Introduction to Pytorch

Zekun Li
PhD Student in Computer Science
University of Minnesota
Deep Learning Applications

• Text Detection
• Image Style Transfer
• Face Pose & Gaze Detection
• Video Synthesis
Deep Learning Applications – Text Detection

Deep neural network based text detector for historical maps

https://github.com/machines-reading-maps/map-kurator
Deep Learning Applications – Map Style Transfer

Convert the OSM map images to the historical style

Deep Learning Applications – Pose & Gaze

A joint model to predict the gaze and face mesh simultaneously
Deep Learning Applications – Video Synthesis

Why choose Pytorch?

- Python-based framework
- Easy to learn and easy to debug
- Dynamic graph structure
- Supports GPU and CPU computation

Figure from https://devopedia.org/deep-learning-frameworks
Outline

• Pytorch Tensors
• Frequently used layers
  • Linear Layer
  • Convolution Layer
• Activation Functions
• Train neural network with Pytorch
Pytorch Tensor

- Tensor is a **multi-dimensional matrix** containing elements of a single data type
- Tensors can be created directly from a list

```python
# Create a tensor directly from a list
a_list = [[0, 1],[2, 3],[4, 5]]
a_tensor = torch.tensor(a_list)
print(a_tensor)
```

- or initialized from Numpy array

```python
# Create a tensor from Numpy array
a_array = np.array([[0, 1],[2, 3],[4, 5]])
data = torch.from_numpy(a_array)
print(data)
```
Pytorch Tensor

• Tensor can be created given the size of **existing tensors**
  • Tensor with ones
    
    ```python
    a.ones = torch.ones_like(a_tensor) # same shape as a_tensor, but with ones
    print(a.ones)
    tensor([[1, 1],
            [1, 1],
            [1, 1]], dtype=torch.int32)
    ```

  • Tensor with random values
    
    ```python
    a.rand = torch.rand_like(a_tensor, dtype=torch.float) # same shape as a_tensor, with rand values
    print(a.rand)
    tensor([[0.6263, 0.3245],
            [0.4140, 0.2188],
            [0.5598, 0.0729]])
    ```
Tensor Data Types

• Each tensor has a data type
• You can **specify** the data type **explicitly** when creating tensor
• If **not** specified, the data type will be **inferred** implicitly

```python
a_list = [[0, 1], [2, 3], [4, 5]]
a_tensor = torch.tensor(a_list, dtype=torch.float32)
print(a_tensor)

tensor([[0., 1.],
        [2., 3.],
        [4., 5.]])
```

```python
a_list = [[0, 1], [2, 3], [4, 5]]
a_tensor = torch.tensor(a_list, dtype=torch.int)
print(a_tensor)

tensor([[0, 1],
        [2, 3],
        [4, 5]], dtype=torch.int32)
```
Tensor Attributes

• Frequently used attributes

```python
a_list = [[0, 1],[2, 3],[4, 5]]
a_tensor = torch.tensor(a_list, dtype=torch.int)
print(f"Shape of tensor: {a_tensor.shape}")
print(f"Datatype of tensor: {a_tensor.dtype}")
print(f"Device tensor is stored on: {a_tensor.device}")
```
Shape of tensor: torch.Size([3, 2])
Datatype of tensor: torch.int32
Device tensor is stored on: cpu

• List all attributes and functions with `dir()`

```python
dir(a_tensor)
['cumprod_','cumsum_','cumsum_','data','data_ptr','deg2rad_','deg2rad_','dense_dim','dequantize','det','detach_','detach_','device','diag','diag_embed','diagflat','diagonal','diff','
```
Tensor Operations

• Pytorch tensors support indexing and slicing operations

```python
a_list = [[0, 1],[2, 3],[4, 5]]
a_tensor = torch.tensor(a_list, dtype=torch.int)
print(a_tensor)
print(a_tensor[2][1])
print(a_tensor[:, 0])
tensor([[0, 1],
        [2, 3],
        [4, 5]], dtype=torch.int32)
tensor(5, dtype=torch.int32)
tensor([[0, 2, 4], dtype=torch.int32])
```
Tensor Operations

- Joining tensors

```python
import torch

# Joining tensors along dimension 1
a_tensor = torch.Tensor([[0, 1, 0, 1, 0, 1],
                          [2, 3, 2, 3, 2, 3],
                          [4, 5, 4, 5, 4, 5]],
                         dtype=torch.int32)

t1 = torch.cat([a_tensor, a_tensor, a_tensor], dim=1)
print(t1)

# Joining tensors along dimension 0
b_tensor = torch.Tensor([[0, 1],
                          [2, 3],
                          [4, 5],
                          [0, 1],
                          [2, 3],
                          [4, 5]],
                         dtype=torch.int32)

t2 = torch.cat([a_tensor, a_tensor, a_tensor], dim=0)
print(t2)
```
Tensor Multiplication

