Report

In this assignment Reinforcement Learning (RL) agents were developed with the aim to learn to play different games. In section A the classic Cart-Pole problem was tackled using variety of learning approaches. In section B agents were trained to play three Atari console games. For results reproducibility, a time seed = 20 is used throughout.

Background:
In RL the agent's goal is to maximise the expected total reward by learning an optimal policy (mapping from states to actions). At time *t* the agent observes a state *s(t),* selects an action *a(t)*, and receives a reward *r(t),* after taking the action in time t+1 the agent then observes the next state *s(t+1).* The expected return after taking action *a(t)* and policy (pi) is defined by the action-value function *Q(s,a)*. The optimal value of the Q function obeys a fundamental recursion known as the Bellman equation.

Problem A: Cart-pole

In this problem the task is to keep a pole balanced indefinitely, but in practice this is limited to 300 episodes. This is done in the Gym environment where the extracted data has 4 attributes and 2 possible actions (left or right). In this implementation, the reward is set to 0 for non-terminating steps and to -1 on termination.

General comments:

- Maximum episode length is set to 300 and the discount factor is kept at 0.99 throughout. However, for parts 1-3 the environment "CartPole-v0" was not used because it was discovered that in the default settings this could only go to 200 episodes instead of the required 300. Therefore, version "CartPole-v1" was used instead, where the maximum number of episodes is 500. Later it was discovered that one could change the max. episodes number in "CartPole-v0", with an additional command. Therefore, for part 4 onwards version v0 was used as per assignment specification.
- Parts 1-3 use the RMSPropOptimizer, (default) which was originally recommended for the task. It was however discovered that gradient descent performs better, thus part 4 onwards use the GradientOptimiser (default), yielding more stable results.

Part 1. Generate three trajectories under a uniform random policy.

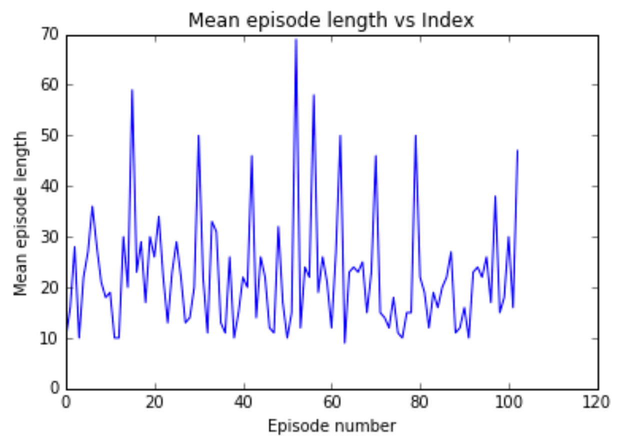
Results:
Episode lengths and the return from the initial state:
Run 1: Episode length was 10 and he corresponding return from initial state was: -0.91352
Run 2: Episode length was 16 and he corresponding return from initial state was: -0.86006
Run 3: Episode length was 28 and he corresponding return from initial state was: -0.76234
Commentary:
It can be seen that the episodes length increases with each run and consequently the reward becomes less negative. This is however a chance result due to the time seed. The results are generally bad, with rewards close to -1 and episode lengths far from the desired 300. This is to be expected from only using the crudest of policies (i.e. random) to balance the pole.

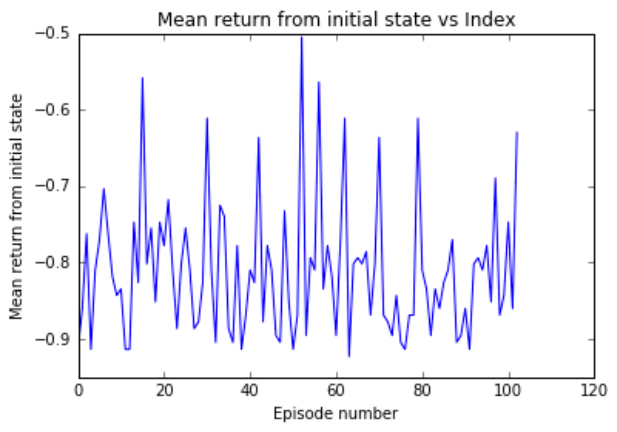
Part 2. Generate 100 episodes under the above random policy, and report the mean and standard deviation of the episode lengths and the return from the starting state.

Results over 100 runs:
Mean episode length was approx.: 22.5 and the corresponding standard deviation was: 11.7
Mean return from initial state was: -0.81 and the corresponding standard deviation was: 0.087
The results over 100 episodes are displayed don graphs below.

Commentary:

Results are still generally unsatisfactory; high variability of results and no convergence, indicating that even running for a 100 episodes is not enough to achieve learning. Again, this is unsurprising given the random policy employed.



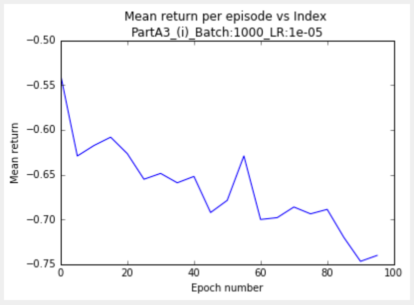
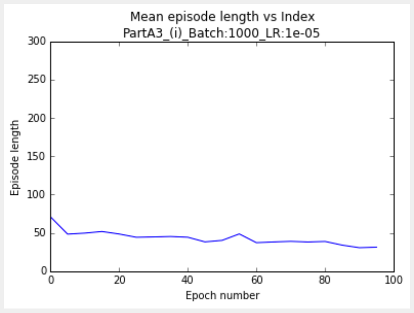
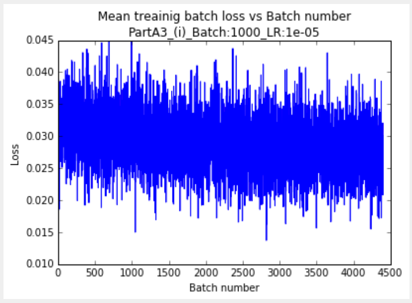
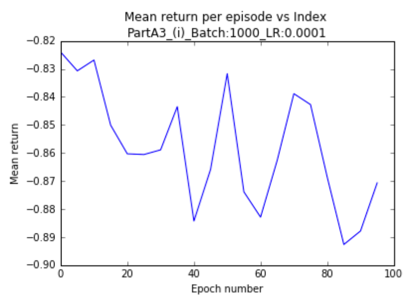
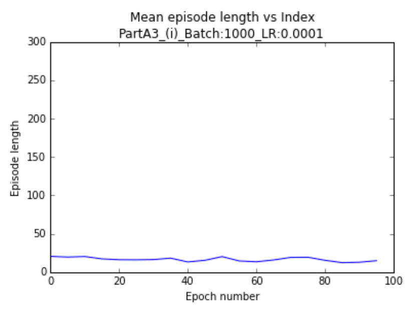
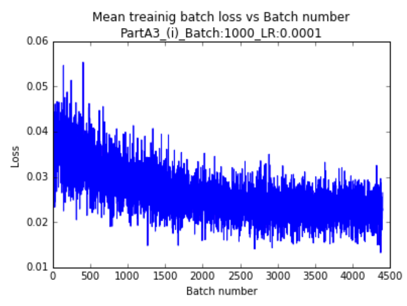


