Learning Workbook: Deep Q-Learning for Pong

Dr. Yves J. Hilpisch[1]

December 13, 2025

[1]Get in touch: `https://linktr.ee/dyjh`. Web page `https://hilpisch.com`. Research, structuring, drafting, and visualizations were assisted by GPT 5.1 as a co-writing tool under human direction.

Contents

1 From Hardwired Pong to Learning Agents

On the C64 we specified every rule: move the paddle when the ball is above or below; bounce off a wall; reset lives manually. In this workbook we let an agent discover these behaviours on its own. The only interface is the Gymnasium API: observations in, actions out, rewards for feedback. The narrative remains playful, but every step is grounded in a runnable Deep Q-Learning (DQL) pipeline.

What You Will Build

- A compact DQL agent that trains on `gymnasium.make("ALE/Pong-v5")` (or a lightweight Pong variant).
- Instrumentation to visualise episodic returns, Q-values, and loss curves while training.
- A quick viewer to watch the learned policy play visually in a notebook or exported video.

Three contrasts frame the time travel from 1982 to 2025:

- **State**: instead of a few sprite coordinates, the agent ingests stacked frames that encode motion.
- **Control**: no hand-written if/else; actions come from a neural network approximating $Q(s, a)$.
- **Learning**: behaviour improves via stochastic gradient descent on Bellman targets, not by typing new BASIC lines.

For readers who prefer to work in a notebook, the folder that contains this workbook also includes a Colab-ready script `colab_learning.ipynb` and a rendered HTML version `colab_learning.html`. Online copies are available at `https://hilpisch.com/videos/colab_learning.ipynb` and `https://hilpisch.com/videos/colab_learning.html`. The notebook holds a complete, runnable implementation of the agent and training loop discussed here; the HTML file serves as a static, browsable reference if you simply want to inspect the code and comments without executing anything.
The rest of the workbook keeps the pacing hands-on: concise theory followed by annotated code you can run immediately.

2 Reinforcement Learning Primer (Fast)

Before building the full agent, this section establishes a compact reinforcement-learning vocabulary and the specific variant we will use. If you have seen RL before, you can skim for notation; if not, treat it as a translation layer between everyday language ("state", "score", "strategy") and the symbols that appear in the code and in the original Atari DQN paper.

2.1 Key Concepts

We keep the vocabulary lean so you can map each term to a line of code.

- **State** s_t: what the agent observes at time t; for Pong this is a stack of preprocessed frames.
- **Action** a_t: discrete moves such as `UP` or `DOWN`; the network outputs a value per action.
- **Reward** r_t: scalar feedback; Pong gives $+1$ for scoring, -1 for losing a point, 0 otherwise.

- **Return** G_t: discounted sum $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$ with discount $0 < \gamma < 1$.
- **Policy**: how actions are chosen. During training we use ϵ-greedy to balance exploration and exploitation.
- **Replay buffer**: a rolling memory of transitions (s, a, r, s', d) sampled uniformly to stabilise learning.

2.2 DQL in One Page

Deep Q-Learning approximates $Q_\theta(s, a)$, the value of taking action a in state s under parameters θ.

1. Initialise two networks with identical weights: `q_online` and `q_target`.
2. For each step: pick a_t with ϵ-greedy from `q_online`, observe (r_t, s_{t+1}, d_t), store the transition.
3. When the buffer is warm: sample a mini-batch, compute Bellman targets $y = r + \gamma(1 - d) \max_{a'} Q_{\text{target}}(s', a')$.
4. Minimise the loss $\frac{1}{2}(Q_{\text{online}}(s, a) - y)^2$ via backpropagation.
5. Periodically copy weights from `q_online` to `q_target` to stabilise the targets.

The following code shows the numeric core; inline comments emphasise intent.

