COMPGI13: Assignment 3

Antonio Remiro Azócar, MSc Machine Learning

18 April 2017

Code/README

Note: One week extension granted due to medical issues.
Tools: Code was developed using Python 3.6.0 and TensorFlow version 0.12.1. The following packages were utilised: matplotlib 2.0.0 (pyplot), numpy 1.12.0, gym 0.8.1, scikit-image 0.12.0 (for image processing in Part B: reducing environment observations in size to images and converting these to grey scale). See the README file, which features detailed information on the programs/model restoring. **Code files:**

- Problem A: `exerciseA1.py`, `exerciseA2.py`, `exerciseA3i.py` (linear), `exerciseA3ii.py` (hidden), `exerciseA4.py`, `exerciseA5i.py` (30 hidden units), `exerciseA5ii.py` (1000 hidden units), `exerciseA6.py`, `exerciseA7.py`, `exerciseA8.py`.
- Problem B: `exerciseB.py`.

Parameter optimisation

Hyperparameters are tuned for each model individually. Learning curves have been utilised to monitor the training process and decide on suitable hyperparameters. The learning rate and optimiser used along with any other modified parameters are reported for each experiment.

Problem A: Cart-pole

The task is to keep a pole balanced indefinitely. This environment has a 4D observation and two discrete actions. The reward is modified to 0 on non-terminating steps and -1 on termination. The maximum episode length is set to 300 and the discount factor is set to 0.99.

A1

The random seeds for numpy and for the gym environment are set to 555. Table 1 reports the episode length and the return from the initial state for the trajectories generated under a uniform random policy.

Trajectory	Episode length	Return from the initial state (3 s.f.)
1	16	-0.851
2	12	-0.886
3	11	-0.895

Table 1: Episode length and return from the initial state for the trajectories generated in **A1**.

A2

The random seeds for numpy and for the gym environment are set to 555. 100 trajectories are generated under the random policy implemented in **A1**. Table 2 reports the mean and standard deviation of the episode lengths and the return from the initial state.

Mean episode length (± st.dev)	Mean return from the initial state (± st.dev)
20.77 ± 11.69	-0.817 ± 0.086

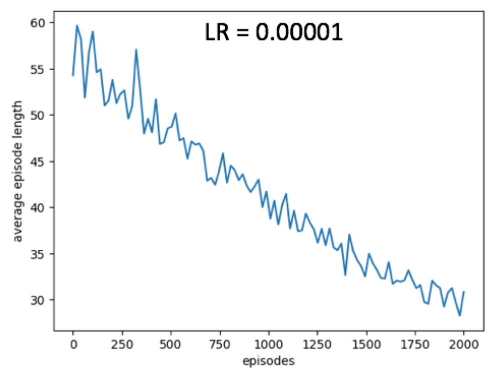
Table 2: Mean and standard deviation of episode lengths/return from the initial state for **A2**.

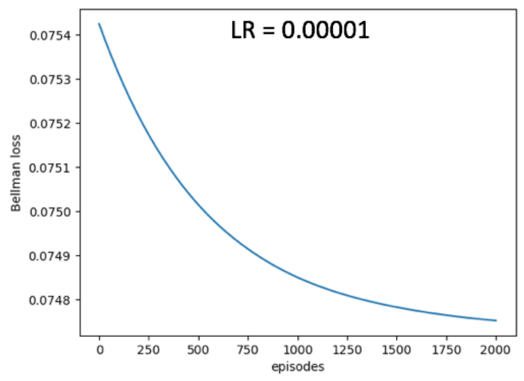
A3

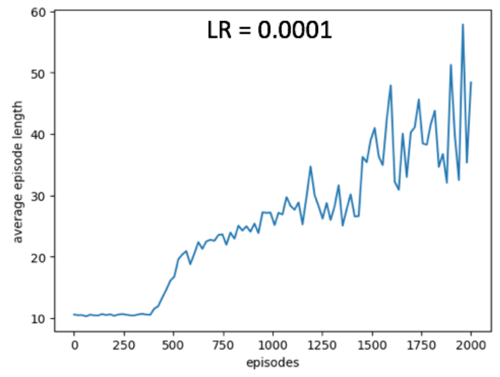
The random seeds for numpy, TensorFlow and for the gym environment is set to 17. The models are trained over 2000 episodes under a uniform random policy (generated trajectory data is loaded from exercise **A2**). A gradient descent optimiser is used with learning rates of 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1} and 0.5. The batch size is set to 256 and weights are initialised from a truncated normal distribution.

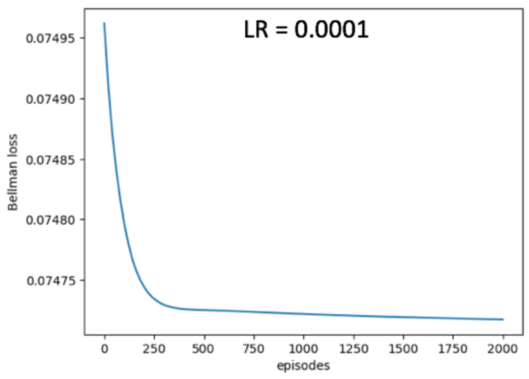
Linear layer with an output per action:

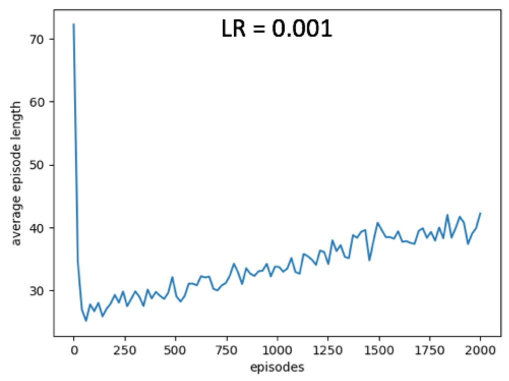
Figures 1-12 present plots of test performance (measured as average episode length) and Bellman loss ($|\delta|$ in the worksheet) during training using the aforementioned learning rates. Note that the average episode length is computed over 100 games/trajectories for most of the exercises (**A4** is an exception).