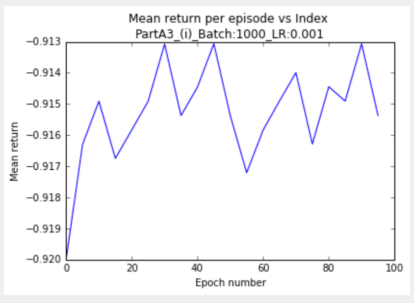
Part 3. Batch Q-learning

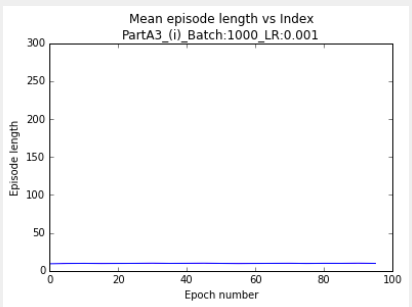
In this part 2000 episodes are generated under a uniformly random policy and batch Q-learning is implemented to advance learning. Batch size of 1000 is used and a range of learning rates is trailed. The value function is trained under two representations: i) a linear layer with one output per action and respectively ii) a hidden layer(100) – linear transformation + ReLU – followed by a linear layer with one output per action.

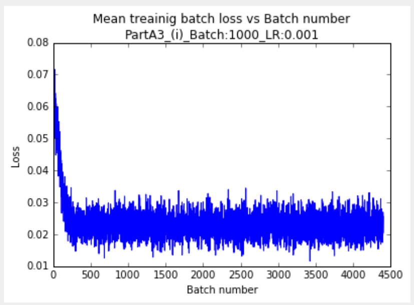
(i) Linear layer with one output per action

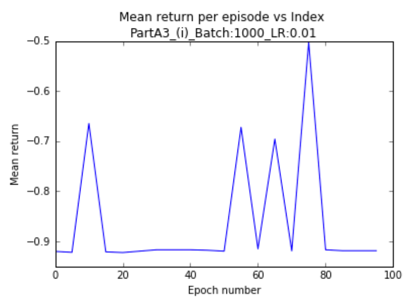
<u>Results are reported graphical below:</u>

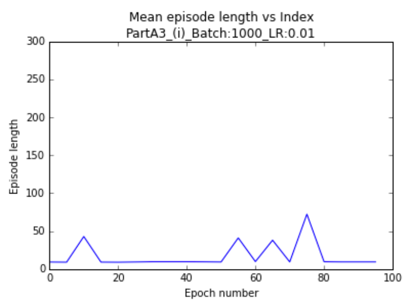
Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		
		

