Deep Learning with PyTorch

NSF Workshop: Deep Learning on GPU

Tony T. Luo

July 2023

Agenda for today (July 24)

1. PyTorch
2. MLP
3. CNN

Lunch break (12:30-1:30PM)

4. Transfer Learning
5. Save/Load models
6. PINN

PyTorch Tutorial

Credit: Q. Xiao

Outline

- Background: What is Pytorch?
- Installation
- Training & Testing Neural Networks in Pytorch
- Dataset & Dataloader
- Tensors
- 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** computation (like NumPy) on **GPUs**
  - **Automatic differentiation** for training deep neural networks

PyTorch Installation

Python / Conda

CUDA (if applicable): https://developer.nvidia.com/cuda-downloads

Create virtual environment (if needed) and activate it

conda create -n pytorch python=3.11

conda activate pytorch

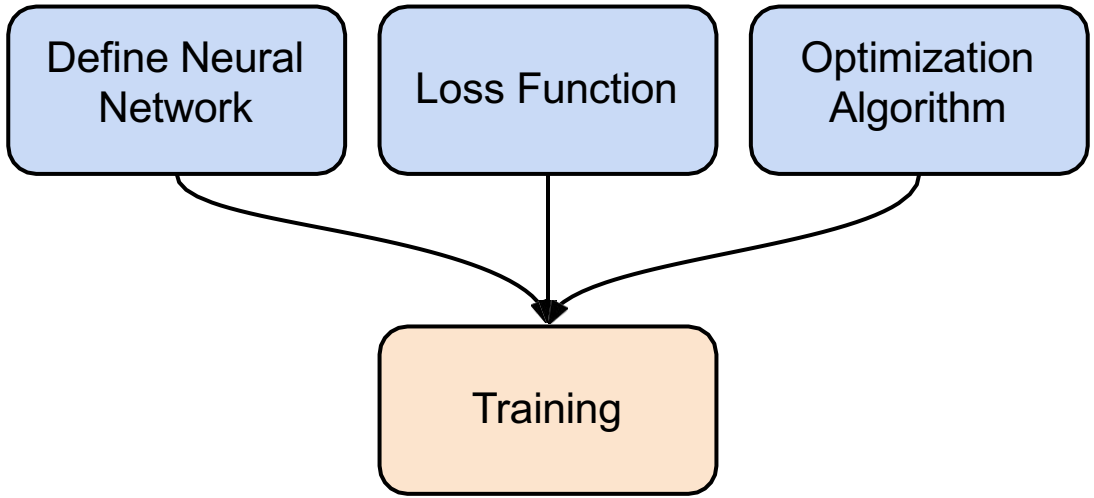
PyTorch

(pytorch) $ *<installation command here>*

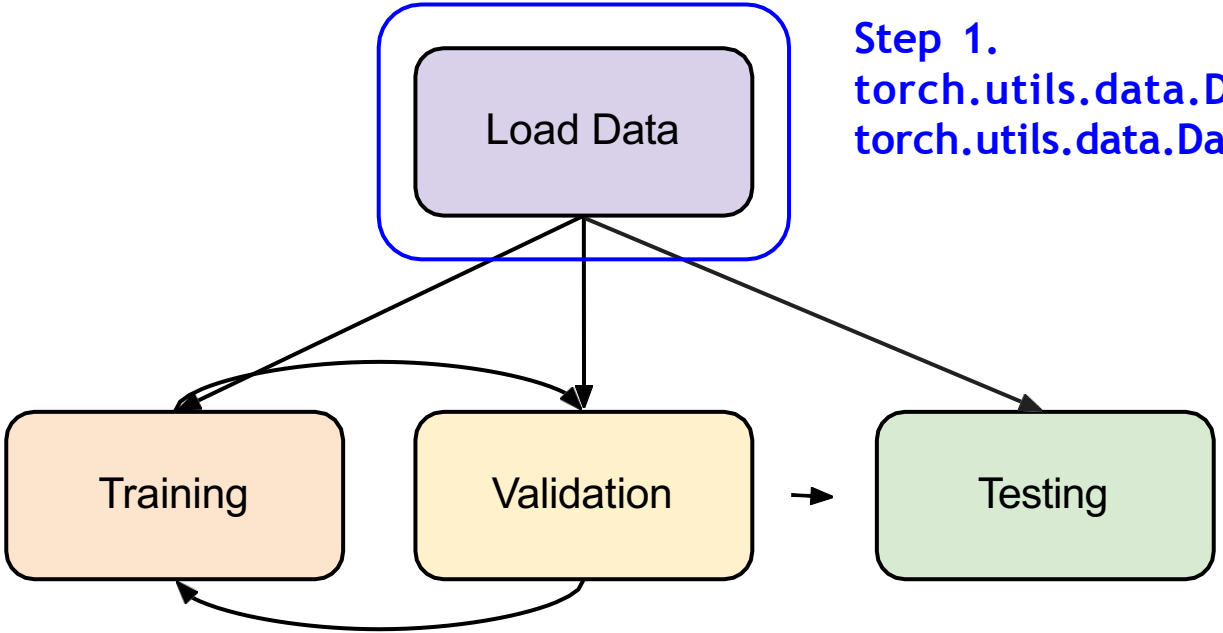
Verification

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

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

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

Training Neural Networks



Training & Testing Neural Networks - in Pytorch



Step 1.
torch.utils.data.Dataset &
torch.utils.data.DataLoader

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

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)
```

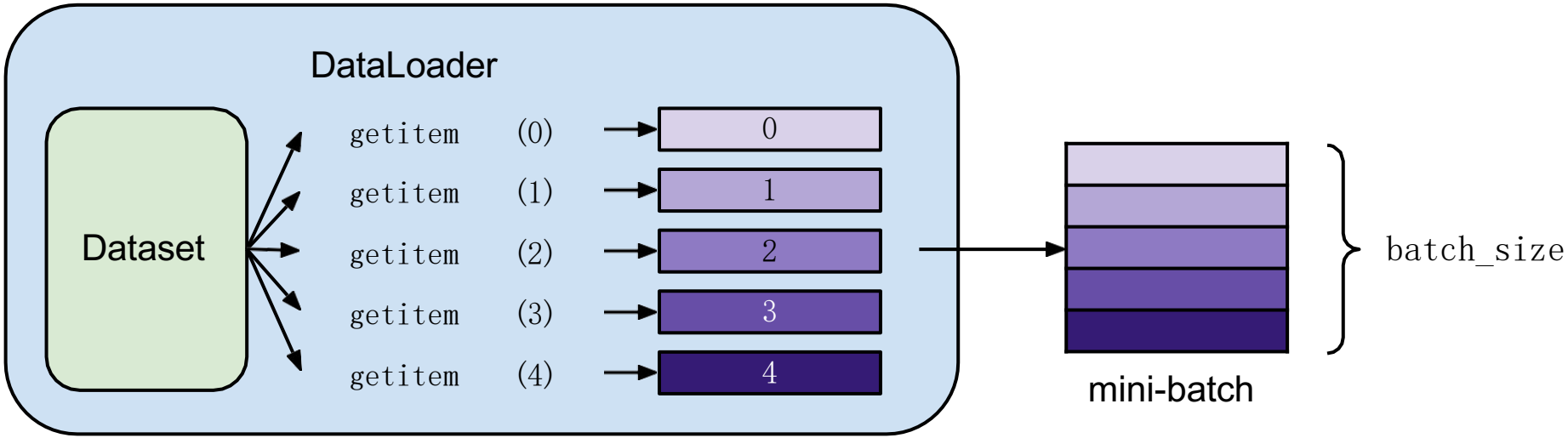
Read data & preprocess

Returns one sample at a time

Returns the size of the dataset

Dataset & Dataloader

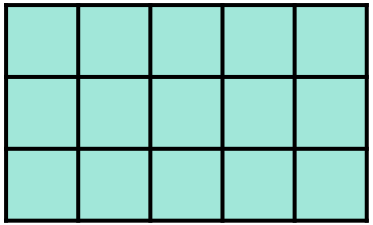
```
dataset = MyDataset(file)

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



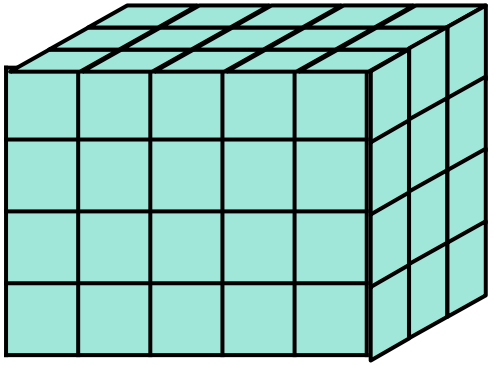
Tensors

- 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

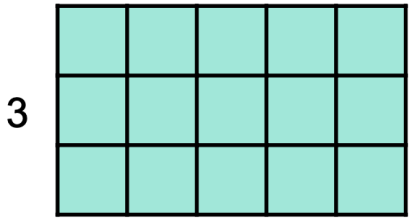
Tensors – Shape of Tensors

- Check with .shape

5

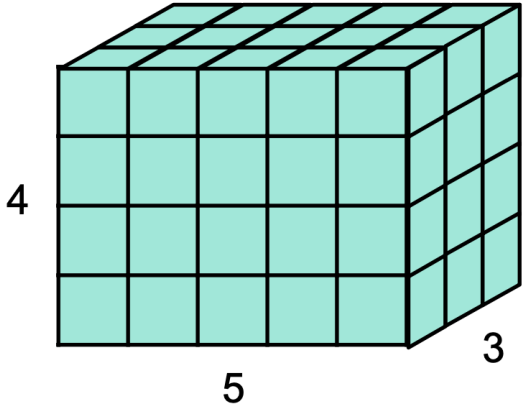
(5,)

dim 0



(3, 5)

dim 0 dim 1



(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]])
```

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

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

- Tensor of constant zeros & ones

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

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

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

shape

