S15 - CapStone

Start Assignment

- Due 9 Oct by 23:59
- Points 2,500
- Submitting a text entry box or a website url
- Available 25 Sep at 7:00 - 9 Oct at 23:59

Session 15 - *CAPSTONE*

Project Title: “Custom DataLoader for Multimodal Datasets”

Project Proposal

Description

In this project, you will build a flexible `DataLoader` class that can load, preprocess, and manage different types of datasets commonly used in AI and machine learning projects, including:

- **Image datasets**: CIFAR-10, CIFAR-100, MNIST
- **Text datasets**: Small text datasets
- **Structured data**: CSV files
- **Unstructured data**: Folders containing multiple files of various formats

Your DataLoader should:

- **Download datasets from online sources** if they are not already present locally.
- **Handle different file formats** and organize data appropriately.
- **Provide data in batches** for efficient processing.
- **Support data augmentation and preprocessing** steps.
- **Be extensible** to accommodate new data types and sources.

This project will give you hands-on experience with data preprocessing, which is a critical step in AI projects, while applying the Python concepts you’ve learned.

Project Breakdown by Concepts

1. **Basics**
   - **Classes**: Implement the DataLoader as a class.
   - **Functions**: Define methods for downloading, loading, and preprocessing data.
   - **Loops and Conditionals**: Iterate over files and data, check for file existence.
2. **Object Mutability and Interning**
   - Manage mutable data structures like lists and dictionaries to store datasets.
   - Understand how changes to objects affect data integrity.
3. **Numeric Types I & II**
   - Handle numerical computations during preprocessing (e.g., normalization).
   - Use booleans and comparison operators for condition checks.
4. **Functional Parameters**
   - Create flexible methods that accept various parameters for data transformations.
   - Use `**kwargs` to pass optional preprocessing functions.
5. **First-Class Functions Part I & II**
   - Use lambda functions for simple data transformations.
   - Employ `map` and `filter` to process data iterables.
6. **Scopes and Closures**
   - Maintain state within data loading functions using closures if necessary.
7. **Decorators**
   - Implement decorators to log the time taken for data loading and preprocessing.
   - Use decorators for caching data to avoid redundant computations.
8. **Tuples and NamedTuples**
   - Use namedtuples to represent data samples with features and labels.
9. **Modules, Packages, and Namespaces**
   - Organize code into modules for loaders, preprocessors, utils, etc.
   - Use packages to separate different components logically.
10. **f-Strings, Timing Functions, and Command Line Arguments**
    - Use f-strings for informative print statements.
    - Accept command-line arguments for configuration (e.g., dataset selection).
11. **Sequence Types I & II and Advanced List Comprehension**

- Manage collections of data samples.
- Use list comprehensions for efficient data processing.

12. **Iterables and Iterators**
    - Implement an iterator protocol in the DataLoader to iterate over data batches.
13. **Generators and Iteration Tools**
    - Use generators to load data on-the-fly without consuming excessive memory.
14. **Context Managers**
    - Use context managers when opening files to ensure they are properly closed.
15. **Exception Handling (Try/Except)**
    - Handle exceptions during file I/O and data processing to prevent crashes.

Starter Code

To help you get started, here's the basic structure of your project with some starter code snippets. You can expand upon these templates and modify them as needed.

Project Structure

```
project_root/
├── dataloader/
│   ├── __init__.py
│   ├── dataloader.py
│   ├── preprocessors.py
│   └── utils.py
├── datasets/
│   └── (Your datasets will be stored here)
├── tests/
│   └── test_dataloader.py
├── main.py
└── requirements.txt
```

1. dataloader/init.py

This file can be left empty or used to initialize the package.

```
# dataloader/__init__.py

from .dataloader import DataLoader
```