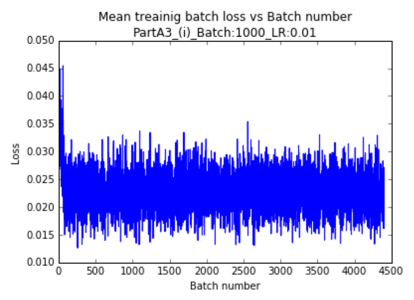


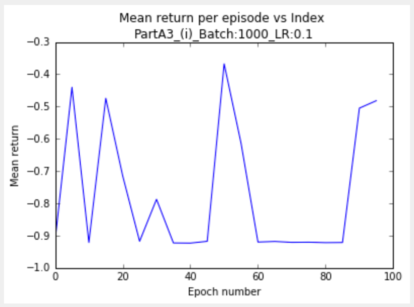


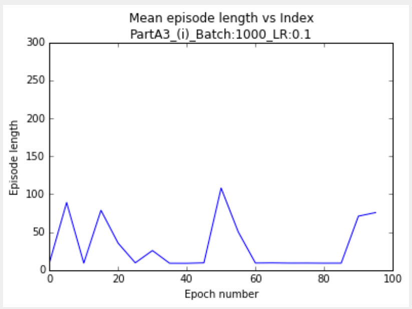


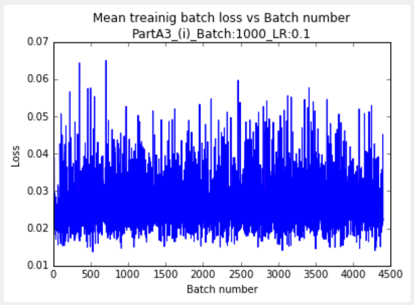


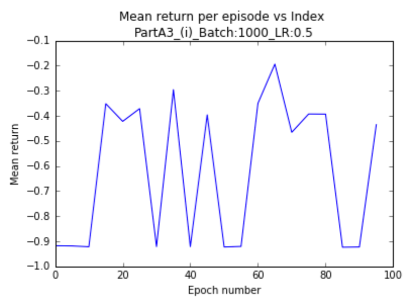


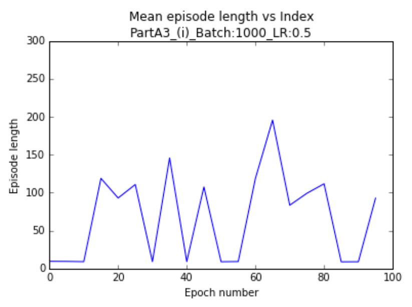


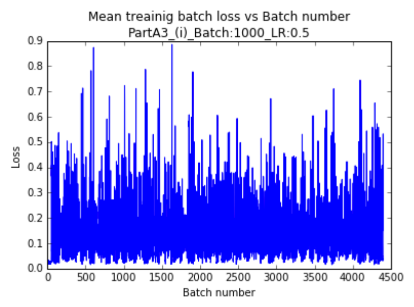












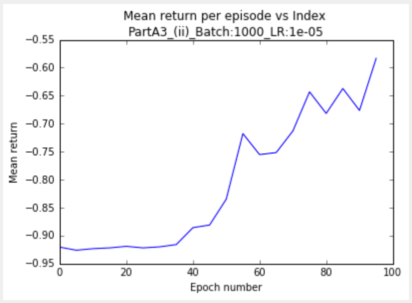
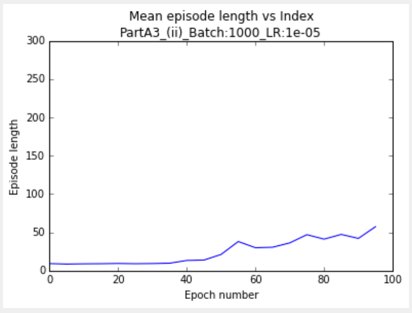
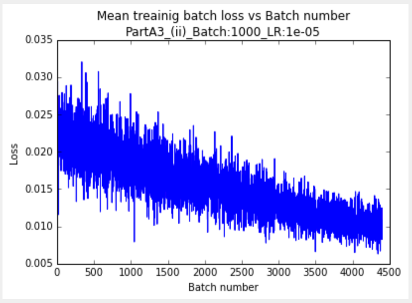
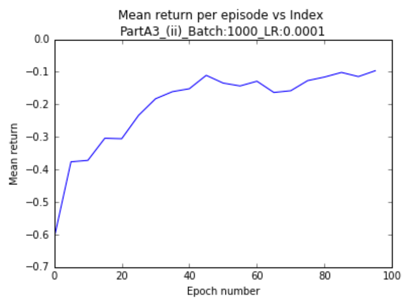
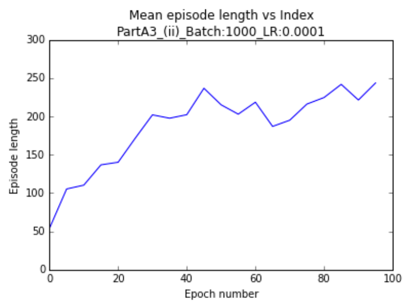
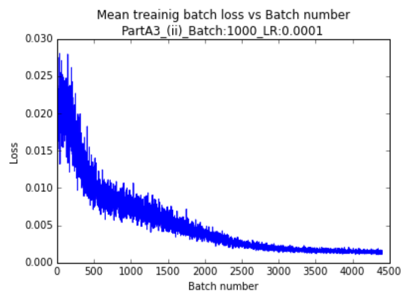
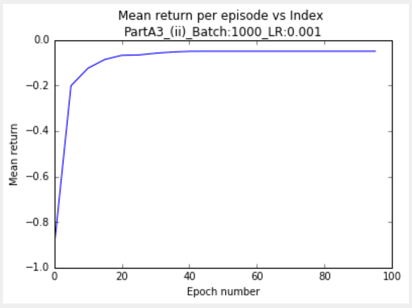
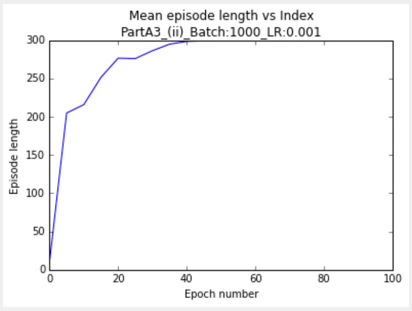
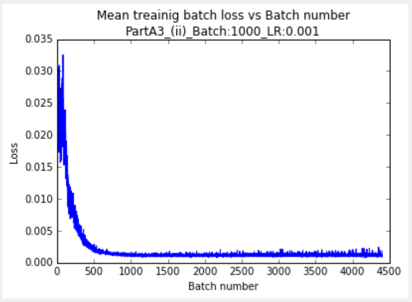
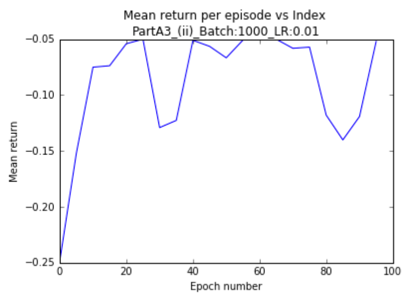
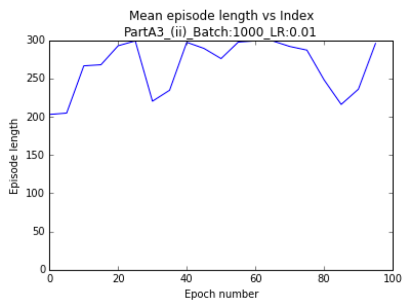
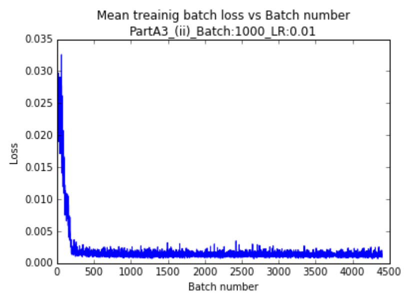
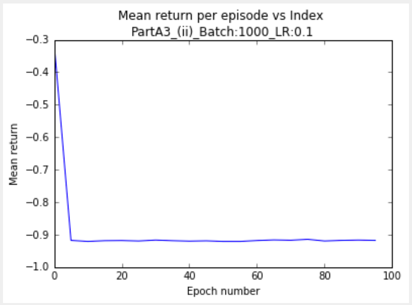
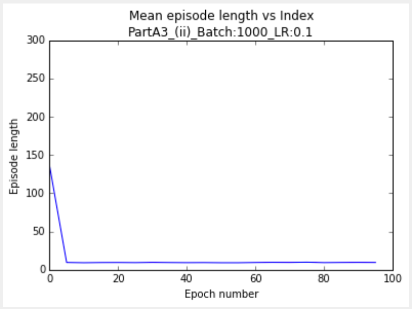
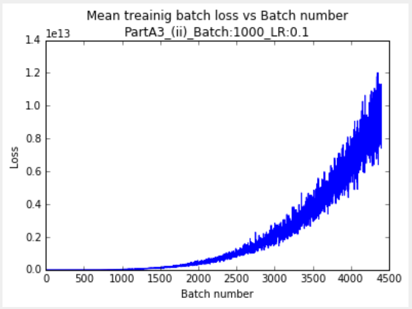
Commentary:

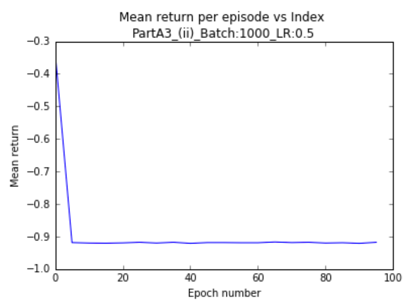
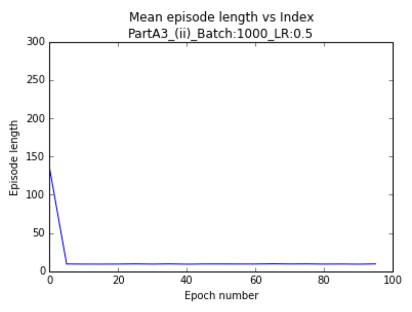
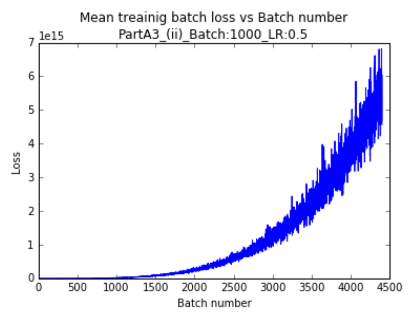
Generally model does not train well, the length of learning episode does not reach the desired 300, and the reward is mostly close to negative one, instead of zero. The performance across learning rates is generally random and highly sensitive to changes in the time seed used (results are displayed for seed = 20). Therefore, no concrete inference can be made about the optimal learning rate for this model. One can however develop the intuition that low learning rates are worse than intermediate and high ones.

This is due to several factors, firstly the model itself is not complex enough and training time is not long enough, secondly the batch size is too big, thirdly the RMSPropOptimizer used is not ideal and gradient descent would perform better. From trail and error, batch sizes of less than 200 would perform better. However, due to the model simplicity and the use of RMSPropOptimizer the results would still be unsatisfactory. Due to time constraints this model was not re-run but if it were than the batch size and optimiser type should be changed. However, it is likely that the refinement of the model itself, undertaken in consecutive parts of the assignment is the best way to proceed to obtain better results.

PART 3(ii) Hidden layer(100) – linear transformation + ReLU – followed by a linear layer with one output per action

Results are reported graphical below:

Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		
		
		
		
		

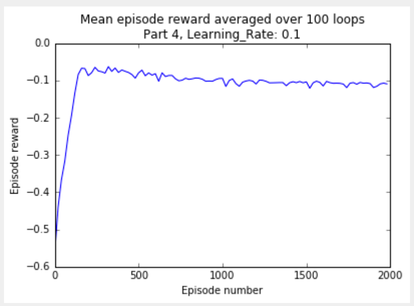
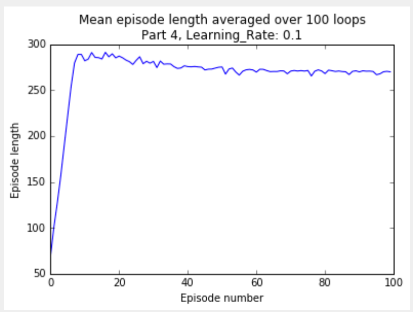
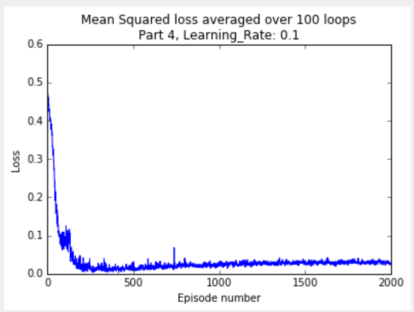
<u>Commentary:</u>
Keeping the same batch size, model trains significantly better than the one from part (i) but only for a few learning rates. It can be deduced that the learning rate of 0.001 preforms best, followed by 0.01. For these rates the episode length reaches 300 and the change in return form the initial state (reward) nears zero. The Bellman loss curve also follows the desired shape. Generally, it was observed that at learning rates below the optimal range the model was stable and still somewhat performing, whereas at high learning rates results became worse than random.
Learning rates of the order of magnitude of about 0.001 possibly going up from this number towards 0.01 could be further explored to fine-tune the model. Another avenue for improvement would be to try gradient descent instead of the RMSPropOptimizer.

Part 4. Online Q learning with a small neural net for function approximation.

In this section we use online Q-learning instead of batch Q learning (part 3), inputs are connected to a single hidden layer of 100 units followed by a ReLU, followed by a layer with one output per action. Additionally, the agent follows an epsilon greedy policy when training, which encourages taking random actions with the frequency dictated by the epsilon constant. This can be though of adding noise to the model and encouraging exploration.

<u>Final setup and hyperparameters:</u>
Model was tuned through trial and error, with the final selection:
Batch size was chosen at 1000, learning rate 0.1, epsilon constant for e_greedy policy during training was set to 0.05 and to 0 for evaluation. Due to long runtime learning rate choice was not extensively tested, with more time lower learning rates could be tried.
Using the gradient descent optimiser instead of the RMSPropOptimizer (both with default settings) improved training time and the stability of results. Therefore gradient descent was used for the reminder of this problem.

<u>Results Average results for a 100 loops with 2000 episodes are reported graphical below:</u>