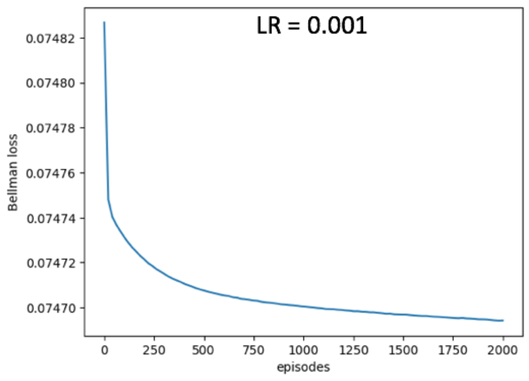


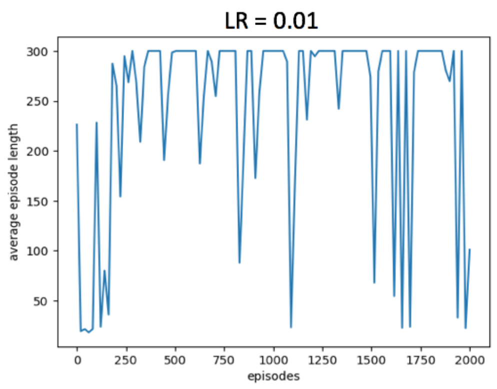


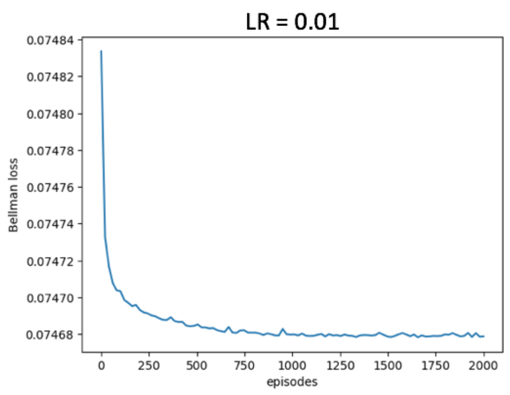


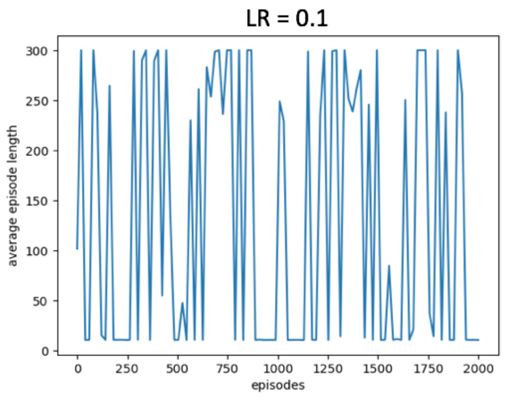


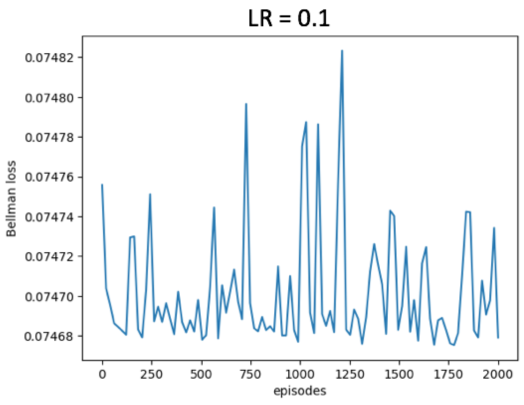


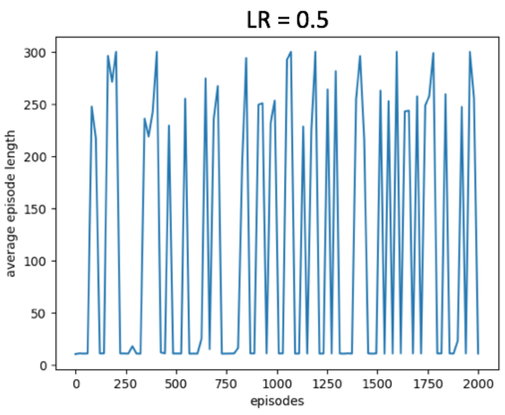


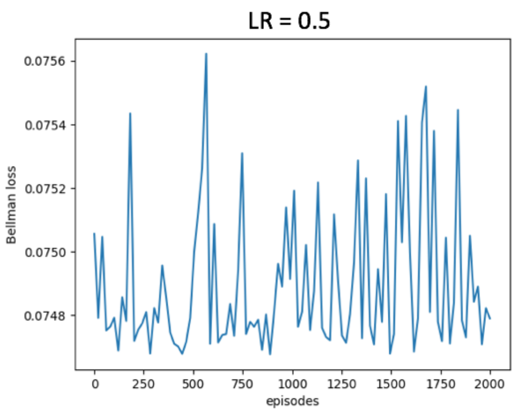








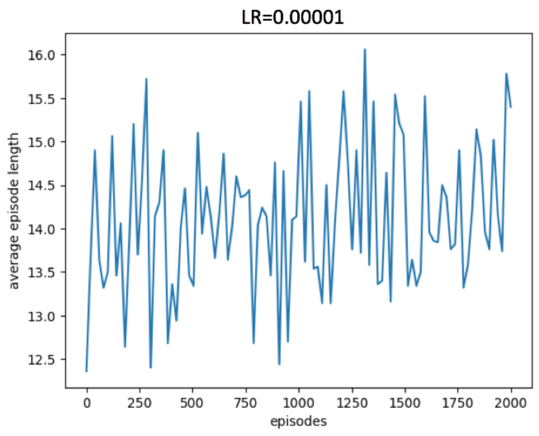


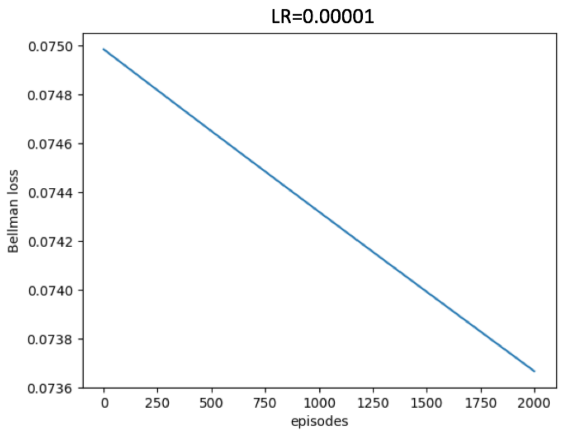


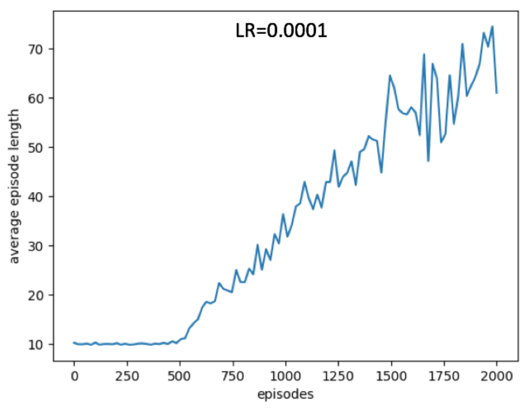
Figures 1-12 (top left to bottom right). Average episode length (left) and Bellman loss (right) plotted over 2000 episodes (every 20 steps) for different learning rates. A linear layer with an output per action is used.

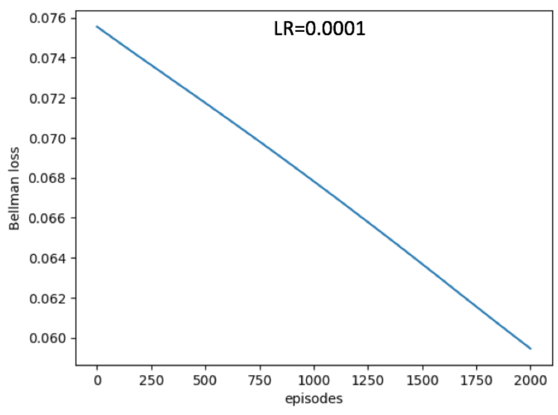
Hidden layer (100) – linear transformation + ReLU – followed by a linear layer:

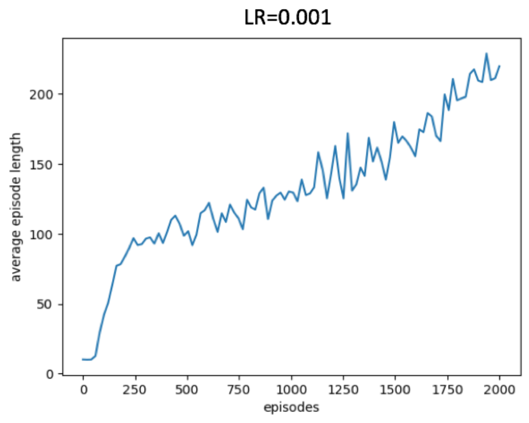
Figures 13-24 present plots of test performance (measured as average episode length) and Bellman loss during training using the aforementioned learning rates.

