

✓ Atari Breakout DQN Training

✓ 1. Dependency Setup

```
!pip install numpy==1.25.2
!pip install tensorflow==2.15.0
!pip install keras==2.15.0
!pip install h5py==3.11.0
!pip install pillow==10.3.0
!pip install gymnasium[atari]==0.29.1
!pip install keras-rl2==1.0.4
!pip install autorom[accept-rom-license]
!AutoROM --accept-license
!echo "y" | AutoROM --accept-license > /dev/null


# Force restart
import IPython
IPython.Application.instance().kernel.do_shutdown(True)
```

 [Show hidden output](#)

✓ 2. keras-rl2 compatibility patch

```
""" This module fixes the compatibility issue between keras-rl2 and gymnasium """
import os
import rl

# Apply the patch to fix keras-rl2 compatibility
rl_path = os.path.dirname(rl.__file__)
callbacks_path = os.path.join(rl_path, 'callbacks.py')
with open(callbacks_path, 'r') as file:
    content = file.read()
fixed_content = content.replace(
    'from tensorflow.keras import __version__ as KERAS_VERSION',
    'from keras import __version__ as KERAS_VERSION'
)
with open(callbacks_path, 'w') as file:
    file.write(fixed_content)
print("✓ keras-rl2 compatibility patch applied")

 ✓ keras-rl2 compatibility patch applied
```

✓ 3. Imports

```
# Import all necessary libraries for the rest of the notebook
import numpy as np
import sys
import re
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
from collections import deque
import time
import datetime

# Imports from keras-rl2
import rl
from rl.processors import Processor
from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy, LinearAnnealedPolicy
from rl.memory import SequentialMemory
from rl.callbacks import ModelIntervalCheckpoint

# Imports from Keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, Input
from keras.optimizers import Adam
from keras.callbacks import Callback
```

```
# Imports from Gymnasium
import gymnasium as gym
from gymnasium.wrappers import AtariPreprocessing
```

```
print("✓ All core libraries imported")
```

```
↗ ✓ All core libraries imported
```

✓ 4. Mount Google Drive (For Saving Models and logs)

```
# Mount Google Drive for saving models and logs
from google.colab import drive
drive.mount('/content/drive')
print("✓ Google Drive mounted")
```

```
# Create directory for saving models
if not os.path.exists('/content/drive/MyDrive/breakout_dqn/logs'):
    os.makedirs('/content/drive/MyDrive/breakout_dqn/logs')
    print("✓ Directory created for saving models and logs")
```

```
print("✓ All setup complete")
```

```
↗ Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
✓ Google Drive mounted
✓ All setup complete
```

✓ 5. Implementation

```
class GymCompatibilityWrapper(gym.Wrapper):
    """
    Wrapper to make gymnasium compatible with keras-rl while supporting reward shaping.

    This wrapper bridges the Gymnasium API with keras-rl expectations and ensures
    the processor receives terminal state information for proper reward shaping.
    """
    def __init__(self, env, processor=None):
        super().__init__(env)
        self.processor = processor

    def step(self, action):
        """
        Update step method to match keras-rl output and enhance reward shaping.

        Adds 'done' flag to info dict when episode terminates, allowing the
        processor to apply end-of-episode reward adjustments.
        """
        obs, reward, terminated, truncated, info = self.env.step(action)
        done = terminated or truncated

        # Signal episode termination to processor
        if done and self.processor is not None:
            info['done'] = True

        return obs, reward, done, info

    def render(self):
        """Update render method to match keras-rl"""
        return self.env.render()

    def reset(self, **kwargs):
        """Update reset method to match keras-rl output"""
        obs, _ = self.env.reset(**kwargs)
        return obs

class AdaptiveRewardScaler:
    def __init__(self, target_min=-1.0, target_best=1.2, decay_factor=0.95, initial_best=1.0):
        """
        Adaptive reward scaler that adjusts based on best performance.

        Args:
            target_min: The minimum (negative) scaled reward
            target_best: The scaled reward for the best performance so far
        """
```

```

        decay_factor: Factor to decay best_reward when resetting (0.95 means 5% decay)
        initial_best: Initial value for best_reward
    """
    self.best_reward = initial_best
    self.target_min = target_min
    self.target_best = target_best
    self.decay_factor = decay_factor

def scale_reward(self, shaped_reward):
    """
    Scale reward relative to best performance seen so far.
    """
    # Update best_reward tracker if we see a new best
    if shaped_reward > self.best_reward:
        self.best_reward = shaped_reward

    # Scale the reward
    if shaped_reward == 0:
        return 0
    elif shaped_reward > 0:
        # Scale positive rewards relative to best seen
        # This ensures the best reward gets target_best value
        return self.target_best * (shaped_reward / self.best_reward)
    else:
        # Scale negative rewards using fixed approach
        return self.target_min * min(shaped_reward / -1.0, 1.0)

def reset_on_target_update(self):
    """
    Slightly decays the best reward to allow for scaling adjustment.
    """
    self.best_reward = max(1.0, self.best_reward * self.decay_factor)

class StackDimProcessor(Processor):
    """
    Custom processor that resolves dimension mismatches and implements reward shaping.
    Reward shaping is designed to mimic human-like motivation in games:
    - Breaking bricks is the primary objective and main source of satisfaction
    - Dying after scoring feels more disappointing than dying without scoring
    - Surviving longer builds anticipation and makes failure more consequential
    These human-like motivational signals help the agent learn faster by providing
    a richer reward landscape while maintaining the proper incentive hierarchy.
    """
    def __init__(self):
        super().__init__()
        self.episode_steps = 0
        self.episode_rewards = 0
        self._is_terminal = False
        self.reward_scaler = AdaptiveRewardScaler(
            target_min=-1.0,
            target_best=1.2,
            decay_factor=0.95,
            initial_best=1.0
        )

    def process_observation(self, observation):
        """Return the observation as is"""
        return observation

    def process_state_batch(self, batch):
        """Fix dimension mismatch between environment obs and model inputs"""
        # If we have a 5D tensor (batch, window_length, height, width, channel)
        if len(batch.shape) == 5:
            # Get dimensions
            batch_size, window_length, height, width, channels = batch.shape
            # Reshape to (batch, height, width, window_length*channels)
            # This stacks the frames along the channel dimension
            return np.reshape(batch, (batch_size, height, width, window_length * channels))
        return batch

    def process_reset(self, observation):
        """Reset episode tracking when environment resets"""
        self.episode_steps = 0
        self.episode_rewards = 0
        self._is_terminal = False
        return observation