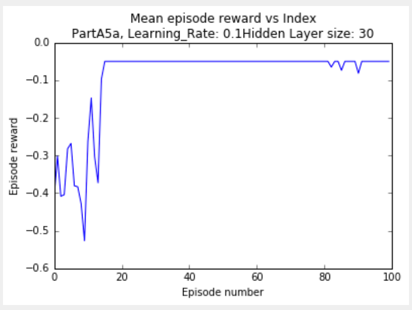
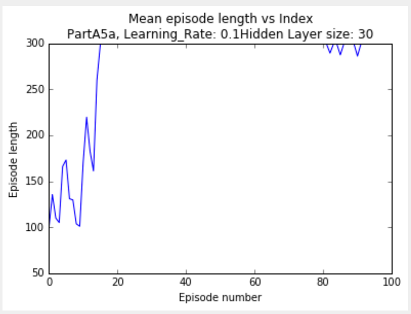
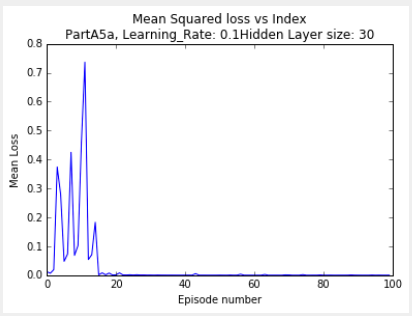
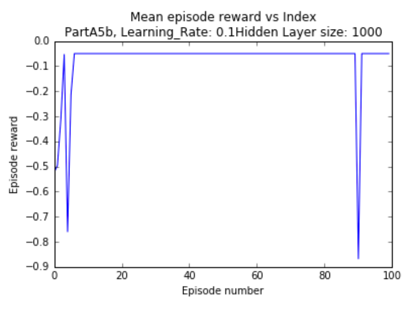
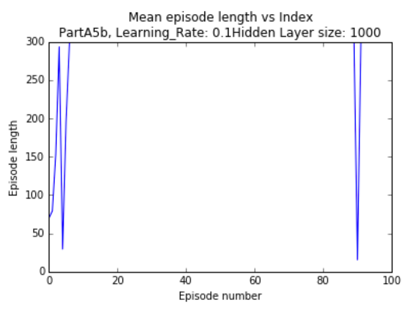
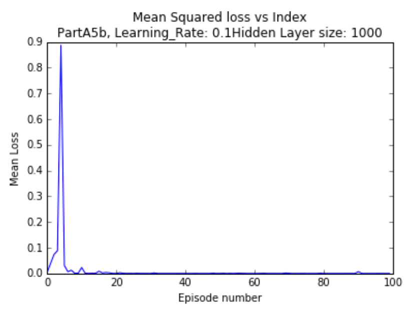
Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		

<u>Commentary:</u>

Generally, results are good; the model trains relatively quickly, performance curves and the loss both follow desired shapes. Rewards and loss are low as required, and episode length is satisfactory, albeit it does not reach 300. Through inspecting the console log for the 100 loops calculated, it seams that relatively large result variability can sometimes occur from run to run. This is possibly due to the random initialisation of variables, where certain weight configurations yield worse training results. For instance, for the last training episodes most of the repetitions of experiment give episode length of approx. 300, however on certain runs the model stays at about 100. This adversely affects the averaged results, even though most instances perform better. On the other hand, for the sections to follow it also shows that a single instance of 'worse' than expected results is possible. This part is the only one where we loop 100 times and average, albeit computationally expensive, the findings illustrate the importance of replaying the experiment and averaging to get a reliable final solution. The also demonstrate that in the sections to follow, unexpectedly 'good' or 'bad' results could be the fruit of an 'unlucky' random initialisation and would not represent the general average case very well.

Part 5. Repeat Part 4 with different hidden layer size.
In this part we explore and compare model performance for different sizes of the hidden layer.
In this section suffix A refers to the small network with 30 hidden units and suffix B refers to the large network with 1000 hidden units. For ease of comparison, the training hyperparameters for both parts A and B were kept the same and in line with part 4. This allows to assess the impact of the size of network on the model performance.

<u>Results are reported graphical below:</u>
Performance and loss plotted over 20 steps, 1st row is part A and 2nd row is part B results.

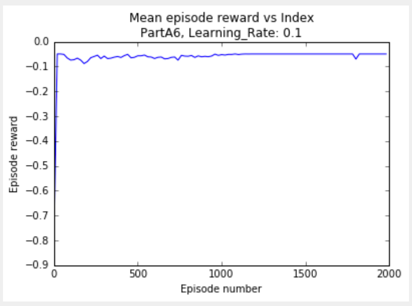
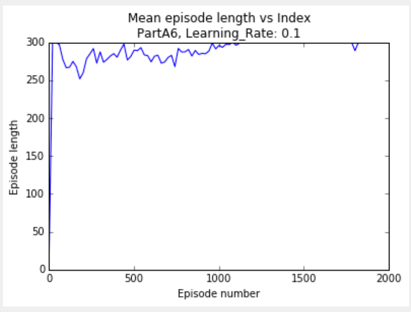
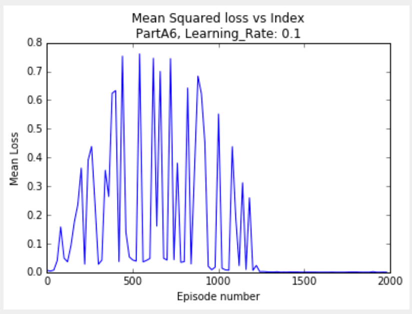
Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		
		

Commentary:
In terms of speed of results convergence the larger hidden unit in part B performed better than the smaller unit. However, the smaller unit is more stable and the large unit suddenly gets fatally 'lost' just towards the end of training, which is undesirable. It should be noted that at lower learning rates (i.e. 0.2), which were also tested both models trained slower but exhibited more stable results. These results demonstrate the importance of choosing a suitable size of the hidden layer for model performance. Comparing with part 4, it can be seen that the hidden layer of size 100 (part 4) is superior to both of those trailed in part 5. Therefore, in line with the assignment specification, this will be the parameter put forward in further sections.

Part 6. Add Experience Replay Buffer
Returning to the 100 unit network from part 4, in this section an experience replay buffer is added to store transitions of agent's experience. The agent will thus train on replayed experience and will only be presented with the real environment upon testing/evaluation.

Final hyperparameters:
Model was tuned through trial and error, with the final selections:
Batch size was chosen at 1000, learning rate 0.1, epsilon constant for e_greedy policy during training was set to 0.05 and to 0 for evaluation. The replay buffer size needed to be set to a large number thus 50000 was chosen. The number of initial episodes for populating the buffer was set at 1200. Other learning rates and batch sizes were tested but the ones chosen were a good compromise for the majority of models employing gradient descent optimisation. Therefore, this parameters were kept unchanged to allow for comparison between parts.

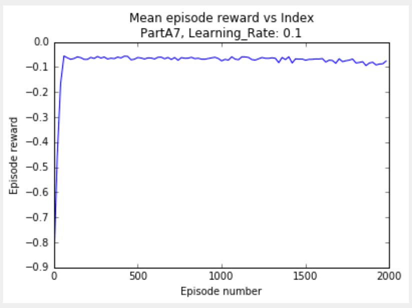
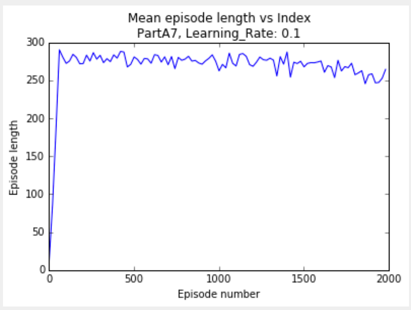
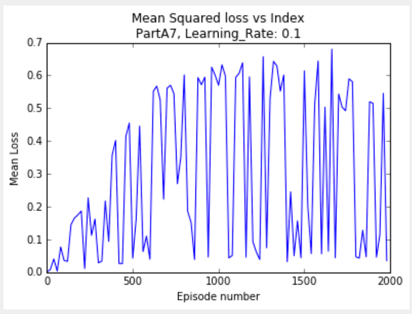
Results are reported graphical below:

Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		

Commentary:
Generally good results, with even faster convergence of performance measures (reward, episode length) to desired values, as compared with previous parts. Mean loss remains low. Upon testing for various hyperparameters and time seeds, it is clear that results re generally more stable and less variable then in part 4. This is to be expected as now the agent is learning from experience instead of just the current environment.