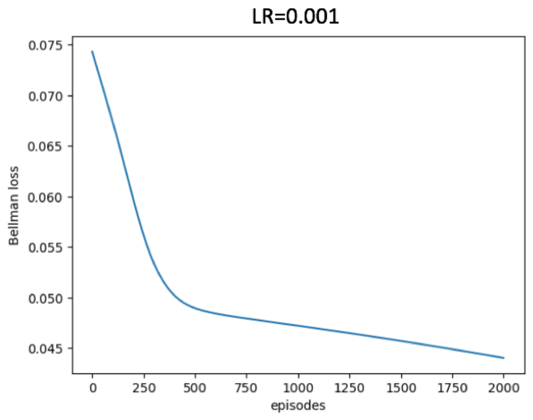


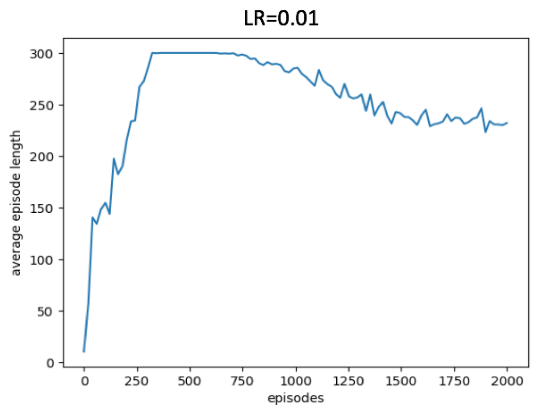


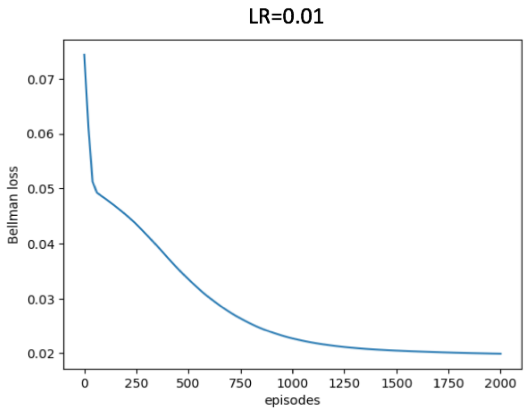


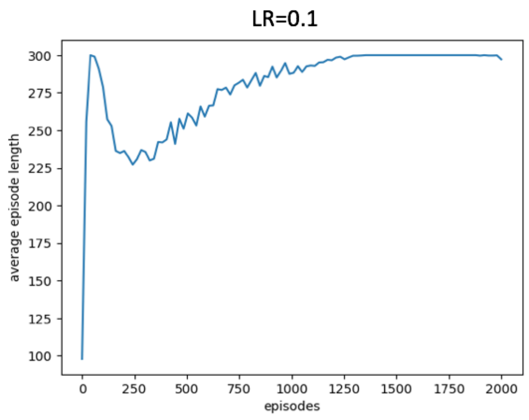


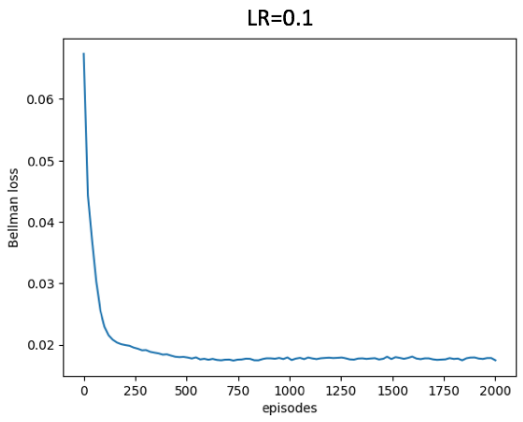


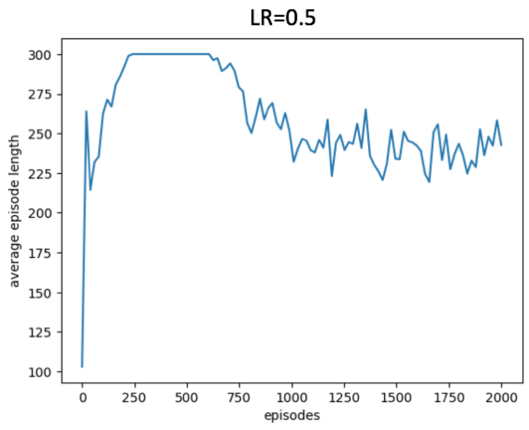


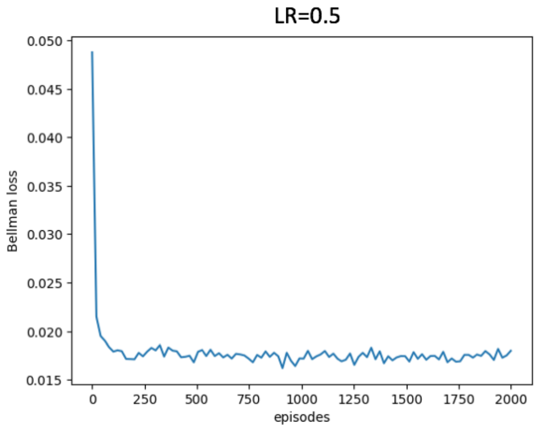








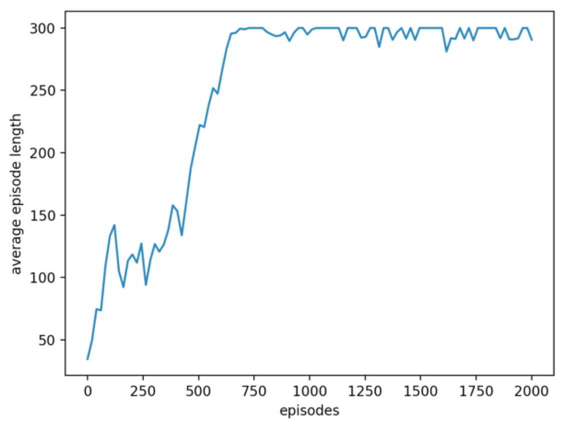


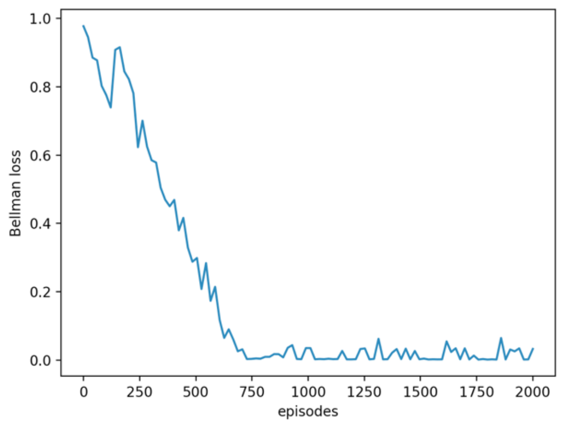


Figures 13-24 (top left to bottom right). Average episode length (left) and Bellman loss (right) plotted over 2000 episodes (every 20 steps) for different learning rates. A hidden layer (100 units) - linear transformation + ReLU - followed by a linear layer is used.

A4

The random seeds for numpy, Tensorflow and for the gym environment are set to 23. The epsilon-greedy Q-learning agent is trained over 2000 episodes with $\epsilon = 0.05$. A gradient descent optimiser is utilised. Its step size is adjusted to compare performances; the final learning rate selected is 0.02. The batch size is set to 256 and weights are initialised from a truncated normal distribution. Control performance (measured as the average episode length) and the Bellman loss are computed every 20 steps for the