**Project Title:**

**"Adaptive Resource Management in Multi-Tasking Operating Systems Using Machine Learning"**

**Overview:**

This project aims to demonstrate how machine learning techniques can enhance the performance of an operating system by intelligently managing resources like CPU, memory, and I/O in a multi-tasking environment. The OS will leverage ML algorithms to predict resource needs based on usage patterns, dynamically adjusting scheduling and resource allocation to optimize system performance.

**Key Components:**

1. **Dynamic Process Scheduling using Machine Learning**:
   * Use ML models to predict which processes need more CPU time based on historical patterns and adjust scheduling priorities dynamically.
   * Compare the performance of ML-based scheduling with traditional scheduling algorithms (e.g., Round Robin, Shortest Job First).
2. **Memory Management Optimization**:
   * Implement ML models to predict page access patterns and optimize paging strategies. For example, use ML to minimize page faults by predicting which memory pages should stay in RAM.
   * Compare this approach to standard page replacement algorithms like Least Recently Used (LRU) or FIFO.
3. **I/O and Disk Scheduling**:
   * Use ML to predict I/O requests (e.g., disk read/write) and optimize disk scheduling (e.g., SCAN, C-SCAN) by predicting the best ordering of requests to reduce seek time.
   * Create an intelligent disk scheduling mechanism that adapts based on system load and usage patterns.
4. **Power and Energy Management**:
   * Implement ML algorithms that monitor CPU usage, battery consumption, and thermal metrics to optimize power consumption. This could involve dynamically adjusting CPU frequency and voltage (Dynamic Voltage and Frequency Scaling - DVFS) based on predicted workloads.
5. **Load Balancing in Multi-Core Systems**:
   * Use ML to predict workloads and distribute processes across multiple CPU cores efficiently.
   * Adapt the load-balancing strategy by learning patterns in process execution times, memory usage, and I/O operations.
6. **Anomaly Detection and Security**:
   * Integrate ML models to detect anomalies in resource usage patterns. For instance, identify unusual spikes in CPU or memory usage that might indicate a security threat (like a DDoS attack or malware).
   * Automatically adjust OS behavior (like throttling suspicious processes) when an anomaly is detected.

**Technical Breakdown:**

**1.Process Scheduling with ML**:

* + Build a **reinforcement learning (RL)** agent that learns the best scheduling decisions over time by receiving feedback based on system performance (e.g., CPU utilization, process wait time).
  + Compare the RL-based scheduling approach with classic OS scheduling algorithms.

Sample code to simulate RL-based scheduling using Q-learning python.

**Dynamic Process Scheduling using Q-learning**

**Code Snippet:**

import numpy as np

# Define actions (e.g., schedule process A, B, or C)

actions = ['A', 'B', 'C']

# Define Q-table (state-action pair)

q\_table = np.zeros((len(actions), len(actions)))

# Learning parameters

learning\_rate = 0.1

discount\_factor = 0.9

exploration\_rate = 0.1

# Reward function based on process completion time and CPU utilization

def reward\_function(process, completion\_time, cpu\_utilization):

return 1 / (completion\_time \* (1 - cpu\_utilization))

# Q-learning update

def update\_q\_table(state, action, reward, next\_state):

q\_predict = q\_table[state, action]

q\_target = reward + discount\_factor \* np.max(q\_table[next\_state, :])

q\_table[state, action] += learning\_rate \* (q\_target - q\_predict)

**Detailed Explanation:**

* **Purpose:** This code implements a Q-learning algorithm for process scheduling in an OS. Q-learning is a type of reinforcement learning where an agent learns to make decisions by receiving rewards based on its actions.
* **Q-table:** A matrix that stores the expected utility (Q-value) of taking a certain action in a given state. Each row corresponds to a state, and each column corresponds to an action.
* **Learning Parameters:**
  + **Learning Rate (0.1):** Determines how much new information overrides old information. A higher value makes the agent learn faster but can lead to instability.
  + **Discount Factor (0.9):** Weighs the importance of future rewards. A value close to 1 makes the agent consider long-term rewards.
  + **Exploration Rate (0.1):** Balances exploration (trying new actions) and exploitation (choosing the best-known action). A higher exploration rate encourages trying new actions.
* **Reward Function:** This function calculates the reward based on process completion time and CPU utilization. The goal is to maximize rewards by minimizing completion time and maximizing CPU usage efficiency.
* **Q-learning Update:** The Q-value for the state-action pair is updated using the Bellman equation. It predicts the expected future reward based on the current reward and the maximum expected future reward.