Part 7. Add a Target Network
In this section we add a target network to the net from part 6. The target network will contain current network parameters copied through every 5 episodes. Hyperparameters are kept exactly the same as in part 6 above to allow for comparison.

Results are reported graphical below:

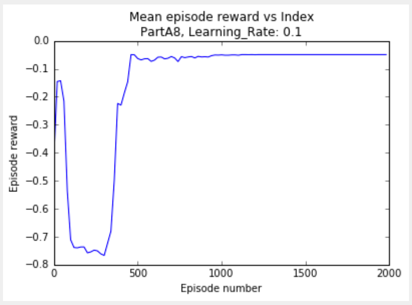
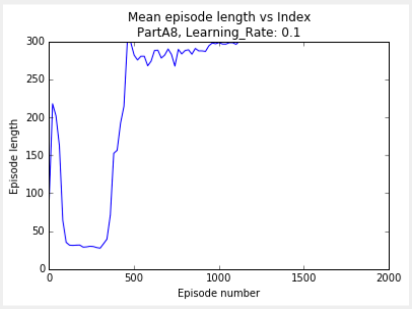
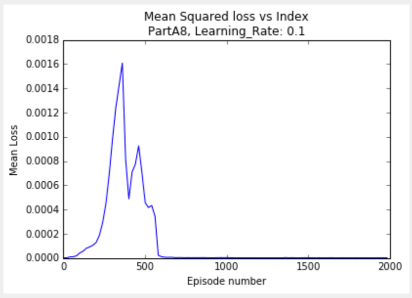
Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		

Commentary:
Generally good results, with fast convergence of performance measures (reward, episode length) to desired values and low Bellman loss. However, results are marginally worse than in part 6 as episode length does not reach the target 300 and the loss fluctuates widely. This indicates that copying network parameters to a target network every 5 steps did not markedly improve performance. This is unexpected as, in principle, adding the target network should decrease result variability and improve results. One way to improve the model is by changing the update step size, which has a large impact on the final performance. If time allowed this would be tested with the idea that larger step sizes could produce more stable results.

Part 8. Apply SARSA (State-Action-Reward-State-Action) instead of Q-learning.
The model was adapted based on model from part 7 above.

Final hyperparameters:
Model was tuned through trial and error, with the final selection:
batch size was chosen at 300, learning rate 0.1, epsilon constant for e_greedy policy during training was set to 0.05 and to 0 for evaluation. The replay buffer size needed to be set to a large number thus 50000 was chosen. The number of initial episodes for populating the buffer was set at 1200. Batch size was reduced from the larger batches used previously, this yelled substantially improved results. This is due to how SARSA works. Other learning rates were tested but 0.1 was a good choice for the majority of models employing gradient descent optimisation. Therefore, this rate was kept to allow comparison.

Results are reported graphical below:

Mean reward per Episode vs Episode number	Mean Episode Length vs Episode number	Mean Bellman loss during training, reported per batch
		

Commentary:
Results are generally good, with rewards quickly reaching near zero levels, episode lengths reaching maximum after approx.1000 epochs and Bellman losses remaining low. This results are competitive with those obtained in part 6. Q-learning assumes that the optimal policy is being followed, whereas SARSA is more sophisticated; it incorporates the agent's control policy into its update of actions. SARSA can take into account the opportunity cost of exploration and learn to counter or encourage taking random decisions. Thus, like in part 7, one would hope that adding additional information to the agents decision making process would yield improved results. It is possible that these learning technique are too sophisticated foe the problem at hand and that the model prom part 6, employing the replay buffer is just good enough.

Problem B: Atari Games

In this section Q learning is applied to three Atari console games: *Pong, Ms.Pacman,* and *Boxing.* The inputs to the model are raw pixels, thus the first step is image preprocessing where observation size is reduced to 28 by 28 and transformed to grayscale. A convolutional neural network is use to approximate the Q function. First layer: filter size is 6x6, stride is 2 and 16 channels followed by a ReLU. Second layer: filter size is 4x4, stride is 2 and 32 channels followed by a ReLU. Followed by flattening and a fully connected layer with 256 units followed by a ReLU. The net ends with a final linear layer that predicts the state-action value function. Training is undertaken with hyperparameters set as per assignment specification: fixed epsilon of 0.1, minibatches of size 32, discount factor of 0.99 . Rewards are clipped to be -1, 0, or 1. The RMSOptimiser is used throughout this section with a learning rate of 0.001 and other parameters kept at default values. The size of the experience replay buffer is kept at values slightly higher than the recommended minimum (with 110000 transitions). A target network is also used with parameters copied over every 5000 steps.

Note on terminology: Frame counts are equivalent to episode length (from the cart-pole problem) and score is equivalent to discounted reward, this names are used in code and plots.

Games Background:

PONG:
Pong is one of the first computer games that ever created. Pong is a two-dimensional sports game that simulates table tennis. It features two paddles (controlled by two player for a player and a computer) and a ball. Players can control the paddle movements and use them to hit the ball back and forth. Points are earned when one fails to return the ball to the other with the aim is for each player to reach 10 points before the opponent.

BOXING:
Boxing displays a top-down view of two boxers who can hit one another if close. Long punches score one point and closer punches score two. A match is completed when one player lands 100 punches or when two minutes have elapsed. In the case of a time-out, the player with the most landed punches is the winner.

PACMAN:
MsPacman is a slightly modified version of the original Pacman game. In this game an agent navigates in a maze while being chased ghosts. The player earns points by eating pellets and avoiding ghosts (loss of life on contact). The agent is positively rewarded
(+1) for collecting bricks. Bonuses also exist, and 'eating' a bonus temporarily freezes the ghosts from moving also scoring extra points. Rounds end after a fixed time and consecutive rounds are faster, the aim is to collect maximum points.

