CYBERAI DEEP COURSE

Month 1: Machine Learning Foundations

Python • C • Rust

Version 1.0
Course Duration: 28 Days
Target Audience: Intermediate to Advanced Programmers
Prerequisites: Basic programming knowledge in at least one language

---

TABLE OF CONTENTS

WEEK 3: ADVANCED TECHNIQUES (Days 15-21)

WEEK 4: INTEGRATION & OPTIMIZATION (Days 22-28)

COURSE OVERVIEW

This intensive 28-day course provides a deep dive into machine learning fundamentals using three powerful programming languages: Python, C, and Rust. Each language offers unique advantages:

- **Python**: Rapid prototyping, extensive libraries, research-friendly
- **C**: Maximum performance, system-level control, embedded systems
- **Rust**: Memory safety, concurrent processing, modern systems programming

Why Multi-Language ML?

1. **Performance Optimization**: Different algorithms benefit from different language strengths
2. **Production Readiness**: Real-world systems often require multiple languages
3. **Career Versatility**: Understanding multiple paradigms makes you invaluable

4. **Deep Understanding**: Implementing algorithms from scratch builds intuition

Course Philosophy

This course emphasizes:

- **Implementation over Theory**: Build working code first, understand theory through practice
- **Performance Awareness**: Always consider computational complexity and optimization
- **Safety & Reliability**: Learn to write robust, production-ready ML code
- **Cross-Platform Skills**: Develop expertise that transfers across ecosystems

---

LEARNING OBJECTIVES

By the end of Month 1, you will be able to:

Technical Skills

1. Implement fundamental ML algorithms from scratch in Python, C, and Rust
2. Compare performance characteristics across languages
3. Design memory-efficient data structures for large datasets
4. Optimize code for different hardware architectures
5. Create Foreign Function Interfaces (FFI) for multi-language projects

Conceptual Understanding

1. Explain the mathematical foundations of core ML algorithms
2. Analyze time and space complexity of different approaches
3. Choose appropriate algorithms for specific problem domains
4. Evaluate model performance using multiple metrics
5. Design robust training and evaluation pipelines

Professional Development

1. Write clean, maintainable, and well-documented ML code
2. Implement comprehensive testing strategies for ML systems
3. Create reproducible experiments and benchmarks
4. Develop debugging skills for complex ML pipelines
5. Build a portfolio of cross-language ML implementations

COURSE STRUCTURE

Daily Lesson Format

Each lesson follows a consistent structure:

1. **Concept Introduction** (15 minutes)
   - Theory overview with visual diagrams
   - Mathematical foundations
   - Real-world applications
2. **Implementation Guide** (45 minutes)
   - Step-by-step code development
   - Language-specific considerations
   - Best practices and common pitfalls
3. **Hands-on Practice** (30 minutes)
   - Guided exercises
   - Code modifications and experiments
   - Performance analysis
4. **Reflection & Analysis** (15 minutes)
   - Critical thinking questions
   - Performance comparisons
   - Next steps and extensions

Assessment Strategy

- **Weekly Quizzes**: 25% of grade
- **Daily Reflections**: 25% of grade
- **Code Implementation**: 40% of grade
- **Final Project**: 10% of grade

DAY 1: INTRODUCTION TO MACHINE LEARNING PARADIGMS

Learning Objectives

- Understand the three main ML paradigms

- Identify appropriate use cases for each approach
- Set up development environments for all three languages

Concept Introduction

Machine Learning can be categorized into three main paradigms:

```
SUPERVISED LEARNING
├── Regression
│   ├── Linear Regression
│   ├── Polynomial Regression
│   └── Neural Networks
└── Classification
    ├── Logistic Regression
    ├── Decision Trees
    └── Support Vector Machines

UNSUPERVISED LEARNING
├── Clustering
│   ├── K-Means
│   ├── Hierarchical
│   └── DBSCAN
└── Dimensionality Reduction
    ├── PCA
    ├── t-SNE
    └── Autoencoders

REINFORCEMENT LEARNING
├── Model-Free Methods
│   ├── Q-Learning
│   └── Policy Gradient
└── Model-Based Methods
    ├── Monte Carlo Tree Search
    └── Dynamic Programming
```

