BME646 and ECE60146: Homework 11

Spring 2026
Due Date: Sunday, May 3, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. **Late submissions will be accepted with penalty: -10 points per late day, up to 2 days.**

1 Introduction

The last four homeworks trained neural networks by showing them labels. Metric learning had you pulling together same-class embeddings and pushing apart different-class ones. Generative modeling had you minimizing an ELBO or matching a noise prediction. Transformers learned from parallel corpora of sentence pairs. BERT and GPT ate terabytes of text with either a masking objective or a next-token objective.

Reinforcement learning does away with labels. You hand the learner an environment and a scalar reward signal, and the learner has to figure out the rest by trying things and observing what happens. There is no training set in the usual sense: the data is whatever the agent has happened to experience so far, and that distribution shifts as the policy improves. This shift is what makes RL hard, and what makes it the right framework for problems where the right answer cannot be written down in advance.

This homework has three tasks and an optional PPO bonus. Task 1 is a warm-up with tabular Q-learning on CartPole. Task 2 replaces the Q-table with a neural network and asks you to land a small rocket softly on a pad (LunarLander). Task 3 jumps over to language models and asks you to train the reward model from Slide 35 of the lecture on a real human-preference dataset, the same one behind OpenAI's "Learning to Summarize from Human Feedback" paper. Along the way you will observe how the network trains on this dataset and be asked to reflect on what you see.

The bonus is an optional PPO implementation on LunarLander for those who want to see a full policy-gradient pipeline.

1.1 Quick reference: lecture slide numbers

Use this table to look up the lecture material for each topic.

HW11 topic	Lecture	Slides
Basic RL vocabulary		
Agent, environment, state, action, reward	RL [1]	48–53
Markov Decision Process	RL	54–56
Future expected reward, quality index Q	RL	57–59
Q-Learning (Tasks 1–2)		
Q-Learning and the update equation	RL	60–66
Tabular Double Q on Cart-Pole	RL	67–82
Neural Q-Learning, replay memory	RL	83–95
RLHF and reward modeling (Task 3)		
RLHF notation, sextuples $(x, y_1, y_2, y_3, y_4, b)$	RL	9–16
Reward modeling loss, Eq. (6)	RL	17–21
Language-model simulator and RewardNetwork	RL	22–35
PPO and policy gradients (Bonus)		
Value-function vs. policy-based methods	RL	96–99
Policy gradients, $\nabla_\theta J(\theta)$	RL	100–108
PPO, advantage function, clipping	RL	36–47

Which slides to read for each task:

Task	Key slides
Task 1: Tabular Double Q on CartPole	RL 60–82
Task 2: DQN on LunarLander	RL 60–66, 83–95
Task 3: Reward modeling	RL 9–35
Bonus: PPO on LunarLander	RL 36–47, 96–108

2 Setting up your environment

2.1 Prerequisites from previous homeworks

Ensure you have:

- Python 3.8+ with PyTorch, NumPy, Matplotlib installed
- A Google Colab account. A GPU runtime is helpful for Task 2 and Task 3 but not strictly required. CartPole (Task 1) runs comfortably on CPU.

2.2 Installing Gymnasium

This homework uses Gymnasium, the maintained fork of OpenAI Gym. The lecture notes were written against the old `gym` package; you should use `gymnasium` instead. The API is almost identical but has two important differences that matter for your code:

- `env.reset()` now returns a tuple `(obs, info)`, not just `obs`.
- `env.step(action)` now returns a 5-tuple `(obs, reward, terminated, truncated, info)` instead of a 4-tuple. An episode ends when *either* `terminated` or `truncated` is true.

Install with:

```
pip install gymnasium
pip install "gymnasium[box2d]"
pip install swig
```

The `box2d` extra is required for LunarLander. If the install fails with a swig error, run `pip install swig` first, then retry. On Colab this usually just works.