Comments to all results:
Results are generally bad, the model does not train and exhibits worst than random performance. Various optimiser hyperparameters were tried, with no significant improvements over default settings (results below). Due to the long run time, parameters have not ben tuned extensively, and given more time an computational resources, one obvious avenue for improvement would be to continue tuning.
Another reason for the bad results is that the pixel image feed for the games was shrunk too much and converting to grayscale removed even more information. It could be that maintaining a more robust image size (say 80 by 80 instead of 28 by 28) and image colour would yield the most substantial improvement. However, this would come at the expense of increased runtime. Furthermore, in reality, a model like this would be run for more agent steps and with a larger replay buffer size. State-of-the-art RL moles would also employ a mix of batch normalisation, input

clipping, prioritised experience reply or a duelling Q-Net architecture. Therefore , it is likely the model use there is just too simple to achieve good results.
Learning curves (part 3 and 4) are expected to be different from supervised learning because RL problems are not supervised learning problems; in RL the agent influences the environment through the actions taken and feedback is delayed.

Part 1. Report the score and frame counts from the three games under a random policy, evaluated on 100 episodes. Report both the average and the standard deviation.

PONG:
The Mean Score (i.e discounted reward) is: -0.98 and the standard deviation is: 0.19.
The Mean Frames (i.e episode length) are: 1227.22 and their standard deviation is: 132.43.

BOXING:
The Mean Score (i.e discounted reward) is: -0.338 and the standard deviation is: 1.111.
The Mean Frames (i.e episode length) are: 2380.6 and their standard deviation is: 14.41.

PACMAN:
The Mean Score (i.e discounted reward) is: 2.423 and the standard deviation is: 0.669.
The Mean Frames (i.e episode length) are: 645.38 and their standard deviation is: 77.28.

General comments to part 1:
In terms of score, Pacman achieves the highest positive score, which is desirable, Boxing performs worse and Pong is the worst. In terms of frames, the absolute value of which is game specific.

Part 2. Report performance on the three games from an initialised but untrained Q-network, evaluated on 100 episodes. Explain why the performance can be different from part one.

PONG:
The Mean Score (i.e discounted reward) is: -0.999 and their standard deviation is near 0.
The Mean Frames (i.e episode length) are: 1019.38 and the standard deviation is: 8.6.

BOXING:
The Mean Score (i.e discounted reward) is: 0.01 and their standard deviation is approx. 0.01.
The Mean Frames (i.e episode length) are: 2380.6 and the standard deviation is: 14.4.

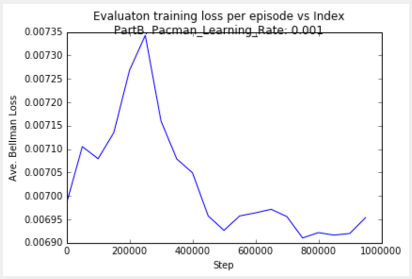
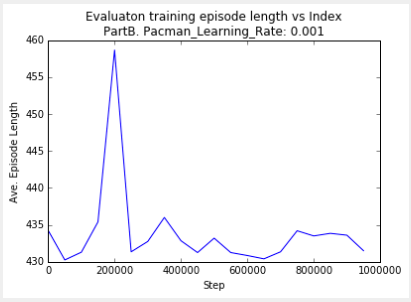
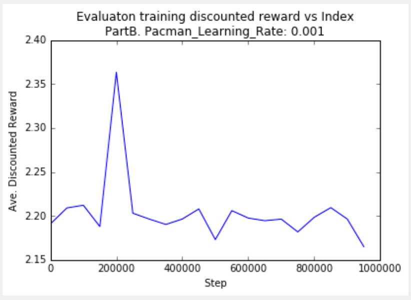
PACMAN:
The Mean Score (i.e discounted reward) is: 0.002 and their standard deviation s near 0.
The Mean Frames (i.e episode length) are: 627.9 and the standard deviation is: 6.7.

General comments to part 2:
In terms of score, Boxing and Pacman again achieve higher positive scores, which is desirable, and Pong performs worse. In terms of frames, the absolute value of which is game specific.

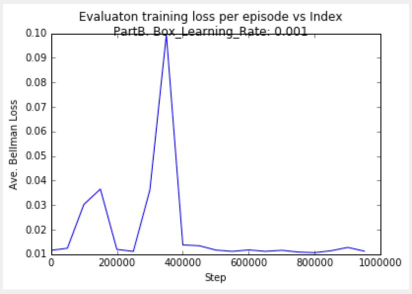
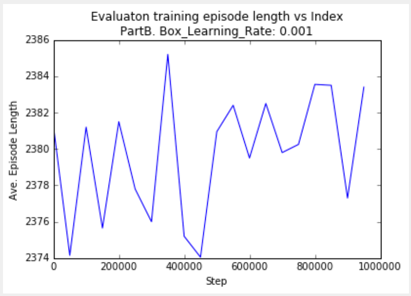
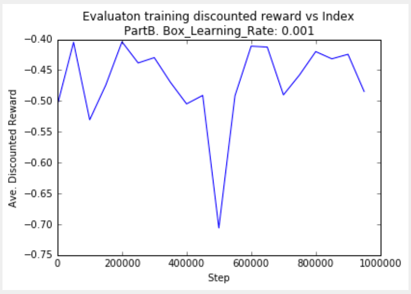
Comparison of parts 1 and 2:
In terms of the mean score, Q learning (part 2) performs the same for Pong, worse for Pacman and better for Boxing as compared with the uniform random policy in part 1. In terms of frames, Pong and Pacman play for s shorter time under the Q learning policy and Boxing remains the same. This general reduction in episode length in part 2, is due to the fact that a Q-learning agent can decide to always take the same action, whereas a random policy is more likely to switch between agent behaviours. Always following the same policy generally leads to a faster termination than under random game play. For example in say the Pong game, this means that agent is more likely to hit a ball at random then by always moving the racket in the same way. However taking the Pong score into consideration, the overall results is bad regardless of the agent policy. In Boxing, the taking the same action seems to be beneficial whereas in Pacman the converse is true.

Part 3. Report the training losses and performance measured every 50,000 agent steps.