aforementioned learning rate. This question has two possible interpretations: **(i)** the average episode length/bellman loss is computed over 100 games/trajectories (as performed in **A3**, **A5-8**), **(ii)** the aforementioned averages are computed and averaged again over performance curves after repeating the experiment 100 times. I believe the correct interpretation is **(ii)**. The average measures have been computed over 20 games per run and we have averaged over performance curves after 35 experiment repeats. This was necessary due to computational and time limitations. Control performance and Bellman loss over the number of episodes are plotted in Figures 25 and 26 respectively. **Note:** Control performance and Bellman loss plots for interpretation **(i)** are provided in the Appendix.

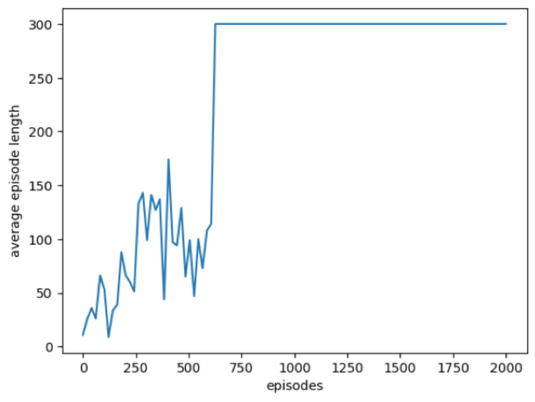


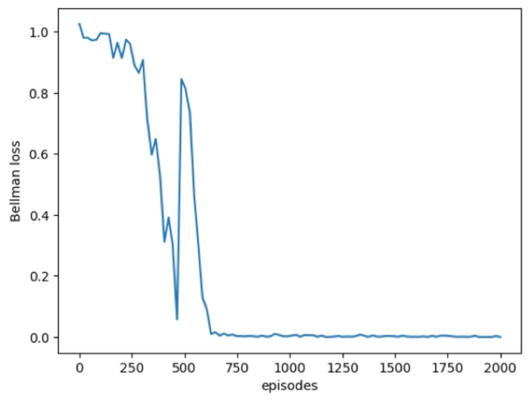


Figures 25 and 26 (left to right). Figure 25: Average episode length plotted over 2000 episodes (every 20 steps) for **A4**. Figure 26: Bellman loss ($|\delta|$ in the worksheet) plotted against the number of episodes for **A4**. We have averaged the result over 35 runs/performance curves.

