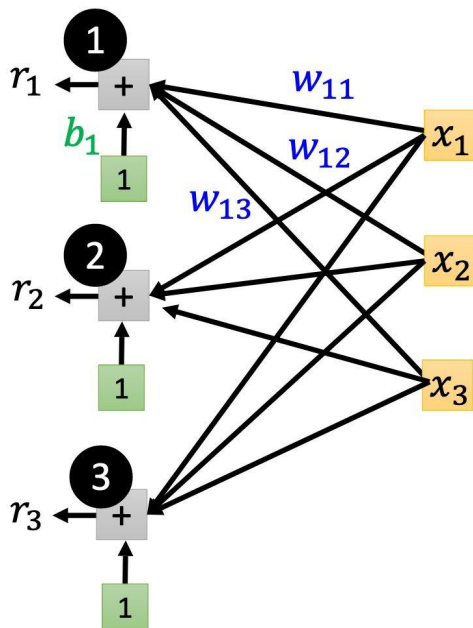
```
nn.Linear(in_features, out_features)
```



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)

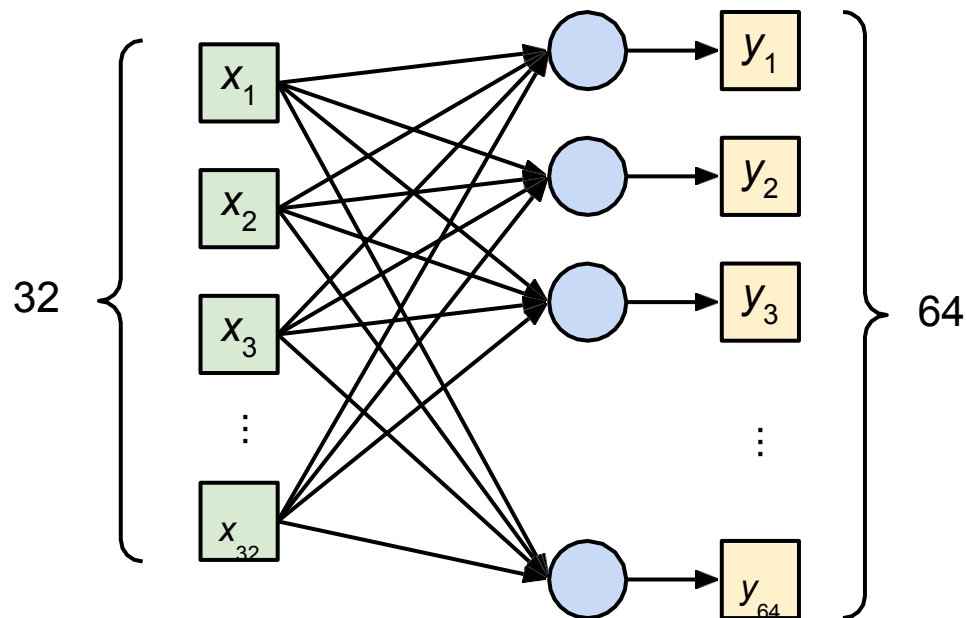


$$b + Wx$$

ref: [last year's lecture video](#)

torch.nn – Neural Network Layers

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



$$\begin{matrix} \text{64x32} \\ \mathbf{W} \end{matrix} \times \begin{matrix} \mathbf{x} \end{matrix} + \begin{matrix} \mathbf{b} \end{matrix} = \begin{matrix} \mathbf{y} \end{matrix}$$

torch.nn – Network Parameters

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

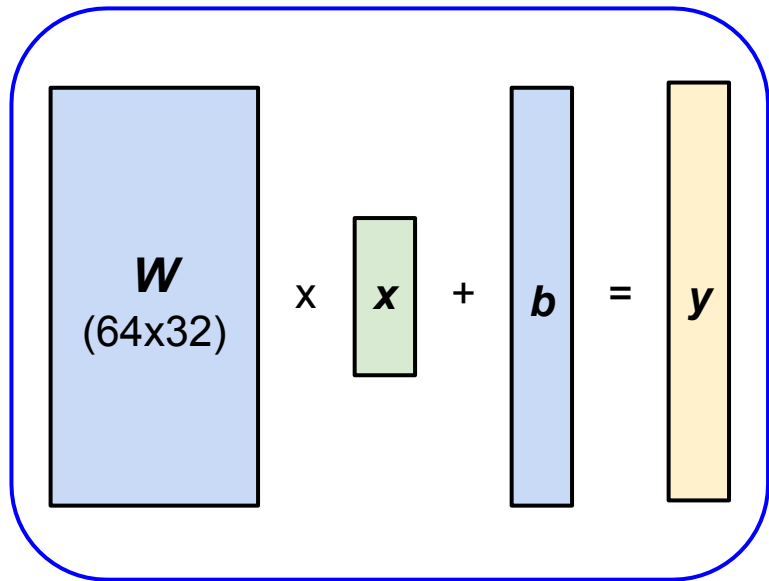
```
>>> layer = torch.nn.Linear(32, 64)
```

```
>>> layer.weight.shape
```

```
torch.Size([64, 32])
```

```
>>> layer.bias.shape
```