• Matrix Multiplication

```python
y1 = a_rand @ a_rand.T
y2 = a_rand.matom(a_rand.T)

y3 = torch.randn(3,3)
torch.matom(a_rand, a_rand.T, out=y3)

print(y1)
print(y2)
print(y3)
```

tensor([[0.4975, 0.3303, 0.3742],
        [0.3303, 0.2193, 0.2477],
        [0.3742, 0.2477, 0.3187]])
tensor([[0.4975, 0.3303, 0.3742],
        [0.3303, 0.2193, 0.2477],
        [0.3742, 0.2477, 0.3187]])
tensor([[0.4975, 0.3303, 0.3742],
        [0.3303, 0.2193, 0.2477],
        [0.3742, 0.2477, 0.3187]])

• Element-wise Multiplication

```python
z1 = a_rand * a_rand
z2 = a_rand.mul(a_rand)

z3 = torch.randn(3,2)
torch.mul(a_rand, a_rand, out=z3)
print(z1)
print(z2)
print(z3)
```

tensor([[0.3922, 0.1053],
        [0.1714, 0.0479],
        [0.3133, 0.0053]])
tensor([[0.3922, 0.1053],
        [0.1714, 0.0479],
        [0.3133, 0.0053]])
tensor([[0.3922, 0.1053],
        [0.1714, 0.0479],
        [0.3133, 0.0053]])
Tensor Gradient

• Pytorch could automatically calculate the gradient of a tensor

```python
# requires_grad=True tells PyTorch to store the gradient
x = torch.tensor([3.], requires_grad=True)

# Currently None since x is not connected to other tensors
print(x.grad)

None

# Calculating the gradient of y with respect to x
y = x * x  # y=x^2
y.backward()
print(x.grad)  # d(y)/d(x) = d(x^2)/d(x) = 2x = 6

tensor([6.])
```
Tensor Gradient

• Gradients will be summed up before making an update

```python
z = x * x * 5  # 5x^2
z.backward()
print(x.grad)  #d(y)/d(x) + d(z)/d(x) = 2x + 10x = 36
```
tensor([[36.]])

• Reset gradient with .grad.zero_() or optimizer.zero_grad() during training

```python
x.grad.zero_()  # zero out the gradient
z = x * x * 5  # 5x^2
z.backward()
print(x.grad)  #d(z)/d(x) = 10x = 30
```
tensor([[30.]])
Linear Layer

- Create a Linear Layer
  - Linear Layer performs the operation $y=Ax+b$
  - $A$ and $b$ are network parameters (weights) initialized randomly
  - If we do not need $b$, set the bias=False

```python
input = torch.ones(32, 200)
# N,H_in -> N,H_out

# Make a linear layers transforming N, H_in dimensional inputs to N, H_out
# dimensional outputs
linear = nn.Linear(200, 100)
linear_output = linear(input)
linear_output.shape

torch.Size([32, 100])
```
Linear Layer

• Create a Linear Layer
  • Linear Layer can also take 3D tensor as input

```python
# Create the inputs
input = torch.ones(32, 3, 200)
# N, *, H_in -> N, *, H_out

# Take N,*,H_in dimensinal inputs and output N,*,H_out tensor
linear = nn.Linear(200, 100)
linear_output = linear(input)
linear_output.shape
```

**Question:** what is the shape of `linear_output`?
Linear Layer

- Create a Linear Layer
  - Linear Layer can also take 3D tensor as input

```
# Create the inputs
input = torch.ones(32, 3, 200)
# N, *, H_in -> N, *, H_out

# Take N,*,H_in dimensional inputs and output N,*,H_out tensor
linear = nn.Linear(200, 100)
linear_output = linear(input)
linear_output.shape

torch.Size([32, 3, 100])
```
Linear Layer

• Shape of the network parameters $A$ and $b$

```python
define nn.Linear(200, 100)
A, b = list(linear.parameters())
print(A.shape)
print(b.shape)
```

**Question:** what is the shape of $A$ and $b$?
Linear Layer

• Shape of the network parameters $A$ and $b$

```python
linear = nn.Linear(200, 100)
A, b = list(linear.parameters())
print(A.shape)
print(b.shape)

torch.Size([100, 200])
torch.Size([100])
```
Convolution Layer

- Convolution Layer
  - `nn.Conv2d`

```python
input = torch.ones(32, 3, 100, 100)  # batch_size, channel, height, width
# Conv2d(in_channels, out_channels, kernel_size, stride, padding, kwargs)
# With square kernels and equal stride
m = nn.Conv2d(3, 16, 3, stride=2)
output = m(input)
print(output.shape)
```

`torch.Size([32, 16, 49, 49])`

Figure from https://github.com/vdumoulin/conv_arithmetic

blue map is input
green map is output
Activation Functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1+e^{-x}} \]

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]

**Leaky ReLU**
\[ \max(0.1x, x) \]

**Maxout**
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[ \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
\end{cases} \]

Figure from https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092
Activation Functions

