1. **tensor\_tutorial**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| torch.empty(5, 3) | Uninitialized 5x3 tensor |
| torch.rand(5, 3) | Random 5x3 tensor |
| torch.zeros(5, 3, dtype=torch.long) | 5x3 tensor of all zeros, data type long |
| torch.tensor([5.5, 3]) | Tensor created from data |
| x.new\_ones(5, 3, dtype=torch.double) | Tensor of ones. By default, has same dtype as x |
| torch.randn\_like(x, dtype=torch.float) | Random tensor, same size as x |
|  |  |
| x.size() | Shape of tensor – this is in fact a tuple |
| x+y | Addition #1 |
| torch.add(x, y) | Addition #2 |
| result = torch.empty(5, 3)  torch.add(x, y, out=result)  print(result) | Addition #3  Providing output tensor as argument |
| y.add\_(x) | Addition #4 – in-place |
| Any in-place operation post-fixed with “\_” | e.g. x.copy\_(y) |
| x[:, 1] | NumPy-like indexing |
| x.view(-1, 8) | Resizing x: -1 means inferring the size from other dimensions |
| x.item() | One-element tensor: get the value as a number |
| torch tensor-NumPy array conversion | Share underlying memory locations |
| a.numpy() | Change torch tensor a to a NumPy array |
| torch.from\_numpy(a) | From NumPy to torch |
| CUDA Tensors | For use with GPU |
| x.to(device) | to method moves tensor to a device |
| z.to(“cpu”, torch.double) | This method can change dtype too |
| torch.device(“cuda”) | CUDA device object |
| torch.cuda.is\_available() | Check if CUDA is available |
| torch.ones\_like(x , device=device) | Directly create tensor with GPU |

1. **autograd\_tutorial**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| Autograd provides automatic differentiation on all tensor operations | Define-by-run framework, backprop is defined by how code is run |
| torch.Tensor is the central class | Set .requires\_grad=True to track operations  Call .backward() to compute gradients  .grad attribute |
| .detach() | Detach from computation history |
| with torch.no\_grad(): | Wrap code block in this to prevent tracking history |
| Function class | Tensor & Function interconnected. They make up an acyclic graph that encodes the computation history  Each tensor has .grad\_fn |
| .backward() | Computing derivatives  Tensor is scalar -> no arguments  More elements -> specify gradient argument, a tensor of matching shape |
| torch.ones(2, 2, requires\_grad=True) |  |
| y = x+2  y.grad\_fn |  |
| a.requires\_grad\_(True) | In-place changing of a’s requires\_grad attribute |
| out.backward() | out is scalar, this is equivalent to out.backward(torch.tensor(1)) |
| x.grad | Gradients d(out)/dx |

1. **neural\_networks\_tutorial**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| torch.nn package | Neural networks constructed using this |
| nn.Module | Contains layers and forward(input) method which returns output |
| net.parameters() | Learnable parameters of the model |
| input = torch.randn(1, 1, 32, 32)  out = net(input)  print(out) | Example forward prop |
| net.zero\_grad()  out.backward(torch.randn(1, 10)) | Zero the gradients  Then backprop with random gradients |
| torch.nn only supports mini-batches | Doesn’t support a single sample  e.g. nn.Conv2d takes 4D Tensor:  nSamples x nChannels x Height x Width |
| input.unsqueeze(0) | Add a fake batch dimension |
| criterion = nn.MSELoss()  loss = criterion(output, target) | Compute MSE Loss between output tensor of network and target tensor |
| loss.grad\_fn.next\_functions[0][0] | Example of accessing grad functions |
| loss.backward() | To backpropagate the error |
| net.zero\_grad() | Clear gradients to avoid them being accumulated |
| torch.optim | Package which implements various optimizers, e.g. SGD, Adam, RMSProp, to update parameters |
| optim.SGD(net.parameters(), lr=0.01) | Creating the optimizer |
| optimizer.step() | Update the weights (at end of training loop) |

**Typical training procedure:**

1. Define network with trainable parameters
2. Iterate over dataset
3. Process input through network
4. Compute loss
5. Backpropagate gradients
6. Update network parameters: weight = weight – learning\_rate \* gradient

**Defining a Network:**

import torch

import torch.nn as nn

import torch.nn.functional as F

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

\*\**Define layers here\*\**

def forward(self, x):

\*\*Apply layers and activations here – forward prop\*\*

net = Net()

1. **cifar10\_tutorial**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| Working with image, audio, text or video data | Use a standard Python library to load it into NumPy and convert to a torch Tensor |
| Pillow, OpenCV | Image data |
| Scipy, Librosa | Audio data |
| Raw Python, Cython, NLTK, SpaCy | Text data |
| torchvision | Specifically for vision  Contains data loaders for common datasets & data transformers |
| torchvision.datasets  torch.utils.data.DataLoader |  |
| torchvision.transforms | Contains the image transformers |
| transforms.Compose([transforms.ToTensor(),  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) | Transforms image to a torch tensor and normalizes it |
| dataiter = iter(trainloader)  images, labels = dataiter.next() | Get a batch of images |
| plt.imshow()  torchvision.utils.make\_grid(images) | Matplotlib and torchvision functions used for visualization |
| criterion = nn.CrossEntropyLoss()  optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) | Loss function & optimizer |
| torch.max(outputs, 1) | Max along dimension 1 (each row)  Returns max value & index |
| device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu") | Define device as first CUDA device available |
| net.to(device) | Send network to GPU |
| inputs, labels = inputs.to(device), labels.to(device) |  |

**Training an Image Classifier:**

1. Load & normalize CIFAR10 training & test datasets using torchvision
2. Define a CNN
3. Define loss function
4. Train network on training data
5. Test network on test data

**Creating CIFAR10 DataLoader:**

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)

*\*\*Similar process for the test data\*\**

**Simple Training Loop for Image DataLoader:**

for epoch in range(2):

running\_loss = 0.0

for i, data in enumerate(trainloader, 0):

inputs, labels = data

optimizer.zero\_grad()

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

**Testing Network on Test Data:**

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

1. **data\_parallel\_tutorial:**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| model = nn.DataParallel(model) | Adding model to multiple GPUs |

**Put Model on Multiple Devices:**

model = Model(input\_size, output\_size)

if torch.cuda.device\_count() > 1:

print("Let's use", torch.cuda.device\_count(), "GPUs!")

model = nn.DataParallel(model)

model.to(device)

1. **data\_loading\_tutorial:**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
| Dataset class: torch.utils.data.Dataset | Abstract class representing dataset |
| Custom dataset should inherit Dataset | Also, should override these 2 methods:   1. \_\_len\_\_ - len(dataset) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**16. transfer\_learning\_tutorial:**

|  |  |
| --- | --- |
| **Code/Detail** | **Description** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |