PyTorch Cheat Sheet

A comprehensive cheat sheet for PyTorch, covering essential concepts, modules, and functions for building and training neural networks.

Core Concepts & Tensor Operations

Tensors

Creating Tensors:

- `torch.tensor(data)`: Creates a tensor from data (list, tuple, array).
- `torch.zeros(size)`: Creates a tensor filled with zeros.
- `torch.ones(size)`: Creates a tensor filled with ones.
- `torch.rand(size)`: Creates a tensor with random values from a uniform distribution (0, 1).
- `torch.randn(size)`: Creates a tensor with random values from a normal distribution (mean 0, std 1).
- `torch.arange(start, end, step)`: Creates a 1D tensor with values from `start` to `end` (exclusive) with step size `step`.
- `torch.linspace(start, end, steps)`: Creates a 1D tensor with `steps` evenly spaced values from `start` to `end` (inclusive).

Tensor Attributes:

- `tensor.shape`: Returns the dimensions of the tensor.
- `tensor.dtype`: Returns the data type of the tensor elements.
- `tensor.device`: Returns the device on which the tensor is stored (CPU or GPU).
- `tensor.requires_grad`: Indicates if the tensor requires gradients (for autograd).

Example:

```
import torch

x = torch.tensor([1, 2, 3],
dtype=torch.float32)
print(x.shape)       # Output: torch.Size([3])
print(x.dtype)       # Output: torch.float32
```

Tensor Operations

Basic 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.
Matrix Operations:	`torch.matmul(a, b)` or `a @ b`: Matrix multiplication. `torch.transpose(a, dim0, dim1)` or `a.T`: Transpose of a matrix.
Reduction Operations:	`torch.sum(a)`: Sum of all elements. `torch.mean(a)`: Mean of all elements. `torch.max(a)`: Maximum element. `torch.min(a)`: Minimum element.
Indexing, Slicing, Joining, Mutating Data:	Standard Python indexing and slicing. `torch.cat([a, b], dim=0)`: Concatenates tensors along a dimension. `torch.stack([a, b], dim=0)`: Stacks tensors along a new dimension. `a.view(new_shape)` or `a.reshape(new_shape)`: Reshapes a tensor.

CUDA Tensors

Moving Tensors to GPU:

- `torch.cuda.is_available()`: Checks if CUDA is available.
- `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`: Sets the device.
- `tensor.to(device)`: Moves a tensor to the specified device.
- `tensor.cuda()`: Moves a tensor to the CUDA device. (Deprecated - use `tensor.to(device)` instead)
- `tensor.cpu()`: Moves a tensor to the CPU device.

Example:

```
import torch

device = torch.device('cuda' if
torch.cuda.is_available() else 'cpu')
x = torch.tensor([1, 2, 3]).to(device)
print(x.device)
```

Autograd and Neural Network Modules

Autograd

Automatic Differentiation:

- `tensor.requires_grad = True`: Enables gradient tracking for a tensor.
- `output = function(tensor)`: Performs operations on tensors with `requires_grad=True`.
- `output.backward()`: Computes the gradients of `output` with respect to the tensors that require gradients.
- `tensor.grad`: Accesses the computed gradients of a tensor.
- `with torch.no_grad():`: Disables gradient calculation within a block of code.
- `tensor.detach()`: Creates a new tensor detached from the computational graph.

Example:

```
import torch

x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
z = y.sum()
z.backward()
print(x.grad)  # Output: tensor([2., 2., 2.])
```

Neural Network Modules (nn.Module)

Defining a Neural Network:

- Create a class that inherits from `nn.Module`.
- Define the layers in the `__init__` method using `nn.Linear`, `nn.Conv2d`, `nn.ReLU`, etc.
- Implement the forward pass in the `forward` method.
- Instantiate the model.
- `model.parameters()`: Returns an iterator over the model's parameters (used by optimizers).

Common Layers:

- `nn.Linear(in_features, out_features)`: Fully connected layer.
- `nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)`: 2D convolutional layer.
- `nn.MaxPool2d(kernel_size, stride)`: 2D max pooling layer.
- `nn.ReLU()`: Rectified Linear Unit activation function.
- `nn.Sigmoid()`: Sigmoid activation function.
- `nn.Tanh()`: Hyperbolic Tangent activation function.
- `nn.BatchNorm1d(num_features)`: Batch Normalization for 1D input.
- `nn.BatchNorm2d(num_features)`: Batch Normalization for 2D input.
- `nn.Dropout(p=0.5)`: Dropout layer (p is the probability of dropping out a neuron).

Example:

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
```

Training and Optimization

Loss Functions

Common Loss Functions:

- `nn.MSELoss()`: Mean Squared Error loss (for regression).
- `nn.CrossEntropyLoss()`: Cross-entropy loss (for classification).
- `nn.BCELoss()`: Binary Cross-Entropy loss (for binary classification).
- `nn.BCEWithLogitsLoss()`: Binary Cross-Entropy with Logits loss (more stable version of BCELoss when using sigmoid output).
- `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)
```

Optimizers

Common Optimizers:

- `torch.optim.SGD(params, lr, momentum, weight_decay)`: Stochastic Gradient Descent.
- `torch.optim.Adam(params, lr, betas, weight_decay)`: Adam optimizer.
- `torch.optim.RMSprop(params, lr, alpha, weight_decay)`: RMSprop optimizer.

Parameters:

- `params`: Iterable of parameters to optimize (e.g., `model.parameters()`).
- `lr`: Learning rate.
- `momentum`: Momentum factor (for SGD).
- `weight_decay`: L2 regularization penalty.

Optimization Step:

1. `optimizer.zero_grad()`: Clears the gradients of all optimized tensors.
2. `loss.backward()`: Computes the gradients of the loss with respect to the model parameters.
3. `optimizer.step()`: Updates the model parameters using the computed gradients.

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(dataloader):
        # Move data to device
        inputs = inputs.to(device)
        labels = labels.to(device)

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

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

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

Datasets and Data Loaders

Datasets (torch.utils.data.Dataset)

Creating a Custom Dataset:

- Create a class that inherits from `torch.utils.data.Dataset`.
- Implement the `__len__` method, which returns the size of the dataset.
- Implement the `__getitem__` method, which returns a data sample given an index.

Example:

```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.data[idx],
self.labels[idx]
```

DataLoaders (torch.utils.data.DataLoader)

Using DataLoaders:

- Create a `DataLoader` instance, passing in a `Dataset`.
- Specify batch size, shuffle, and number of worker processes.

Parameters:

- `dataset`: Dataset from which to load the data.
- `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(data, labels)
dataloader = DataLoader(dataset,
batch_size=32, shuffle=True, num_workers=4)

for inputs, labels in dataloader:
    # Process the batch
    pass
```

Pre-trained Models

Using Pre-trained Models:

- `torchvision.models`: Provides access to pre-trained models.
- Download a pre-trained model (e.g., ResNet, AlexNet, VGG).
- Modify the model for your specific task (e.g., replace the classification layer).
- Optionally, freeze some layers during training to prevent them from being updated.

Example:

```
import torchvision.models as models

# Load pre-trained ResNet-18
model = models.resnet18(pretrained=True)

# Modify the classification layer
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, num_classes)
```