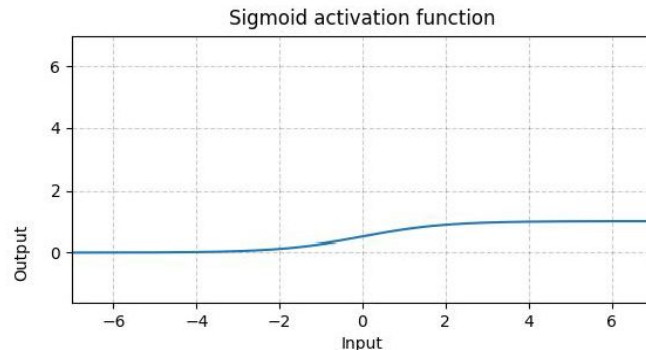
```
torch.Size([64])
```



torch.nn – Non-Linear Activation Functions

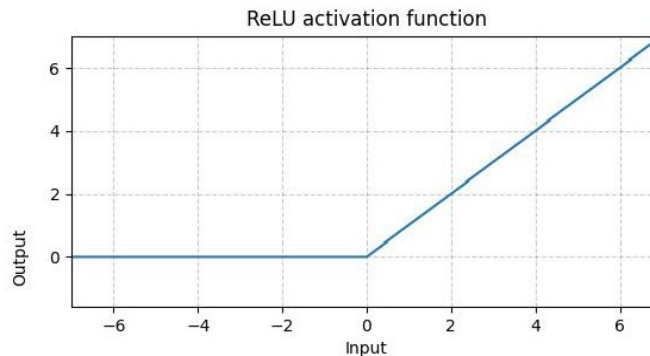
- Sigmoid Activation

`nn.Sigmoid()`



- ReLU Activation

`nn.ReLU()`



See [here](#) to learn about why we need activation functions.

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)
```

Initialize your model & define layers

Forward 是干什么的？

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)
```

