In [ ]:
```
import polars as pd
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
import numpy as np
import random
from torch.nn.functional import relu
from tqdm import tqdm
```

In [ ]:
```
import gymnasium as gym
from ale_py import ALEInterface
from ale_py.roms import DoubleDunk

ale = ALEInterface()
ale.loadROM(DoubleDunk)
```

In [ ]:
```
replay_buffer_df = pd.DataFrame()
for i in range(0, 1000):
    file_path = 'data/data1_compressed/data' + str(i) + '.json'
    df = pd.read_json(file_path)
    replay_buffer_df = pd.concat([replay_buffer_df, df])
```

In [ ]:
```
df.head()
```

Out[ ]: shape: (5, 5)

state	action	new_state	reward	done
list[i64]	i64	list[i64]	f64	bool
[130, 137, … 177]	8	[130, 137, … 188]	0.0	false
[130, 137, … 188]	6	[130, 137, … 188]	0.0	false
[130, 137, … 188]	7	[130, 137, … 188]	0.0	false
[130, 137, … 188]	3	[130, 137, … 188]	0.0	false
[130, 137, … 188]	9	[130, 137, … 188]	0.0	false

In [ ]:
```
states = np.stack(df['state'].to_numpy())
actions = df['action'].to_numpy()
next_states = np.stack(df['new_state'].to_numpy())
rewards = df['reward'].to_numpy()
dones = df['done'].to_numpy()
```

In [ ]:
```
from torch.utils.data import DataLoader, Dataset

class ReplayDataset(Dataset):
    def __init__(self, states, actions, next_states, rewards, dones):
        self.states = torch.FloatTensor(states)
```

```
        self.actions = torch.LongTensor(actions)
        self.next_states = torch.FloatTensor(next_states)
        self.rewards = torch.FloatTensor(rewards)
        self.dones = torch.FloatTensor(dones)

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

    def __getitem__(self, idx):
        return (self.states[idx], self.actions[idx], self.next_states[idx], self.re

dataset = ReplayDataset(states, actions, next_states, rewards, dones)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

In [ ]:
```
class DeepQNetwork(nn.Module):
    def __init__(self, input_dim, action_space):
        super(DeepQNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 32)
        self.fc5 = nn.Linear(32, action_space)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = torch.relu(self.fc4(x))
        x = self.fc5(x)
        return x
```

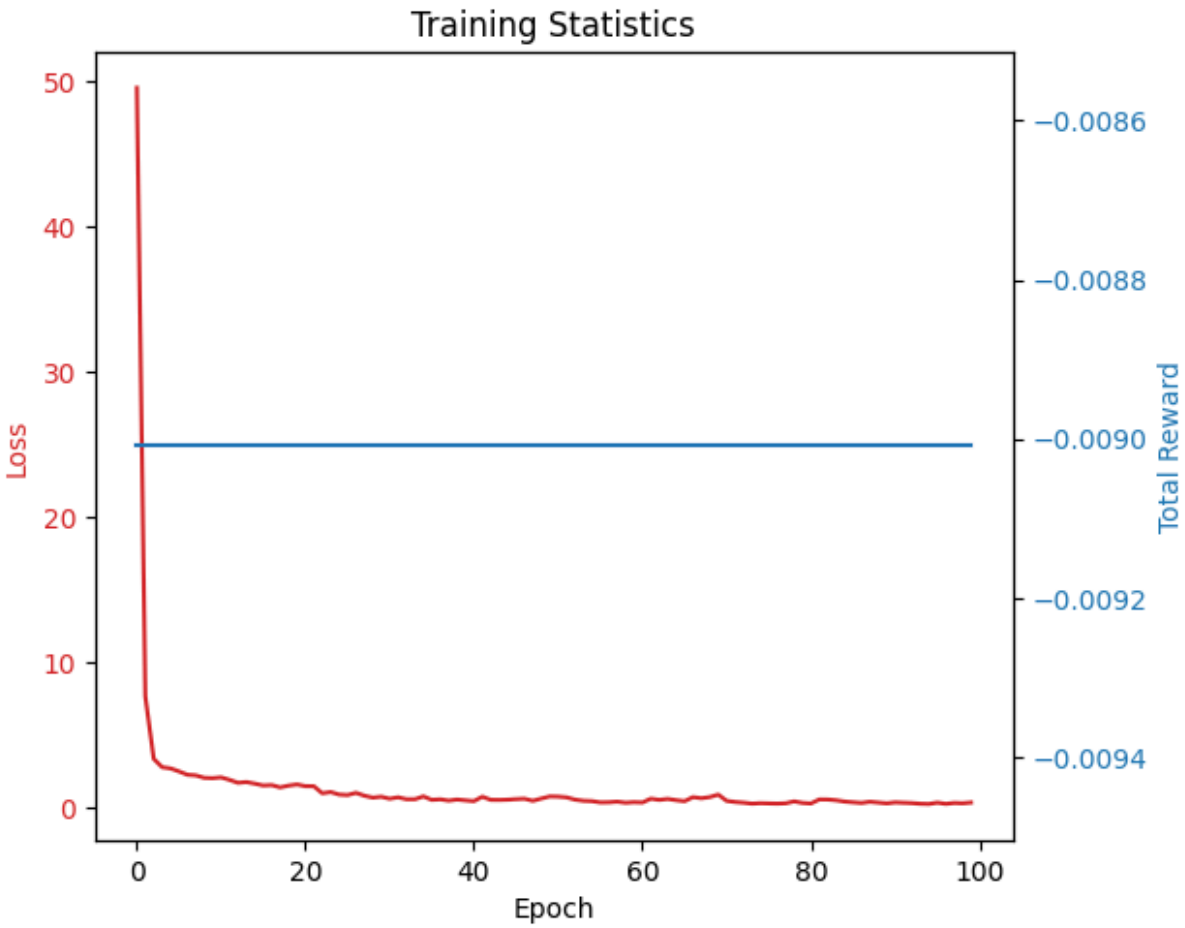
In [ ]:
```
def train_dqn(dataloader, num_epochs, gamma, target_update_freq):
    input_dim = 128
    action_space = 18