```

```

def process_reward(self, reward):
    """
    Shape rewards to provide meaningful learning signals between sparse game rewards.
    Modified to ensure strong negative signal for deaths without creating perverse incentives.
    """
    # Track accumulated rewards and steps
    self.episode_rewards += reward
    self.episode_steps += 1

    # Base reward (from breaking bricks)
    shaped_reward = reward

    # Terminal state detection (end of episode/life loss)
    if hasattr(self, '_is_terminal') and self._is_terminal:
        # Fixed penalty for all deaths: -0.5
        # This creates a consistent, strong negative signal that doesn't
        # penalize scoring behavior
        end_adjustment = -0.5

        # Optional: Small bonus for lasting longer (but still keeping net negative)
        survival_factor = min(1.0, self.episode_steps / 500)
        survival_bonus = 0.1 * survival_factor

        # Final adjustment is still negative but rewards survival
        end_adjustment += survival_bonus # At most reduces penalty to -0.4

    shaped_reward += end_adjustment

    # Reset episode tracking
    self.episode_steps = 0
    self.episode_rewards = 0
    self._is_terminal = False

    # Use adaptive scaling instead of clipping
    return self.reward_scaler.scale_reward(shaped_reward)

def process_info(self, info):
    """Process game information to detect episode termination."""
    # Track terminal state for next reward processing
    if 'done' in info and info['done']:
        self._is_terminal = True
    return info

class EpisodicTargetNetworkUpdate(Callback):
    """
    Custom callback to update the target network after a specific number of episodes.
    This overrides the default step-based update mechanism in DQNAgent.
    """
    def __init__(self, update_frequency=10, verbose=0):
        """
        Args:
            update_frequency: Number of episodes between target network updates
            verbose: Verbosity level (0=silent, 1=progress bar, 2=one line per epoch)
        """
        super(EpisodicTargetNetworkUpdate, self).__init__()
        self.update_frequency = update_frequency
        self.episodes_since_update = 0
        self.verbose = verbose

    def on_episode_end(self, episode, logs={}):
        """Called at the end of each episode."""
        self.episodes_since_update += 1

        # Check if it's time to update the target network
        if self.episodes_since_update >= self.update_frequency:
            # Update target network by manually copying weights
            # In keras-rl2, we need to directly access and update the target model weights
            target_weights = self.model.target_model.get_weights()
            online_weights = self.model.model.get_weights()

            # Manual update
            for i in range(len(target_weights)):
                target_weights[i] = online_weights[i]

            # Set the updated weights
            self.model.target_model.set_weights(target_weights)