To render video of a trained episode (useful for Task 2), also install:

```
apt-get install -y xvfb
pip install imageio imageio-ffmpeg
```

2.3 Environments used in this homework

CartPole-v1 (Task 1). The pole-and-cart problem from the lecture. State is a 4-vector (cart position, cart velocity, pole angle, pole angular velocity). Two actions (push left, push right). Reward is +1 per timestep the pole stays up. The environment considers CartPole-v1 "solved" at an average return of 475 over 100 episodes.

LunarLander-v3 (Task 2 and Bonus). You control a small lander and try to land it softly on a pad between two flags. State is an 8-vector: lander (x, y) position, (x, y) velocity, angle, angular velocity, and two booleans indicating whether each leg is touching the ground. Four discrete actions: do nothing, fire left engine, fire main engine, fire right engine. Reward is shaped: you get points for moving toward the pad, bonus points for landing, a penalty for crashing, and a small fuel cost per engine firing. The environment is considered "solved" at an average return of 200 over 100 consecutive episodes.

```
import gymnasium as gym

env = gym.make("LunarLander-v3")
obs, info = env.reset(seed=0)
print(obs.shape)  # (8,)
print(env.action_space.n)  # 4
```

Note on environment versions. If your Gymnasium install only has LunarLander-v2, that is fine: `gym.make("LunarLander-v2")` works identically for the purposes of this homework. Just note which version you used in your report.

3 Programming tasks (100 points + 20 bonus)

3.1 Task 1: Tabular Double Q-Learning on CartPole (15 points)

Objective: Run the Double Q-learning implementation from Slides 78–81 of the RL lecture on CartPole, reproduce the training curve, and study one hyperparameter ablation.

3.1.1 Reproducing the baseline (8 points)

The lecture posts a complete implementation of Double Q-learning with a discretized state space and replay memory (Slides 78–81, Rohan Sarkar's code). Copy this into a file `tabular_dq.py` and run it on CartPole.

Two small edits are needed to port the lecture code from `gym` to `gymnasium`:

```
# OLD (from the lecture slides):
state = env.reset()
next_state, reward, done, info = env.step(action)

# NEW (gymnasium):
state, info = env.reset()
next_state, reward, terminated, truncated, info = env.step(
    action)
done = terminated or truncated
```

Use `CartPole-v1` instead of `v0`. Train for at least 1000 episodes.

Deliverables:

- Your `tabular_dq.py` (you may cite the lecture code)
- A plot of episode duration (y-axis) vs. episode number (x-axis) for the training run, with a moving average overlaid. The plot should show the duration rising from small values near the start to higher values as training proceeds.
- The average episode duration over the last 100 training episodes, reported as a single number.

3.1.2 One ablation (7 points)

Pick *one* of the following and rerun training with the modified setting. Use the same seed and the same number of episodes as your baseline so the comparison is clean.

(a) Change the discount factor γ from 0.85 to either 0.5 or 0.99. Report what happens.

(b) Change the state discretization: use a coarser grid (e.g., 2 bins per dimension instead of 3) or a finer one. Report what happens.

(c) Replace the double-Q update with standard single Q-learning: use one table Q and always write $Q[s][a] \leftarrow Q[s][a] + \alpha\big(r + \gamma \max_{a'} Q[s'][a'] - Q[s][a]\big)$. Report what happens.

Deliverables:

- A single plot showing the training curves of the baseline and the ablation on the same axes, with a legend.
- One paragraph (5–8 sentences) explaining what you changed and what effect it had. If the ablation made learning worse, say why you think so based on what you changed. If it made no visible difference, say that.

Grading:

- Baseline run with curve and final number (8 points)
- Ablation with comparison plot and discussion (7 points)

3.2 Task 2: DQN on LunarLander (35 points)

Objective: Implement a Deep Q-Network following the structure on Slides 83–95 of the lecture (with the code itself on Slides 92–94), but with LunarLander's continuous 8-dimensional state instead of a discretized CartPole state. Train it until the environment is solved or you run out of episodes.

3.2.1 Background

The tabular approach of Task 1 does not extend to LunarLander. The state is an 8-vector of real numbers, and a useful discretization would require an enormous Q-table with very few visits per cell. The neural-network approach

from Slides 83–95 replaces the table with a small feedforward network $Q_\phi(s)$ that outputs a value for each of the four actions given a state s.

