AI Basics

PyTorch Cheatsheet

++ Code Examples ++

```
## Deep Learning
from torch import *
```

#Python

/sumitkhanna

Python

Follow me on
Sumit Khanna for more updates

Comprehensive Guide to PyTorch Framework

PyTorch is an open-source deep learning framework that provides a flexible and efficient platform for building and training neural networks. Developed by Facebook's AI Research lab (FAIR), PyTorch has gained widespread adoption in both academia and industry due to its dynamic computation graph, ease of use, and strong support for GPU acceleration. The latest version of PyTorch introduces several new features and performance optimizations that make it even more powerful and versatile.

Cheat Sheet of Top Useful PyTorch Methods

Method Name	Definition
`torch.tensor`	Creates a tensor from data
`torch.zeros`	Returns a tensor filled with zeros, with the shape defined by the argument
`torch.ones`	Returns a tensor filled with ones, with the shape defined by the argument
`torch.rand`	Returns a tensor filled with random numbers between 0 and 1
`torch.randn`	Returns a tensor filled with random numbers from a normal distribution
`torch.linspace`	Returns a 1D tensor with values spaced linearly within a given interval
`torch.arange`	Returns a 1D tensor with values spaced by a given step size
`torch.eye`	Returns a 2D tensor with ones on the diagonal and zeros elsewhere
`torch.matmul`	Performs matrix multiplication between two tensors
`torch.mm`	Performs matrix multiplication between two 2D tensors
`torch.bmm`	Performs batch matrix multiplication

Method Name	Definition
`torch.cat`	Concatenates a sequence of tensors along a specified dimension
`torch.stack`	Stacks a sequence of tensors along a new dimension
`torch.split`	Splits a tensor into chunks
`torch.chunk`	Splits a tensor into a specified number of chunks
`torch.reshape`	Returns a tensor with a new shape, with data unchanged
`torch.view`	Returns a new tensor with the same data but a different shape
`torch.transpose`	Returns a tensor with dimensions transposed
`torch.t`	Transposes the last two dimensions of a tensor
`torch.permute`	Permutes the dimensions of a tensor
`torch.unsqueeze`	Returns a tensor with a dimension of size one inserted at the specified position
`torch.squeeze`	Returns a tensor with all the dimensions of size 1 removed
`torch.flatten`	Flattens a tensor to a single dimension
`torch.norm`	Returns the norm of a tensor
`torch.max`	Returns the maximum value of all elements in a tensor
`torch.min`	Returns the minimum value of all elements in a tensor
`torch.sum`	Returns the sum of all elements in a tensor
`torch.mean`	Returns the mean of all elements in a tensor
`torch.std`	Returns the standard deviation of all elements in a tensor
`torch.var`	Returns the variance of all elements in a tensor
`torch.argmax`	Returns the indices of the maximum value of all elements along a specified axis
`torch.argmin`	Returns the indices of the minimum value of all elements along a specified axis
`torch.nn.Linear`	A fully connected layer in a neural network
`torch.nn.Conv2d`	A 2D convolutional layer in a neural network
`torch.nn.ReLU`	A ReLU activation function in a neural network
`torch.nn.Sigmoid`	A Sigmoid activation function in a neural network
`torch.nn.CrossEntropyLoss`	A cross-entropy loss function for classification problems

Method Name	Definition
`torch.nn.MSELoss`	A mean squared error loss function for regression problems
`torch.optim.SGD`	Stochastic gradient descent optimizer
`torch.optim.Adam`	Adam optimizer
`torch.optim.lr_scheduler.StepLR`	A learning rate scheduler that decays the learning rate by a factor every few epochs
`torch.autograd.grad`	Computes and returns the gradients of specified tensors
`torch.autograd.backward`	Computes the gradient of the current tensor w.r.t. graph leaves
`torch.cuda.is_available`	Returns a boolean indicating whether CUDA is available for GPU acceleration
`torch.cuda.device`	Context manager for selecting a CUDA device
`torch.save`	Saves an object to a disk file
`torch.load`	Loads an object saved with `torch.save` from a disk file
`torch.nn.Module`	Base class for all neural network modules
`torch.nn.Sequential`	A sequential container to stack multiple layers
`torch.nn.Dropout`	A layer that randomly zeroes some of the elements of the input tensor with a probability
`torch.nn.BatchNorm2d`	A layer that normalizes the input for each mini-batch
`torch.utils.data.DataLoader`	A data loader to load datasets in batches
`torch.utils.data.Dataset`	An abstract class representing a dataset