```
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()
```

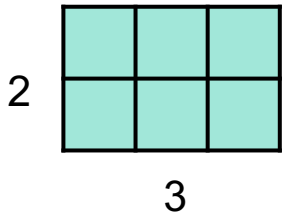
- Mean

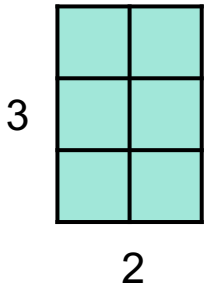
```
y = x.mean()
```

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])
```

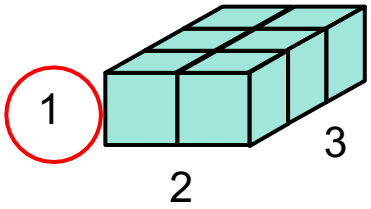


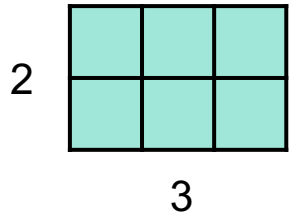


Tensors – Common Operations

- **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])
```

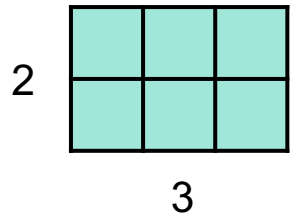


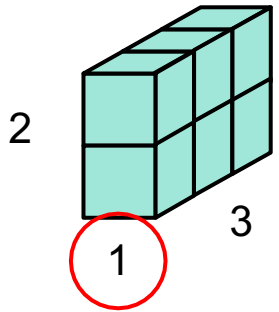


Tensors – Common Operations

- **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])
```