• Sigmoid Activation

```python
print(linear_output.shape)
print(linear_output[0,0:20])
sigmoid = nn.Sigmoid()
sig_output = sigmoid(linear_output)
print(sig_output[0,0:20])
```

torch.Size([32, 3, 100])
tensor([ 0.1973, -0.1327,  1.2161,  0.5312, -1.1714,  0.1625, -0.1284, -0.1617,
        0.6658,  0.5343, -0.0825,  0.3412, -0.1179,  0.8846,  0.6028,  1.4662,
        -0.8332, -0.0781,  0.2253,  0.5549], grad_fn=<SliceBackward0>)
tensor([ 0.5492,  0.4669,  0.7714,  0.6298,  0.2366,  0.5405,  0.4679,  0.4597,  0.6606,
        0.6305,  0.4794,  0.5845,  0.4706,  0.7078,  0.6463,  0.8125,  0.3030,  0.4805,
        0.5561,  0.6353], grad_fn=<SliceBackward0>)
```

Notice that the range of sig_output and linear_output is different!
Build the Neural Network

- `__init__()`
  - Declare the layers to use

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

This example is from https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
Build the Neural Network

- `__init__()`
  - Declare the layers to use
- `forward()`
  - Construct the network

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```
Define Loss function and Optimizer

• For classification tasks, it is common to use cross entropy loss
• Common optimizers are Stochastic Gradient Descent (SGD) and Adam

```python
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```
Load and Normalize the Dataset

- Define transformation
  
  ```python
  transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
  ```

- Load the dataset
  
  ```python
  trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
     download=True, transform=transform)
  trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
     shuffle=True, num_workers=2)
  ```

  Prepare the inputs and GTs **one sample** at a time

  Collect the inputs and GTs into **minibatches**
Load and Normalize the Dataset

• Define transformation

```python
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

• Load the dataset

```python
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, 
                                         download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, 
                                         shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, 
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, 
                                       shuffle=False, num_workers=2)
```
Custom Dataset

- Pytorch has pre-defined classes for benchmark datasets

```python
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
```

- To process your own data, you need to write a custom dataset class

```python
from torch.utils.data.dataset import Dataset

class MyCustomDataset(Dataset):
    def __init__(self, ...):
        # stuff

    def __getitem__(self, index):
        # stuff
        return (img, label)

    def __len__(self):
        return count # of how many examples(images?) you have
```
Custom Dataset Example

```python
class LandmarkDataset(Dataset):
    def __init__(self, image_paths, transform=False):
        self.image_paths = image_paths
        self.transform = transform

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

    def __getitem__(self, idx):
        image_filepath = self.image_paths[idx]
        image = cv2.imread(image_filepath)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        label = image_filepath.split('/')[-2]
        label = class_to_idx[label]
        if self.transform is not None:
            image = self.transform(image=image)['image']

        return image, label
```

This example is from https://towardsdatascience.com/custom-dataset-in-pytorch-part-1-images-2df3152895
Train the Network

```python
for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print(('[%d, %5d] loss: %.3f' %
                    (epoch + 1, i + 1, running_loss / 2000)))
            running_loss = 0.0

    print('Finished Training')
```
Train the Network

```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

    print('Finished Training')
```
Train the Network

```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

    print('Finished Training')
```
Train the Network

```python
for epoch in range(2): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999: # print every 2000 mini-batches
            print('Epoch [%d, %5d] Loss: %.3f' %
                (epoch + 1, i + 1, running_loss / 2000))
        running_loss = 0.0

    print('Finished Training')
```
Save and Load Model

- Save model weights
  - Model weights are stored in an internal state dictionary

  ```python
  torch.save(model.state_dict(), 'model_weights.pth')
  ```

- Load model weights

  ```python
  model.load_state_dict(torch.load('model_weights.pth'))
  model.eval()
  ```
Summary: Essential Components

• Dataset
• Model
• Loss function
• Optimizer
Jupyter notebook Tutorials

• CIFAR-10 Tutorial:
  • [https://yaoyichi.github.io/spatial-ai/lab/CIFAR10_Tutorial.ipynb](https://yaoyichi.github.io/spatial-ai/lab/CIFAR10_Tutorial.ipynb)

• Transfer Learning Tutorial
  • [https://yaoyichi.github.io/spatial-ai/lab/transfer_learning_tutorial.ipynb](https://yaoyichi.github.io/spatial-ai/lab/transfer_learning_tutorial.ipynb)
How to use Google Colab?

Go to [https://colab.research.google.com/](https://colab.research.google.com/), click New notebook to create a live jupyter notebook instance.
How to use Google Colab?

To enable GPU, go to Runtime-> Change runtime type, set the Hardware accelerator to be GPU.
Acknowledgement

• Some materials are adapted from
  • Pytorch official tutorial
  • Stanford CS231N course
  • Stanford CS224N course
These materials are released under a CC-BY License

https://creativecommons.org/licenses/by/2.0/

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

Please credit as: Li, Zekun Introduction to Spatial Artificial Intelligence. Available from https://yaoyichi.github.io/spatial-ai.html

If you use an individual slide, please place the following at the bottom: “Credit: https://yaoyichi.github.io/spatial-ai.html

We welcome your feedback and contributions.

Credit: http://www.datascience4all.org/