=

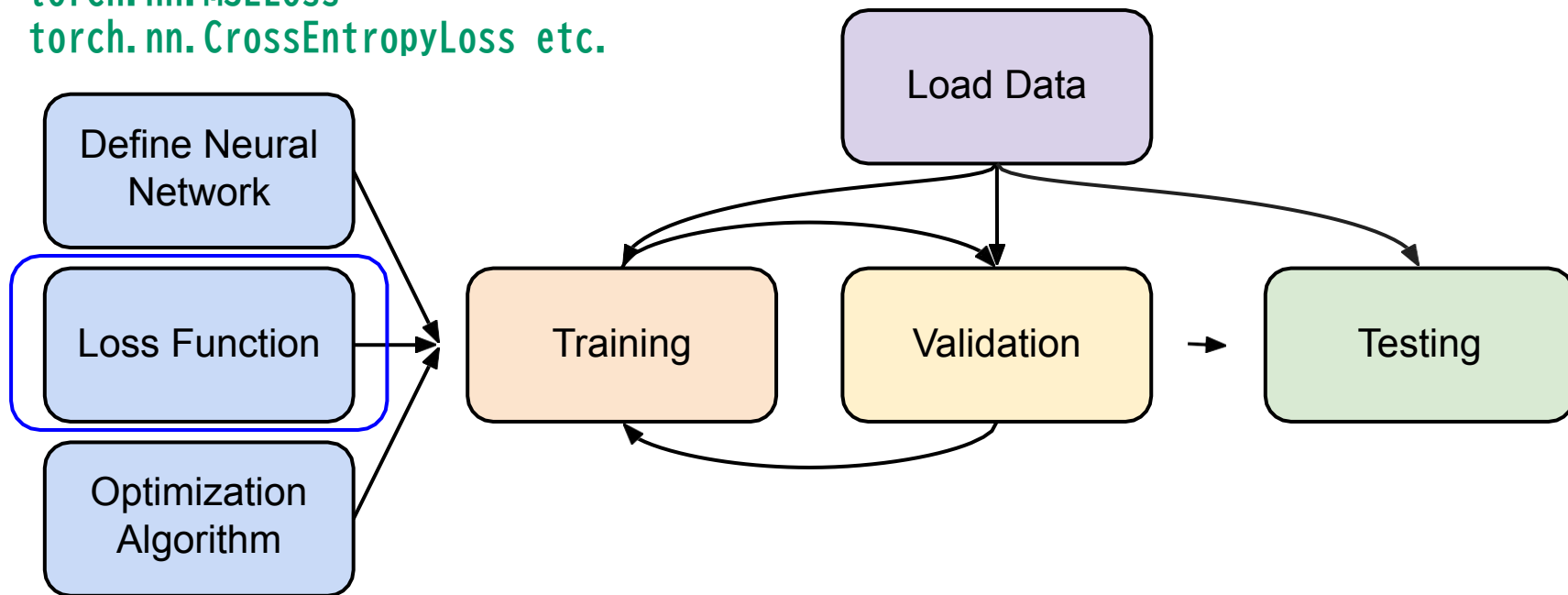
```
class MyModel(nn.Module):  
    def __init__(self):  
        super(MyModel, self).__init__()  
        self.layer1 = nn.Linear(10, 32)  
        self.layer2 = nn.Sigmoid()  
        self.layer3 = nn.Linear(32, 1)  
  
    def forward(self, x):  
        out = self.layer1(x)  
        out = self.layer2(out)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

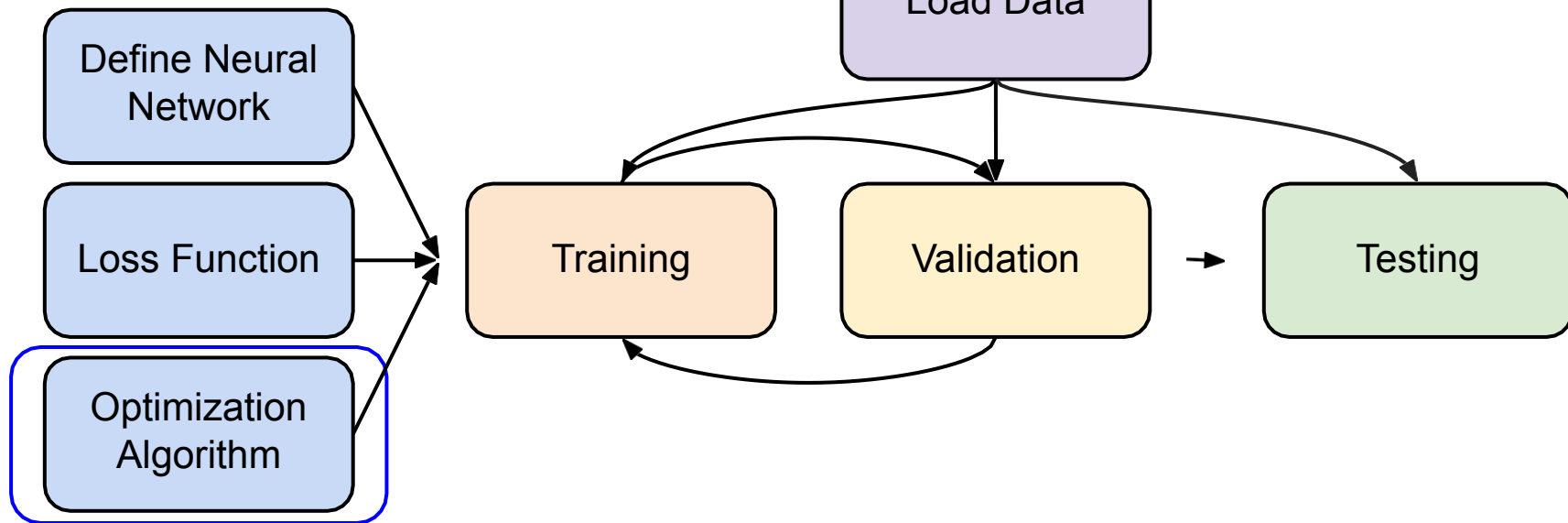
- Cross Entropy (for classification tasks)