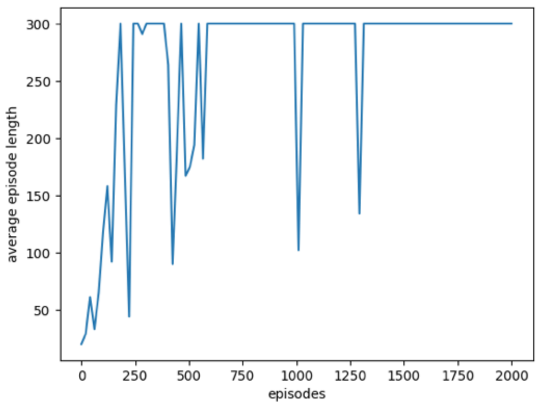
A5

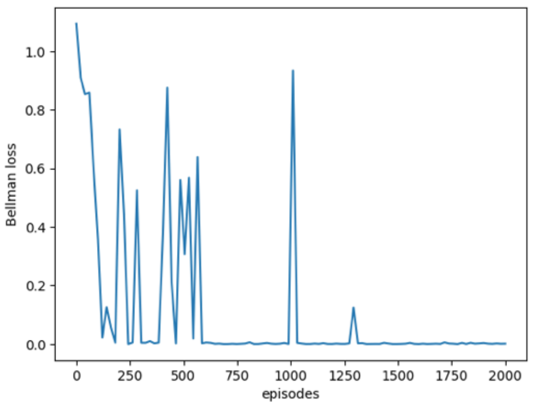
The random seed for numpy, Tensorflow and for the gym environment is set to 23. ϵ is set to 0.05 in the greedy policy. The optimiser used is gradient descent with a learning rate of 0.02. The batch size is set to 256. Weights are initialised from a truncated normal distribution. **Hidden layer of 30 units:** The test performance (measured as the average episode length) and Bellman loss every 20 steps during a total of 2000 episodes are plotted in Figures 27 and 28 respectively. **Hidden layer of 1000 units:** The test performance (measured as the average episode length) and Bellman loss every 20 steps during a total of 2000 episodes are plotted in Figures 29 and 30 respectively.





Figures 27 and 28 (left to right). Figure 27: Average episode length plotted over 2000 episodes (every 20 steps) for the hidden layer of 30 units. Figure 28: Bellman loss over 2000 episodes using hidden layer (30).



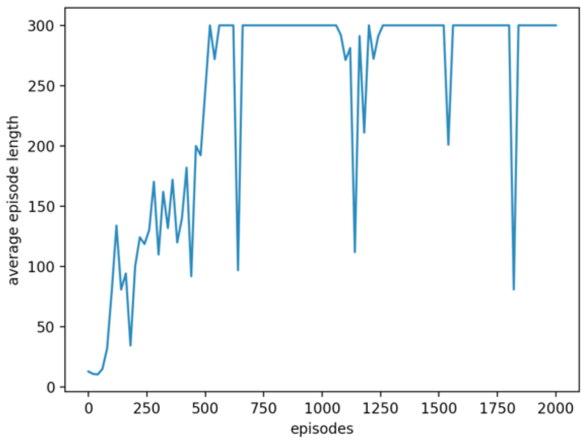


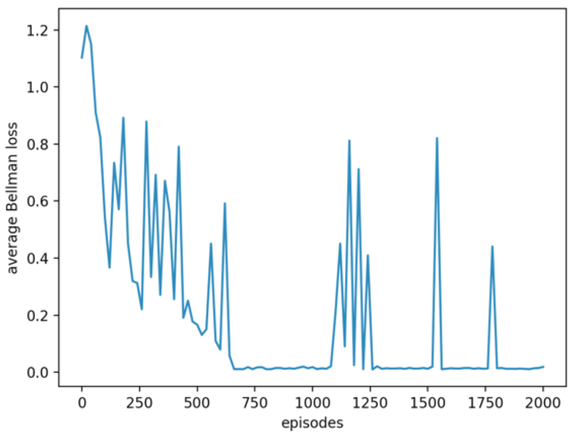
Figures 29 and 30 (left to right). Figure 29: Average episode length plotted over 2000 episodes (every 20 steps) for the hidden layer of 1000 units. Figure 28: Bellman loss over 2000 episodes using hidden layer (1000).

Setup comparison: In this particular example, faster convergence is reached using the hidden layer of 1000 units than that of 30. However, the hidden layer of 1000 units appears to be less stable once it has reached convergence; this is perhaps due to overfitting. In addition, we notice that in some of our runs, the Bellman loss undergoes a sudden increase, as clearly observed in Figure 30. I suspect this may be attributed the exploding gradient problem. The model does not continue learning once this occurs.

A6

Note: Whereas in **A1-A5** the Bellman loss is interpreted as $|\delta|$, from now onwards it denotes $\frac{1}{2}\delta^2$.
A hidden layer of 100 units - linear transformation + ReLU - followed by a linear layer is utilised. The batch size is set to 256 and the buffer size is set to 65,536. The optimiser used is RMSprop with a step size of 0.0002, a decay of 0.935 and a momentum of 0.8. The random seeds for numpy, TensorFlow and the gym enviornment are set to 23. In the greedy policy, ϵ is set to 0.05. Weights are initialised from a truncated normal distribution. The average test performance (episode length) and Bellman loss every 20 steps during a total of 2000 episodes are plotted in Figures 31 and 32.



