



Deep RL agents in PyTorch

Graded Assignment 2

DTE-2505-1 25H

Neural Networks

Tatiana Gorokhova

14.11.2025

Deep Q-Learning Snake Agent

Repository description

- This repository contains a Reinforcement Learning implementation of the Snake game, originally built with multiple agent types (Q-learning, Monte-Carlo, ADP, Deep Q-Learning). The implementation uses Tensorflow framework.
The focus is on testing different RL methods in the same grid-world environment.

Assignment Description

For this project, the task was to:

- Take the Deep Q-Learning agent from the original repo
- Rewrite it into PyTorch
- Clean the repository by removing unused agents, logs, and training scripts
- Run training experiments and evaluate how well the DQN agent learns to play Snake
- The goal is to produce a working, simplified PyTorch DQN agent and document the results.

Project summary

Removed (Not Needed for This Assignment):

- PolicyGradientAgent
- AdvantageActorCriticAgent
- SupervisedLearningAgent
 - All inherited from the old TensorFlow DeepQLearningAgent → not compatible
 - Out of scope for the DQN-only assignment
- supervised_training.py
 - Trained TensorFlow agents - removed
- Old logs, saved TF models, unused directories
 - No longer relevant after rewriting
- HamiltonianCycleAgent - hard-coded heuristic agent
- BreadthFirstSearchAgent - non-learning search agent
- No learning, do not use TensorFlow but are out of scope for the assignment so they are removed for simplicity

Kept (Needed for DQN Functionality)

- Base Agent class
 - Required parent class for the new DQN agent
 - Environment and snake game logic
-
- ## Changed
- agent.py:
 - New TorchDQN class
 - PyTorch neural network replacing old TF model
 - DeepQLearningAgent
 - Rewritten to work with PyTorch
 - Updated methods
 - training.py
 - Removed imports of deleted agents
 - Removed loops checking multiple agent types
 - Simplified to run only DQN
 - utilities.py & game_visualization.py
 - Cleaned unused imports and outdated TF-related components

agent.py: Utilities

huber_loss

- Custom implementation of the Huber loss, originally written for TensorFlow and now adapted for PyTorch.
- Used in DQN because it is:
 - Quadratic for small errors → stable gradient
 - Linear for large errors → less sensitive to outliers
- Helps avoid exploding gradients and makes Q-value updates smoother and more stable.

mean_loss

- Calculates the average Huber loss across a training batch.
- Useful for:
 - Reducing noise in updates
 - Reporting clean, consistent training metrics

agent.py: class TorchDQN

Why we need TorchDQN class:

- Replaces the old Keras/TensorFlow model with a PyTorch implementation.
- Keras built the model dynamically by applying build-in functions on tensors. PyTorch prefers an explicit nn.Module with a forward() method.
- We also needed one reusable model object that:
 - Reads the same JSON config as before
 - Handles channel order correctly for PyTorch
 - Outputs Q-values for all actions.

What TorchDQN class does:

- Parses JSON model config
 - Reads the same architecture description as the original code
 - Extracts layer definitions and parameters
- Finds and builds layers
 - Convolution, pooling, flatten, and dense (linear) layers
 - Uses those config keys to create corresponding nn.Module layers
- Maps activations to PyTorch
 - Converts activation names (e.g. "relu", "tanh") to PyTorch functions
- Handles data layout
 - Fixes channel order (e.g. from HWC to CHW) if needed
- Implements forward()
 - Defines how input goes through conv → pool → flatten → dense stack
 - Returns Q-values with shape (batch_size, n_actions)

agent.py: class DeepQLearningAgent:

DeepQLearningAgent inherits from the base Agent class

What the Agent Does:

- Connects the Snake environment to the TorchDQN neural network
- Converts board state → tensors → Q-value predictions
- Selects actions using ϵ -greedy
- Performs network updates + target network sync
- Saves and loads trained models
- Manages the entire DQN training loop

What I Kept

- normalize_board()
 - Converts integer board to normalized tensor
 - Kept identical input encoding → model sees same structured data as original code
- Inheritance from Agent
 - Maintains same agent interface for consistency

