


✓ Atari Breakout DQN Training

✓ 1. Check GPU Availability

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)
print("GPU is", "available" if tf.config.list_physical_devices('GPU') else "NOT available")
```

 TensorFlow version: 2.18.0
GPU is available

✓ 2. Install Required Packages

```
# Run this cell and restart runtime when prompted
!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
```

 [Show hidden output](#)

✓ 3. Install Atari ROMs

```
# Create a directory for saving models
```

✓ 4. Mount Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

 [Show hidden output](#)

Next steps: [Explain error](#)

✓ 5. keras-rl2 compatibility patch

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

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("Fixed!")
```

Fixed!

✓ 6. Implementation

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, Input
from keras.optimizers.legacy import Adam # Using legacy optimizer for compatibility
from rl.processors import Processor
from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy, LinearAnnealedPolicy
from rl.memory import SequentialMemory
import gymnasium as gym
from gymnasium.wrappers import AtariPreprocessing

class GymCompatibilityWrapper(gym.Wrapper):
    """Wrapper to make gymnasium compatible with keras-rl"""
    def __init__(self, env):
        super().__init__(env)

    def step(self, action):
        """Update step method to match keras-rl output"""
```

```

    obs, reward, terminated, truncated, info = self.env.step(action)
    done = terminated or truncated
    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 StackDimProcessor(Processor):
    """Custom processor to resolve dimension mismatches and clip rewards"""
    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_reward(self, reward):
        """Clip reward to [-1, 1] to stabilize training"""
        return np.clip(reward, -1.0, 1.0)

def make_env(env_id):
    """Creates a wrapped Atari environment for use with keras-rl2"""
    env = gym.make(env_id)

    # Apply Atari preprocessing
    env = AtariPreprocessing(
        env,
        noop_max=30,          # No-op action for up to 30 frames
        frame_skip=4,         # Skip every 4 frames
        screen_size=84,       # Resize to 84x84
        terminal_on_life_loss=True, # End episode on life loss
        grayscale_obs=True,   # Convert to grayscale
        grayscale_newaxis=True, # Keep the channel dimension
        scale_obs=False,      # Do not scale observations
    )

```

```

# Make compatible with keras-rl
env = GymCompatibilityWrapper(env)

return env

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

```

▼ 7. Training Function

```

def train_dqn(steps=1000000, save_path='/content/drive/MyDrive/breakout_dqn'):
    """
    Train a DQN agent on Breakout and automatically resume from latest checkpoint
    """

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

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

    # Regular expression to extract step count from filenames
    import re
    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
    env = 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)

dqn = DQNAgent(
    model=model,
    nb_actions=n_actions,
    memory=memory,
    nb_steps_warmup=50000,
    target_model_update=10000,
    policy=policy,
    enable_double_dqn=True,
    processor=StackDimProcessor()
)

# Compile DQN agent
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_ch
    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 nee
    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
while step_count < steps:
    # Determine how many steps to train in this batch
    batch_steps = min(checkpoint_interval, steps - step_count)

    # Train for a batch of steps
    dqn.fit(env, nb_steps=batch_steps, visualize=False, verbose=2)

    # 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}")

# 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}")

env.close()
return dqn
```

✓ 7. Test Fucntion

```
def test_model(weights_path, episodes=5):
    """Test a trained model on Breakout"""
    # Create environment
    env = 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=StackDimProcessor()
)

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

# Load weights
dqn.load_weights(weights_path)

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

env.close()
```

✓ 8. Train the Agent

```
train_dqn(3000000) # 1M steps for testing, 5M for full training
```

```
... 80428/100000: episode: 2071, duration: 2.750s, episode steps: 25, steps per second:
    80456/100000: episode: 2072, duration: 2.478s, episode steps: 28, steps per second:
```

✓ 9. Test a Trained Agent

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