TD target computation (ready to adapt)

```
with torch.no_grad():
    next_q = q_target(next_states)  # Q(s', a') from the slowly moving target net
    max_next_q, _ = next_q.max(dim=1)  # greedy value over next actions
    targets = rewards + gamma * (1 - dones) * max_next_q  # Bellman target y

q_values = q_online(states)  # Q(s, a) from the online net
action_q = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)  # select Q(s,a)
loss = torch.nn.functional.mse_loss(action_q, targets)  # squared TD error
loss.backward()  # propagate gradients
```

2.3 Atari DQN in Context

The recipe above closely mirrors the landmark Atari work by Mnih et al. at DeepMind.[2] In that paper, a single convolutional neural network learned to play many Atari 2600 games directly from pixels and score signals, using:

- stacked, preprocessed 84×84 grayscale frames as state,
- a CNN backbone almost identical to **??**,
- experience replay and a separate target network, as in the pseudocode above,
- an ϵ-greedy exploration schedule with a small residual exploration rate.

Our Pong agent is a scaled-down, single-game instance of the same idea: it uses the Atari Gymnasium environment instead of the raw Arcade Learning Environment, trains only on Pong rather than dozens of titles, and runs for hundreds of thousands rather than tens of millions of environment steps. The point of this workbook is not to reproduce headline benchmarks, but to make the moving parts of deep Q-Learning concrete enough that the Atari result feels like a natural extension rather than magic.

[2] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv:1312.5602, 2013.

3 Environment Setup with Gymnasium

Before we can talk about agents and networks, we need a place for them to act. In this workbook that place is the Gymnasium Atari interface: a thin wrapper around the classic Arcade Learning Environment that presents Pong as a standard reinforcement-learning environment. The goal of this section is to show how a few composable wrappers—for statistics, preprocessing, and frame stacking—turn the raw game into a clean stream of observations and rewards that a DQN can work with.

We assume a Python environment with PyTorch (or JAX) plus Gymnasium Atari extras:

- Install: `pip install gymnasium[atari] autorom[accept-rom-license] torch torchvision` then run `python -m AutoROM --accept-license` to download Atari ROMs.
- Rendering: set `render_mode="rgb_array"` for headless capture; use `"human"` for live windows.
- Hardware: a CPU technically suffices for Pong, but training will be much slower; a GPU dramatically shortens wall-clock time even though the algorithm and results are the same.

The code below instantiates Pong, applies minimal preprocessing (resizing, grayscaling, frame skipping), and then wraps observations into stacks of frames so that the agent can infer motion and velocity. Each wrapper does one job, and the combined effect is to turn noisy pixels into a compact, learning-ready state.

Environment creation and preprocessing

```
import collections
import gymnasium as gym  # gymnasium API
import numpy as np
from gymnasium.wrappers import RecordEpisodeStatistics, AtariPreprocessing

class FrameStackWrapper(gym.Wrapper):
    def __init__(self, env, num_stack: int = 4):
        super().__init__(env)
        self.num_stack = num_stack
        self.frames = collections.deque(maxlen=num_stack)
        low = np.repeat(env.observation_space.low, num_stack, axis=0)
        high = np.repeat(env.observation_space.high, num_stack, axis=0)
        self.observation_space = gym.spaces.Box(low=low, high=high, dtype=env.
            observation_space.dtype)

    def reset(self, **kwargs):
        obs, info = self.env.reset(**kwargs)
        self.frames.clear()
        for _ in range(self.num_stack):
            self.frames.append(obs)
        return self._get_obs(), info

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(action)
        self.frames.append(obs)
        return self._get_obs(), reward, terminated, truncated, info

    def _get_obs(self):
        return np.concatenate(list(self.frames), axis=0)

def make_env(seed: int = 0, render_mode: str = "rgb_array"):
```

```
    env = gym.make("ALE/Pong-v5", render_mode=render_mode)  # raw env (Gymnasium Atari)
    env = RecordEpisodeStatistics(env)  # track returns/lengths automatically
    env = AtariPreprocessing(
        env,
        frame_skip=4,
        screen_size=84,
        grayscale_obs=True,
        scale_obs=False,
    )  # standard DQN preprocessing
    env = FrameStackWrapper(env, num_stack=4)  # history for velocity perception
    env.action_space.seed(seed)  # reproducible action sampling
    env.observation_space.seed(seed)  # reproducible start states
    return env

env = make_env(seed=42)  # create a training env instance
obs, info = env.reset(seed=42)  # first observation, wrapped and stacked
```

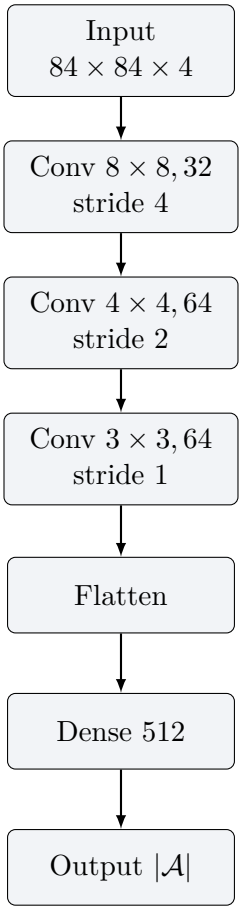

Figure 1: Planned DQL network sketch.