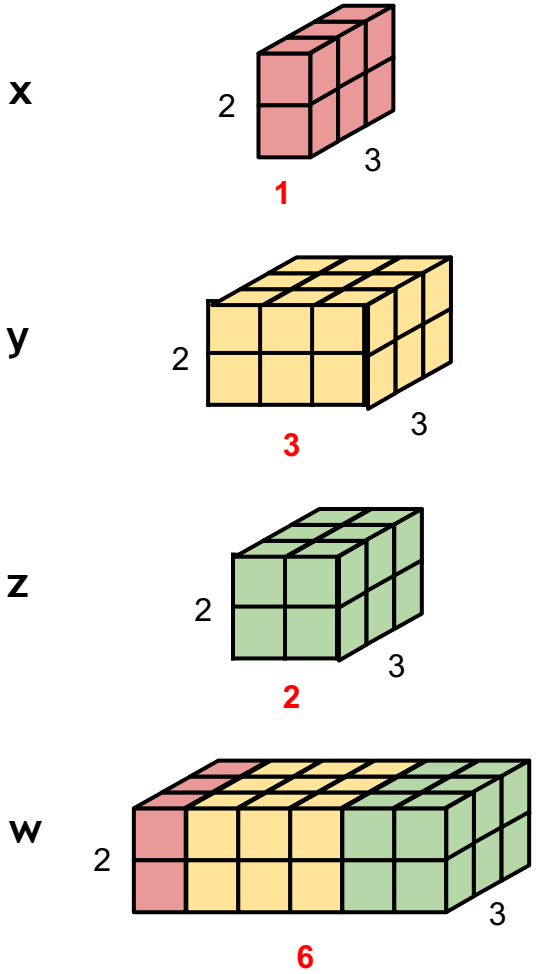




Tensors – Common Operations

- **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])
```



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 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.

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

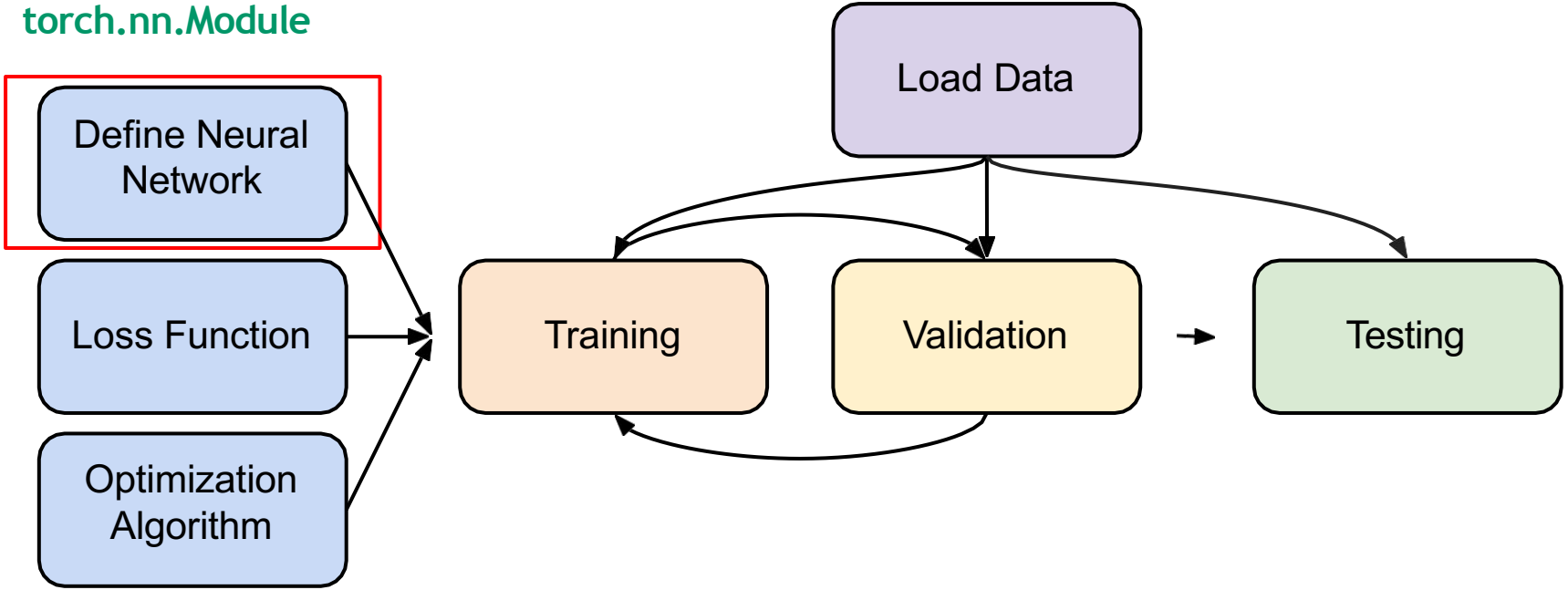
2
$$z = \sum_i \sum_j x_{i,j}^2$$

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

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