2. dataloader/dataloader.py

```
# dataloader/dataloader.py

import os
import sys
import requests
from collections import namedtuple
```

```
from contextlib import contextmanager
from .preprocessors import default_preprocess
from .utils import download_file, timer

DataSample = namedtuple('DataSample', ['features', 'label'])

class DataLoader:
    def __init__(self, dataset_name='MNIST', batch_size=32, shuffle=True, **kwargs):
        self.dataset_name = dataset_name
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.kwargs = kwargs
        self.data = []
        self.index = 0
        self.load_data()

    @timer
    def load_data(self):
        if not os.path.exists(f'datasets/{self.dataset_name}'):
            self.download_dataset()
        # Implement data loading logic
        self.data = self.preprocess_data(self.read_data())

    @timer
    def download_dataset(self):
        # Implement dataset download logic
        print(f"Downloading {self.dataset_name} dataset...")
        # Use download_file from utils.py

    def read_data(self):
        # Implement data reading logic
        return []

    def preprocess_data(self, data):
        # Implement data preprocessing logic
        preprocess_func = self.kwargs.get('preprocess_func', default_preprocess)
        return [preprocess_func(sample) for sample in data]

    def __iter__(self):
        self.index = 0
        if self.shuffle:
            import random
            random.shuffle(self.data)
        return self

    def __next__(self):
        if self.index < len(self.data):
            batch = self.data[self.index:self.index + self.batch_size]
            self.index += self.batch_size
            return batch
        else:
            raise StopIteration
```

3. dataloader/preprocessors.py

```
# dataloader/preprocessors.py

def default_preprocess(sample):
    # Implement default preprocessing
    return sample

def normalize(sample):
    # Implement normalization logic
    return sample

def augment(sample):
    # Implement data augmentation logic
    return sample
```

4. dataloader/utils.py

```
# dataloader/utils.py

import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        print(f"Function '{func.__name__}' took {time.time() - start_time:.2f}s to complete.")
        return result
    return wrapper

def download_file(url, dest_path):
    # Implement file download logic
    pass
```

5. main.py (http://main.py/)

```
# main.py

import sys
from dataloader import DataLoader

def main():
    # Parse command-line arguments
    dataset_name = sys.argv[1] if len(sys.argv) > 1 else 'MNIST'
    batch_size = int(sys.argv[2]) if len(sys.argv) > 2 else 32

    # Initialize DataLoader
    data_loader = DataLoader(dataset_name=dataset_name, batch_size=batch_size)

    # Iterate over data
    for batch in data_loader:
        # Process the batch
        print(f"Processing batch of size {len(batch)}")

if __name__ == '__main__':
    main()
```

Test Cases Plan

Below are the test cases in actual Python code that you can use to test your project. These test cases are designed to check both the functionality of your DataLoader and the application of the specific Python concepts.