`torch.tensor`

The `torch.tensor` method is used to create a tensor from data. It is one of the most fundamental operations in PyTorch.

```python
import torch

# Create a tensor from a list
data = [1, 2, 3, 4]
tensor = torch.tensor(data)
print(tensor)
```

`torch.zeros`

The `torch.zeros` method returns a tensor filled with zeros. The shape of the tensor is defined by the input arguments.

```python
import torch

# Create a 2x3 tensor filled with zeros
zeros_tensor = torch.zeros(2, 3)
print(zeros_tensor)
```

`torch.ones`

The `torch.ones` method returns a tensor filled with ones. The shape of the tensor is defined by the input arguments.

```python
import torch

# Create a 2x3 tensor filled with ones
ones_tensor = torch.ones(2, 3)
print(ones_tensor)
```

`torch.rand`

The `torch.rand` method returns a tensor filled with random numbers between 0 and 1.

```python
import torch

# Create a 3x3 tensor filled with random numbers between 0 and 1
rand_tensor = torch.rand(3, 3)
print(rand_tensor)
```

`torch.randn`

The `torch.randn` method returns a tensor filled with random numbers from a normal distribution.

```python
import torch

# Create a 2x3 tensor filled with random numbers from a normal distribution
randn_tensor = torch.randn(2, 3)
print(randn_tensor)
```

`torch.linspace`

The `torch.linspace` method returns a 1D tensor with values spaced linearly within a given interval.

```python
import torch

# Create a tensor with 5 values spaced linearly between 0 and 10
linspace_tensor = torch.linspace(0, 10, steps=5)
print(linspace_tensor)
```

`torch.arange`

The `torch.arange` method returns a 1D tensor with values spaced by a given step size.

```python
import torch

# Create a tensor with values from 0 to 10 with a step size of 2
arange_tensor = torch.arange(0, 10, step=2)
print(arange_tensor)
```

`torch.eye`

The `torch.eye` method returns a 2D tensor with ones on the diagonal and zeros elsewhere (an identity matrix).

```python
import torch

# Create a 3x3 identity matrix
eye_tensor = torch.eye(3)
print(eye_tensor)
```

`torch.matmul`

The `torch.matmul` method performs matrix multiplication between two tensors.

```python
import torch

# Define two 2D tensors
tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[5, 6], [7, 8]])

# Perform matrix multiplication
matmul_result = torch.matmul(tensor1, tensor2)
print(matmul_result)
```

`torch.mm`

The `torch.mm` method is similar to `torch.matmul`, but specifically for 2D tensors.

```python
import torch

# Define two 2D tensors
tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[5, 6], [7, 8]])

# Perform matrix multiplication
mm_result = torch.mm(tensor1, tensor2)
print(mm_result)
```

`torch.bmm`

The `torch.bmm` method performs batch matrix multiplication.

```python
import torch

# Define two 3D tensors
tensor1 = torch.randn(10, 3, 4)
tensor2 = torch.randn(10, 4, 5)

# Perform batch matrix multiplication
bmm_result = torch.bmm(tensor1, tensor2)
print(bmm_result.size())  # Should print (10, 3, 5)
```

`

torch.cat`

The `torch.cat` method concatenates a sequence of tensors along a specified dimension.

```python
import torch

# Define two tensors
tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[5, 6]])

# Concatenate along the first dimension
cat_result = torch.cat((tensor1, tensor2), dim=0)
print(cat_result)
```

`torch.stack`

The `torch.stack` method stacks a sequence of tensors along a new dimension.

```python
import torch

# Define two tensors
tensor1 = torch.tensor([1, 2])
tensor2 = torch.tensor([3, 4])

# Stack along a new dimension
stack_result = torch.stack((tensor1, tensor2), dim=0)
print(stack_result)
```

`torch.split`

The `torch.split` method splits a tensor into chunks.

```python
import torch

# Define a tensor
tensor = torch.tensor([1, 2, 3, 4, 5, 6])

# Split into three chunks
split_result = torch.split(tensor, 2)
for chunk in split_result:
    print(chunk)
```

`torch.chunk`

The `torch.chunk` method splits a tensor into a specified number of chunks.

```python
import torch

# Define a tensor
tensor = torch.tensor([1, 2, 3, 4, 5, 6])

# Chunk the tensor into three parts
chunk_result = torch.chunk(tensor, 3)
for chunk in chunk_result:
    print(chunk)
```

`torch.reshape`

The `torch.reshape` method returns a tensor with a new shape, without changing the data.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Reshape the tensor to a different shape
reshape_result = torch.reshape(tensor, (3, 2))
print(reshape_result)
```

`torch.view`

The `torch.view` method returns a new tensor with the same data but a different shape.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# View the tensor with a different shape
view_result = tensor.view(3, 2)
print(view_result)
```

`torch.transpose`

The `torch.transpose` method returns a tensor with dimensions transposed.

```python
import torch

# Define a 2D tensor
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Transpose the tensor
transpose_result = torch.transpose(tensor, 0, 1)
print(transpose_result)
```

`torch.t`

The `torch.t` method transposes the last two dimensions of a tensor.

```python
import torch

# Define a 2D tensor
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Transpose the last two dimensions
t_result = torch.t(tensor)
print(t_result)
```

`torch.permute`

The `torch.permute` method permutes the dimensions of a tensor.

```python
import torch

# Define a 3D tensor
tensor = torch.randn(2, 3, 4)

# Permute the dimensions
permute_result = torch.permute(tensor, (2, 0, 1))
print(permute_result.size())  # Should print torch.Size([4, 2, 3])
```

`torch.unsqueeze`

The `torch.unsqueeze` method returns a tensor with a dimension of size one inserted at the specified position.

```python
import torch

# Define a tensor
tensor = torch.tensor([1, 2, 3, 4])

# Unsqueeze the tensor to add a new dimension
unsqueeze_result = torch.unsqueeze(tensor, dim=0)
print(unsqueeze_result)
```

`torch.squeeze`

The `torch.squeeze` method returns a tensor with all the dimensions of size 1 removed.

```python
import torch

# Define a tensor
tensor = torch.tensor([[[1, 2, 3, 4]]])

# Squeeze the tensor to remove dimensions of size 1
squeeze_result = torch.squeeze(tensor)
print(squeeze_result)
```

`torch.flatten`

The `torch.flatten` method flattens a tensor to a single dimension.

```python
import torch

# Define a 2D tensor
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Flatten the tensor
flatten_result = torch.flatten(tensor)
print(flatten_result)
```

`torch.norm`

The `torch.norm` method returns the norm of a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([1, 2, 3, 4], dtype=torch.float32)

# Calculate the norm of the tensor
norm_result = torch.norm(tensor)
print(norm_result)
```

`torch.max`

The `torch.max` method returns the maximum value of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]])

# Find the maximum value
max_result = torch.max(tensor)
print(max_result)
```

`torch.min`

The `torch.min` method returns the minimum value of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]])

# Find the minimum value
min_result = torch.min(tensor)
print(min_result)
```

`torch.sum`

The `torch.sum` method returns the sum of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]])

# Find the sum of all elements
sum_result = torch.sum(tensor)
print(sum_result)
```

`torch.mean`

The `torch.mean` method returns the mean of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)

# Find the mean of all elements
mean_result = torch.mean(tensor)
print(mean_result)
```

`torch.std`

The `torch.std` method returns the standard deviation of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)

# Find the standard deviation of all elements
std_result = torch.std(tensor)
print(std_result)
```

`torch.var`

The `torch.var` method returns the variance of all elements in a tensor.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)

# Find the variance of all elements
var_result = torch.var(tensor)
print(var_result)
```

`torch.argmax`

The `torch.argmax` method returns the indices of the maximum value of all elements along a specified axis.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]])

# Find the index of the maximum value
argmax_result = torch.argmax(tensor)
print(argmax_result)
```

`torch.argmin`

The `torch.argmin` method returns the indices of the minimum value of all elements along a specified axis.

```python
import torch

# Define a tensor
tensor = torch.tensor([[1, 2], [3, 4]])

# Find the index of the minimum value
argmin_result = torch.argmin(tensor)
print(argmin_result)
```

`torch.nn.Linear`

The `torch.nn.Linear` method creates a fully connected layer in a neural network.

```python
import torch
import torch.nn as nn

# Define a fully connected layer
linear = nn.Linear(in_features=2, out_features=3)

# Define an input tensor
input_tensor = torch.tensor([[1.0, 2.0]])

# Pass the input tensor through the linear layer
output_tensor = linear(input_tensor)
print(output_tensor)
```

`torch.nn.Conv2d`

The `torch.nn.Conv2d` method creates a 2D convolutional layer in a neural network.

```
import torch
import torch.nn as nn

# Define a 2D convolutional layer
conv2d = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3)

# Define an input tensor
input_tensor = torch.randn(1, 1, 5, 5)

# Pass the input tensor through the convolutional layer
output_tensor = conv2d(input_tensor)
print(output_tensor.size())  # Should print torch.Size([1, 1, 3, 3])
```

`torch.nn.ReLU`

The `torch.nn.ReLU` method applies the ReLU activation function in a neural network.

```
import torch
import torch.nn as nn

# Define a ReLU activation function
relu = nn.ReLU()

# Define an input tensor
input_tensor = torch.tensor([-1.0, 0.0, 1.0])

# Apply the ReLU activation function
output_tensor = relu(input_tensor)
print(output_tensor)
```

`torch.nn.Sigmoid`

The `torch.nn.Sigmoid` method applies the Sigmoid activation function in a neural network.

```
import torch
import torch.nn as nn

# Define a Sigmoid activation function
sigmoid = nn.Sigmoid()

# Define an input tensor
input_tensor = torch.tensor([-1.0, 0.0, 1.0])

# Apply the Sigmoid activation function
output_tensor = sigmoid(input_tensor)
print(output_tensor)
```

`torch.nn.CrossEntropyLoss`

The `torch.nn.CrossEntropyLoss` method creates a cross-entropy loss function for classification problems.

```
import torch
import torch.nn as nn

# Define a cross-entropy loss function
loss_fn = nn.CrossEntropyLoss()

# Define the inputs (logits) and target labels
inputs = torch.tensor([[0.5, 1.5, 2.0]])
targets = torch.tensor([2])

# Compute the loss
loss = loss_fn(inputs, targets)
print(loss)
```

`torch.nn.MSELoss`

The `torch.nn.MSELoss` method creates a mean squared error loss function for regression problems.

```
import torch
import torch.nn as nn

# Define a mean squared error loss function
loss_fn = nn.MSELoss()

# Define the inputs and target values
inputs = torch.tensor([0.5, 1.5, 2.0])
targets = torch.tensor([1.0, 2.0, 3.0])

# Compute the loss
loss = loss_fn(inputs, targets)
print(loss)
```

`torch.optim.SGD`

The `torch.optim.SGD` method creates a stochastic gradient descent optimizer.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple linear model
model = nn.Linear(in_features=2, out_features=1)

# Define a stochastic gradient descent optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Define a simple input tensor and target
input_tensor = torch.tensor([[1.0, 2.0]])
target = torch.tensor([[1.0]])

# Forward pass
output = model(input_tensor)

# Compute loss
loss = nn.MSELoss()(output, target)

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

print(output)
```

`torch.optim.Adam`

The `torch.optim.Adam` method creates an Adam optimizer.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple linear model
model = nn.Linear(in_features=2, out_features=1)

# Define an Adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Define a simple input tensor and target
input_tensor = torch.tensor([[1.0, 2.0]])
target = torch.tensor([[1.0]])

# Forward pass
output = model(input_tensor)

# Compute loss
loss = nn.MSELoss()(output, target)

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

print(output)
```

`torch.optim.lr_scheduler.StepLR`

The `torch.optim.lr_scheduler.StepLR` method creates a learning rate scheduler that decays the learning rate by a factor every few epochs.

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple linear model
model = nn.Linear(in_features=2, out_features=1)

# Define an Adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Define a learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

# Define a simple input tensor and target
input_tensor = torch.tensor([[1.0, 2.0]])
target = torch.tensor([[1.0]])

# Training loop
for epoch in range(20):
    # Forward pass
    output = model(input_tensor)

    # Compute loss
    loss = nn.MSELoss()(output, target)

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

    # Step the learning rate scheduler
    scheduler.step()

    print(f"Epoch {epoch+1}, Loss: {loss.item()}, LR: {scheduler.get_last_lr()}")
```

`torch.autograd.grad`

The `torch.autograd.grad` method computes and returns the gradients of specified tensors.

```python
import torch

# Define a simple function
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2

# Compute gradients
grad = torch.autograd.grad(outputs=y, inputs=x)
print(grad)
```

`torch.autograd.backward`

The `torch.autograd.backward` method computes the gradient of the current tensor w.r.t. graph leaves.

```
import torch

# Define a simple function
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2

# Perform backpropagation
y.backward()

# Print the gradient
print(x.grad)
```

`torch.cuda.is_available`

The `torch.cuda.is_available` method returns a boolean indicating whether CUDA is available for GPU acceleration.

```
import torch

# Check if CUDA is available
cuda_available = torch.cuda.is_available()
print(cuda_available)
```

`torch.cuda.device`

The `torch.cuda.device` method is a context manager for selecting a CUDA device.

```
import torch

if torch.cuda.is_available():
    with torch.cuda.device(0):
        # Create a tensor and move it to the GPU
        tensor = torch.tensor([1, 2, 3], device=torch.device('cuda'))
        print(tensor)
```

`torch.save`

The `torch.save` method saves an object to a disk file.

```python
import torch

# Define a simple tensor
tensor = torch.tensor([1, 2, 3])

# Save the tensor to a file
torch.save(tensor, 'tensor.pth')
```

`torch.load`

The `torch.load` method loads an object saved with `torch.save` from a disk file.

```python
import torch

# Load the tensor from a file
tensor = torch.load('tensor.pth')
print(tensor)
```

`torch.nn.Module`

The `torch.nn.Module` method is the base class for all neural network modules.

```python
import torch
import torch.nn as nn

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)

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

# Instantiate the neural network
model = SimpleNN()
print(model)
```

`torch.nn.Sequential`

The `torch.nn.Sequential` method is a sequential container to stack multiple layers.

```
import torch
import torch.nn as nn

# Define a simple sequential model
model = nn.Sequential(
    nn.Linear(2, 2),
    nn.ReLU(),
    nn.Linear(2, 1)
)
print(model)
```

`torch.nn.Dropout`

The `torch.nn.Dropout` method creates a dropout layer that randomly zeroes some of the elements of the input tensor with a probability.

```
import torch
import torch.nn as nn

# Define a dropout layer
dropout = nn.Dropout(p=0.5)

# Define an input tensor
input_tensor = torch.randn(2, 3)

# Apply dropout
output_tensor = dropout(input_tensor)
print(output_tensor)
```

`torch.nn.BatchNorm2d`

The `torch.nn.BatchNorm2d` method creates a layer that normalizes the input for each mini-batch.

```
import torch
import torch.nn as nn

# Define a batch normalization layer
batch_norm = nn.BatchNorm2d(num_features=3)

# Define an input tensor
input_tensor = torch.randn(2, 3, 4, 4)

# Apply batch normalization
output_tensor = batch_norm(input_tensor)
print(output_tensor)
```

`torch.utils.data.DataLoader`

The `torch.utils.data.DataLoader` method creates a data loader to load datasets in batches.