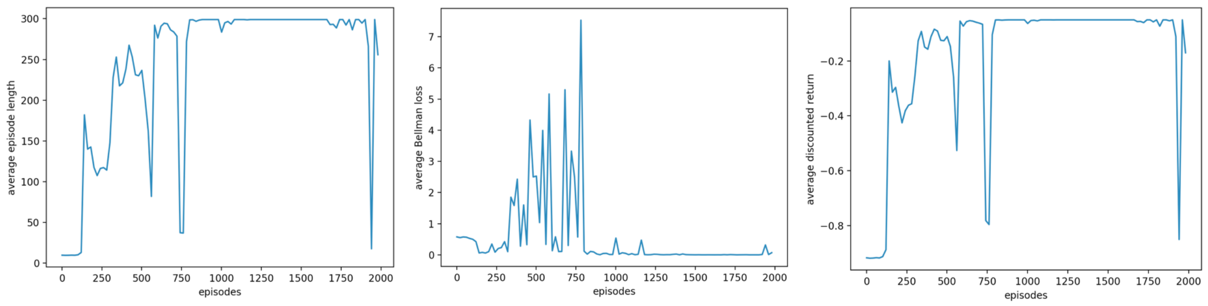


Figures 31 and 32 (left to right). Figure 31: Average episode length plotted over 2000 episodes (every 20 steps) for the hidden layer of 1000 units. Figure 32: Bellman loss over 2000 episodes using hidden layer (1000).

Compare performance with your original A4 agent: For this particular run (Figures 31 and 32), we see no significant differences with most runs performed with the original agent. In fact, **A6** seems to have less stable performance than the individual run performed for **A4** (in the appendix). This is surprising. One expects the agent trained on the 'replayed experience' prior to deployment to converge more stably. Perhaps using a larger buffer size would give a more robust policy. Alternatively, this instability could well be a an attribute of the individual run and not generalise.

A7

A target network is added to the network from part **A6**. The same parametric model is used as in **A6**. Current network parameters are copied to the target network every 5 episodes. The test performance (average episode length), average Bellman loss and average discounted empirical return every 20 steps during a total of 2000 episodes are plotted in Figures 33, 34 and 35 respectively.

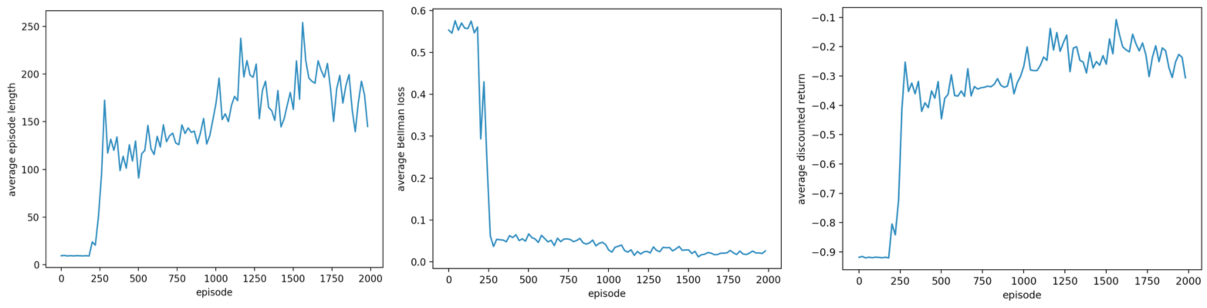


Figures 33-35 (left to right): Figure 34: Average episode length plotted over 2000 episodes (every 20 steps) for **A7**. Figure 35: Average Bellman loss over 2000 episodes. Figure 36: Average discounted return from the initial state over 2000 episodes.

Setup comparison (between **A6** and **A7**): Despite the exploding Bellman loss values when commencing training, **A7** appears to give greater stability later (with less sudden value dips than **A6**). This is perhaps due to how the experience buffer performs, in a sense, minibatch online learning. The experience buffer reuses previous events to train the neural network (this perhaps also causes the many consecutive peaks in the first 800 episodes). The replay experience buffer also makes the code more efficient.

A8

SARSA is used instead of Q-learning. The same parameters are used as in **A6** and **A7**. The test performance (average episode length), average Bellman loss and average discounted empirical return every 20 steps during a total of 2000 episodes are plotted in Figures 36, 37 and 38 respectively. While Q-learning is off-policy, SARSA is on-policy i.e. it does not always assign maximum reward to the Q-value. SARSA should, in principle, guarantee more stable global convergence. However, this should come at the expense of not attaining maximum values for episode length/discounted return. These suspicions are confirmed by our results. Figures 36 and 38 arguably present more stability for average episode length and discounted return than Figures 34 and 36 i.e there are less sharp dips in value. On the other hand, average episode length and discounted return do not reach their 'maximum' values (attained in **A7**). They may have perhaps done so running the experiment for a greater number of episodes. Figure 37 confirms SARSA guaranteeing a more stable convergence of the Bellman loss (in contrast to Figure 34).



Figures 37-39 (left to right): Figure 37: Average episode length plotted over 2000 episodes (every 20 steps) for **A8**. Figure 38: Average Bellman loss over 2000 episodes. Figure 39: Average discounted return from the initial state over 2000 episodes.

Problem B: Atari Games

For all experiments, the discount factor is set to 0.99 and the environmental rewards are clipped to be -1,0 or 1. The minibatch size is set to 32. An RMSprop optimiser with a step size of 0.001 is utilised (except for Ms. Pacman in **B3/4**, in which case a step size of 0.0001 is used). The target network is updated every 5,000 steps and the agent is evaluated every 50,000 steps. The following adjustments are made to the parameters provided by the worksheet. Firstly, whereas a fixed epsilon of 0.1 is utilised in the greedy policy in **B1** and **B2**, for **B3/4** the value of epsilon is decreased throughout the training (it is initialised at 1 and gradually decays to 0.1). The machine is able to handle a minibatch size of 3,200 and an experience replay buffer storing 3 million transitions. Weights are initialised from a truncated normal distribution. The TensorFlow and numpy seeds are set to 55.

Note 1: For **B1**, the *mean score* has been interpreted to be the mean discounted reward. For **B2**, **B3**, **B4** it has been interpreted as the actual in-game score.

B1

Random policy: The average score (discounted reward) and frame counts for the three games under a random policy, evaluated on 100 episodes, are presented in Table 3.

Game	Mean Frame Counts (environment steps) per episode	Mean Score (Discounted Reward)
Ms. Pac-Man	634.1 ± 98.6	2.39 ± 0.64
Boxing	2394.6 ± 23.7	-0.46 ± 1.07
Pong	$1254.5 \pm \pm 109.2$	-0.95 ± 0.21

Table 3: Average (± st.dev) discounted reward and frame counts for three games evaluated at 100 episodes.

