Deep Learning with Pytorch

A concise reference for deep learning practitioners using PyTorch, covering core concepts, modules, and common operations.

Core Concepts & Tensor Operations

Tensors

Creating Tensors:

- `torch.tensor(data)`: Create a tensor from data (list, tuple, array).
- `torch.zeros(size)`: Create a tensor filled with zeros.
- `torch.ones(size)`: Create a tensor filled with ones.
- `torch.rand(size)`: Create a tensor with random values (uniform distribution).
- `torch.randn(size)`: Create a tensor with random values (normal distribution).
- `torch.empty(size)`: Create an uninitialized tensor.

Tensor Attributes:

- `.shape`: Returns the shape of the tensor.
- `.dtype`: Returns the data type of the tensor.
- `.device`: Returns the device on which the tensor is stored (CPU or GPU).

Moving Tensors:

- `.to(device)`: Moves the tensor to the specified device (e.g., `torch.device('cuda')`).
- `.cpu()`: Moves the tensor to the CPU.
- `.cuda()`: Moves the tensor to the GPU.

Basic Operations

Arithmetic:

- `torch.add(a, b)` or `a + b`: Element-wise addition.
- `torch.sub(a, b)` or `a - b`: Element-wise subtraction.
- `torch.mul(a, b)` or `a * b`: Element-wise multiplication.
- `torch.div(a, b)` or `a / b`: Element-wise division.
- `torch.pow(a, b)` or `a ** b`: Element-wise exponentiation.

Matrix Operations:

- `torch.matmul(a, b)` or `a @ b`: Matrix multiplication.
- `torch.transpose(a, dim0, dim1)`: Transpose the tensor.
- `torch.inverse(a)`: Inverse of a matrix.
- `torch.det(a)`: Determinant of a matrix.

Slicing and Indexing:

- `a[index]`: Accessing a single element.
- `a[start:end]`: Slicing a tensor.
- `a[mask]`: Indexing with a boolean mask.
- `torch.gather(input, dim, index)`: Gathers values along an axis specified by dim.

Reshaping:

- `a.view(new_shape)`: Reshapes the tensor without changing its data.
- `a.reshape(new_shape)`: Returns a tensor with the same data and number of elements as input, but with the specified shape.
- `a.squeeze()`: Removes dimensions of size one.
- `a.unsqueeze(dim)`: Adds a dimension of size one at the specified position.

Autograd

Automatic Differentiation:

- `requires_grad=True`: Enable gradient tracking for a tensor.
- `.backward()`: Compute gradients of a tensor with respect to the graph leaves.
- `.grad`: Access the computed gradients.
- `with torch.no_grad():`: Disable gradient calculation within a block.

Example:

```
x = torch.randn(3, requires_grad=True)
y = x + 2
z = y * y * 2
z = z.mean()
z.backward()
print(x.grad) # Gradients of z w.r.t. x
```

Neural Network Modules

Linear Layers

`torch.nn.Linear(in_features, out_features)`: Applies a linear transformation to the incoming data: `y = xW^T + b`.

- `in_features`: Size of each input sample.
- `out_features`: Size of each output sample.
- `weight`: The learnable weights of the module.
- `bias`: The learnable bias of the module.

Example:

```
import torch.nn as nn
linear = nn.Linear(20, 30) # Input size 20, output size 30
input = torch.randn(128, 20) # Batch of 128 samples, each of size 20
output = linear(input)
print(output.size()) # torch.Size([128, 30])
```

Activations

- `torch.nn.ReLU()`: Rectified Linear Unit.
- `torch.nn.Sigmoid()`: Sigmoid function.
- `torch.nn.Tanh()`: Hyperbolic Tangent function.
- `torch.nn.LeakyReLU()`: Leaky ReLU.
- `torch.nn.Softmax(dim)`: Softmax function (often used for classification, `dim` specifies the dimension to normalize across).

Example:

```
import torch.nn as nn
relu = nn.ReLU()
input = torch.randn(2)
output = relu(input)
print(output)
```

Usage:
Activation functions are typically applied element-wise after linear transformations to introduce non-linearity into the model.

Convolutional Layers

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`: Applies a 2D convolution over an input signal composed of several input planes.

- `in_channels`: Number of input channels.
- `out_channels`: Number of output channels.
- `kernel_size`: Size of the convolutional kernel.
- `stride`: Stride of the convolution.
- `padding`: Zero-padding added to both sides of the input.

