WEEK 5: Machine Learning PyTorch Tutorial

Department of Computer Science and Information Engineering
Ubiquitous Sensing and Cloud Computing Lab

Outline

- Prerequisites
- What is PyTorch?
- Tensor manipulation
- Overview of the DNN Training Procedure
- Saving/Loading a Neural Network

Prerequisites

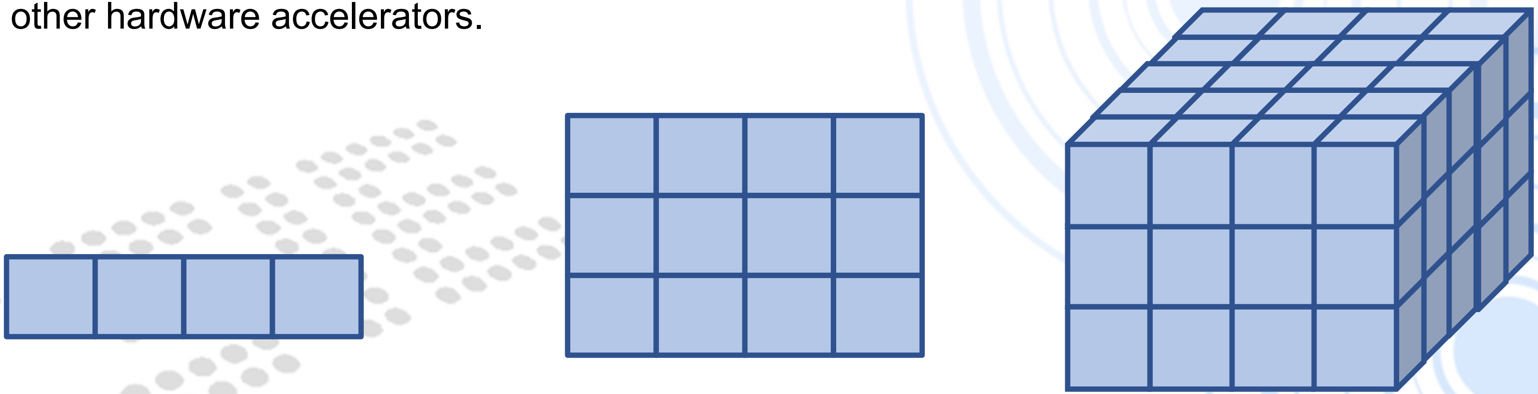
- We assume you are already familiar with…
  - Python3: if-else, loop, function, file IO, class, …
  - Numpy: array & array operations

What is PyTorch?

- PyTorch is an open source machine learning framework based on the Torch library
- PyTorch provides two high-level features:
  - **Tensor** computing (like NumPy) with strong acceleration via graphics processing units (GPU)
  - Deep neural networks built on a type-based automatic differentiation system

What is Tensor?

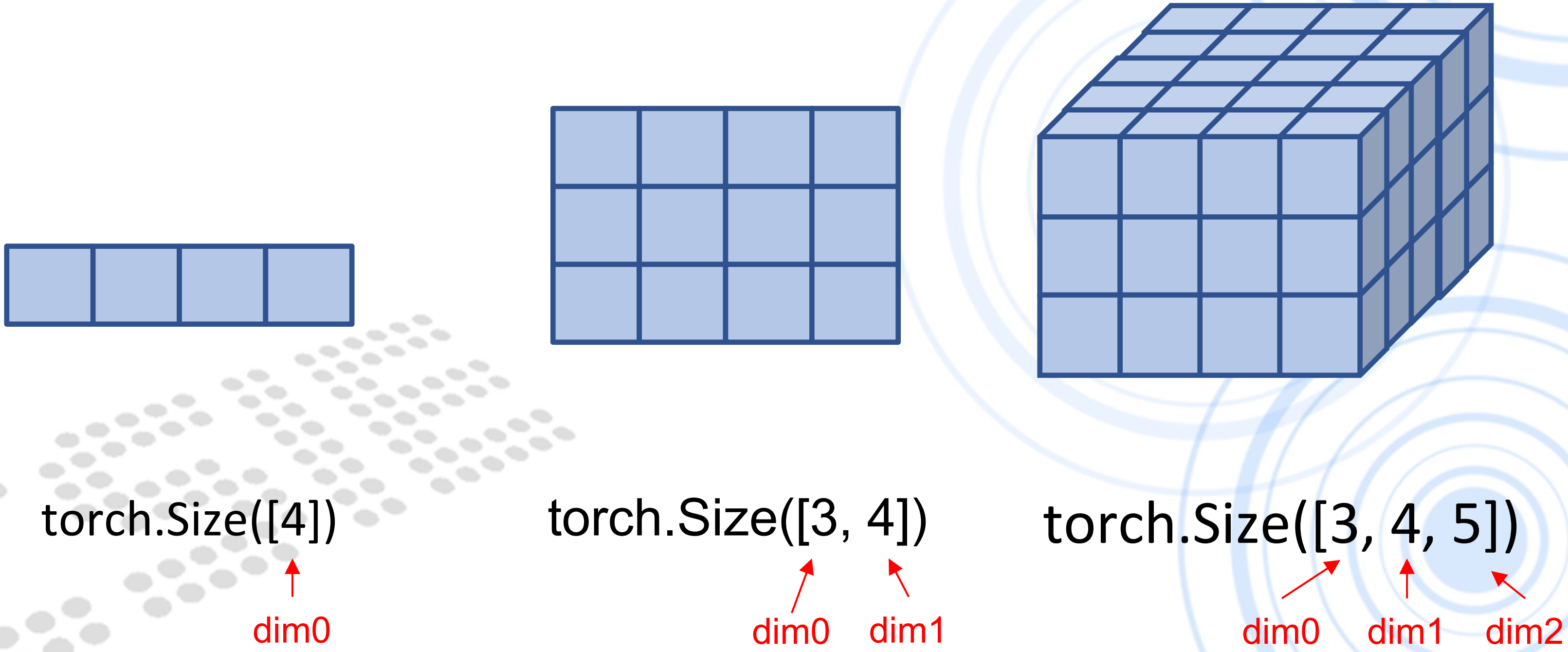
- Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.
- Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.



1-d tensor　　2-d tensor　　3-d tensor

Department of Computer Science and Information Engineering
Ubiquitous Sensing and Cloud Computing Lab

Shape of Tensors



Tensor Constructor

- Direct from data

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

```
tensor([[1, 2],
        [3, 4]])
```

- From a Numpy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
tensor([[1, 2],
        [3, 4]])
```

Tensor Constructor

```
shape = (2, 3, )
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

printf(f"Random Tensor: \n {rand_tensor} \n")
printf(f"Ones Tensor: \n {ones_tensor} \n")
printf(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.8012, 0.4547, 0.4156],
         [0.6645, 0.1763, 0.3860]])

Ones Tensor:
 tensor([[1., 1., 1.],
         [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
         [0., 0., 0.]])
```

Attributes of a Tensor

```
tensor = torch.rand(3, 4)

Shape of tensor:
>>> tensor.shape
torch.Size([3, 4])

Datatype of tensor:
>>> tensor.dtype
torch.float32

Device tensor is stored on:
>>> tensor.device
cpu
```

Operations on Tensors

```
tensor = torch.ones(4, 4)

First row:
>>> tensor[0]
tensor([1., 1., 1., 1.])

First column:
>>> tensor[:, 0]
tensor([1., 1., 1., 1.])