Implementation Guide

Python Environment Setup

```
python
```

```
# requirements.txt
numpy==1.24.3
pandas==2.0.3
scikit-learn==1.3.0
matplotlib==3.7.2
jupyter==1.0.0
pytest==7.4.0

# Installation
pip install -r requirements.txt

# Verification script
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt

print(f"NumPy version: {np.__version__}")
print(f"Pandas version: {pd.__version__}")
print(f"Scikit-learn version: {sklearn.__version__}")

# Test basic functionality
data = np.random.randn(1000, 5)
df = pd.DataFrame(data, columns=['A', 'B', 'C', 'D', 'E'])
print(f"Test data shape: {df.shape}")
print("Python environment ready!")
```

C Development Setup

```
c
```

```c
// test_environment.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Basic matrix structure for ML operations
typedef struct {
    double **data;
    int rows;
    int cols;
} Matrix;

Matrix* create_matrix(int rows, int cols) {
    Matrix *mat = malloc(sizeof(Matrix));
    mat->rows = rows;
    mat->cols = cols;
    mat->data = malloc(rows * sizeof(double*));

    for(int i = 0; i < rows; i++) {
        mat->data[i] = malloc(cols * sizeof(double));
    }
    return mat;
}

void free_matrix(Matrix *mat) {
    for(int i = 0; i < mat->rows; i++) {
        free(mat->data[i]);
    }
    free(mat->data);
    free(mat);
}

void print_matrix(Matrix *mat) {
    for(int i = 0; i < mat->rows; i++) {
        for(int j = 0; j < mat->cols; j++) {
            printf("%.2f ", mat->data[i][j]);
        }
        printf("\n");
    }
}

int main() {
```

```c
    printf("C ML Environment Test\n");

    // Create test matrix
    Matrix *test = create_matrix(3, 3);

    // Fill with test data
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            test->data[i][j] = (double)(i * 3 + j);
        }
    }

    printf("Test Matrix:\n");
    print_matrix(test);

    free_matrix(test);
    printf("C environment ready!\n");

    return 0;
}
```

Compilation: `gcc -o test_environment test_environment.c -lm`

Rust Development Setup

```rust
```

```rust
// Cargo.toml
[package]
name = "ml_course"
version = "0.1.0"
edition = "2021"

[dependencies]
ndarray = "0.15"
rand = "0.8"
serde = { version = "1.0", features = ["derive"] }
csv = "1.2"

// src/main.rs
use ndarray::{Array2, Axis};
use rand::Rng;

#[derive(Debug, Clone)]
pub struct Matrix {
    data: Array2<f64>,
}

impl Matrix {
    pub fn new(rows: usize, cols: usize) -> Self {
        Matrix {
            data: Array2::zeros((rows, cols)),
        }
    }

    pub fn from_vec(data: Vec<Vec<f64>>) -> Self {
        let rows = data.len();
        let cols = data[0].len();
        let flat: Vec<f64> = data.into_iter().flatten().collect();

        Matrix {
            data: Array2::from_shape_vec((rows, cols), flat).unwrap(),
        }
    }

    pub fn random(rows: usize, cols: usize) -> Self {
        let mut rng = rand::thread_rng();
        let data = Array2::from_shape_fn((rows, cols), |_| rng.gen::<f64>());
        Matrix { data }
    }
```

```rust
    pub fn shape(&self) -> (usize, usize) {
        (self.data.nrows(), self.data.ncols())
    }

    pub fn print(&self) {
        println!("{:.2}", self.data);
    }
}

fn main() {
    println!("Rust ML Environment Test");

    // Create test matrix
    let test_data = vec![
        vec![1.0, 2.0, 3.0],
        vec![4.0, 5.0, 6.0],
        vec![7.0, 8.0, 9.0],
    ];

    let matrix = Matrix::from_vec(test_data);
    println!("Test Matrix:");
    matrix.print();

    // Create random matrix
    let random_matrix = Matrix::random(5, 3);
    println!("\nRandom Matrix:");
    random_matrix.print();

    println!("Rust environment ready!");
}
```

Hands-on Practice

1. **Environment Verification**: Run all three setup scripts and verify output
2. **Performance Comparison**: Create 1000x1000 matrices in each language and measure creation time
3. **Memory Usage**: Monitor memory consumption for large matrix operations

Reflection Questions

1. What are the trade-offs between Python's ease of use and C's performance?
2. How does Rust's ownership model affect ML algorithm implementation?

3. When would you choose each language for different ML tasks?
4. What challenges do you anticipate in implementing algorithms from scratch?