Training & Testing Neural Networks – in Pytorch

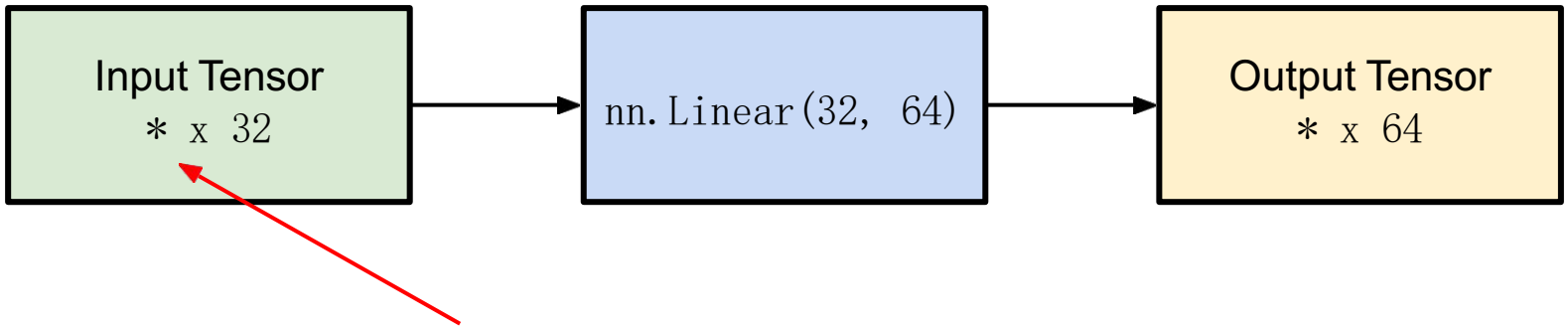
Step 2.
torch.nn.Module



torch.nn – Network Layers

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

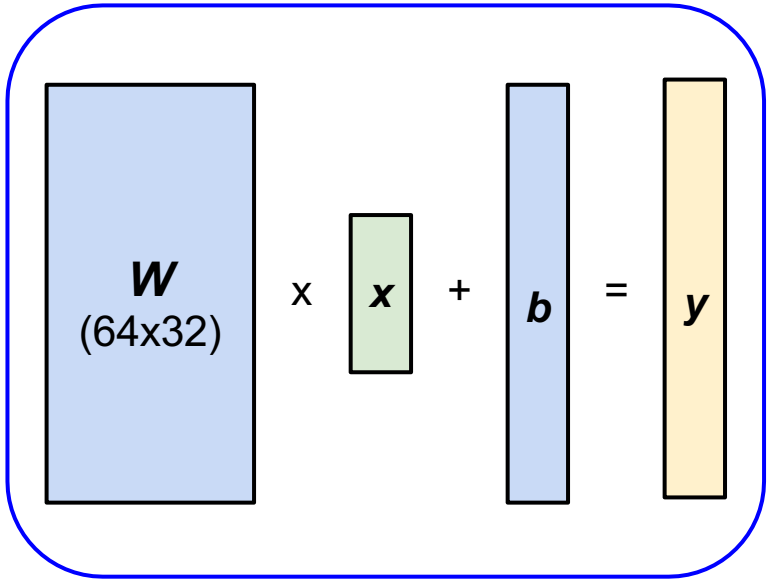
  `nn.Linear(in_features, out_features)`



torch.nn – Network Layers

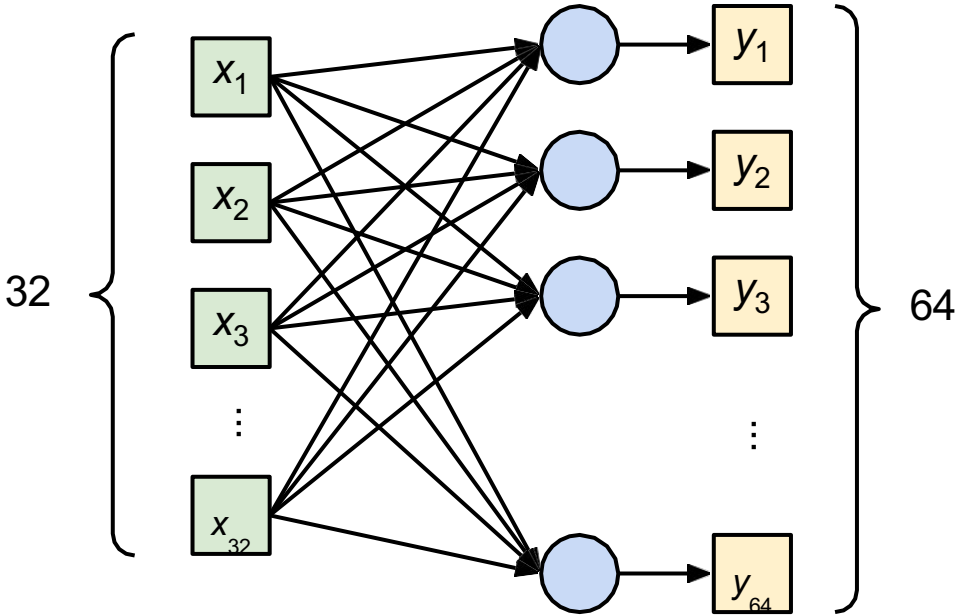
- Linear Layer (**Fully-connected** Layer)

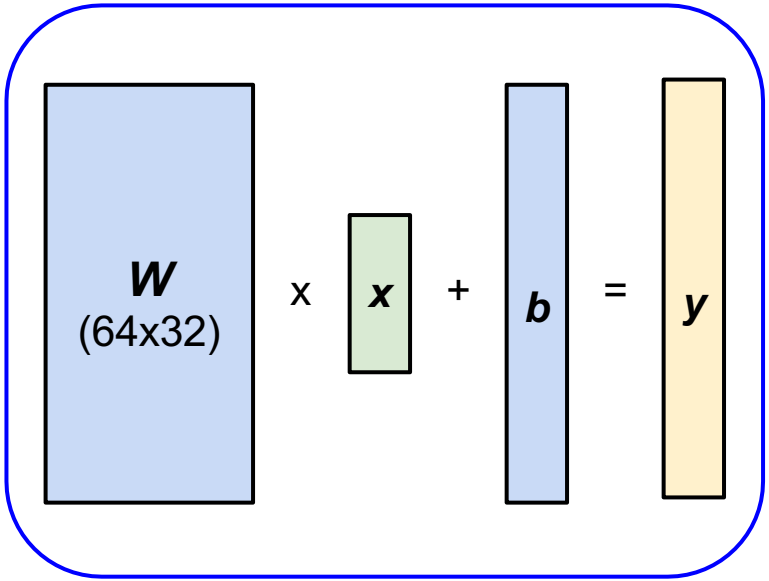
```
>>> layer = torch.nn.Linear(32, 64)