Two stabilization tricks from the lecture are required for LunarLander to work:

- **Experience replay**: store transitions $(s, a, r, s', \text{done})$ in a buffer and sample minibatches at random rather than training on the most recent transition. This decorrelates consecutive samples. See the `ReplayMemory` class on Slide 92.
- **Target network**: maintain a second copy Q_{ϕ^-} of the Q-network whose weights are held fixed for many steps and used to compute the TD target. Without this, the target moves every time the online network updates, and training becomes unstable. The lecture code on Slide 94 does not use a separate target network; you should add one for LunarLander (the environment is harder than CartPole and training without a target network often diverges).

The Bellman target for a transition $(s, a, r, s', \text{done})$ is

$$y = r + \gamma\,(1 - \text{done})\,\max_{a'} Q_{\phi^-}(s', a')$$

and the loss is $\mathcal{L}(\phi) = \big(Q_\phi(s, a) - y\big)^2$ averaged over the minibatch. Detach the target so autograd treats y as a constant (for example, `y = y.detach()`, or wrap the target forward pass in `torch.no_grad()`). Without the detach, gradients propagate through the target network unnecessarily; and if you ever collapse ϕ^- back into ϕ (for example, use the online network for both), those stray gradients will corrupt the update. The lecture code on Slide 94 calls `.detach()` explicitly on the target forward pass for this reason.

3.2.2 Implementation requirements (30 points)

Implement DQN with the following components:

1. A Q-network with two hidden layers of 128 units and ReLU activations. Input is the 8-dim state, output is a 4-dim vector of action values.
2. A replay buffer of size at least 50,000 transitions.
3. A target network that is either (a) copied from the online network every N steps (hard update, try $N = 500$ or $N = 1000$), or (b) updated via Polyak averaging with $\tau = 0.005$. Either is fine; say which you used.

4. ϵ-greedy exploration with ϵ annealed from 1.0 to 0.05 linearly over the first 10% to 30% of training.

5. The Adam optimizer with learning rate in the range 10^{-4} to 10^{-3}.

Train for up to 1000 episodes or until the environment is solved (average return ≥ 200 over the last 100 episodes), whichever comes first. On a Colab T4 this takes roughly 15–30 minutes.

```python
# Suggested skeleton
import torch
import torch.nn as nn
import gymnasium as gym
from collections import deque
import random

class QNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=4):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 128), nn.ReLU(),
            nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, action_dim),
        )
    def forward(self, s):
        return self.net(s)

class ReplayBuffer:
    def __init__(self, capacity=50000):
        self.buf = deque(maxlen=capacity)
    def push(self, s, a, r, s_next, done):
        self.buf.append((s, a, r, s_next, done))
    def sample(self, batch_size):
        return random.sample(self.buf, batch_size)
    def __len__(self):
        return len(self.buf)

env = gym.make("LunarLander-v3")
# ... build Q-network, target network, optimizer ...
# ... training loop with epsilon-greedy, replay, target updates
    ...
```

Deliverables:

- Your `dqn_lander.py`
- A training curve showing episode return vs. episode number. Overlay a moving average over 100 episodes.

- The average return over the last 100 training episodes, reported as a single number. State whether you solved the environment.
- A recorded video (or animated GIF) of one episode played by your trained agent. A rough Colab recipe is in the FAQ.
- A saved checkpoint file (e.g., `dqn_lander.pt`) containing your trained Q-network weights.

3.2.3 Brief analysis (5 points)

Answer each question in one short paragraph (3–5 sentences each):

- **Why a target network?** What would go wrong if you used the same Q_ϕ to produce both the predicted value and the TD target? Frame your answer in terms of what happens to the regression target during a single gradient step.
- **ϵ-greedy decay.** You annealed ϵ from 1.0 to 0.05 over training. Why not start at $\epsilon = 0.05$? Why not stay at $\epsilon = 1.0$?