DAY 2: PYTHON ML ENVIRONMENT SETUP & NUMPY FUNDAMENTALS

Learning Objectives

- Master NumPy array operations for ML
- Understand vectorization and broadcasting
- Implement basic statistical functions
- Create efficient data pipelines

Concept Introduction

NumPy is the foundation of the Python ML ecosystem. Understanding its core concepts is crucial for efficient ML implementations:

Key Concepts:

- **Vectorization**: Operate on entire arrays without explicit loops
- **Broadcasting**: Perform operations on arrays with different shapes
- **Memory Layout**: Understanding C-order vs Fortran-order for performance
- **Data Types**: Choosing appropriate dtypes for memory efficiency

Implementation Guide

Advanced NumPy Operations

```python
```

```python
import numpy as np
import time
import matplotlib.pyplot as plt

class MLUtils:
    """Utility class for common ML operations using NumPy"""

    @staticmethod
    def normalize_data(X, method='standardize'):
        """
        Normalize data using different methods

        Args:
            X: Input data (samples x features)
            method: 'standardize', 'minmax', or 'unit'
        """
        if method == 'standardize':
            return (X - np.mean(X, axis=0)) / np.std(X, axis=0)
        elif method == 'minmax':
            min_vals = np.min(X, axis=0)
            max_vals = np.max(X, axis=0)
            return (X - min_vals) / (max_vals - min_vals)
        elif method == 'unit':
            return X / np.linalg.norm(X, axis=1, keepdims=True)
        else:
            raise ValueError("Method must be 'standardize', 'minmax', or 'unit'")

    @staticmethod
    def create_polynomial_features(X, degree=2):
        """Create polynomial features up to given degree"""
        n_samples, n_features = X.shape

        # Start with original features
        features = [X]

        # Add polynomial combinations
        for d in range(2, degree + 1):
            for i in range(n_features):
                features.append(X[:, i:i+1] ** d)

        return np.hstack(features)

    @staticmethod
```

```python
    def train_test_split(X, y, test_size=0.2, random_state=None):
        """Split data into training and testing sets"""
        if random_state:
            np.random.seed(random_state)

        n_samples = X.shape[0]
        test_samples = int(n_samples * test_size)

        # Random indices
        indices = np.random.permutation(n_samples)
        test_idx = indices[:test_samples]
        train_idx = indices[test_samples:]

        return X[train_idx], X[test_idx], y[train_idx], y[test_idx]

# Demonstration of vectorized operations
def vectorization_demo():
    """Compare vectorized vs loop-based operations"""

    # Create large arrays
    n = 1000000
    a = np.random.randn(n)
    b = np.random.randn(n)

    # Timing vectorized operation
    start = time.time()
    c_vec = a * b + np.sin(a) - np.exp(b * 0.1)
    vec_time = time.time() - start

    # Timing loop-based operation
    start = time.time()
    c_loop = np.zeros(n)
    for i in range(n):
        c_loop[i] = a[i] * b[i] + np.sin(a[i]) - np.exp(b[i] * 0.1)
    loop_time = time.time() - start

    print(f"Vectorized time: {vec_time:.4f}s")
    print(f"Loop time: {loop_time:.4f}s")
    print(f"Speedup: {loop_time/vec_time:.1f}x")

    # Verify results are the same
    print(f"Results equal: {np.allclose(c_vec, c_loop)}")

# Broadcasting examples
```

```python
def broadcasting_demo():
    """Demonstrate NumPy broadcasting"""

    # Example 1: Matrix + vector
    matrix = np.random.randn(5, 3)
    vector = np.array([1, 2, 3])

    result = matrix + vector  # Broadcasting happens automatically
    print("Matrix shape:", matrix.shape)
    print("Vector shape:", vector.shape)
    print("Result shape:", result.shape)

    # Example 2: Centering data
    data = np.random.randn(100, 4)
    means = np.mean(data, axis=0)
    centered = data - means  # Broadcasting

    print(f"Original means: {means}")
    print(f"Centered means: {np.mean(centered, axis=0)}")

# Memory-efficient operations
def memory_optimization_demo():
    """Demonstrate memory-efficient NumPy operations"""

    # Use views instead of copies when possible
    large_array = np.random.randn(10000, 1000)

    # This creates a view (no copy)
    subset_view = large_array[1000:2000, :]

    # This creates a copy
    subset_copy = large_array[1000:2000, :].copy()

    print(f"Original array size: {large_array.nbytes / 1e6:.1f} MB")
    print(f"View shares memory: {np.shares_memory(large_array, subset_view)}")
    print(f"Copy shares memory: {np.shares_memory(large_array, subset_copy)}")

    # In-place operations to save memory
    # Instead of: large_array = large_array * 2
    large_array *= 2  # In-place multiplication

if __name__ == "__main__":
    print("NumPy Fundamentals Demo")
    print("=" * 40)
```

```
# Run demonstrations
print("\n1. Vectorization Performance:")
vectorization_demo()

print("\n2. Broadcasting Examples:")
broadcasting_demo()

print("\n3. Memory Optimization:")
memory_optimization_demo()

# Test utility functions
print("\n4. ML Utilities Test:")

# Create sample data
X = np.random.randn(1000, 5)
y = np.random.randn(1000)

# Test normalization
X_norm = MLUtils.normalize_data(X, 'standardize')
print(f"Normalized data mean: {np.mean(X_norm, axis=0)}")
print(f"Normalized data std: {np.std(X_norm, axis=0)}")

# Test polynomial features
X_small = np.random.randn(100, 2)
X_poly = MLUtils.create_polynomial_features(X_small, degree=3)
print(f"Original features: {X_small.shape[1]}")
print(f"Polynomial features: {X_poly.shape[1]}")

# Test train/test split
X_train, X_test, y_train, y_test = MLUtils.train_test_split(X, y, test_size=0.3)
print(f"Train size: {X_train.shape[0]}")
print(f"Test size: {X_test.shape[0]}")
```

Advanced NumPy Techniques

python

```python
import numpy as np
from scipy.linalg import solve
import warnings

class AdvancedNumPy:
    """Advanced NumPy techniques for ML"""

    @staticmethod
    def efficient_distance_matrix(X):
        """Compute pairwise distances efficiently"""
        # Using broadcasting to avoid explicit loops
        # ||a - b||^2 = ||a||^2 + ||b||^2 - 2a·b

        X_norm_sq = np.sum(X**2, axis=1, keepdims=True)
        distances = X_norm_sq + X_norm_sq.T - 2 * np.dot(X, X.T)

        # Handle numerical errors (small negative values)
        distances = np.maximum(distances, 0)
        np.fill_diagonal(distances, 0)

        return np.sqrt(distances)

    @staticmethod
    def batch_processing(data, batch_size, process_func):
        """Process large arrays in batches to manage memory"""
        n_samples = data.shape[0]
        results = []

        for i in range(0, n_samples, batch_size):
            batch = data[i:i + batch_size]
            result = process_func(batch)
            results.append(result)

        return np.vstack(results) if len(results) > 1 else results[0]

    @staticmethod
    def numerical_gradient(func, x, h=1e-8):
        """Compute numerical gradient using central differences"""
        grad = np.zeros_like(x)

        for i in range(len(x)):
            x_plus = x.copy()
            x_minus = x.copy()
```

```python
            x_plus[i] += h
            x_minus[i] -= h

            grad[i] = (func(x_plus) - func(x_minus)) / (2 * h)

        return grad

    @staticmethod
    def stable_softmax(x):
        """Numerically stable softmax implementation"""
        # Subtract max to prevent overflow
        x_shifted = x - np.max(x, axis=-1, keepdims=True)
        exp_x = np.exp(x_shifted)
        return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

    @staticmethod
    def moving_average(data, window_size):
        """Compute moving average using convolution"""
        kernel = np.ones(window_size) / window_size
        return np.convolve(data, kernel, mode='valid')

# Performance profiling utilities
class MLProfiler:
    """Profiling utilities for ML operations"""

    def __init__(self):
        self.timings = {}

    def time_operation(self, name, func, *args, **kwargs):
        """Time an operation and store results"""
        start = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - start

        if name not in self.timings:
            self.timings[name] = []
        self.timings[name].append(elapsed)

        return result

    def report(self):
        """Generate timing report"""
        print("Performance Report")
        print("-" * 40)
```

```python
        for name, times in self.timings.items():
            mean_time = np.mean(times)
            std_time = np.std(times)
            print(f"{name}: {mean_time:.4f}s ± {std_time:.4f}s ({len(times)} runs)")

# Example usage and benchmarks
def run_advanced_examples():
    """Run advanced NumPy examples"""

    profiler = MLProfiler()

    # Generate test data
    n_samples, n_features = 5000, 100
    X = np.random.randn(n_samples, n_features)

    print("Advanced NumPy Techniques Demo")
    print("=" * 50)

    # 1. Efficient distance computation
    X_small = X[:100]  # Use smaller subset for distance matrix
    distances = profiler.time_operation(
        "distance_matrix",
        AdvancedNumPy.efficient_distance_matrix,
        X_small
    )
    print(f"Distance matrix shape: {distances.shape}")

    # 2. Batch processing
    def dummy_process(batch):
        return np.mean(batch, axis=1, keepdims=True)

    results = profiler.time_operation(
        "batch_processing",
        AdvancedNumPy.batch_processing,
        X, 1000, dummy_process
    )
    print(f"Batch processing result shape: {results.shape}")

    # 3. Numerical gradient
    def quadratic(x):
        return np.sum(x**2)

    x_test = np.array([1.0, 2.0, 3.0])
```

```python
    grad = profiler.time_operation(
        "numerical_gradient",
        AdvancedNumPy.numerical_gradient,
        quadratic, x_test
    )
    print(f"Gradient: {grad}")
    print(f"Expected (2*x): {2 * x_test}")

    # 4. Stable softmax
    logits = np.array([1000, 1001, 1002])  # Large values that would cause overflow
    stable_probs = AdvancedNumPy.stable_softmax(logits)
    print(f"Stable softmax: {stable_probs}")

    # 5. Moving average
    data = np.random.randn(1000)
    smoothed = AdvancedNumPy.moving_average(data, window_size=10)
    print(f"Original data length: {len(data)}")
    print(f"Smoothed data length: {len(smoothed)}")

    # Show performance report
    print("\n")
    profiler.report()

if __name__ == "__main__":
    run_advanced_examples()
```