```
criterion = nn.CrossEntropyLoss()
```

- `loss = criterion(model_output, expected_value)`

Training & Testing Neural Networks – in Pytorch

Step 4.
`torch.optim`



torch.optim

- Gradient-based **optimization algorithms** that adjust network parameters to reduce error. (See [Adaptive Learning Rate](#) lecture video)
- E.g. Stochastic Gradient Descent (SGD)

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

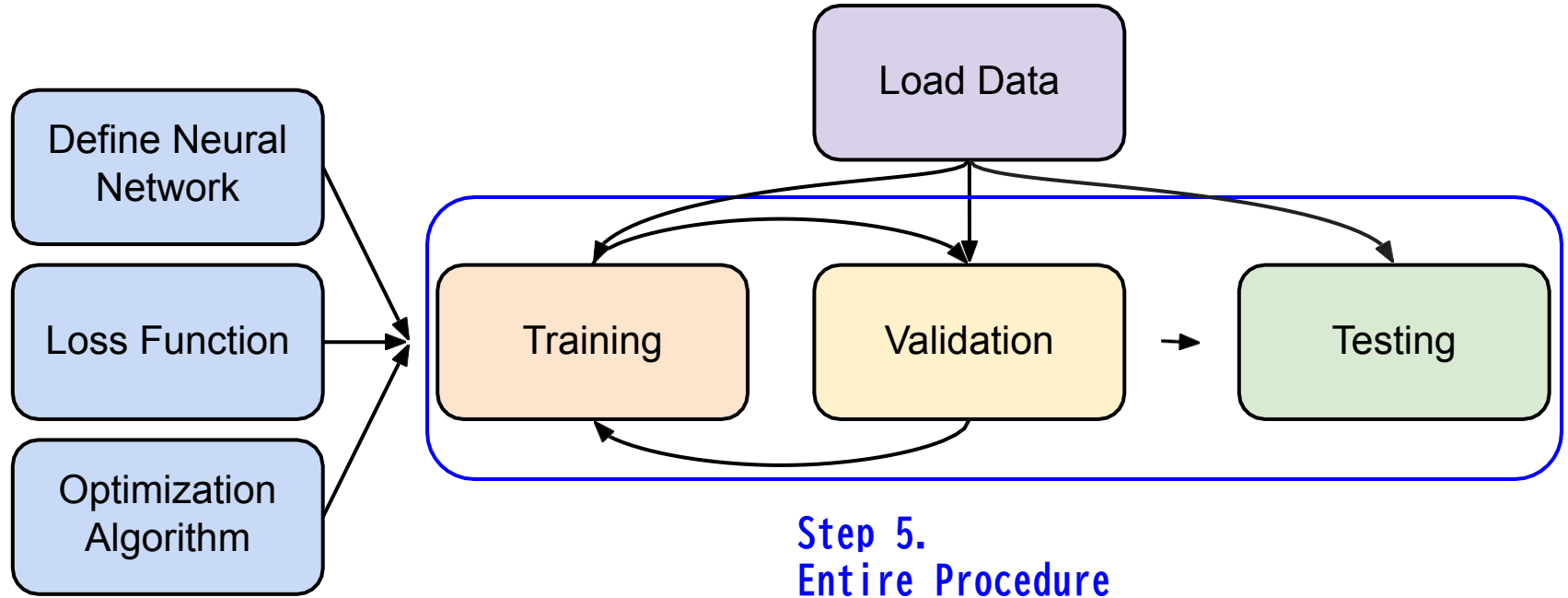
torch.optim

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

- For every batch of data:
 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.

See [official documentation](#) for more optimization algorithms.

Training & Testing Neural Networks – in Pytorch



Neural Network Training Setup

```
dataset = MyDataset(file)
```

read data via MyDataset

```
tr_set = DataLoader(dataset, 16, shuffle=True)
```

put dataset into Dataloader

```
model = MyModel().to(device)
```

construct model and move to device (cpu/cuda)

```
criterion = nn.MSELoss()
```

set loss function

```
optimizer = torch.optim.SGD(model.parameters(), 0.1)
```

set optimizer

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 through the dataloader

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 dv_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(dv_set.dataset)
```

compute averaged loss

Neural Network Testing Loop

```
model.eval()
```

set model to evaluation mode

```
preds = []
```

```
for x in tt_set:
```

iterate through the dataloader

```
    x = x.to(device)
```

move data to device (cpu/cuda)

```
    with torch.no_grad():
```

disable gradient calculation

```
        pred = model(x)
```

forward pass (compute output)

```
        preds.append(pred.cpu())
```

collect prediction

Notice - `model.eval()`, `torch.no_grad()`

- `model.eval()`
Changes behaviour of some model layers, such as dropout and batch normalization.
- `with torch.no_grad()`
Prevents calculations from being added into gradient computation graph.
Usually used to prevent accidental training on validation/testing data.

Save/Load Trained Models

- Save

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

- Load

```
ckpt = torch.load(path)  
model.load_state_dict(ckpt)
```

More About PyTorch

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

More About PyTorch

- Useful github repositories using PyTorch
 - [Huggingface Transformers](#) (transformer models: BERT, GPT, ...)
 - [Fairseq](#) (sequence modeling for NLP & speech)
 - [ESPnet](#) (speech recognition, translation, synthesis, ...)
 - Most implementations of recent deep learning papers
 - ...

References

- [Machine Learning 2022 Spring Pytorch Tutorial](#)
- [Official Pytorch Tutorials](#)
- <https://numpy.org/>

Any questions?