Grading:

- DQN implementation, training curve, video, checkpoint (30 points)
- Two analysis paragraphs (5 points)

3.3 Task 3: Reward modeling on real human preferences (50 points)

Objective: Train Prof Kak's `RewardNetwork` from Slide 35 on a real human-preference dataset: OpenAI's *Learning to Summarize from Human Feedback* comparisons. You will study the code's behavior during training and reflect on what you observe.

3.3.1 Background: what is reward modeling, and who is the "critic"?

A quick taxonomy, because these terms get tangled. In the standard RL picture from Tasks 1–2, an agent takes actions in an environment and receives rewards that are *given* by the environment (land the rocket $\rightarrow +100$, crash $\rightarrow -100$). Training the agent is just learning a policy that maximizes these rewards.

In RLHF the setup is different. The "environment" is a language model generating text in response to prompts, and there is no built-in reward for "this response was helpful" or "this response was not safe." So before you can apply RL to a language model, you need to *learn* the reward function from human data. That learning step is what this task does.

Concretely: you collect a dataset where, for each prompt, a human has looked at two or more candidate responses and picked the one they preferred. You then train a small neural network (the "reward model") to predict which response the human would pick. This network takes a (prompt, response) pair and outputs a scalar $r(x, y)$. After training, $r(x, y)$ stands in for human judgment during the later RL step.

The word "critic" does not appear in this task. A critic is an RL term for a value function used alongside an actor (policy) in actor-critic methods like PPO. Task 3 does not do any RL: it is supervised learning on a ranking dataset, and its output (the reward model) is what the *next* step of a full RLHF pipeline would use. The next step (PPO fine-tuning of the language model) is beyond the scope of this homework; the optional Bonus task does PPO, but on LunarLander, where the reward comes from the environment and no reward model is needed.

3.3.2 The dataset

You will use `openai/summarize_from_feedback` from Hugging Face. This is the preference data behind the 2020 Stiennon et al. paper "Learning to Summarize from Human Feedback," which was the blueprint for the later ChatGPT-style RLHF pipelines.

Each example has three fields you will use:

- `info.post`: a Reddit post (the "prompt" x).
- `summaries`: a list of two candidate summaries, each with a `text` field.
- `choice`: 0 or 1, indicating which of the two summaries a human annotator preferred.

You will take the first few hundred examples from the `comparisons` split, embed each (post, summary) pair with a pretrained language model, and train the reward network to score the chosen summary higher than the rejected one.

3.3.3 One small adaptation of Slide 20

The lecture defines the reward-modeling loss for sextuples with four candidate responses (Eq. 6, Slide 20):

$$\mathcal{L} = -\sum_{j=1}^{M} \log \frac{e^{r(x,y_b)}}{\sum_{k=1}^{4} e^{r(x,y_k)}}$$

This dataset has only two candidates per prompt (chosen vs. rejected). When you specialize Eq. (6) to two responses, the sum in the denominator has two terms, and the expression reduces to

$$\mathcal{L} = -\sum_{j=1}^{M} \log \sigma\big(r(x, y_{\text{chosen}}) - r(x, y_{\text{rejected}})\big)$$

where σ is the logistic sigmoid. This is the Bradley–Terry preference model and it is what RLHF papers use. It is mathematically the same loss as Eq. (6) when you set the number of candidates to two.

3.3.4 Starter code

We provide `lm_utils.py`, which handles the pieces that are not the focus of the task:

- Loads `distilgpt2` (an 82M-parameter distilled GPT-2 from Hugging Face) as a frozen sentence embedder. Its weights are never updated.
- Takes any string and returns a 768-dimensional vector by mean-pooling the last-layer hidden states. This vector is the input feature for the reward network.
- Loads the `openai/summarize_from_feedback` dataset and extracts (post, chosen, rejected) triples.

Note: Both `distilgpt2` and the dataset are downloaded from Hugging Face the first time the notebook runs. Make sure your Colab runtime has internet access. On a T4 GPU, embedding 500 (post, chosen, rejected) triples takes about 3 minutes. On CPU, about 10 minutes. No GPU is strictly required.