Last column:
>>> tensor[..., -1]
tensor([1., 1., 1., 1.])
```

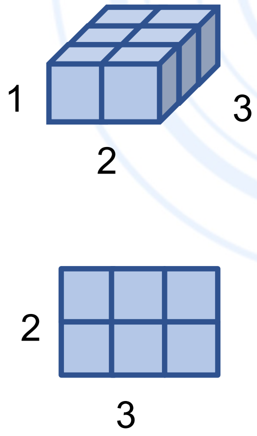
```
tensor = torch.ones(4, 4)

>>> tensor[:, 1] = 0
>>> print(tensor)
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Operations on Tensors

- Squeeze: remove all the dimensions of **input** of size **1**

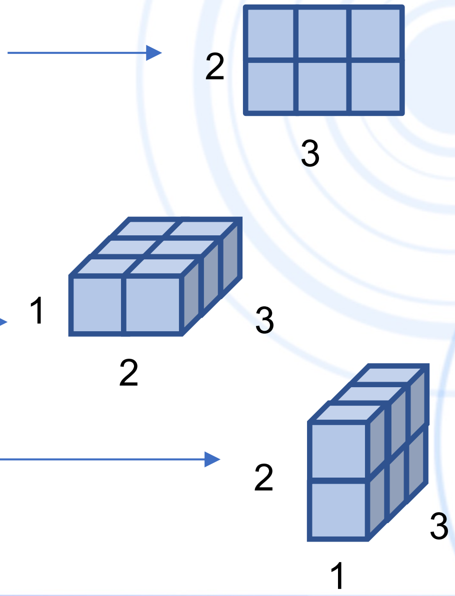
```
>>> x = torch.zeros([1, 2, 3])
>>> x.shape
torch.Size([1, 2, 3])
>>> x = x.squeeze(dim=0)
>>> x.shape
torch.Size([2, 3])
```



Operations on Tensors

- Unsqueeze: **insert** a dimension of size **1** at the specified position

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



Joining tensors

- torch.cat: Concatenate a sequence of tensors along a given dimension

```
>>> tensor = torch.ones(4, 4)
>>> tensor[:, 1] = 0
>>> print(tensor)
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

>>> t1 = torch.cat([tensor, tensor, tensor], dim=1)
>>> print(t1)
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

Arithmetic operations

- Add
```
>>> z = x + y
```
- Subtraction
```
>>> z = x - y
```
- Power
```
>>> y = x.pow(2)
```
- Summation
```
>>> y = x.sum()
```
- Mean
```
>>> y = x.mean()
```

Arithmetic operations

- This computes the matirx multiplication between two tensors. y1, y2 will have same value

```
>>> y1 = tensor @ tensor.T
>>> y2 = tensor.matmul(tensor.T)
```

- This computes the element-wise product. z1, z2 will have same value

```
>>> z1 = tensor * tensor
>>> z2 = tensor.mul(tensor)
```

Single-element tensors

- Convert one-element tensor to a Python numerical value using **item()**

```
>>> agg = tensor.sum()
>>> agg_item = agg.item()
>>> print(agg_item, type(agg_item))
12.0 <class ‘float’>
```

Single-element tensors

- Convert one-element tensor to a Python numerical value using **item()**

```
>>> agg = tensor.sum()
>>> agg_item = agg.item()
>>> print(agg_item, type(agg_item))
12.0 <class ‘float’>
```

Training on GPU

- By default, tensors are created on the CPU. We move tensors to the GPU using **.to** method

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to(“cuda”)
```

What is cuda?
CUDA (or Compute Unified Device Architecture) is a parallel computing https://en.wikipedia.org/wiki/CUDA

Build the neural network

Step 1
Prepare Dataset

Write Dataset & DataLoader

```
torch.utils.data.Dataset
torch.utils.data.DataLoader
```

Step 2
Training Model

Write Model

Set Loss func

Set Optimizer

Step 3
Val/Test Model

Dataset & DataLoader

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file):
        self.img_labels = pd.read_csv(annotations_file)