```

```

        # Also update reward scaler if processor has one
        if hasattr(self.model.processor, 'reward_scaler'):
            self.model.processor.reward_scaler.reset_on_target_update()

        # Reset counter
        self.episodes_since_update = 0

        if self.verbose >= 1:
            print(f"\nTarget network updated after {self.update_frequency} episodes")

def make_env(env_id):
    """
    Creates a wrapped Atari environment with reward shaping for faster learning.

    The environment includes human-like motivational signals that help
    the agent learn from sparse rewards by providing a richer feedback landscape.
    """
    env = gym.make(env_id)

    # Apply Atari preprocessing

    # NOTE:

    # Setting noop_max=7 strikes a balance for Breakout:
    # Based on rate of movement for the paddle and ball; enough randomization to
    # avoid fixed starting paddle positions that could create unwinnable ball
    # trajectories (which would disproportionately penalize the agent with negative
    # learning signals), while keeping training efficient.
    env = AtariPreprocessing(
        env,
        noop_max=7,
        frame_skip=4,
        screen_size=84,
        terminal_on_life_loss=True, # End episode on life loss
        grayscale_obs=True,
        grayscale_newaxis=True,
        scale_obs=False,
    )

    # Create processor for dimension handling and reward shaping
    processor = StackDimProcessor()

    # Make compatible with keras-rl, passing the processor reference
    env = GymCompatibilityWrapper(env, processor)

    return env, processor

def model_template(state_shape, n_actions):
    """Defines the DQN model architecture for policy and target networks"""
    model = Sequential()
    model.add(Input(shape=state_shape))
    model.add(Conv2D(32, (8, 8), strides=4, activation='relu'))
    model.add(Conv2D(64, (4, 4), strides=2, activation='relu'))
    model.add(Conv2D(64, (3, 3), strides=1, activation='relu'))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(n_actions, activation='linear'))
    return model

```

✓ 6. Training Monitor

```

# Training Monitor for efficient Progress Tracking
class TrainingMonitor:
    """
    Memory-efficient training monitor using rolling statistics.

    Tracks metrics using running averages and periodic sampling
    to minimize memory usage and computational overhead.
    """
    def __init__(self, log_dir='/content/drive/MyDrive/breakout_dqn/logs',
                 window_size=100, log_interval=10):
        self.log_dir = log_dir
        self.window_size = window_size

```

```

self.log_interval = log_interval # Only log every N episodes to CSV
os.makedirs(log_dir, exist_ok=True)

# Rolling windows for recent metrics (fixed memory usage)
self.reward_window = deque(maxlen=window_size)
self.length_window = deque(maxlen=window_size)
self.q_window = deque(maxlen=window_size)
self.loss_window = deque(maxlen=window_size)
self.sps_window = deque(maxlen=window_size)

# Running statistics (constant memory regardless of training length)
self.episode_count = 0
self.total_steps = 0
self.max_reward = float('-inf')
self.max_reward_episode = 0

# For checkpoint statistics
self.checkpoint_episodes = []
self.checkpoint_rewards = []
self.checkpoint_steps = []
self.checkpoint_q_values = []

# Timing
self.last_checkpoint_time = time.time()
self.last_checkpoint_steps = 0

# Create the CSV log file
self.log_file = os.path.join(log_dir, 'training_log.csv')
self.create_log_file()

def create_log_file(self):
    """Initialize the CSV log file with headers"""
    with open(self.log_file, 'w') as f:
        f.write('episode,total_steps,reward,length,duration,loss,mean_q,epsilon,steps_per_second\n')

def on_episode_end(self, episode, logs):
    """Record metrics at the end of each episode using efficient rolling stats"""
    # Extract metrics
    reward = logs.get('episode_reward', 0)
    steps = logs.get('nb_steps', 0)
    duration = logs.get('duration', 0)
    loss = logs.get('loss', None)
    mean_q = logs.get('mean_q', None)
    epsilon = logs.get('mean_eps', None)
    sps = steps / max(duration, 0.001) # Steps per second

    # Update counters
    self.episode_count += 1
    self.total_steps += steps

    # Update rolling windows (fixed memory usage)
    self.reward_window.append(reward)
    self.length_window.append(steps)
    self.sps_window.append(sps)

    if loss is not None:
        self.loss_window.append(loss)
    if mean_q is not None:
        self.q_window.append(mean_q)

    # Track maximum reward
    if reward > self.max_reward:
        self.max_reward = reward
        self.max_reward_episode = self.episode_count

    # Log to CSV periodically (not every episode)
    if self.episode_count % self.log_interval == 0:
        with open(self.log_file, 'a') as f:
            f.write(f'{self.episode_count},{self.total_steps},{reward},{steps},{duration},{loss},{mean_q},{epsilon},{sps}\n')

def on_checkpoint(self, step_count):
    """Generate summary visualizations at checkpoint intervals with minimal data"""
    # Calculate performance
    now = time.time()
    time_elapsed = now - self.last_checkpoint_time
    steps_done = step_count - self.last_checkpoint_steps
    steps_per_sec = steps_done / max(time_elapsed, 0.001)

```

```

# Store checkpoint metrics (minimal data points)
self.checkpoint_episodes.append(self.episode_count)
self.checkpoint_rewards.append(self._get_window_avg(self.reward_window))
self.checkpoint_steps.append(step_count)
if self.q_window:
    self.checkpoint_q_values.append(self._get_window_avg(self.q_window))

# Create timestamp for files
timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

# Create visualizations
self.generate_plots(step_count, timestamp)

# Generate summary statistics
summary = self.generate_summary(step_count, steps_per_sec, time_elapsed)

# Update checkpoint tracking
self.last_checkpoint_time = now
self.last_checkpoint_steps = step_count

return summary

def _get_window_avg(self, window):
    """Compute average of a window deque efficiently"""
    if not window:
        return 0
    return sum(window) / len(window)

def generate_plots(self, step_count, timestamp):
    """Create visualization plots using only checkpoint data and current windows"""
    # Create figure with multiple subplots
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Plot 1: Episode rewards (using checkpoints and current window)
    ax = axes[0, 0]
    # Plot checkpoint data (sparse historical data)
    if self.checkpoint_rewards:
        ax.plot(self.checkpoint_episodes, self.checkpoint_rewards, 'b-o', label='Checkpoint Avg')

    # Plot recent episodes in detail (from rolling window)
    recent_indices = list(range(self.episode_count - len(self.reward_window) + 1,
                               self.episode_count + 1))
    ax.plot(recent_indices, list(self.reward_window), 'g-', alpha=0.5, label='Recent Episodes')

    ax.set_title(f'Episode Rewards (Max: {self.max_reward} at #{self.max_reward_episode})')
    ax.set_xlabel('Episode')
    ax.set_ylabel('Reward')
    ax.grid(True)
    if self.checkpoint_rewards or self.reward_window:
        ax.legend()

    # Plot 2: Episode lengths (current window only)
    ax = axes[0, 1]
    if self.length_window:
        ax.plot(recent_indices, list(self.length_window))
        ax.set_title(f'Recent Episode Lengths (Avg: {self._get_window_avg(self.length_window):.1f})')
        ax.set_xlabel('Episode')
        ax.set_ylabel('Steps')
        ax.grid(True)
    else:
        ax.text(0.5, 0.5, 'No episode length data available',
               horizontalalignment='center', verticalalignment='center')

    # Plot 3: Mean Q-values (checkpoints + current window)
    ax = axes[1, 0]
    if self.checkpoint_q_values:
        ax.plot(self.checkpoint_episodes[len(self.checkpoint_episodes)-len(self.checkpoint_q_values):],
               self.checkpoint_q_values, 'b-o', label='Checkpoint Avg')

    if self.q_window:
        ax.plot(recent_indices, list(self.q_window), 'g-', alpha=0.5, label='Recent Episodes')
        ax.set_title(f'Mean Q-Values (Recent Avg: {self._get_window_avg(self.q_window):.4f})')
        ax.set_xlabel('Episode')
        ax.set_ylabel('Q-Value')
        ax.grid(True)
        ax.legend()