    policy_net = DeepQNetwork(input_dim, action_space)
    target_net = DeepQNetwork(input_dim, action_space)

    target_net.load_state_dict(policy_net.state_dict())
    optimizer = optim.Adam(policy_net.parameters())

    epoch_losses = []
    epoch_rewards = []

    for epoch in tqdm(range(num_epochs)):
        epoch_loss = 0
        total_reward = 0
        for states, actions, next_states, rewards, dones in dataloader:
            q_values = policy_net(states)
            next_q_values = target_net(next_states)

            q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)
            next_q_values = next_q_values.max(1)[0]
            expected_q_values = rewards + gamma * next_q_values * (1 - dones)
```

```
            loss = nn.functional.mse_loss(q_values, expected_q_values)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            total_reward += rewards.sum().item()

        epoch_losses.append(epoch_loss / len(dataloader))
        epoch_rewards.append(total_reward / len(dataloader.dataset))

        if epoch % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

    return policy_net, epoch_losses, epoch_rewards
        # print(f"Epoch {epoch}, Loss: {epoch_loss / len(dataloader)}")
```

In [ ]:
```
policy_net, epoch_losses, epoch_rewards = train_dqn(dataloader, num_epochs=100, gam
```

```
100%|██████████| 100/100 [00:10<00:00,  9.97it/s]
```

In [ ]:
```
import matplotlib.pyplot as plt

def plot_training_statistics(epoch_losses, epoch_rewards):
    fig, ax1 = plt.subplots()

    color = 'tab:red'
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss', color=color)
    ax1.plot(epoch_losses, color=color)
    ax1.tick_params(axis='y', labelcolor=color)

    ax2 = ax1.twinx()
    color = 'tab:blue'
    ax2.set_ylabel('Total Reward', color=color)
    ax2.plot(epoch_rewards, color=color)
    ax2.tick_params(axis='y', labelcolor=color)

    fig.tight_layout()
    plt.title('Training Statistics')
    plt.show()

plot_training_statistics(epoch_losses, epoch_rewards)
```



In [ ]:
```python
def get_action_probabilities(policy_net, state):
    policy_net.eval()
    with torch.no_grad():
        state = torch.FloatTensor(state).unsqueeze(0)
        q_values = policy_net(state)
        action_probabilities = torch.nn.functional.softmax(q_values, dim=1)
    return action_probabilities.squeeze().numpy()
```

In [ ]:
```python
# Assuming 'states' is your array of states from the dataset
sample_state = states[0]  # Take the first state as an example

action_probabilities = get_action_probabilities(policy_net, sample_state)
print(f"Action probabilities for the sample state: {action_probabilities}")
print(f"Predicted action: {np.argmax(action_probabilities)}")
```

```
Action probabilities for the sample state: [0.05837047 0.04450282 0.05951525 0.04499
35  0.05246726 0.0188166
 0.09047823 0.0369099  0.10082109 0.0707422  0.03155356 0.06568328
 0.08694156 0.05604778 0.07531142 0.04262663 0.02428259 0.03993593]
Predicted action: 8
```