B2

Q-network: The average score (in-game score) and frame counts for the three games under an untrained Q-network, evaluated on 100 episodes, are presented in Table 4.

Game	Mean Frame Counts (environment steps) per episode	Mean Score
Ms. Pac-Man	435.3 ± 11.2	6.00 ± 0.00
Boxing	2381.5 ± 16.1	-37.67 ± 0.47
Pong	1078.8 ± 18.5	-21.00 ± 0.00

Table 4: Average (± st.dev) discounted reward and frame counts for three games evaluated at 100 episodes.

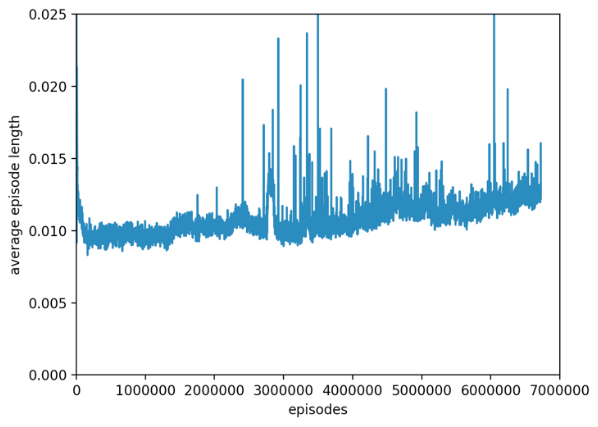
The performance of **B2** is worse than that of **B1** if we pay attention to the mean frame counts (environment steps) per episode in all three games. An inspection of in-game scores and discounted reward during training confirms this hypothesis. Under the random policy, actions are chosen from a normal distribution. This random choice process results in more profound exploration of the state space. An initialised trained network, on the other hand, may fail to explore the state space, getting stuck in 'dead zones' and losing games with low scores.

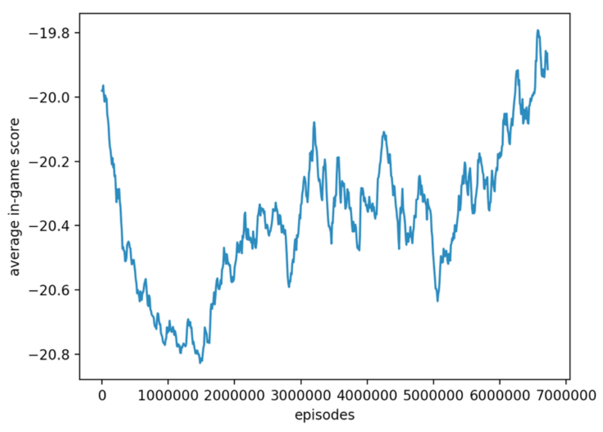
B3 and B4

Note 1: Ignore the incorrect y-axis label in the Figures corresponding to this subsection. 'Average episode length' should be replaced by 'batch loss'.
Note 2: Ignore the incorrect x-axis tick mark labels in the Figures corresponding to this subsection. These are 10^3 greater than they should i.e. 1,000,000 episodes should be 1,000.

Pong

This game is played for nearly 7,000 episodes. Pong has the least stable/robust training of the three games and is the model that takes longest to train. Figures 40 and 41 plot the agent batch loss (every 500 steps) and the average in-game score (every 50k steps) during the training process. The final in-game score is -19.93.

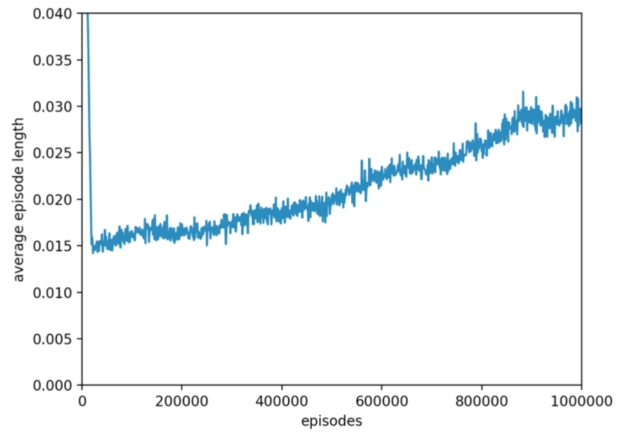


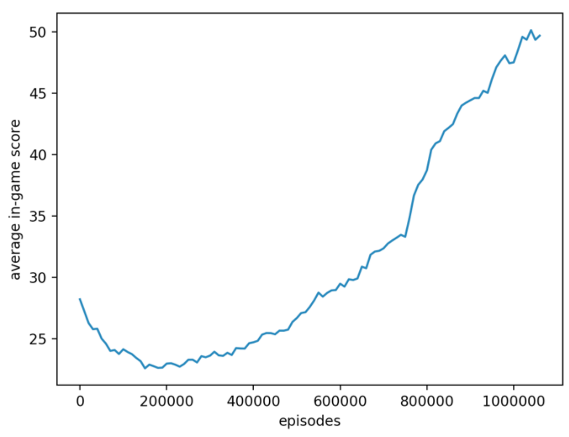


Figures 40,41: Agent loss and in-game scores tracked during training for Pong.

Boxing

This game is played for 1,000 episodes. It trains more quickly than Pong and has the most stable/robust training process of the three games. Figures 42 and 43 present the agent loss (every 500 steps) and average in-game score (recorded every 50k steps) during the training process. The final in-game score is 49.65.