What I Modified (PyTorch)

- Model setup & forward pipeline
 - `__init__`, `reset_models`, `prepare_input`,
- Action selection & target network logic
 - `_get_model_outputs`,
`move`, `agent_model`, `update_target_net`
- Action probability calculation
 - `get_action_proba`
- Model saving, loading, printing
 - `save_model`, `load_model`, `print_model`
- Training loop inside the agent
 - `train_agent`

class DeepQLearningAgent: model setup and input handling

__init__

- Initializes the DQN agent
- Sets hyperparameters (γ , ϵ , lr, update frequency, batch size, etc.)
- Loads the model configuration JSON
- Creates two networks:
 - online network → self.model
 - target network → self.target_net
- Selects optimizer (Adam)
- Updated for PyTorch: no TF sessions, no Keras model, direct nn.Module usage

prepare_input

- Converts raw board state into a PyTorch tensor
- Adds batch dimension
- Fixes channel order (HWC → CHW if needed)
- Casts to float32
- Moves tensor to device (CPU/GPU)
- Needed because TensorFlow used NHWC; PyTorch uses NCHW

reset_models

- Recreates the online and target networks from the JSON config
- Ensures both nets have identical structure
- Syncs target network weights with the online network
- Changed: now uses TorchDQN() instead of TensorFlow model builder

class DeepQLearningAgent: action selection and target network

_get_model_outputs

- Prepares board tensor → passes it through the online network
- Uses the model's implementation
- Returns raw Q-values for all actions PyTorch forward()
- Rewritten to replace TF's: model.predict(board)

move

- The decision-making core of the agent
- Steps:
 - Calls `_get_model_outputs` to obtain Q-values
 - Chooses `argmax(Q)` as best action
 - Returns both the action and Q-values
- Provides replacement for TensorFlow's graph execution

_agent_model

- reads the JSON, then builds and returns a new TorchDQN model with the correct architecture.

update_target_net

- Periodically copies online network weights → target network
- Prevents feedback instability during Q-learning
- Essential for stabilizing TD-targets during training

class DeepQLearningAgent: action probability

get_action_proba

- Computes the probability distribution over actions based on Q-values
- Steps:
 1. Takes the Q-values from `_get_model_outputs`
 2. Applies softmax to convert them into normalized probabilities
 3. Returns a vector of action probabilities (sums to 1)

save_model

- Saves the online and target network weights to disk
- Stores .pt files for reproducibility and grading
- Replaces TensorFlow checkpoint system with torch method

load_model

- Loads saved weights back into TorchDQN models
- Allows running a trained agent without retraining
- Ensures consistent architecture + weights

print_model

- Prints model architecture (layer types, sizes)
- Helpful for debugging and verifying that PyTorch correctly parsed the JSON config
- Replaces Keras `.summary()` calls

class DeepQLearningAgent: train_agent

What train_agent Does Now:

1. Sample replay buffer & build tensors

- Takes a minibatch of transitions:
 $(s, a, r, s', \text{done})(s, a, r, s', \text{done})$
- Converts them to PyTorch tensors and moves them to the correct device
- Normalizes states (e.g. state / 4.0)

2. Handle actions

- Supports both one-hot and index action formats
- Always ends up with an index tensor a_idx of shape (B, 1)

3. Compute TD target

- Computes: $y = r + \gamma \cdot \max(a')Q(s', a')(1 - \text{done})$
- Uses target network if enabled
- Masks illegal moves with -inf before max to ignore them

4. Predictions

- Gets $Q(s, \cdot)$ from the online network
- Uses gather to select $Q(s, a)$ for each action in the batch

5. Loss & optimization

- Computes loss using custom Huber loss (mean_huber_loss_torch)
- Standard PyTorch training step:
 - `optimizer.zero_grad()`
 - `loss.backward()`
 - `optimizer.step()`

6. Return

- Returns a scalar loss value
- Used for logging and plotting training curves

training.py: cleanup

Before (Original Version):

- Supported many different agents:
 - Q-learning, MC, ADP, DeepQLearning, PolicyGradient, A2C, Supervised, etc.
- Large if/else blocks to check agent type
- Multiple training loops and logging logic mixed together
- TensorFlow-specific pieces tied to the old DeepQLearningAgent

Now (Simplified Version):

- Focuses on one agent only:
 - DeepQLearningAgent (PyTorch DQN)
- Removed:
 - Imports of deleted agents (PolicyGradient, A2C, Supervised, etc.)
 - All loops that switched behavior based on agent type
- Kept:
 - Single main training loop:
 - create env + DQN agent
 - run episodes
 - call train_agent()
 - log: iteration, reward_mean, length_mean, games, loss

Training results

Tracking features:

- Autosaving online + target models every 500 iterations
- CSV logging of: iteration, reward mean, snake length mean, number of games, loss

First Test Run (short – 1000 iterations, see image on the right)

- Quick test run to verify code and environment
- Agent mostly achieved reward ≈ -1 , snake length ≈ 2
- Confirmed that the pipeline worked, but needed more training for real learning

Full Run — 200,000 Iterations

Final Training Statistics:

- Reward range: Min: -1.0; Max: 23.75
→ Shows clear improvement and meaningful learned behaviour
- Loss range: Min: 0.001; Max: 0.24
→ Loss decreases and stabilizes → convergence
- Snake length range: Min: 2.0; Max: 26.75
→ Agent reliably finds food and survives longer

```
iteration,reward_mean,length_mean,games,loss
50,-1.0,2.0,9.0,0.12205212563276291
100,-1.0,2.0,8.0,0.07257646322250366
150,-1.0,2.0,8.0,0.05780348926782608
200,-1.0,2.0,8.0,0.04835512489080429
250,-1.0,2.0,8.0,0.02249867282807827
300,-1.0,2.0,8.0,0.009632958099246025
350,-1.0,2.0,8.0,0.009949893690645695
400,-1.0,2.0,8.0,0.004947303328663111
450,-1.0,2.0,9.0,0.004115710500627756
500,-1.0,2.0,9.0,0.012053604237735271
550,-1.0,2.0,11.0,0.009371839463710785
600,-1.0,2.0,11.0,0.019227461889386177
650,-1.0,2.0,9.0,0.014113394543528557
700,-1.0,2.0,8.0,0.0028047808445990086
750,-1.0,2.0,8.0,0.0030090450309216976
800,-1.0,2.0,8.0,0.008918952196836472
850,-1.0,2.0,8.0,0.02016604319214821
900,-0.9,2.1,10.0,0.01137166004627943
950,-1.0,2.0,9.0,0.017438044771552086
1000,-1.0,2.0,8.0,0.017791882157325745
```

Conclusion

Objective 1: Rewrite Deep Q-Learning Agent in PyTorch

- Successful conversion of the original TensorFlow/Keras implementation
- Created a clean, modular TorchDQN class
- Rebuilt the DeepQLearningAgent around PyTorch modules, tensors, and optimizer flow

Objective 2: Simplify and Clean the Repository

- Removed unused agents, TF models, logs, and training scripts
- Kept only components relevant to the DQN assignment
- Simplified training.py to a single-agent PyTorch training loop

Objective 3: Ensure Functionality and Trainability

- All functionality (action selection, target net, replay sampling, saving/loading) restored in PyTorch
- Long training run (200k iterations) demonstrates successful learning:
 - Reward improved from -1 to 23.75
 - Snake length up to 26.75
 - Loss stabilized at low values

Objective 4: Interpret and Present Training Results

- Model checkpoints saved automatically every 500 iterations
- Training statistics logged for analysis
- Final model and metrics provide clear evidence of learning progress

References

Original Repository

Snake-RL (TensorFlow version)

Source code for the original multi-agent Snake RL framework.

Used as the baseline for architecture, agent structure, and JSON model configuration.

<https://github.com/DragonWarrior15/snake-rl>

Assignment Repository

GA02 — PyTorch Deep Q-Learning Rewrite

Contains the cleaned project, rewritten DQN agent in PyTorch, simplified training loop, and final trained models.

<https://github.com/t-ango/ga2>