Hands-on Practice

1. **Vectorization Challenge**: Implement a function to compute cosine similarity between all pairs of vectors without using explicit loops.
2. **Memory Optimization**: Create a function that processes a 10GB dataset (simulated) using only 1GB of RAM through batching.
3. **Broadcasting Mastery**: Implement batch normalization using only NumPy broadcasting.

Reflection Questions

1. When might vectorized operations actually be slower than loops?
2. How does NumPy's memory layout affect performance for different operations?
3. What are the trade-offs between in-place operations and creating new arrays?
4. How can understanding NumPy internals help in debugging ML algorithms?

DAY 3: DATA STRUCTURES FOR ML IN PYTHON

Learning Objectives

- Design efficient data structures for ML workflows
- Implement custom data loaders and preprocessors
- Understand memory management for large datasets
- Create reusable ML pipeline components

Concept Introduction

Efficient data structures are crucial for ML performance. Python offers several options, each with specific advantages:

Core Data Structures for ML:

- **NumPy Arrays**: Dense numerical computation
- **Pandas DataFrames**: Structured data with labels
- **Sparse Matrices**: Memory-efficient storage for sparse data
- **Generators**: Memory-efficient data streaming
- **Custom Classes**: Domain-specific optimizations

Implementation Guide

Custom Dataset Class

python

```python
import numpy as np
import pandas as pd
from typing import Generator, Tuple, Optional, Union
from abc import ABC, abstractmethod
import pickle
import gzip
from pathlib import Path

class BaseDataset(ABC):
    """Abstract base class for ML datasets"""

    def __init__(self, transform=None, target_transform=None):
        self.transform = transform
        self.target_transform = target_transform

    @abstractmethod
    def __len__(self) -> int:
        """Return the size of the dataset"""
        pass

    @abstractmethod
    def __getitem__(self, idx: Union[int, slice]) -> Tuple[np.ndarray, np.ndarray]:
        if isinstance(idx, slice):
            # Handle slice indexing
            start, stop, step = idx.indices(len(self))
            file_batch = [self.file_paths[i] for i in range(start, stop, step)]
            label_batch = self.labels[idx]
        else:
            file_batch = [self.file_paths[idx]]
            label_batch = self.labels[idx:idx+1]

        # Load data from files
        X_batch = []
        for file_path in file_batch:
            data = np.load(file_path)  # Assuming .npy files
            X_batch.append(data)

        X_batch = np.array(X_batch)

        if self.transform:
            X_batch = self.transform(X_batch)
        if self.target_transform:
            label_batch = self.target_transform(label_batch)
```

```
        return X_batch, label_batch

class DataLoader:
    """Efficient data loader with batching and shuffling"""

    def __init__(self, dataset: BaseDataset, batch_size: int = 32,
                 shuffle: bool = True, drop_last: bool = False):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.drop_last = drop_last

    def __len__(self) -> int:
        if self.drop_last:
            return len(self.dataset) // self.batch_size
        else:
            return (len(self.dataset) + self.batch_size - 1) // self.batch_size

    def __iter__(self) -> Generator[Tuple[np.ndarray, np.ndarray], None, None]:
        indices = np.arange(len(self.dataset))

        if self.shuffle:
            np.random.shuffle(indices)

        for i in range(0, len(indices), self.batch_size):
            batch_indices = indices[i:i + self.batch_size]

            if self.drop_last and len(batch_indices) < self.batch_size:
                break

            # Get batch data
            X_batch = []
            y_batch = []

            for idx in batch_indices:
                X, y = self.dataset[idx]
                X_batch.append(X.squeeze())
                y_batch.append(y.squeeze())

            yield np.array(X_batch), np.array(y_batch)

# Data preprocessing pipeline components
class DataTransformer:
```

```
    """Base class for data transformations"""

    def __init__(self):