```

```

else:
    ax.text(0.5, 0.5, 'No Q-values recorded yet\n(still in warmup phase)',
            horizontalalignment='center', verticalalignment='center')
    ax.set_title('Mean Q-Values (Not Available)')

# Plot 4: Training progress (steps vs episodes)
ax = axes[1, 1]
if self.checkpoint_episodes:
    ax.plot(self.checkpoint_episodes, self.checkpoint_steps, 'b-o')
    ax.set_title('Training Progress')
    ax.set_xlabel('Episodes')
    ax.set_ylabel('Total Steps')
    ax.grid(True)

    # Add second y-axis for SPS
    if self.sps_window:
        ax2 = ax.twinx()
        ax2.plot(recent_indices, list(self.sps_window), 'r-', alpha=0.5)
        ax2.set_ylabel('Steps/Second', color='r')
        ax2.tick_params(axis='y', labelcolor='r')
else:
    ax.text(0.5, 0.5, 'No checkpoint data available yet',
            horizontalalignment='center', verticalalignment='center')

plt.tight_layout()

# Save the figure
plt.savefig(os.path.join(self.log_dir, f'training_progress_{step_count}_{timestamp}.png'))
plt.close()

def generate_summary(self, step_count, steps_per_sec, time_elapsed):
    """Generate checkpoint summary statistics from rolling windows"""
    # Summary statistics use only current windows (constant memory)
    avg_reward = self._get_window_avg(self.reward_window)
    avg_length = self._get_window_avg(self.length_window)
    avg_q = self._get_window_avg(self.q_window) if self.q_window else None

    # Return formatted summary
    summary = {
        'step_count': step_count,
        'episodes_completed': self.episode_count,
        'avg_reward_recent': avg_reward,
        'max_reward_all_time': self.max_reward,
        'avg_episode_length': avg_length,
        'steps_per_second': steps_per_sec,
        'time_elapsed_minutes': time_elapsed / 60
    }

    if avg_q is not None:
        summary['avg_q_value'] = avg_q

    # Save summary to file
    with open(os.path.join(self.log_dir, f'summary_{step_count}.txt'), 'w') as f:
        for key, value in summary.items():
            f.write(f"{key}: {value}\n")

    return summary

def compare_runs(self, other_log_file, output_path=None):
    """Compare current run with another training run"""
    try:
        # Load just the necessary data from CSV files (memory efficient)
        current_df = pd.read_csv(self.log_file, usecols=['episode', 'reward'])
        other_df = pd.read_csv(other_log_file, usecols=['episode', 'reward'])

        # Calculate rolling averages
        window = min(100, len(current_df), len(other_df))
        current_df['reward_avg'] = current_df['reward'].rolling(window=window, min_periods=1).mean()
        other_df['reward_avg'] = other_df['reward'].rolling(window=window, min_periods=1).mean()

        # Create comparison plot
        plt.figure(figsize=(10, 6))
        plt.plot(current_df['episode'], current_df['reward_avg'], 'b-', linewidth=2, label='Current Run')
        plt.plot(other_df['episode'], other_df['reward_avg'], 'r-', linewidth=2, label='Comparison Run')
        plt.title('Training Strategy Comparison')
        plt.xlabel('Episode')
        plt.ylabel(f'Avg Reward ({window} ep window)')
    
```



```

plt.legend()
plt.grid(True)

if output_path:
    plt.savefig(output_path)
plt.show()

# Return statistics for comparison
return {
    'current_run': {
        'episodes': len(current_df),
        'avg_reward': current_df['reward'].mean(),
        'max_reward': current_df['reward'].max(),
        'final_avg': current_df['reward_avg'].iloc[-1] if not current_df.empty else 0
    },
    'comparison_run': {
        'episodes': len(other_df),
        'avg_reward': other_df['reward'].mean(),
        'max_reward': other_df['reward'].max(),
        'final_avg': other_df['reward_avg'].iloc[-1] if not other_df.empty else 0
    }
}

except Exception as e:
    print(f"Error comparing runs: {e}")
    return None

```

7. Training Function

```

def patch_dqn_for_continuous_training(dqn_agent):
    """
    Patch DQNAgent to allow continuous training without warmup reset.
    This modifies the DQNAgent instance to skip warmup on subsequent fit() calls.
    """
    # Store original fit method
    original_fit = dqn_agent.fit

    # Flag to track if we've already done the warmup
    dqn_agent._warmup_done = False

    # Define patched fit method
    def patched_fit(env, nb_steps, **kwargs):
        # If we've already done warmup in a previous fit call
        if dqn_agent._warmup_done:
            # Temporarily set warmup steps to 0
            original_warmup_steps = dqn_agent.nb_steps_warmup
            dqn_agent.nb_steps_warmup = 0

            # Call original fit
            result = original_fit(env, nb_steps, **kwargs)

            # Restore original warmup steps
            dqn_agent.nb_steps_warmup = original_warmup_steps
            return result
        else:
            # First time training, do normal warmup
            result = original_fit(env, nb_steps, **kwargs)
            # Mark warmup as done for future fit calls
            dqn_agent._warmup_done = True
            return result

    # Replace the fit method with our patched version
    dqn_agent.fit = patched_fit.__get__(dqn_agent)
    return dqn_agent

def train_dqn(steps=1000000, save_path='/content/drive/MyDrive/breakout_dqn'):
    """
    Train a DQN agent on Breakout with human-like reward shaping for faster learning.

    Uses reward signals that mimic human motivation in games to accelerate learning
    while maintaining proper incentive alignment between objectives.
    """
    # Verify GPU setup
    physical_devices = tf.config.list_physical_devices('GPU')