4 Network Architecture and Hyperparameters

With the environment prepared, we now need a function that maps each stacked frame to a value for every possible action. For Atari-scale Pong, a lightweight convolutional neural network (CNN) is enough to extract useful spatial and motion features from the $84 \times 84 \times 4$ input (see Figure 1). You can think of this network as a learned feature extractor plus a small decision head: the early layers detect edges and blobs moving on the screen, and the final layers combine

those signals into a single number per action representing "how good" that move seems in the current state.

To stay close to the original Atari DQN recipe, we follow the classic convolutional stem:

- $\text{Conv}_{8\times8,\,32}$ with stride 4 $\rightarrow$ ReLU
- $\text{Conv}_{4\times4,\,64}$ with stride 2 $\rightarrow$ ReLU
- $\text{Conv}_{3\times3,\,64}$ with stride 1 $\rightarrow$ ReLU
- Flatten $\rightarrow$ Dense (512) $\rightarrow$ ReLU $\rightarrow$ Output $|\mathcal{A}|$

On top of the architecture itself, a few training hyperparameters have outsized influence on stability and speed. It is worth making them explicit and thinking about them as a small configuration surface rather than magic numbers:

- Learning rate (e.g. 1e$-$4 with Adam), discount $\gamma = 0.99$, batch size 32 or 64.
- Replay buffer size (e.g. 100 000), warmup steps before learning (e.g. 20 000).
- Target network update frequency (e.g. every 1 000 gradient steps).
- ϵ-greedy schedule: start at 1.0, decay to 0.05 over $\approx$ 1e6 frames.

The following code sketches the model in PyTorch with inline comments for each layer.

Pong DQN backbone (PyTorch)

```python
import torch
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, action_dim: int):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(4, 32, kernel_size=8, stride=4),  # 4 stacked frames -> 32 maps
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),  # downsample + deepen
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),  # final conv block
            nn.ReLU(),
            nn.Flatten(),  # flatten to vector
            nn.Linear(64 * 7 * 7, 512),  # 84x84 -> 7x7 after strides
            nn.ReLU(),
            nn.Linear(512, action_dim)  # one logit per discrete action
        )

    def forward(self, x):
        x = x / 255.0  # normalize pixel values to [0,1]
        return self.net(x)

# Example: create networks
# q_online = DQN(action_dim=env.action_space.n).to(device)
# q_target = DQN(action_dim=env.action_space.n).to(device)
# q_target.load_state_dict(q_online.state_dict())  # initial sync
```

5 Training Loop Walkthrough

With state representation and a Q-network in place, the final ingredient is the learning loop itself. At a high level, every iteration of DQL follows the same rhythm: act in the environment, store the resulting transition, learn from a mini-batch sampled from the replay buffer, and periodically synchronise the target network. This section walks through a minimal but complete implementation of that rhythm. The intention is not that you memorise every line, but that you can point to each step in the code and say "this is where acting happens", "this is where learning happens", and "this is where stability tricks such as the target network are applied".

The following code outlines a trainer that you can drop into a Colab or local script. It assumes the environment and model definitions from the previous sections and uses conservative defaults that mirror the configuration in `colab_learning.ipynb`.