>>> layer.weight.shape
torch.Size([64, 32])
>>> layer.bias.shape
torch.Size([64])
```



torch.nn – Network Layers

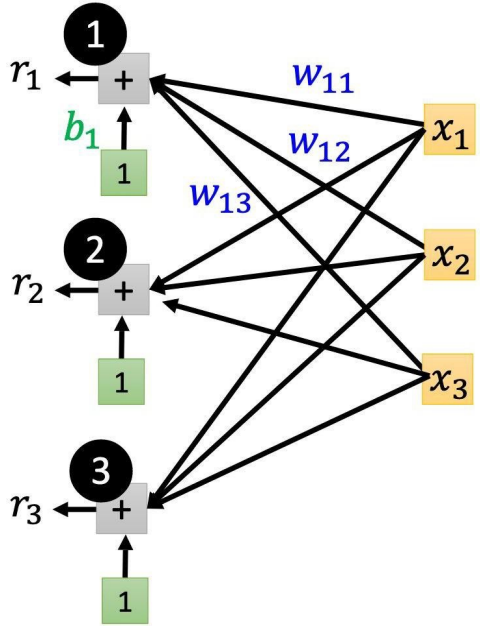
- Linear Layer (**Fully-connected** Layer)





torch.nn – Network Layers

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

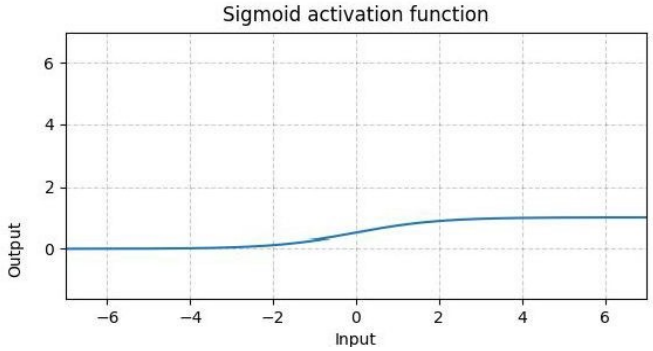


$$\boldsymbol{b} + \boldsymbol{W}\,\boldsymbol{x}$$

torch.nn – Non-Linear Activation Functions

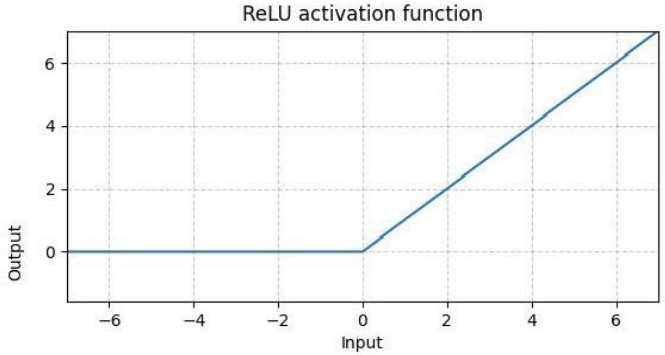
- Sigmoid Activation