```

```

print("GPU devices detected by TensorFlow:", physical_devices)

# Option to force CPU mode if GPU still doesn't work
# os.environ['CUDA_VISIBLE_DEVICES'] = '-1' # Uncomment this line to force CPU

# Check if any GPU devices were found. If not, force CPU mode.
if not physical_devices:
    print("No GPU devices detected by TensorFlow. Forcing CPU mode.")
    os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    # Re-check devices after forcing CPU (should show none)
    print("GPU devices after forcing CPU:", tf.config.list_physical_devices('GPU'))
    print("Using CPU mode for training.")
    # Reduce steps if using CPU to make training faster
    if steps > 1000000:
        print(f"Reducing steps from {steps} to 1000000 for CPU training")
        steps = 1000000
else:
    print("GPU detected and will be used for training.")

# Create save directory if it doesn't exist
os.makedirs(save_path, exist_ok=True)

# Initialize memory-efficient training monitor
monitor = TrainingMonitor(
    log_dir=os.path.join(save_path, 'logs'),
    window_size=100, # Only keep last 100 episodes in memory
    log_interval=10 # Only log every 10 episodes to reduce I/O
)

# Find the latest checkpoint (if any)
latest_step = 0
latest_checkpoint = None

# Regular expression to extract step count from filenames
weight_pattern = re.compile(r'breakout_dqn_weights_(\d+)\.h5')

# Check for existing checkpoint files
if os.path.exists(save_path):
    for filename in os.listdir(save_path):
        match = weight_pattern.match(filename)
        if match:
            step_count = int(match.group(1))
            if step_count > latest_step:
                latest_step = step_count
                latest_checkpoint = os.path.join(save_path, filename)

# Create environment with reward shaping
env, processor = make_env('BreakoutNoFrameskip-v4')

# Set window length for frame stacking
window_length = 4

# Calculate input shape for model
state_shape = (84, 84, window_length)
n_actions = env.action_space.n

# Build DQN model
model = model_template(state_shape, n_actions)
model.summary()

# Use annealed exploration policy for better results
policy = LinearAnnealedPolicy(
    EpsGreedyQPolicy(),
    attr='eps',
    value_max=1.0,
    value_min=0.1,
    value_test=0.05,
    nb_steps=1000000
)

# Configure agent
memory = SequentialMemory(limit=1000000, window_length=window_length)

# Configure agent with no automatic target updates
dqn = DQNAgent(
    model=model,
    nb_actions=n_actions,

```

```

memory=memory,
nb_steps_warmup=50000,
target_model_update=1000000000, # Disable automatic updates, we'll use our callback
policy=policy,
enable_double_dqn=True,
processor=processor
)

# Compile DQN agent
# Ensure Adam from legacy optimizers is used
dqn.compile(Adam(learning_rate=0.00025), metrics=['mae'])

# Load weights if checkpoint exists
if latest_checkpoint:
    print(f"Found checkpoint at step {latest_step}. Resuming training from {latest_checkpoint}")
    # Ensure the optimizer state is not loaded if structure changed, but should be fine here
    dqn.load_weights(latest_checkpoint)
else:
    print("No checkpoint found. Starting training from scratch.")
    latest_step = 0

# Update remaining steps
remaining_steps = steps - latest_step
if remaining_steps <= 0:
    print(f"Training already completed ({latest_step} steps). No further training needed.")
    return dqn

print(f"Training for {remaining_steps} more steps (total target: {steps})")

# Manual checkpointing
checkpoint_interval = 100000
step_count = latest_step

# Custom training with checkpoints and monitoring
while step_count < steps:
    # Determine how many steps to train in this batch
    batch_steps = min(checkpoint_interval, steps - step_count)

    # Setup custom callback for monitoring
    class MonitorCallback(Callback):
        def on_episode_end(self, episode, logs={}):
            monitor.on_episode_end(episode, logs)

    # Create episodic target update callback
    episode_update_callback = EpisodicTargetNetworkUpdate(
        update_frequency=30, # Update target network every 30 episodes
        verbose=1
    )

    # Train for a batch of steps
    # Ensure callbacks are passed correctly
    dqn.fit(env, nb_steps=batch_steps, visualize=False, verbose=2,
           callbacks=[MonitorCallback(), episode_update_callback])

    # Update step count
    step_count += batch_steps

    # Save checkpoint
    filename = f'breakout_dqn_weights_{step_count}.h5'
    filepath = os.path.join(save_path, filename)
    dqn.save_weights(filepath, overwrite=True)
    print(f"Model saved at step {step_count} to {filepath}")

    # Generate and display checkpoint summary
    summary = monitor.on_checkpoint(step_count)
    print("\n==== TRAINING PROGRESS SUMMARY =====")
    for key, value in summary.items():
        print(f"{key}: {value}")
    print("=====\n")

# Save final model weights
final_path = os.path.join(save_path, 'breakout_dqn_final.h5')
dqn.save_weights(final_path, overwrite=True)
print(f"Final model saved to {final_path}")

# Final visualization

```

```

timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
monitor.generate_plots(step_count, timestamp)

env.close()
return dqn

```

8. Test Fucntion

```

def test_model(weights_path, episodes=5):
    """Test a trained model on Breakout"""
    # Create environment with reward shaping
    env, processor = make_env('BreakoutNoFrameskip-v4')

    # Set window length for frame stacking
    window_length = 4

    # Calculate input shape for model
    state_shape = (84, 84, window_length)
    n_actions = env.action_space.n

    # Build DQN model
    model = model_template(state_shape, n_actions)

    # Configure agent
    memory = SequentialMemory(limit=10000, window_length=window_length)
    policy = EpsGreedyQPolicy(eps=0.05) # Low exploration for testing

    dqn = DQNAgent(
        model=model,
        nb_actions=n_actions,
        memory=memory,
        nb_steps_warmup=100,
        target_model_update=10000,
        policy=policy,
        enable_double_dqn=True,
        processor=processor
    )

    # Compile DQN agent
    dqn.compile(Adam(learning_rate=0.00025), metrics=['mae'])

    # Patch the agent to avoid warmup resets
    dqn = patch_dqn_for_continuous_training(dqn)

    # Load weights
    dqn.load_weights(weights_path)

    # Test for episodes
    dqn.test(env, nb_episodes=episodes, visualize=True)

    env.close()