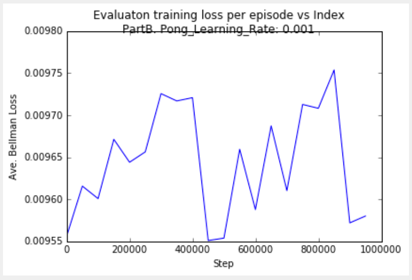
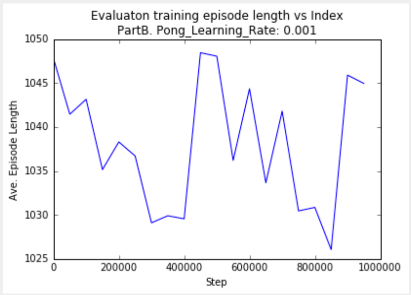
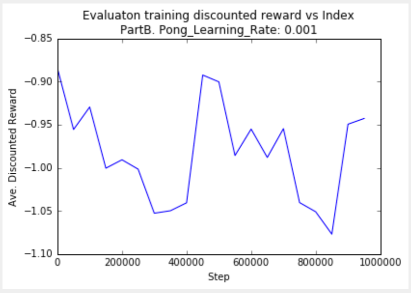
PACMAN:
Mean episode length: 644.22, Mean return from initial state: 3.42 Results are summarised in the graphs below with the training evaluation undertaken every 50,000 steps:

Training Evaluation Loss	Training Eval. Episode Length	Training Eval. Reward
		

BOXING:
Mean episode length: 2378.18, Mean return from initial state: -0.65. Results are summarised in the graphs below with the training evaluation undertaken every 50,000 steps:

Training Evaluation Loss	Training Eval. Episode Length	Training Eval. Reward
		

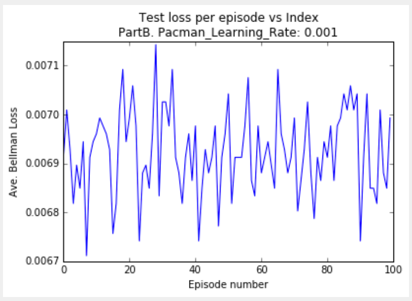
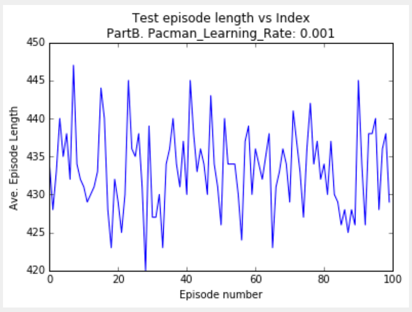
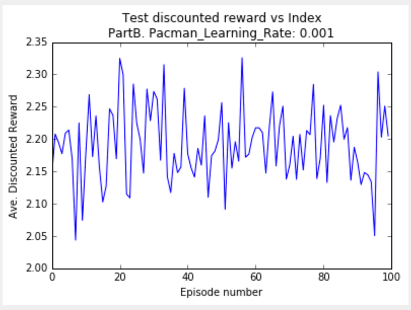
PONG:
Mean episode length: 1147.82, Mean return from initial state: -0.92. Results are summarised in the graphs below with the training evaluation undertaken every 50,000 steps:

Training Evaluation Loss	Training Eval. Episode Length	Training Eval. Reward
		

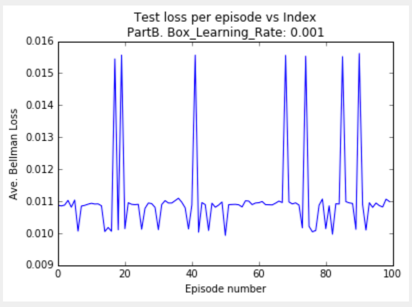
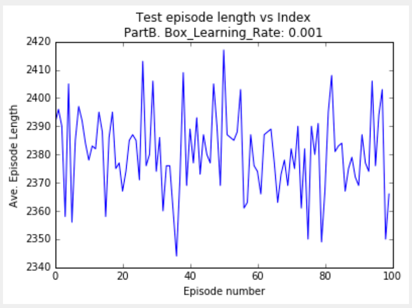
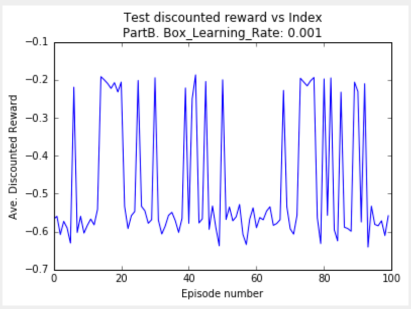
General comments to part 3:
Curves generally exhibit random performance indicating that the model has not trained.

Part 4. Test the trained agent, reporting the final loss and performance averaged over 100 episodes. Sadly none of the tested modifications mentioned earlier, helped attain good results.

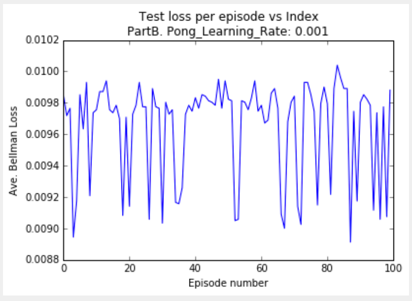
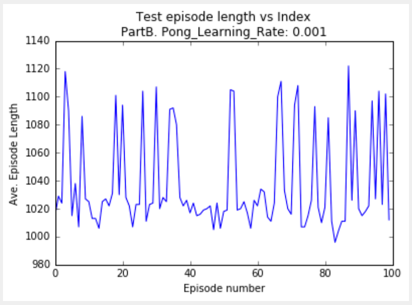
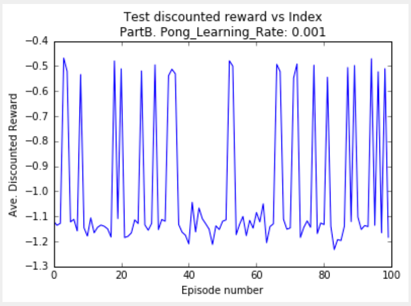
PACMAN:
Performance evaluation results averaged over a 100 episodes:
Mean episode length: 433.28
Mean return from initial state: 2.192

Testing Evaluation Loss	Testing Eval. Episode Length	Testing Eval. Reward
		

BOXING:
Performance evaluation results averaged over a 100 episodes:
Mean episode length: 2380.6
Mean return from initial state: -0.477

Testing Evaluation Loss	Testing Eval. Episode Length	Testing Eval. Reward
		

PONG:
Performance evaluation results averaged over a 100 episodes:
Mean episode length: 1037.62
Mean return from initial state: -0.996

Testing Evaluation Loss	Testing Eval. Episode Length	Testing Eval. Reward
		

General comments to part 4:
Curves generally exhibit random performance indicating that the model has not trained.