Example:

```
import torch.nn as nn
conv = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1) # Input channels 3, output channels 16
input = torch.randn(1, 3, 64, 64) # Batch of 1 image, 3 channels, 64x64 resolution
output = conv(input)
print(output.size()) # torch.Size([1, 16, 64, 64])
```

Pooling Layers

- `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0)`: Applies a 2D max pooling over an input signal.
- `torch.nn.AvgPool2d(kernel_size, stride=None, padding=0)`: Applies a 2D average pooling over an input signal.

Example:

```
import torch.nn as nn
pool = nn.MaxPool2d(kernel_size=2, stride=2)
input = torch.randn(1, 16, 64, 64)
output = pool(input)
print(output.size()) # torch.Size([1, 16, 32, 32])
```

Usage:
Pooling layers reduce the spatial dimensions of the input and help extract dominant features.

Model Building & Training

Defining a Model

Using `torch.nn.Module`:
Models are defined as classes that inherit from `torch.nn.Module`. The forward pass is defined in the `forward` method.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 3)
        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
```

Loss Functions

- `torch.nn.CrossEntropyLoss()`: Commonly used for multi-class classification.
- `torch.nn.MSELoss()`: Mean Squared Error loss, used for regression.
- `torch.nn.BCELoss()`: Binary Cross Entropy loss, used for binary classification.
- `torch.nn.L1Loss()`: L1 Loss (Mean Absolute Error).

Example:

```
import torch.nn as nn
loss_fn = nn.CrossEntropyLoss()
output = model(input)
loss = loss_fn(output, target)
loss.backward()
```

Optimizers

`torch.optim`:
PyTorch provides various optimization algorithms.

- `torch.optim.SGD(params, lr, momentum=0)`: Stochastic Gradient Descent.
- `torch.optim.Adam(params, lr, betas=(0.9, 0.999), eps=1e-08)`: Adam optimizer.
- `torch.optim.RMSprop(params, lr, alpha=0.99, eps=1e-08)`: RMSprop optimizer.

Example:

```
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=0.001)
optimizer.zero_grad()
output = model(input)
loss = loss_fn(output, target)
loss.backward()
optimizer.step()
```

Training Loop

Typical Training Loop:

```
for epoch in range(num_epochs):
    for i, (inputs, labels) in enumerate(train_loader):
        # Move data to device
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward and optimize
        loss.backward()
        optimizer.step()

        # Print statistics
        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, len(train_loader), loss.item()))
```

Data Loading and Preprocessing

Datasets

`torch.utils.data.Dataset`:
Base class for all datasets in PyTorch. You can create custom datasets by inheriting from this class and overriding the `__len__` and `__getitem__` methods.

Example:

```
from torch.utils.data import Dataset
from PIL import Image
import os

class CustomDataset(Dataset):
    def __init__(self, root_dir,
transform=None):
        self.root_dir = root_dir
        self.image_paths =
[os.path.join(root_dir, file) for file
in os.listdir(root_dir) if
file.endswith('.png')]
        self.transform = transform

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

    def __getitem__(self, idx):
        image_path =
self.image_paths[idx]
        image =
Image.open(image_path).convert('RGB')
        if self.transform:
            image =
self.transform(image)
        label = 0  # Replace with your
label loading logic
        return image, label
```

DataLoaders

`torch.utils.data.DataLoader`:
Provides an iterable over the dataset, with features like batching, shuffling, and parallel data loading.

- `dataset`: The Dataset object to load data from.
- `batch_size`: How many samples per batch to load.
- `shuffle`: Set to `True` to have the data reshuffled at every epoch.
- `num_workers`: How many subprocesses to use for data loading.

Example:

```
from torch.utils.data import DataLoader
dataset = CustomDataset(root_dir='data',
transform=transform)
dataloader = DataLoader(dataset,
batch_size=32, shuffle=True,
num_workers=4)

for images, labels in dataloader:
    # Process batch
    pass
```

Transforms

`torchvision.transforms`:
Provides common image transformations for preprocessing data.

- `transforms.ToTensor()`: Convert a PIL Image or NumPy ndarray to tensor.
- `transforms.Normalize(mean, std)`: Normalize a tensor image with mean and standard deviation.
- `transforms.Resize(size)`: Resize the input image to the given size.
- `transforms.RandomHorizontalFlip()`: Horizontally flip the given PIL Image randomly with a given probability.
- `transforms.Compose(transforms)`: Composes several transforms together.

Example:

```
from torchvision import transforms
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,
0.456, 0.406], std=[0.229, 0.224,
0.225])
])
```