```

9. Analytics Dashboard Function

```

def analyze_training_logs(log_path='/content/drive/MyDrive/breakout_dqn/logs/training_log.csv'):
    """
    Generate interactive analytics dashboard from training logs.

    This function loads saved log data and creates visualizations to analyze
    training performance with minimal memory usage.
    """
    try:
        # Load the training log efficiently (only load what we need)
        df = pd.read_csv(log_path)

        # Check if data exists
        if len(df) == 0:
            print("No training data found in log file.")
            return

        # Calculate rolling averages
        window = min(100, len(df))

```

```

df['reward_avg'] = df['reward'].rolling(window=window, min_periods=1).mean()
df['length_avg'] = df['length'].rolling(window=window, min_periods=1).mean()

if 'mean_q' in df.columns and not df['mean_q'].isna().all():
    df['q_avg'] = df['mean_q'].rolling(window=window, min_periods=1).mean()

# Create visualizations
plt.figure(figsize=(15, 12))

# Plot 1: Episode rewards over time
plt.subplot(2, 2, 1)
plt.plot(df['episode'], df['reward'], 'b-', alpha=0.3)
plt.plot(df['episode'], df['reward_avg'], 'r-', linewidth=2)
plt.title('Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.grid(True)

# Plot 2: Reward distribution histogram
plt.subplot(2, 2, 2)
plt.hist(df['reward'], bins=20)
plt.axvline(df['reward'].mean(), color='r', linestyle='dashed', linewidth=2)
plt.title(f'Reward Distribution (Mean: {df["reward"].mean():.2f})')
plt.xlabel('Reward')
plt.ylabel('Count')

# Plot 3: Episode length over time
plt.subplot(2, 2, 3)
plt.plot(df['episode'], df['length'], 'b-', alpha=0.3)
plt.plot(df['episode'], df['length_avg'], 'r-', linewidth=2)
plt.title('Episode Length Over Time')
plt.xlabel('Episode')
plt.ylabel('Steps')
plt.grid(True)

# Plot 4: Q-values over time (if available)
plt.subplot(2, 2, 4)
if 'mean_q' in df.columns and not df['mean_q'].isna().all():
    plt.plot(df['episode'], df['mean_q'], 'b-', alpha=0.3)
    plt.plot(df['episode'], df['q_avg'], 'r-', linewidth=2)
    plt.title('Mean Q-Value Over Time')
    plt.grid(True)
    plt.xlabel('Episode')
    plt.ylabel('Q-Value')
else:
    plt.text(0.5, 0.5, 'No Q-values recorded yet',
            horizontalalignment='center', verticalalignment='center')
    plt.title('Mean Q-Values (Not Available)')

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/breakout_dqn/logs/training_analytics.png')
plt.show()

# Generate summary statistics
print("\n==== TRAINING ANALYTICS SUMMARY =====")
print(f"Total Episodes: {len(df)}")
print(f"Total Steps: {df['total_steps'].max()}")
print(f"Average Reward: {df['reward'].mean():.2f}")
print(f"Max Reward: {df['reward'].max()}")
print(f"Average Episode Length: {df['length'].mean():.2f}")
print(f"Last 100 Episodes Average Reward: {df['reward'].tail(100).mean():.2f}")
if 'mean_q' in df.columns and not df['mean_q'].isna().all():
    print(f"Average Q-Value: {df['mean_q'].mean():.4f}")
print("=====\n")

return df
except Exception as e:
    print(f"Error analyzing logs: {e}")
    return None

def compare_training_strategies(log_path1, log_path2, labels=None, output_path=None):
    """
    Compare two different training strategies side by side.

    Args:
        log_path1: Path to first training log CSV
        log_path2: Path to second training log CSV