```
nn.Sigmoid()
```



- ReLU Activation

```
nn.ReLU()
```



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

Compute output of your NN

torch.nn – Build your own neural network

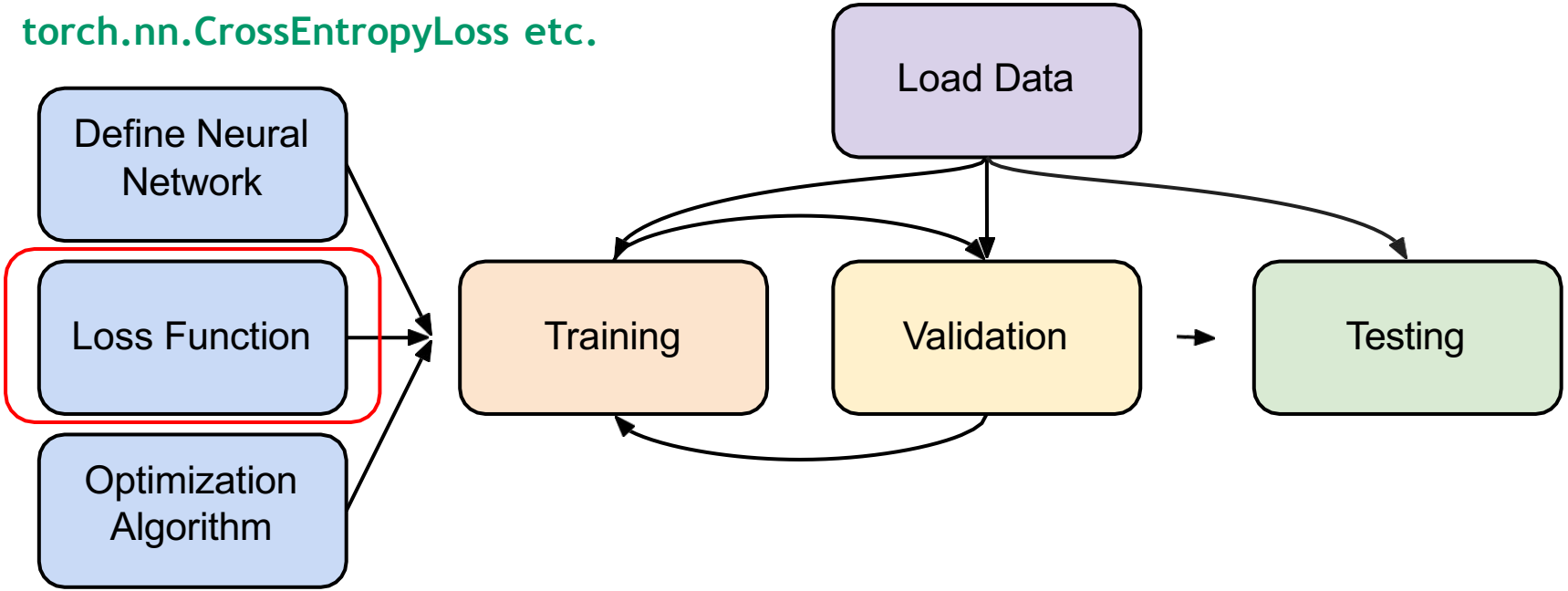
```
import torch.nn as nn