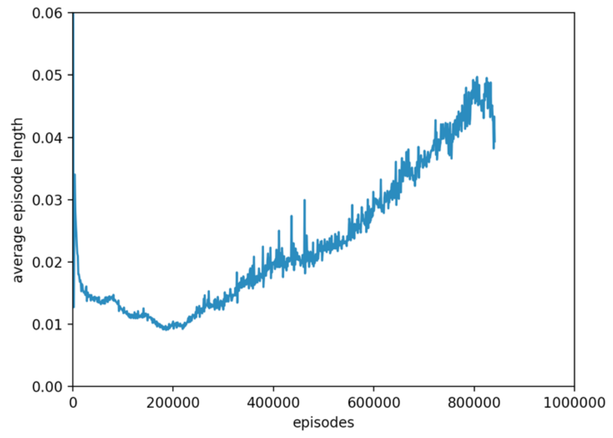


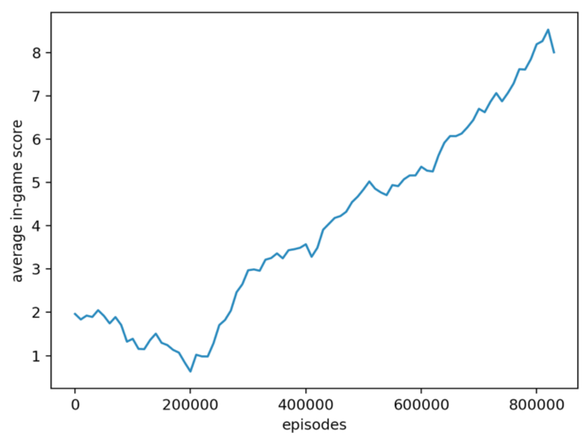


Figures 42,43: Agent loss and in-game scores tracked during training for Boxing.

Ms. Pacman

This game is played for just over 800 episodes. Ms.Pacman has the most stable training process of all games and has the fastest training rate (the curve which most promptly starts going up). Figures 44 and 45 present the loss (recorded every 500 steps) and average in-game score (recorded every 50k steps) during the training process. The final in-game score is 0.038.





Figures 44,45: Agent loss and in-game scores tracked during training for Ms. Pacman.

Discussion

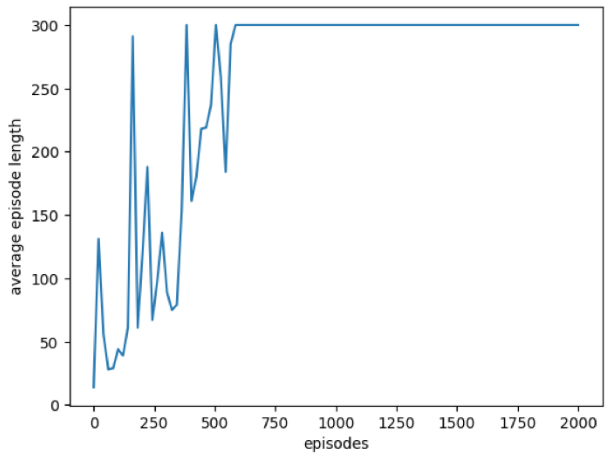
As expected, performance for this exercise is very game-dependent. It is a good sign to see the games with greater expected intrinsic difficulty to be slower to train e.g. Pong, the game less prone to "button-bashing" is the slowest and least stable model to train; Ms. Pacman has the curve which picks up fastest in training (at around 200k episodes). Performance/policy is far from optimal for all games even after 1 million steps; this is expected for comparatively short times. Most importantly, we see a sensible strategy/trend emerging for all games. It seems possible they eventually converge to an optimal policy if given sufficient training time. We have obtained a coherent policy for all games, something which is not as trivial as it seems. In Pacman, the state space must be navigated without bumping into walls or getting stuck under greedy policy avoiding ghosts. In Pong, one must 'attempt' to reach the ball, even while failing at it.

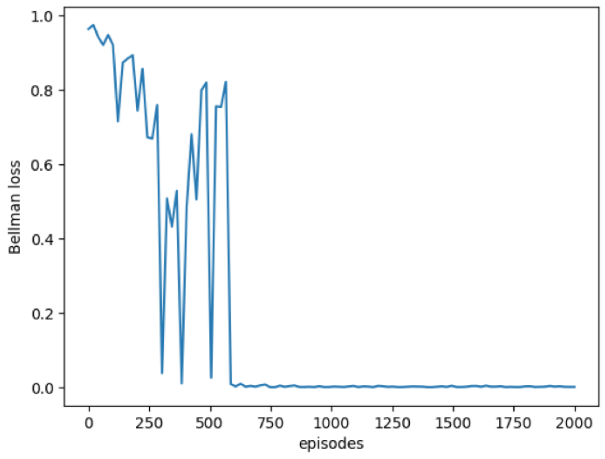
Modifications to training regime: All games were run for at least 1e6 steps. Modifications made included resizing the state space to an 84×84 image before greyscaling (instead of a 28×28 image). This was inspired by the DQN Atari Nature paper. Special care was taken for the ball in Pong; this game required an extra step to threshold the values and extra care in not cutting out the pixels behind the players. This scenario originally stopped my agent from learning rewards; not able to associate these with winning or losing a point. I find the games too 'dense' to be rescaled to 28×28. With sufficient time and computational resources, an 84×84 image can still attain a good compromise in terms of affording a decent number of samples in memory and not losing much on resolution/information. Additionally, the batch size was increased to 3,200 and a replay buffer of size 3,000,000 was fit into memory.

Difference with supervised learning: The curves indicate that the agents are learning, with increased in-game scores and increased rewards during training. In reinforcement learning , the environment is defined by the state space (all possible states) and actions in the state space. The reward depends on the state/action; the agent wants to find the optimal policy maximising the reward. The priority of the agent is exploration: taking an action to change state and receiving a reward/punishment for it. In RL, we give priority to controlling our environment. These characteristics make RL problems much slower and computationally intensive to train (as we have seen from our learning curves). On the other hand, supervised learning is stateless, with no temporal dependence. It focus less on exploration, trying to predict already specified classes. RL may end up exploiting better options than those specified for SL but at a much higher computational cost (it should guarantee more converging eventually to an optimal policy though).

Appendix

A4: We presume this interpretation of exercise to be incorrect. The average episode length/bellman loss is computed over 100 games/trajectories (as performed in **A3**, **A5-8** - no averaging between performance curves). Control performance and Bellman loss over the number of episodes are plotted in Figures 46 and 47 respectively.





Figures 46 and 47 (left to right). Figure 46: Average episode length plotted over 2000 episodes (every 20 steps) for **A4**. Figure 47: Bellman loss ($|\delta|$ in the worksheet) plotted against the number of episodes for **A4**. One individual run.