1. **Memory Management with Predictive Page Replacement**:
   * Train an ML model to predict which memory pages will be accessed next (similar to LRU but based on predictive modeling).
   * Use a dataset of memory access patterns to train a classifier like **LSTM** or **Recurrent Neural Networks (RNNs)** to predict page access sequences.
   * Upon prediction, proactively load these pages into memory to minimize page faults.

from keras.models import Sequential

from keras.layers import LSTM, Dense

import numpy as np

# Example of training an LSTM to predict page access

model = Sequential()

model.add(LSTM(128, input\_shape=(10, 1))) # Input is the last 10 page accesses

model.add(Dense(1, activation='softmax'))

model.compile(loss='mse', optimizer='adam')

# Simulated training data: sequences of page accesses

X\_train = np.random.rand(1000, 10, 1)

y\_train = np.random.randint(0, 100, (1000, 1)) # Predict the next page

model.fit(X\_train, y\_train, epochs=10)

# Use this model to predict next page to access

def predict\_next\_page(past\_accesses):

return model.predict(past\_accesses)

**Detailed Explanation:**

* **Purpose:** This LSTM (Long Short-Term Memory) model predicts future memory page accesses, reducing page faults and improving memory efficiency.
* **Model Architecture:**
  + **Sequential Model:** A linear stack of layers where each layer has one input and one output.
  + **LSTM Layer (128 units):** Captures long-term dependencies in sequential data, making it suitable for time-series data like memory accesses.
  + **Dense Layer:** Outputs the prediction for the next memory page to access. The softmax activation function is used for multi-class classification, but here it could be adjusted to reflect a regression output.
* **Training Data:** Simulated data consists of sequences of past page accesses. The model learns patterns in this data to predict future accesses.
* **Prediction Function:** After training, this function uses the model to predict the next memory page based on the last 10 accesses.

**I/O and Disk Scheduling with Supervised Learning**

**Concept Overview:** This section focuses on using supervised learning to optimize disk scheduling by predicting the optimal order of disk I/O requests based on historical access patterns.

**Detailed Explanation:**

* **Data Collection:** Historical I/O requests are logged, including timestamps and the order of requests.
* **Model Training:** A supervised learning model (e.g., Random Forest or Support Vector Machine) is trained on this data to learn the relationships between requests.
* **Optimal Ordering:** The model predicts the best order for servicing requests, reducing seek time and improving overall I/O efficiency.

**Power Management using Reinforcement Learning**:

import random

# State: CPU utilization, temperature

states = ['low', 'medium', 'high']

actions = ['increase\_freq', 'decrease\_freq', 'maintain']

q\_table = np.zeros((len(states), len(actions)))

# Reward based on power consumption and performance

def reward\_function(cpu\_utilization, power\_consumption):

return -power\_consumption if cpu\_utilization < 0.7 else power\_consumption

# Q-learning update structure similar to the previous example

def update\_q\_table(state, action, reward, next\_state):

# Similar structure as before

pass

**Detailed Explanation:**

* **Purpose:** This code establishes a reinforcement learning framework for optimizing CPU power management through Dynamic Voltage and Frequency Scaling (DVFS).
* **States and Actions:**
  + **States:** Represent CPU utilization levels (low, medium, high), which affect power consumption and performance.
  + **Actions:** Options include increasing, decreasing, or maintaining CPU frequency.
* **Reward Function:** The reward encourages power-saving actions when CPU utilization is low. If the CPU is under high load, the reward reflects the necessity of maintaining performance despite higher power consumption.
* **Q-learning Update:** Similar to the previous Q-learning implementation, this function will update the Q-values based on observed rewards to learn optimal power management strategies.

**Load Balancing across Cores**

**Concept Overview:** This component uses machine learning to predict workloads for efficient distribution across multiple CPU cores.

**Detailed Explanation:**

* **Workload Prediction:** An ML model (e.g., clustering or regression) is trained to predict the intensity of workloads based on historical execution times and resource usage.
* **Dynamic Load Balancing:** Processes are allocated to CPU cores based on predicted workloads to ensure that no single core is overwhelmed, which can lead to performance bottlenecks.

**Anomaly Detection and Security**

**Concept Overview:** Anomaly detection models identify unusual spikes in resource usage that may signal security threats or system overloads.

**Detailed Explanation:**

* **Data Monitoring:** The system continuously monitors CPU, memory, and I/O usage.
* **Model Training:** A supervised learning model (e.g., Isolation Forest or Autoencoder) is trained on historical usage data to identify patterns and detect anomalies.
* **Anomaly Detection:** The trained model flags resource usage that deviates significantly from established patterns, indicating potential security threats or performance issues.
* **Response Mechanism:** When an anomaly is detected, the system can automatically adjust behavior (e.g., throttling suspicious processes) or alert administrators.

**Example Pseudo-code:**

python

from sklearn.ensemble import IsolationForest

# Simulated resource usage data

data = [[0.1, 0.3], [0.2, 0.4], [0.15, 0.35], [0.8, 0.9]] # Normal usage

model = IsolationForest(contamination=0.1)

model.fit(data)

# Predicting anomalies

new\_data = [[0.9, 0.8]] # New resource usage to check

predictions = model

### Implementation: Algorithms like k-means clustering can group processes with similar resource needs, allowing for Conclusion

The integration of machine learning into operating system resource management demonstrates significant potential for improving performance, efficiency, and security. By leveraging ML models for scheduling, memory management, I/O operations, and power management, we can create a more adaptive and intelligent OS environment. Future work could extend these techniques to more complex systems and distributed environments.

**Team members:**

**Shashwat Singh(23BCE1517)**

**Ananya Bhattacharyya(23BCE1503)**