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()
```

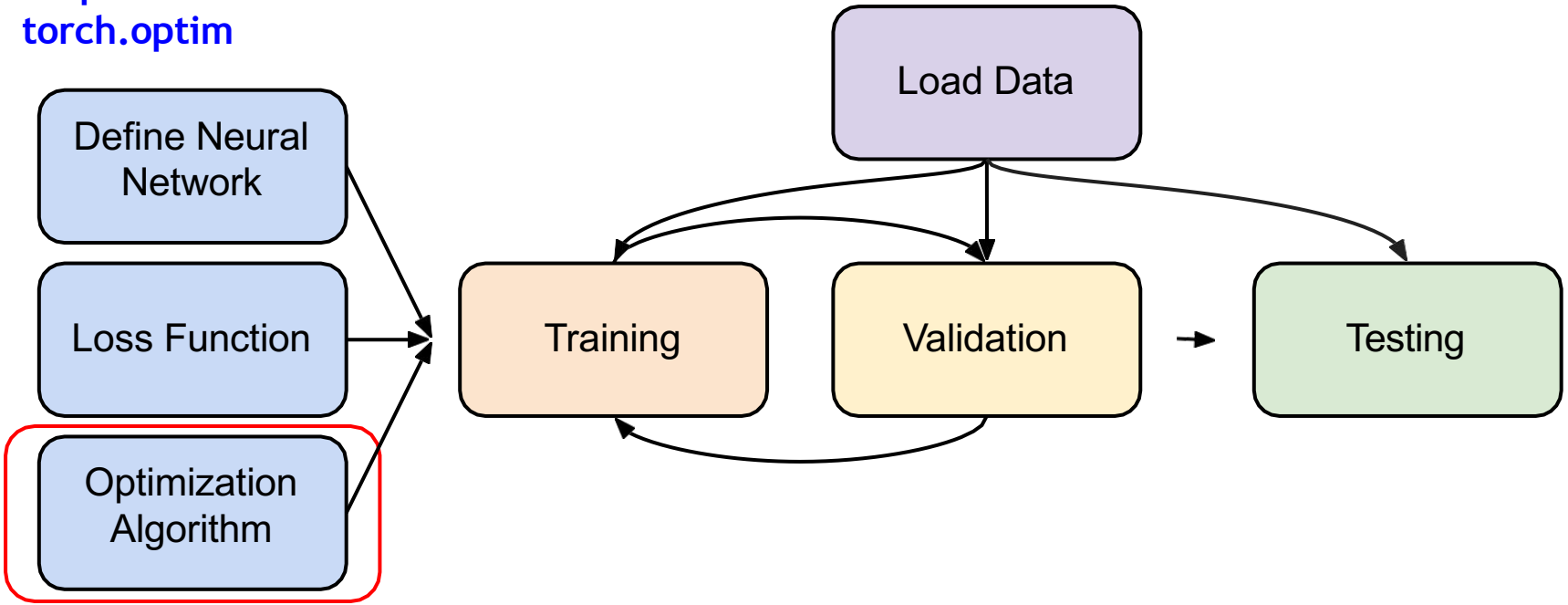
- Cross Entropy (for classification tasks)

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

- `loss = criterion(predicted_value, 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.

- `E.g. Stochastic Gradient Descent (SGD)`

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

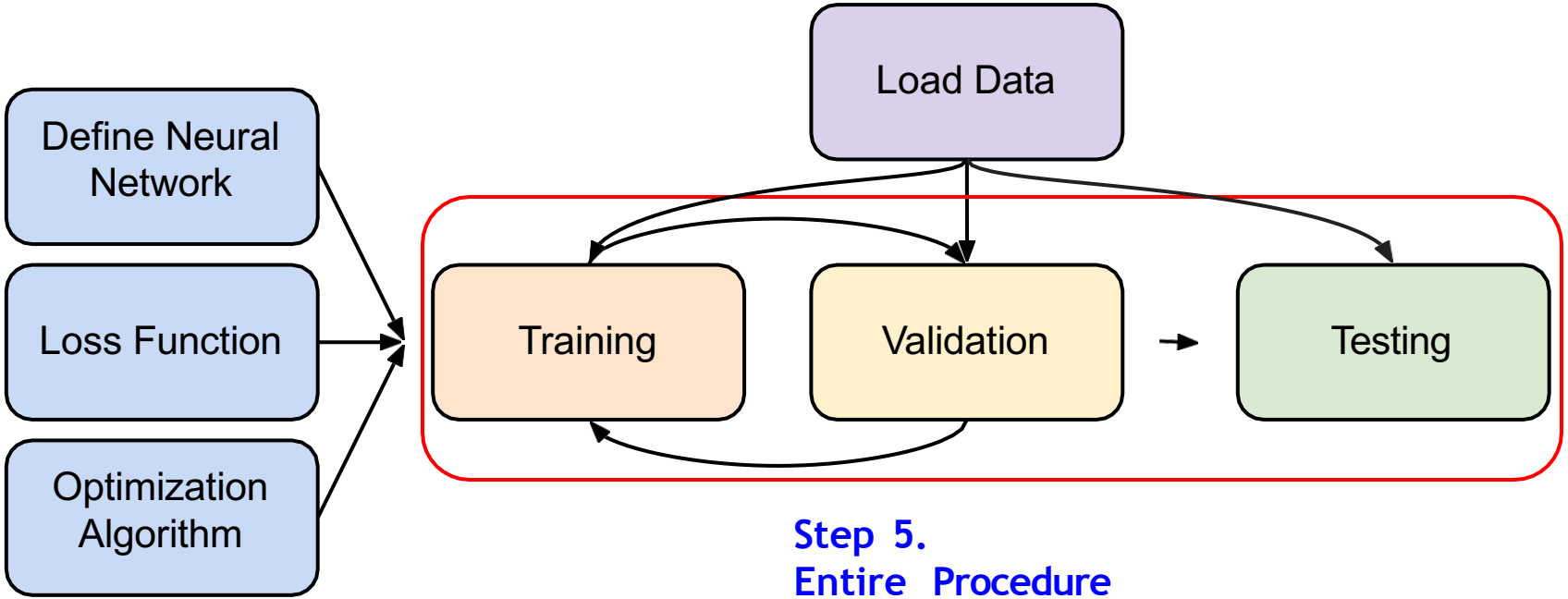
torch.optim

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

- 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



Step 5.
Entire Procedure

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