        self.fitted = False

    def fit(self, X):
        """Fit the transformer to data"""
        return self

    def transform(self, X):
        """Transform the data"""
        raise NotImplementedError

    def fit_transform(self, X):
        """Fit and transform in one step"""
        return self.fit(X).transform(X)

class StandardScaler(DataTransformer):
    """Standardize features by removing mean and scaling to unit variance"""

    def __init__(self):
        super().__init__()
        self.mean_ = None
        self.std_ = None

    def fit(self, X):
        """Compute mean and std for standardization"""
        self.mean_ = np.mean(X, axis=0)
        self.std_ = np.std(X, axis=0)

        # Avoid division by zero
        self.std_ = np.where(self.std_ == 0, 1, self.std_)
        self.fitted = True

        return self

    def transform(self, X):
        """Apply standardization"""
        if not self.fitted:
            raise ValueError("Scaler must be fitted before transform")

        return (X - self.mean_) / self.std_

    def inverse_transform(self, X):
```

```python
        """Reverse the standardization"""
        if not self.fitted:
            raise ValueError("Scaler must be fitted before inverse_transform")

        return X * self.std_ + self.mean_

class MinMaxScaler(DataTransformer):
    """Scale features to a given range"""

    def __init__(self, feature, np.ndarray]:
        """Get item(s) from the dataset"""
        pass

    def __iter__(self):
        """Make dataset iterable"""
        for i in range(len(self)):
            yield self[i]

class MemoryDataset(BaseDataset):
    """Dataset that stores all data in memory"""

    def __init__(self, X: np.ndarray, y: np.ndarray,
                 transform=None, target_transform=None):
        super().__init__(transform, target_transform)
        self.X = X
        self.y = y

        if len(X) != len(y):
            raise ValueError("X and y must have the same length")

    def __len__(self) -> int:
        return len(self.X)

    def __getitem__(self, idx: Union[int, slice]) -> Tuple[np.ndarray, np.ndarray]:
        if isinstance(idx, slice):
            X_batch = self.X[idx]
            y_batch = self.y[idx]
        else:
            X_batch = self.X[idx:idx+1]
            y_batch = self.y[idx:idx+1]

        if self.transform:
            X_batch = self.transform(X_batch)
        if self.target_transform:
```

```python
            y_batch = self.target_transform(y_batch)

        return X_batch, y_batch

    def shuffle(self, random_state=None):
        """Shuffle the dataset in place"""
        if random_state:
            np.random.seed(random_state)

        indices = np.random.permutation(len(self))
        self.X = self.X[indices]
        self.y = self.y[indices]

    def train_test_split(self, test_size=0.2, random_state=None):
        """Split dataset into train and test sets"""
        if random_state:
            np.random.seed(random_state)

        n_samples = len(self)
        n_test = int(n_samples * test_size)

        indices = np.random.permutation(n_samples)
        test_indices = indices[:n_test]
        train_indices = indices[n_test:]

        train_dataset = MemoryDataset(
            self.X[train_indices],
            self.y[train_indices],
            self.transform,
            self.target_transform
        )

        test_dataset = MemoryDataset(
            self.X[test_indices],
            self.y[test_indices],
            self.transform,
            self.target_transform
        )

        return train_dataset, test_dataset

class FileDataset(BaseDataset):
    """Dataset that loads data from files on demand"""
```

```python
    def __init__(self, file_paths: list, labels: list,
                 transform=None, target_transform=None):
        super().__init__(transform, target_transform)
        self.file_paths = file_paths
        self.labels = np.array(labels)

        if len(file_paths) != len(labels):
            raise ValueError("file_paths and labels must have the same length")

    def __len__(self) -> int:
        return len(self.file_paths)

    def __getitem__(self, idx: Union[int, slice]) -> Tuple[np.ndarray
```