```python
import torch
from torch.utils.data import DataLoader, TensorDataset

# Define a simple dataset
data = torch.randn(100, 2)
targets = torch.randn(100, 1)
dataset = TensorDataset(data, targets)

# Create a data loader
data_loader = DataLoader(dataset, batch_size=10, shuffle=True)

# Iterate through the data loader
for batch_data, batch_targets in data_loader:
    print(batch_data.size(), batch_targets.size())
```

`torch.utils.data.Dataset`

The `torch.utils.data.Dataset` method is an abstract class representing a dataset.

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

# Define a custom dataset
class CustomDataset(Dataset):
    def __init__(self, data, targets):
        self.data = data
        self.targets = targets

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

    def __getitem__(self, index):
        return self.data[index], self.targets[index]

# Create a dataset
data = torch.randn(100, 2)
targets = torch.randn(100, 1)
dataset = CustomDataset(data, targets)

# Get the length of the dataset
print(len(dataset))

# Get the first item from the dataset
print(dataset[0])
```

Code Examples Using PyTorch for Common Use Cases

1. Simple Linear Regression

```
import torch
import torch.nn as nn
import torch.optim as optim

#


 Create a dataset
x_train = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
y_train = torch.tensor([[2.0], [4.0], [6.0], [8.0]])

# Define a linear model
model = nn.Linear(in_features=1, out_features=1)

# Define loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(100):
    # Forward pass
    y_pred = model(x_train)

    # Compute loss
    loss = criterion(y_pred, y_train)

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

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")

# Print model parameters
print(f"Weights: {model.weight.item()}, Bias: {model.bias.item()}")
```

2. Multilayer Perceptron for Classification

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Create a classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
y_test = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

# Define a multilayer perceptron model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(20, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

# Instantiate the model, define loss and optimizer
model = MLP()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(100):
    # Forward pass
    y_pred = model(X_train)

    # Compute loss
    loss = criterion(y_pred, y_train)

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

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")

# Evaluate the model
with torch.no_grad():
    y_pred_test = model(X_test)
    y_pred_label = y_pred_test.round()
    accuracy = (y_pred_label.eq(y_test).sum() / y_test.shape[0]).item()
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

3. Convolutional Neural Network for Image Classification

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64*7*7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.maxpool(x)
        x = self.relu(self.conv2(x))
        x = self.maxpool(x)
        x = x.view(-1, 64*7*7)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Prepare the dataset and data loader
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=Tru
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Instantiate the model, define loss and optimizer
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(5):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, target)

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

        if batch_idx % 100 == 0:
            print(f"Epoch {epoch+1}, Batch {batch_idx+1}, Loss: {loss.item()}")
```

4. Transfer Learning with Pretrained Models

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Load a pretrained ResNet18 model
model = torchvision.models.resnet18(pretrained=True)

# Freeze all layers except the final layer
for param in model.parameters():
    param.requires_grad = False

# Replace the final layer with a new fully connected layer
model.fc = nn.Linear(model.fc.in_features, 10)

# Prepare the dataset and data loader
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=transform, download=T
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)

# Training loop
for epoch in range(5):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, target)

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

        if batch_idx % 100 == 0:
            print(f"Epoch {epoch+1}, Batch {batch_idx+1}, Loss: {loss.item()}")
```

5. Recurrent Neural Network for Sequence Modeling

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple RNN model
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out

# Create some dummy data
X_train = torch.randn(100, 10, 1)  # 100 sequences of length 10
y_train = torch.randn(100, 1)       # 100 target values

# Instantiate the model, define loss and optimizer
model = SimpleRNN(input_size=1, hidden_size=20, output_size=1)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
for epoch in range(100):
    # Forward pass
    y_pred = model(X_train)

    # Compute loss
    loss = criterion(y_pred, y_train)

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

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

6. Custom Dataset for Loading Images

```python
import torch
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import os

# Define a custom dataset for loading images
class CustomImageDataset(Dataset):
    def __init__(self, img_dir, transform=None):
        self.img_dir = img_dir
        self.transform = transform
        self.img_names = os.listdir(img_dir)

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

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_names[idx])
        image = Image.open(img_path)
        if self.transform:
            image = self.transform(image)
        return image

# Instantiate the dataset and data loader
transform = transforms.Compose([transforms.Resize((64, 64)), transforms.ToTensor()])
dataset = CustomImageDataset(img_dir='./

images', transform=transform)
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)

# Iterate through the data loader
for batch in data_loader:
    print(batch.size())
```