```

```

labels: Tuple of (label1, label2) for the legend
output_path: Path to save comparison image
"""
try:
    # Load logs efficiently
    df1 = pd.read_csv(log_path1)
    df2 = pd.read_csv(log_path2)

    # Use default labels if none provided
    if labels is None:
        labels = ('Strategy 1', 'Strategy 2')

    # Calculate rolling averages
    window = min(100, len(df1), len(df2))
    df1['reward_avg'] = df1['reward'].rolling(window=window, min_periods=1).mean()
    df2['reward_avg'] = df2['reward'].rolling(window=window, min_periods=1).mean()

    # Create comparison plot
    plt.figure(figsize=(12, 8))

    # Rewards
    plt.subplot(2, 1, 1)
    plt.plot(df1['episode'], df1['reward_avg'], 'b-', linewidth=2, label=labels[0])
    plt.plot(df2['episode'], df2['reward_avg'], 'r-', linewidth=2, label=labels[1])
    plt.title('Reward Comparison')
    plt.xlabel('Episode')
    plt.ylabel(f'Avg Reward ({window} ep window)')
    plt.legend()
    plt.grid(True)

    # Episode lengths
    plt.subplot(2, 1, 2)
    df1['length_avg'] = df1['length'].rolling(window=window, min_periods=1).mean()
    df2['length_avg'] = df2['length'].rolling(window=window, min_periods=1).mean()
    plt.plot(df1['episode'], df1['length_avg'], 'b-', linewidth=2, label=labels[0])
    plt.plot(df2['episode'], df2['length_avg'], 'r-', linewidth=2, label=labels[1])
    plt.title('Episode Length Comparison')
    plt.xlabel('Episode')
    plt.ylabel('Avg Length (steps)')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()

    if output_path:
        plt.savefig(output_path)
    plt.show()

    # Compare statistics
    print("\n===== TRAINING STRATEGY COMPARISON =====")
    print(f"{labels[0]} vs {labels[1]}")
    print(f"Episodes: {len(df1)} vs {len(df2)}")
    print(f"Final Avg Reward: {df1['reward_avg'].iloc[-1]:.2f} vs {df2['reward_avg'].iloc[-1]:.2f}")
    print(f"Max Reward: {df1['reward'].max():.1f} vs {df2['reward'].max():.1f}")
    print(f"Avg Episode Length: {df1['length'].mean():.1f} vs {df2['length'].mean():.1f}")
    print("=====\n")

except Exception as e:
    print(f"Error comparing training strategies: {e}")

```

▼ 10. Train the Agent

```
▶train_dqn(300000) # 1M steps for testing, 5M for full training
```

```

... 62716/100000: episode: 1583, duration: 1.771s, episode steps: 55, steps per second: 31, episode reward: 0.705, mean reward: 0.013 [
62744/100000: episode: 1584, duration: 0.965s, episode steps: 28, steps per second: 29, episode reward: -0.489, mean reward: -0.017 [
62832/100000: episode: 1585, duration: 3.027s, episode steps: 88, steps per second: 29, episode reward: 1.906, mean reward: 0.022 [
62937/100000: episode: 1586, duration: 3.328s, episode steps: 105, steps per second: 32, episode reward: 1.918, mean reward: 0.018 [
62976/100000: episode: 1587, duration: 1.267s, episode steps: 39, steps per second: 31, episode reward: -0.479, mean reward: -0.012 [
63006/100000: episode: 1588, duration: 0.977s, episode steps: 30, steps per second: 31, episode reward: -0.492, mean reward: -0.016 [
63032/100000: episode: 1589, duration: 0.865s, episode steps: 26, steps per second: 30, episode reward: -0.494, mean reward: -0.019 [

```

Target network updated after 30 episodes

```

63058/100000: episode: 1590, duration: 0.865s, episode steps: 26, steps per second: 30, episode reward: -0.495, mean reward: -0.019 [
63089/100000: episode: 1591, duration: 1.006s, episode steps: 31, steps per second: 31, episode reward: -0.495, mean reward: -0.016 [

```

```
63116/100000: episode: 1592, duration: 0.893s, episode steps: 27, steps per second: 30, episode reward: -0.494, mean reward: -0.018 [
63141/100000: episode: 1593, duration: 0.910s, episode steps: 25, steps per second: 27, episode reward: -0.495, mean reward: -0.020 [
63167/100000: episode: 1594, duration: 0.936s, episode steps: 26, steps per second: 28, episode reward: -0.495, mean reward: -0.019 [
```

✓ 11. Test a Trained Agent

```
test_model('/content/drive/MyDrive/breakout_dqn/breakout_dqn_final.h5')
```

✓ Analytics Usage

```
# Example usage after training completes
# Analyze a single training run
analyze_training_logs('/content/drive/MyDrive/breakout_dqn/logs/training_log.csv')

# Example of comparing two different training strategies
# compare_training_strategies(
#     '/content/drive/MyDrive/breakout_dqn_reward_shaping/logs/training_log.csv',
#     '/content/drive/MyDrive/breakout_dqn_baseline/logs/training_log.csv',
#     labels=('With Reward Shaping', 'Baseline DQN'),
#     output_path='/content/drive/MyDrive/breakout_dqn/strategy_comparison.png'
# )
```