DQL training loop scaffold

```
import collections
import random
import torch
import torch.optim as optim

Transition = collections.namedtuple(
    "Transition", ["state", "action", "reward", "next_state", "done"]
)

class ReplayBuffer:
    def __init__(self, capacity: int):
        self.buffer = collections.deque(maxlen=capacity)
    def add(self, *args):
        self.buffer.append(Transition(*args))  # store one transition
    def sample(self, batch_size: int):
        batch = random.sample(self.buffer, batch_size)  # uniform sampling
        return Transition(*zip(*batch))
    def __len__(self):
        return len(self.buffer)

buffer = ReplayBuffer(capacity=100_000)  # experience store
optimizer = optim.Adam(q_online.parameters(), lr=1e-4)  # optimizer for Q-network
epsilon = 1.0  # start fully exploring
steps_done = 0

def epsilon_greedy(state_tensor, epsilon: float):
    if random.random() < epsilon:
        return env.action_space.sample()  # random action for exploration
    with torch.no_grad():
        q_vals = q_online(state_tensor.unsqueeze(0).to(device))  # batch dim
        return int(q_vals.argmax(dim=1).item())  # greedy action

num_episodes = 10_000
target_sync_freq = 1000  # gradient steps between target updates

for episode in range(num_episodes):
    state, _ = env.reset()  # new episode, get initial stacked frames
    episode_return = 0.0
    done = False
    while not done:
        state_np = np.array(state, copy=False)  # avoid extra copies
```

```
        state_t = torch.tensor(state_np, dtype=torch.float32, device=device)
        action = epsilon_greedy(state_t, epsilon)  # pick action
        next_state, reward, terminated, truncated, _ = env.step(action)  # env step
        done = terminated or truncated  # episode end flag
        buffer.add(state, action, reward, next_state, done)  # store transition
        state = next_state  # move forward
        episode_return += reward  # track score
        steps_done += 1  # global step counter

        if len(buffer) > 20_000:  # warmup before learning
            batch = buffer.sample(batch_size=32)  # grab a mini-batch
            states_b = torch.tensor(
                np.array(batch.state), dtype=torch.float32, device=device
            )
            actions_b = torch.tensor(batch.action, dtype=torch.int64, device=device)
            rewards_b = torch.tensor(batch.reward, dtype=torch.float32, device=device)
            next_states_b = torch.tensor(
                np.array(batch.next_state), dtype=torch.float32, device=device
            )
            dones_b = torch.tensor(batch.done, dtype=torch.float32, device=device)

            with torch.no_grad():
                next_q = q_target(next_states_b)  # target net for stability
                max_next_q, _ = next_q.max(dim=1)
                targets = rewards_b + 0.99 * (1 - dones_b) * max_next_q  # TD target

            q_vals = q_online(states_b)
            action_q = q_vals.gather(1, actions_b.unsqueeze(1)).squeeze(1)  # Q(s,a)
            loss = torch.nn.functional.mse_loss(action_q, targets)  # TD error

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

            if steps_done % target_sync_freq == 0:
                q_target.load_state_dict(q_online.state_dict())  # sync target net

        decay = (1.0 - 0.05) / 1_000_000  # per-step epsilon decay
        epsilon = max(0.05, epsilon - decay)  # linear decay floor

    # Optional: log episode_return, epsilon, and steps_done here
```

What Makes This Agent "Deep Q-Learning"

Stepping back from the code, four ingredients distinguish this agent from a classic tabular Q-learner:

- **Deep function approximation:** a convolutional neural network maps high-dimensional image stacks directly to Q-values, rather than storing values in a table indexed by discrete states.
- **Experience replay:** the replay buffer breaks temporal correlations by training on shuffled mini-batches drawn from many episodes, which stabilises learning.
- **Target network:** a slowly updated copy of the Q-network provides targets; this reduces feedback loops where the network chases its own moving predictions.
- **ϵ-greedy exploration:** a decaying but non-zero exploration rate ensures the agent continues to sample new trajectories instead of getting stuck in a deterministic rut.

Once you recognise these four ideas, most DQN variants—Double DQN, dueling heads, prioritized replay—look like careful refinements rather than completely new algorithms.

6 Seeing the Agent Play

Once an agent has learned a policy, the most satisfying way to inspect it is simply to watch it play. In practice, this means running the environment in evaluation mode (no exploration noise, no gradient updates), stepping it until the episode ends, and recording frames to a video file.

Using the Colab notebook `colab_learning.ipynb`, a DQL agent was trained on `"ALE/Pong-v5"` for 800 000 environment steps (four blocks of 200 000), taking roughly 3–4 hours of wall-clock time on a Colab GPU runtime. During training, the mean return over the last ten episodes hovered around small positive and negative values and remained noisy, but a final evaluation run of the trained policy achieved a return of 15.0 in one episode. The evaluation rollout was captured to MP4:

- rollout video: `https://hilpisch.com/videos/pong_dqn.mp4`,
- final checkpoint: `https://hilpisch.com/videos/dqn_pong_checkpoint.pt`.