3.3.5 3.1 Load the dataset and look at it (8 points)

Load the comparisons split, extract the first $N = 500$ examples, and print the post, chosen summary, and rejected summary for three of them.

```
from lm_utils import load_preference_dataset

triples = load_preference_dataset(n_examples=500)
print(f"loaded {len(triples)} triples")

for i in range(3):
    t = triples[i]
    print(f"\n=== Example {i+1} ===")
    print(f"POST:     {t['post'][:200]!r}...")
    print(f"CHOSEN:   {t['chosen']!r}")
    print(f"REJECTED: {t['rejected']!r}")
```

Deliverables:

- Code that loads the dataset.
- Printed output for three examples showing the post, chosen summary, and rejected summary.
- Two or three sentences describing what the examples look like. Are the chosen summaries visibly better to you? In what way?

3.3.6 3.2 Embed everything (12 points)

For each triple, produce three 768-dimensional vectors: one for the post, one for the chosen summary, one for the rejected summary. The helper function `embed_triples` does this in a single call.

```
from lm_utils import load_gpt2, embed_triples

tokenizer, base, device = load_gpt2()
embedded = embed_triples(triples, tokenizer, base, device)
print(f"embedded {len(embedded)} triples")
print(f"post embedding shape:  {embedded[0]['post_emb'].shape}"
    )
```

Deliverables:

- The total number of embedded triples.
- The shape of one example post embedding. This is a quick sanity check that the embeddings came out at the expected dimension.
- A split into train and validation sets. Use the first 80% for training and the last 20% for held-out evaluation.

3.3.7 3.3 Train the RewardNetwork (20 points)

The `RewardNetwork` class is taken from Slide 35 of the lecture. Copy the class as shown into your `train_reward_model.py` without modifications.

For each training triple, build two flat 2304-dim input vectors, each formed by concatenating three 768-dim embeddings (post, SEP, response):

- [post_emb, sep_emb, chosen_emb] $\rightarrow$ feed through the network $\rightarrow r_{\text{chosen}}$
- [post_emb, sep_emb, rejected_emb] $\rightarrow$ feed through the network $\rightarrow r_{\text{rejected}}$

The separator embedding sep_emb can be a fixed pretrained embedding for some generic token (the starter code uses the embedding of the string `"SEP"`). Compute the pairwise loss

$$\ell = -\log \sigma(r_{\text{chosen}} - r_{\text{rejected}})$$

and take a gradient step on the reward network's parameters (not on the embedding model, which is frozen). Train for 2000 gradient steps with Adam and a learning rate of 10^{-3}. Use one triple per gradient step (no minibatching). With 400 training triples, this works out to roughly 5 passes over the training set.

Deliverables:

- Your `train_reward_model.py`.
- A loss curve plot showing the per-step loss and a moving average with window 200.
- The initial loss and the final (last-200-step average) loss. Record whatever numbers you actually get.

3.3.8 3.4 Evaluate and critique (10 points)

On the *validation* set (the 20% you held out), compute one number: the fraction of triples for which $r_{\text{chosen}} > r_{\text{rejected}}$. Random guessing gives 50%. A reward model that has learned something about human preferences should exceed 50%.

Then answer these questions. Keep each answer short (a few sentences each).

1. Report your validation accuracy and what happened to the training loss. How did the values you saw compare to what you would expect from a reward model that is learning to capture human preferences?

2. Look carefully at the `RewardNetwork` class from Slide 35. Work out what values the forward pass can return: start with the three `clamp(min=0)` operations, then the final `sigmoid`, and describe the range of outputs. Given the pairwise loss $-\log \sigma(r_{\text{chosen}} - r_{\text{rejected}})$, how does this output range affect the loss values you can reach during training?

3. Suggest one modification to the `RewardNetwork` that you think would expand the range of its outputs and allow the loss to decrease further. You are not required to implement it or verify that it works; a short description is enough.