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

    def __getitem__(self, idx):
        img_path = self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        return image, label
```

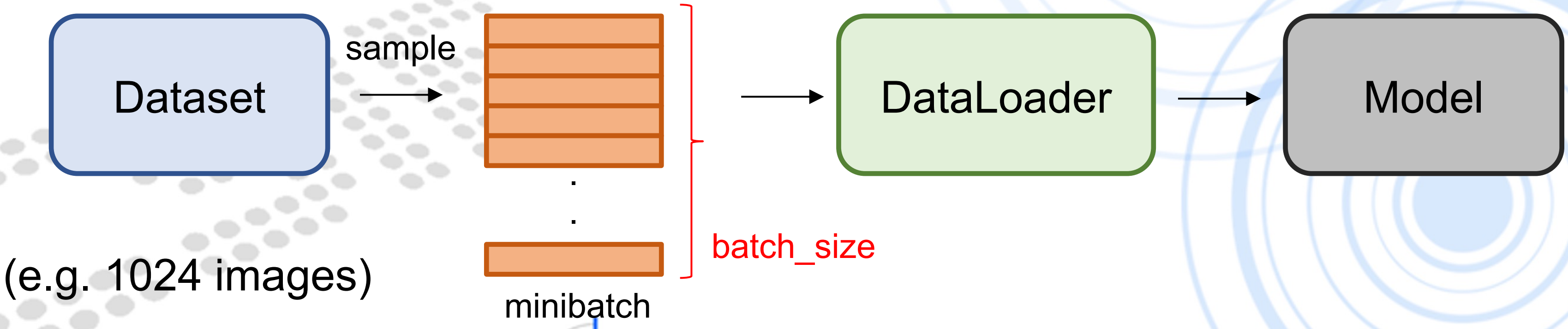
__init__:
initialize the image, the annotations file, and both transforms

__len__:
return the number of samples in dataset

__getitem__:
loads and returns a sample from dataset at the given index **idx**

Dataset & DataLoader

```
from torch.utils.data import DataLoader

dataset = CustomImageDataset(annotations_file)
train_dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```



Torchvision

```
# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='../../data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='../../data',
                                          train=False,
                                          transform=transforms.ToTensor())
```

```
[ ] # Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Build the neural network

Step 1
Prepare Dataset

Write Dataset & DataLoader

```
torch.utils.data.Dataset
torch.utils.data.DataLoader
```

Step 2
Training Model

- Write Model
- Set Loss func
- Set Optimizer

Step 3
Val/Test Model

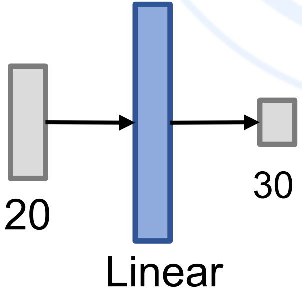
Training Model(Linear function)

```
torch.nn.Linear(in_features, out_features,
                bias=True, device=None, dtype=None)
```

Applies a linear transformation to the incoming data: $y = xA^T + b$

```
# Example:
>>> linear = torch.nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = linear(input)
>>> print(output.size())
torch.Size([128, 30])
```

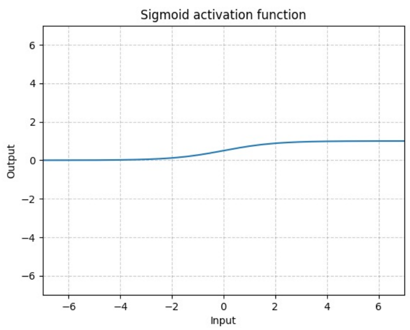
Input: (*, Hin)
output(*, Hout)



Training Model(Activation function)

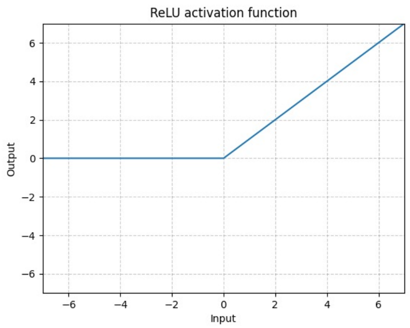
Sigmoid:

```
# Example
>>> sigmoid = nn.Sigmoid()
>>> input = torch.rand(2)
>>> output = sigmoid(input)
```



RELU:

```
# Example
>>> relu = nn.ReLU()
>>> input = torch.randn(2)
>>> output = relu(input)
```



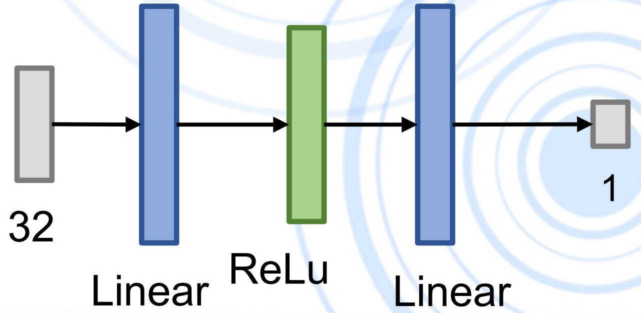
Training Model(Build Model)

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
         )

    def forward(self, x):
        logits = self.linear_relu_stack(x)
        return logits
```

__init__:
initialize the neural network layers

__forward__:
define how the model is to be run, from input to output

nn.Sequential:



Training Model(Set Loss)

```
>>> loss_fn1 = nn.MSELoss()
>>> loss_fn2 = nn.CrossEntroLoss()
```

Training Model(Set Optimizer)

TORCH.OPTIM:

optimizer will hold the current state and will update the parameters based on the computed gradients

```
>>> optimizer = optim.SGD(model.parameters(), lr=0.01, momemtum=0.9)
>>> optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Training Model(Overall)

```
# Create the dataset
dataset = CustomImageDataset(annotations_file)

# read data from dataset
train_dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

model = Classifier().to(device)    # set model to device
criterion = nn.MSELoss()    # Initialize the loss function

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

Training Model(Overall)

```
# Initialize the EPOCHS
EPOCHS = 100

# Iterate EPOCHS times
for epoch in range(EPOCHS)
    model.train()   # set model to train mode
    for x, y in train_dataloader:   # Iterate over the training set
        optimizer.zero_grad()   # reset the gradients
        x, y = x.to(device), y.to(device)   # move the data to device
        pred = model(x)    # get output from the model
        loss = criterion(pred, y)    # compute loss
        loss.backward()    # compute gradient
        optimizer.step()    # update model paremeters
```

Build the neural network

Step 1
Prepare Dataset

Write Dataset & DataLoader

```
torch.utils.data.Dataset
torch.utils.data.DataLoader
```

Step 2
Training Model

Write Model

Set Loss func

Set Optimizer

Step 3
Val/Test Model

Evaluate Model(Validation Set)

```
model.eval()  # set model to evaluate mode
total_loss = 0

for x, y in val_dataloader:  # Iterate over the validation set
    x, y = x.to(device), y.to(device)  # move data to device
    with torch.no_grad():  # disable gradient calculation
        pred = model(x)  # get output from the model
        loss = criterion(pred, y)  # compute loss
    total_loss += loss.cpu().item() * len(x)  # compute total loss
    avg_loss = total_loss / len(val_dataloader.dataset)  # compute average loss
```

Evaluate Model(Testing Set)

```
model.eval() # set model to evaluate mode
preds = []
for x in tt_set:
        x = x.to(device)  # move data to device
    with torch.no_grad():  # disable gradient calculation
        pred = model(x)  # get output from the model
        preds.append(pred.cpu())  # collect predictions
```

Saving & Loading Model for Inference

Save:

```
>>> torch.save(model.state_dict(), PATH)
```

Load:

```
>>> model = TheModelClass(*args, **kwargs)
>>> model.load_state_dict(torch.load(PATH))
>>> model.eval()
```

Colab Material

- tensor_tutorial
- pytorch tutorial
- pytorch lab

Reference

- https://pytorch.org/
- https://pytorch.org/tutorials/
- https://github.com/Atcold/pytorch-Deep-Learning
- https://github.com/yunjey/pytorch-tutorial