7. Autoencoder for Dimensionality Reduction

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define a simple autoencoder model
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Linear(64, 12),
            nn.ReLU(),
            nn.Linear(12, 3)
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(),
            nn.Linear(12, 64),
            nn.ReLU(),
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.Linear(256, 784),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Instantiate the model, define loss and optimizer
model = Autoencoder()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(5):
    for batch_idx, (data, _) in enumerate(train_loader):
        # Flatten the data
        data = data.view(-1, 784)

        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, data)

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

        if batch_idx % 100 == 0:
            print(f"Epoch {epoch+1}, Batch {batch_idx+1}, Loss: {loss.item()}")
```

8. Generative Adversarial Network (GAN) for Image Generation

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Define the generator model
class Generator(nn.Module):
    def __init__(self, z_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(z_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        return self.model(x)

# Define the discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

# Hyperparameters
z_dim = 64
batch_size = 64
lr = 0.0002
```

```
# Load the dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=Tru
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

# Instantiate the models and optimizers
generator = Generator(z_dim)
discriminator = Discriminator()
g_optimizer = optim.Adam(generator.parameters(), lr=lr)
d_optimizer = optim.Adam(discriminator.parameters(), lr=lr)
criterion = nn.BCELoss()

# Training loop
for epoch in range(5):
    for batch_idx, (real_images, _) in enumerate(train_loader):
        # Train the discriminator
        real_images = real_images.view(-1, 784)
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        d_optimizer.zero_grad()
        outputs = discriminator(real_images)
        real_loss = criterion(outputs, real_labels)
        real_loss.backward()

        z = torch.randn(batch_size, z_dim)
        fake_images = generator(z)
        outputs = discriminator(fake_images)
        fake_loss = criterion(outputs, fake_labels)
        fake_loss.backward()
        d_optimizer.step()

        # Train the generator
        g_optimizer.zero_grad()
        z = torch.randn(batch_size, z_dim)
        fake_images = generator(z)
        outputs = discriminator(fake_images)
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        g_optimizer.step()

        if batch_idx % 100 == 0:
            print(f"Epoch {epoch+1}, Batch {batch_idx+1}, D Loss: {real_loss+fake_loss}, G Loss: {g_los
```

9. Sequence-to-Sequence Model for Translation

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define an encoder-decoder model
class Seq2Seq(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(Seq2Seq, self).__init__()
        self.encoder = nn.RNN(input_dim, hidden_dim, batch_first=True)
        self.decoder = nn.RNN(hidden_dim, output_dim, batch_first=True)

    def forward(self, x):
        _, hidden = self.encoder(x)
        output, _ = self.decoder(hidden)
        return output

# Create some dummy data
X_train = torch.randn(100, 10, 5)  # 100 sequences of length 10 with 5 features
y_train = torch.randn(100, 10, 3)  # 100 sequences of length 10 with 3 features

# Instantiate the model, define loss and optimizer
model = Seq2Seq(input_dim=5, hidden_dim=20, output_dim=3)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
for epoch in range(100):
    # Forward pass
    y_pred = model(X_train)

    # Compute loss
    loss = criterion(y_pred, y_train)

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

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

10. Object Detection with YOLOv5

```
# Note: This example assumes that you have YOLOv5 set up in your environment.
# YOLOv5 can be installed via the official GitHub repository: https://github.com/ultralytics/yolov5

import torch
from yolov5 import YOLOv5

# Load the YOLOv5 model (pretrained)
model = YOLOv5("yolov5s.pt")

# Perform inference on an image
img = "image.jpg"
results = model(img)

# Print results
results.show()
results.print()

# Extract the detected objects' bounding boxes and labels
boxes = results.xyxy[0].numpy()  # bounding boxes
labels = results.names  # labels
```

Follow me on

Sumit Khanna for more updates