Grading:

- Dataset loading and examples (8 points)
- Embedding and train/val split (12 points)
- Training loop and loss curve (20 points)
- Validation accuracy and critique (10 points)

4 Bonus Task: PPO on LunarLander (20 points)

This task is entirely optional. Points earned here are added on top of the 100-point base from Tasks 1–3.

Objective: Implement PPO on LunarLander and compare it against your DQN from Task 2. PPO is the policy-gradient algorithm that sits under the hood of most production RLHF pipelines; seeing it work on a non-language environment first gives you a cleaner picture of what the clipping and the advantage actually do, separate from any language-model machinery.

Scope warning. A correct PPO implementation has more moving parts than Q-learning: a value network for the advantage baseline, generalized advantage estimation (GAE), the clipped surrogate objective, multiple epochs over each rollout, and mini-batching within each epoch. Plan accordingly.

4.1 What to implement

1. An actor network $\pi_\theta(a \mid s)$ (two hidden layers of 128 units, softmax over 4 actions) and a separate critic network $V_\phi(s)$ with the same body but a scalar output.

2. Generalized advantage estimation (GAE) for the advantage A_t. A reasonable default is $\lambda = 0.95$, $\gamma = 0.99$.

3. The clipped surrogate objective from Slides 42–43 of the lecture, with $\epsilon = 0.2$.

4. 4–10 epochs of SGD per rollout, with minibatches of size 64.

5. A value-function loss $\frac{1}{2}(V_\phi(s_t) - G_t)^2$ added to the total loss, plus an entropy bonus to encourage exploration.

You are not required to derive any of this from scratch. The CleanRL project [8] publishes a clean, single-file PPO implementation you may study and adapt. If you do, cite it clearly in your report.

4.2 Deliverables

- Your `ppo_lander.py`
- A training curve comparison: DQN (Task 2) and PPO (this task) on the same axes. Use moving averages.
- Average return over the last 100 episodes for PPO.
- One paragraph comparing PPO against DQN on LunarLander: which solved the environment faster, which was easier to tune, which you would pick for a new problem.

Grading:

- Working PPO implementation with training curve (15 points)
- Comparison plot and written comparison (5 points)

5 Submission instructions

5.1 What to submit in the report

Read this section carefully. Incomplete submissions will lose points. Every item listed below must be present in your submission.

1. **Task 1: Tabular Double Q on CartPole (15 points)**
   (a) **Code:** `tabular_dq.py`

(b) **Figure:** Baseline training curve with moving average

(c) **Text:** Final average duration over last 100 episodes

(d) **Figure:** Baseline vs. ablation on the same plot

(e) **Text:** Ablation discussion paragraph

2. **Task 2: DQN on LunarLander (35 points)**

   (a) **Code:** `dqn_lander.py`

   (b) **Figure:** DQN training curve with moving average

   (c) **Text:** Final 100-episode average return; whether solved

   (d) **Media:** Video or GIF of a trained episode

   (e) **File:** Saved checkpoint `dqn_lander.pt`

   (f) **Text:** Target network analysis

   (g) **Text:** ϵ-greedy decay analysis

3. **Task 3: Reward modeling on real human preferences (50 points)**

   (a) **Code:** `train_reward_model.py`

   (b) **Text:** Three example (post, chosen, rejected) triples

   (c) **Text:** Two or three sentences on what you observe in the examples

   (d) **Text:** Number of embedded triples and post-embedding shape

   (e) **Figure:** Training loss curve with moving average

   (f) **Text:** Initial and final loss values

   (g) **Text:** Validation accuracy on held-out 20%

   (h) **Text:** Short reflection on the training behavior and a proposed modification that would expand the RewardNetwork's output range

4. **Bonus Task: PPO on LunarLander (20 bonus points, optional)**

   (a) **Code:** `ppo_lander.py`

   (b) **Figure:** DQN vs. PPO comparison plot, same axes

   (c) **Text:** PPO final 100-episode average

   (d) **Text:** Comparison paragraph

5.2 Code files to submit

1. `tabular_dq.py`
2. `dqn_lander.py`
3. `train_reward_model.py`
4. `ppo_lander.py` **(only if attempting the Bonus Task)**
5. `dqn_lander.pt`: saved Q-network weights from Task 2
6. `requirements.txt`: dependencies (must include `gymnasium`, `torch`, `numpy`, `matplotlib`, `transformers`, `datasets`)
7. `README.md`: instructions to run your code, including Colab setup steps and any seeds you used

Note: You do *not* submit `lm_utils.py`. It is part of the provided starter code, not your work.