tests/test_dataloader.py

```
# tests/test_dataloader.py

import unittest
import os
import sys
sys.path.append('..')  # Adjust the path as needed
from dataloader import DataLoader
from dataloader.utils import timer
from dataloader.preprocessors import default_preprocess
from collections import namedtuple

class TestDataLoader(unittest.TestCase):
```

```
    def test_dataloader_initialization(self):
        """Test Case 1: DataLoader Class Initialization and Data Downloading"""
        data_loader = DataLoader(dataset_name='MNIST', batch_size=32)
        self.assertIsInstance(data_loader, DataLoader)
        self.assertEqual(data_loader.dataset_name, 'MNIST')
        self.assertTrue(os.path.exists('datasets/MNIST'))

    def test_flexible_method_parameters(self):
        """Test Case 2: Flexible Method Parameters"""
        def custom_preprocess(sample, factor=2):
            # Example of using positional and keyword arguments
            return sample * factor

        data_loader = DataLoader(preprocess_func=custom_preprocess)
        self.assertTrue(callable(data_loader.kwargs['preprocess_func']))

    def test_data_normalization(self):
        """Test Case 3: Data Normalization"""
        from dataloader.preprocessors import normalize

        sample_data = [0, 128, 255]
        normalized_data = list(map(normalize, sample_data))
        for value in normalized_data:
            self.assertTrue(0.0 <= value <= 1.0)

    def test_lambda_functions(self):
        """Test Case 4: Use of Lambda Functions"""
        data = [1, 2, 3, 4, 5]
        transformed_data = list(map(lambda x: x * 2, data))
        self.assertEqual(transformed_data, [2, 4, 6, 8, 10])

    def test_closures(self):
        """Test Case 5: Implementation of Closures"""
        def make_multiplier(factor):
            def multiply(number):
                return number * factor
            return multiply

        times_three = make_multiplier(3)
        self.assertEqual(times_three(10), 30)

    def test_decorators(self):
        """Test Case 6: Use of Decorators for Logging"""
        @timer
        def sample_function():
            return True

        result = sample_function()
        self.assertTrue(result)

    def test_namedtuples(self):
        """Test Case 7: Use of NamedTuples"""
        DataSample = namedtuple('DataSample', ['features', 'label'])
        sample = DataSample(features=[1, 2, 3], label=0)
        self.assertEqual(sample.features, [1, 2, 3])
        self.assertEqual(sample.label, 0)

    def test_project_modularization(self):
        """Test Case 8: Proper Project Modularization"""
        # Check if modules can be imported
        try:
            from dataloader import dataloader
            from dataloader import preprocessors
            from dataloader import utils
        except ImportError:
            self.fail("Modules not properly organized")

    def test_fstrings(self):
        """Test Case 9: Use of f-Strings"""
        name = 'DataLoader'
        message = f"Initializing {name}"
```

```
        self.assertEqual(message, "Initializing DataLoader")

    def test_custom_iterator(self):
        """Test Case 10: Implementation of Custom Iterator"""
        data_loader = DataLoader(batch_size=10)
        iterator = iter(data_loader)
        batch = next(iterator)
        self.assertEqual(len(batch), 10)

    def test_generators(self):
        """Test Case 11: Use of Generators for Lazy Loading"""
        def data_generator():
            for i in range(10):
                yield i

        gen = data_generator()
        self.assertEqual(next(gen), 0)
        self.assertEqual(next(gen), 1)

    def test_list_comprehensions(self):
        """Test Case 12: Use of List Comprehensions"""
        data = [i for i in range(5)]
        self.assertEqual(data, [0, 1, 2, 3, 4])

    def test_context_managers(self):
        """Test Case 13: Use of Context Managers"""
        try:
            with open('testfile.txt', 'w') as f:
                f.write('Test')
            self.assertTrue(os.path.exists('testfile.txt'))
        finally:
            if os.path.exists('testfile.txt'):
                os.remove('testfile.txt')

    def test_exception_handling(self):
        """Test Case 14: Exception Handling"""
        try:
            x = 1 / 0
        except ZeroDivisionError:
            self.assertTrue(True)
        else:
            self.fail("ZeroDivisionError not raised")

    def test_command_line_arguments(self):
        """Test Case 15: Command-Line Arguments"""
        # Simulate command-line arguments
        sys.argv = ['main.py', 'CIFAR-10', '64']
        from main import main
        try:
            main()
            self.assertTrue(True)
        except Exception as e:
            self.fail(f"main() raised Exception unexpectedly: {e}")

if __name__ == '__main__':
    unittest.main()
```

Instructions for Running Test Cases

1. **Ensure your project structure is as specified.**
2. **Install any required dependencies.**
   - Create a `requirements.txt` file with necessary packages, e.g., `requests`, `pillow`, etc.
   - Install dependencies using:

```
pip install -r requirements.txt
```

3. **Run the test cases.**
   - Navigate to the project root directory.
   - Run the tests using:

```
python -m unittest tests/test_dataloader.py
```

By following this plan, you'll have a project that not only meets the requirements but also helps you apply and reinforce the Python concepts you've learned. The test cases provided will help you verify the correctness of your implementation and ensure that you're making use of the required concepts.

Good luck with your project! You can find all the files here (https://canvas.instructure.com/courses/9532398/files/271084320?wrap=1) (https://canvas.instructure.com/courses/9532398/files/271084320/download?download_frd=1) .

Submission

Once done, upload to github and share back the actions URL with us. Your readme MUST have a screenshot of the actions test results as well as a label that all tests were cleared.

VIDEOS

STUDIO

GM