```
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 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 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.

- **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/

PyTorch Tutorial (2)

Common Errors

Credit: W. Shi

Common Errors – Tensor on Different Devices

```
model = torch.nn.Linear(5,1).to("cuda:0")
x = torch.randn(5)
y = model(x)
```

Error: Tensor … is on CPU, but expected to be on GPU

=> send the tensor to GPU

```
x = torch.randn(5).to("cuda:0")
y = model(x)
print(y.shape)
```

Common Errors - Mismatched Dimensions

```
x = torch.randn(4,5)
y = torch.randn(5,4)
z = x + y
```

The size of tensor x (5) must match the size of tensor y (4) at non-singleton dimension 1

the shape of a tensor is incorrect

=> use **`transpose, squeeze, unsqueeze`** to align the dimensions

```
y = y.transpose(0,1)
z = x + y
print(z.shape)
```

Common Errors - Cuda Out of Memory

```
import torch
import torchvision.models as models
resnet18 = models.resnet18().to( "cuda:0" ) # Neural Networks for Image
data = torch.randn( 512,3,244,244) # Create fake data (512
out = resnet18(data.to( "cuda:0" )) # Use Data as Input and Feed to
print(out.shape)
```

CUDA out of memory. Tried to allocate 350.00 MiB (GPU 0; 14.76 GiB total capacity; 11.94 GiB already allocated; 123.75 MiB free; 13.71 GiB reserved in total by PyTorch)

=> The batch size of data is too large to fit in the GPU. Reduce the batch size.
⇒ If even batch size=1 still has this error, reduce model size
⇒ If error persists, consider buying a better GPU ☺

Common Errors - Mismatched Tensor Type

```
import torch.nn as nn
L = nn.CrossEntropyLoss()
preds = torch.randn(5,5)
labels = torch.Tensor([1,2,3,4,0])
lossval = L(preds, labels) # Calculate CrossEntropyLoss between preds and labels
```

expected scalar type Long but found Float

=> labels must be long tensors, cast it to type “Long” to fix this issue

```
labels = labels.long()
lossval = L(preds, labels)
print(lossval)
```

Coding Demo:

1_mlp_Day1.ipynb
2_cnn_Day1.ipynb
3_transfer_Day1.ipynb
4_save_load_Day1.ipynb
5_PINNs_Day1.ipynb