5.3 Autograder interface requirements

The autograder will import your `dqn_lander.py` and run a short verification. If the file name or function signature does not match, the autograder will fail and you will lose points.

5.3.1 Required file name

Submit exactly this file name (case-sensitive):

- `dqn_lander.py`

5.3.2 `dqn_lander.py`

Must contain:

- `QNet`: your Q-network class. Must be constructible as `QNet()` with no required arguments (assume 8-dim state, 4-dim action).
- `select_action(state, model)`: takes a NumPy array of shape `(8,)` and a `QNet` instance. Returns an integer in {`0, 1, 2, 3`} corresponding to the greedy action under the model (no exploration noise at test time).

The autograder will run:

```
import torch
import numpy as np
from dqn_lander import QNet, select_action

model = QNet()
model.load_state_dict(torch.load("dqn_lander.pt",
                                 map_location="cpu"))
model.eval()

state = np.random.randn(8).astype(np.float32)
action = select_action(state, model)

assert isinstance(action, (int, np.integer))
assert 0 <= action <= 3
```

The autograder does *not* check whether your agent actually solves LunarLander. Task performance is graded from your training curve and report. The autograder only checks that the interface works.

5.3.3 Autograder summary

File	Required names	Autograder tests
`dqn_lander.py` `dqn_lander.pt`	`QNet`, `select_action`	Loads checkpoint, calls `select_action`, checks type and range of return value

Important notes:

- Do **not** rename the file, class, or function.
- Do **not** put executable code at the module level. Guard scripts with `if __name__ == '__main__':` blocks.
- **Cite any code** you borrow or adapt from CleanRL, Gymnasium examples, or the lecture.

5.4 Code quality requirements

- Well-commented code explaining what each section does
- Meaningful variable and function names
- Docstrings for all classes and functions

- Code should run without modification (given the Colab setup from Section 2.2)
- Follow PEP 8 style guidelines

5.5 Report formatting

- **Include code snippets in the report for every task.** For each task, show the code you wrote alongside the outputs it produced. If the code snippet is not present, you will lose points on that task.
- Place output directly next to the corresponding code block.
- Use figures and tables with captions.
- Number equations, figures, and tables.
- Use proper citations when referencing papers, code repositories, or course materials.

5.6 Additional guidelines

- Convert Jupyter notebooks to .py files. **Do NOT submit .ipynb**
- If submitting late, submit only once
- **Do NOT submit video files larger than 5MB**. If your trained-agent GIF is large, reduce its frame rate or duration.
- Your code and report must be your own work
- Document your compute setup (CPU/GPU, estimated training times) in your README

6 Frequently asked questions (FAQ)

Q: The Box2D install fails on Colab with a swig error.

A: Run `pip install swig` first, then `pip install "gymnasium[box2d]"` again. On rare occasions a Colab runtime restart is needed after the swig install.

Q: My DQN reward goes up to about 100, then crashes back to -200. What went wrong?

A: Almost certainly a missing or buggy target network. Without one, the target moves every gradient step, and a small over-estimate compounds into a large one over a few updates. Add a target network if you haven't, and copy weights from the online network every 500 or 1000 steps.

Q: My DQN is training but the episode returns are extremely noisy.

A: This is normal. LunarLander rewards are noisy by design. Always plot a moving average of 100 episodes on top of the raw episode returns. Judge "solved" from the moving average, not from any single episode.

Q: My reward model (Task 3) training loss does not decrease much.