You can download the checkpoint, load it into the same network architecture shown in **??**, and then reproduce or extend the evaluation.

The snippet below records an evaluation episode to disk. The details (paths, codec) can be adjusted to your environment.

Recording a trained DQN policy to MP4

```
import imageio.v2 as imageio
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

env = make_env(render_mode="rgb_array")  # reuse the wrapper from before
q_eval = DQN(action_dim=env.action_space.n).to(device)
checkpoint = torch.load("dqn_pong_checkpoint.pt", map_location=device)
q_eval.load_state_dict(checkpoint["model_state"])  # adapt to your checkpoint dict
q_eval.eval()

frames = []
```

```
state, _ = env.reset(seed=0)
done = False
episode_return = 0.0

while not done:
    state_t = torch.tensor(state, dtype=torch.float32, device=device)
    with torch.no_grad():
        q_vals = q_eval(state_t.unsqueeze(0))
        action = int(q_vals.argmax(dim=1).item())  # greedy action, no exploration
    next_state, reward, terminated, truncated, _ = env.step(action)
    frames.append(env.render())  # RGB frame for the video
    episode_return += reward
    done = terminated or truncated
    state = next_state

print("Evaluation return:", episode_return)
imageio.mimwrite("pong_dqn.mp4", frames, fps=60)  # save the rollout
```

Interpreting the 800 000-Step Run

Several features of the training log are worth noting:

- The exploration rate ϵ plateaus at 0.05, so even late in training the agent makes random moves 5% of the time; this adds variability to returns.
- Mean return over the last ten training episodes fluctuates between modestly positive and negative values; learning is noisy, and some policies perform well only on a subset of opponents or opening conditions.
- The separate evaluation—with exploration switched off and the latest checkpoint loaded—achieves a clean win with return 15.0, showing that the underlying policy is much stronger than raw training averages might suggest.

This pattern is typical for Atari-style deep RL runs: training curves are informative but not definitive, and targeted evaluations with fixed seeds and saved checkpoints are essential for understanding actual capabilities.

7 Evaluation and Next Steps

Training logs tell only part of the story. To understand what your agent is really doing, it helps to track a handful of simple signals over time and to see how they move together. This closing section suggests a short list of such signals and sketches how you might extend or refine the basic DQN setup once you are comfortable with the core loop.

For a fuller picture of learning dynamics, you can log:

- episode returns and lengths over time,
- moving averages of the TD loss,
- histograms of Q-values to detect divergence or saturation.

Plotting these quantities against environment steps or episodes often reveals phases: an early period of pure exploration, a mid-phase where returns improve rapidly, and a late phase where learning plateaus or becomes unstable.

The 800 000-step run mentioned earlier showed exactly this pattern: early episodes near random play, a middle stretch with occasional streaks of high scores, and a later region where

the TD loss stabilised around $\approx 2 \times 10^{-3}$ while returns continued to fluctuate. Such behaviour is consistent with a DQN agent that has discovered a reasonable policy but still juggles exploration noise and function-approximation quirks.

From here, natural next experiments include:

- **Double DQN:** decouple action selection and evaluation when computing targets to reduce overestimation bias.
- **Dueling networks:** split the network head into value and advantage streams, which can speed up learning in some settings.
- **Prioritized replay:** sample more frequently from transitions with large TD errors, focusing updates where the network is most surprised.
- **Hyperparameter sweeps:** vary replay-buffer size, learning rate, and ϵ decay to see which combinations stabilise training curves.

Even small ablations—such as turning off the target network or training without a replay buffer—are instructive; they quickly demonstrate why the original Atari DQN recipe combined all of these elements. When you log these metrics in a notebook or dashboard, turning them into simple line plots (for example, reward versus environment steps) makes it much easier to spot phases of learning and diagnose instability, even if this workbook does not prescribe a specific visualisation.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing Atari with Deep Reinforcement Learning." *arXiv preprint* arXiv:1312.5602, 2013. Available at `https://arxiv.org/abs/1312.5602`.