A: This is a behavior worth examining rather than a sign that your code is broken. Work through what values the `RewardNetwork` can output given its architecture, and look carefully at what the sigmoid does to your ability to separate the chosen response from the rejected one. Your observations feed into part 3.4.

Q: The Hugging Face download for `openai/summarize_from_feedback` is slow.

A: The full dataset is large; the starter code downloads the `comparisons` split and then keeps only the first 500 examples. On Colab the first run takes a few minutes; subsequent runs use the cached copy. If you hit a network error, wait a minute and retry.

Q: Can I use a larger pretrained model than `distilgpt2` for the embedder?

A: You can, but the observations you report in part 3.4 depend on the behavior of the reward network itself, not on the quality of the embeddings. Stick with `distilgpt2` so your results are comparable to other students'.

Q: How do I record a video of my trained agent on Colab?

A: Something like the following works:

```
import gymnasium as gym
import imageio
env = gym.make("LunarLander-v3", render_mode="rgb_array")
frames = []
obs, info = env.reset(seed=42)
done = False
while not done:
    action = select_action(obs, model)
    obs, reward, terminated, truncated, info = env.step(action)
    frames.append(env.render())
    done = terminated or truncated
imageio.mimsave("lander.gif", frames, fps=30)
```

Q: Can I use `stable-baselines3` or another high-level RL library?

A: No. The point of Task 2 is to implement DQN yourself so you understand what is happening. For the *Bonus* PPO task you may use CleanRL as a reference (with citation); CleanRL is a single-file implementation that does not abstract anything away. `stable-baselines3` hides too much and is not permitted.

Q: Do I need a GPU?

A: CartPole (Task 1) runs fine on CPU. DQN on LunarLander (Task 2) runs on CPU in about 30–60 minutes and on a Colab T4 GPU in about 15–30 minutes. Task 3 (reward modeling) takes about 10 minutes on CPU, 3 minutes on a T4.

Q: What if I get different results on different random seeds?

A: RL is notoriously sensitive to random seeds. Report the seed you used. You are not required to average across seeds for this homework, but if your submitted run looks like an outlier compared to a couple of other seeds you tried, mention that in your report.

Q: Can I work in a group?

A: No. This is individual work.

Q: Can I reuse code from previous homeworks?

A: Yes. Plotting utilities, training-loop boilerplate, and so on are fine to carry over. Training pipelines for supervised learning will not directly apply, but any utility code will.

Q: The lecture code on Slides 78–81 uses `gym`, not `gymnasium`. Do I need to port the whole thing?

A: Only the two lines shown in Task 1. The `gym.make`, `action_space.sample`, and `action_space.n` calls all work unchanged in Gymnasium. You only need to adjust the two places where the API changed: `reset()` and `step()`.

References

[1] Avinash Kak, *Reinforcement Learning: Incorporating Human Preferences in the Fine-Tuning of Large Language Models*. Available at: https://engineering.purdue.edu/kak/pdf-kak/Reinforcement.pdf

[2] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. Available at: http://incompleteideas.net/book/the-book-2nd.html

[3] Volodymyr Mnih et al., *Human-level control through deep reinforcement learning*, Nature, 2015. Available at: https://www.nature.com/articles/nature14236

[4] Hado van Hasselt, Arthur Guez, and David Silver, *Deep Reinforcement Learning with Double Q-learning*, AAAI, 2016. Available at: https://arxiv.org/abs/1509.06461

[5] Ronald J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, Machine Learning, 1992.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, *Proximal Policy Optimization Algorithms*, 2017. Available at: https://arxiv.org/abs/1707.06347

[7] Daniel M. Ziegler et al., *Fine-Tuning Language Models from Human Preferences*, 2019. Available at: https://arxiv.org/abs/1909.08593

[8] Shengyi Huang et al., *CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms*, Journal of Machine Learning Research, 2022. Available at: https://github.com/vwxyzjn/cleanrl

[9] Mark Towers et al., *Gymnasium: A Standard Interface for Reinforcement Learning Environments*, 2023. Available at: https://gymnasium.farama.org/