Project 2: Training an AI to Solve LunarLander v2

Thomas Melanson

Georgia Tech OMSCS

Git commit hash: e3e3e1488def22ecbec583538a7b38b59c0f13c0

June 29, 2020

1 Introduction

The goal of this paper was to utilize Reinforcement Learning principles to successfully play the Lunar Lander OpenAI gym (defined as scoring an average of 200 or above over 100 episodes). This was accomplished via a Double Deep Q network, trained over 1500 episodes of Lunar Lander training. This paper will first introduce the concept of Q learning in general, then go over Deep Q and Double Deep Q networks. Finally, the Lunar Lander problem is explained in full. In the Methods section, deep Q learning is discussed in the context of the paper by Mnih et. al., which trains a deep convolutional network to play various Atari games. The DQN implemented is compared and contrasted to the Mnih et. al. model, in both the network itself and in hyperparameters. It also briefly discusses the use of the CartPole problem as a sanity check for the DQN, and why it is an easier problem to solve. Finally, the Results section analyzes the effects of three hyperparameters : gamma, epsilon, and replay memory size. It also shows the performance of the optimal set of hyperparameters.

2 Background

2.1 Q Learning

In reinforcement learning, a good way to determine the "value" of taking an action at a certain state is by using Q values. Q values are, in classical reinforcement learning, values learned for a specific state, taking a specific action. For any given state s and action a, the Q value is often written as $Q(s, a)$. For a value of Q, the update equation when training on a greedy policy can be written as follows:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha(R_t + \gamma \max_a Q(s', a) - Q(s, a))$$

where the term $R_t + \gamma max_a Q(s', a) - Q(s, a)$ is the error term used for the update, and α is the learning rate.

Upon many successive iterations, when the Q update is based on the greedy future estimate, the Q value will eventually become the optimal value of taking the action a in state s. This optimal value function is denoted as $q*$.

Q learning is categorized as an off-policy Temporal Difference Control algorithm. Temporal Difference learning updates based on estimated values of intermediate states rather than the full reward sequence of a given episode. This way, the agent can learn from previous sequences involving the intermediate states. Q learning is also off-policy control because it can learn the values according to a policy (in this case, $q*$) without using this policy.

2.2 Exploration-Exploitation

In the exploration-exploitation dilemma, the optimal value can only be found through exploration of the state space and only optimized through exploitation of the learned values. For example, in the Lunar Lander case discussed later, the optimal reward can only be found through trying several different landings, but the value of the few successful landings has to be used to derive a consistent policy.

One solution [2] is the idea of an ϵ-greedy policy. This algorithm chooses a random action ϵ percent of the time and otherwise chooses the greedy action with respect to Q. That way, while the agent is trying to find the optimal action, it will occasionally explore the less optimal routes to find a potential hidden reward.

2.3 The Deadly Triad

The original Q learning algorithm created a table that mapped each (state, action) pair to a value. However, in environments with large state spaces, this would be cumbersome to learn because each state-action pair would need to be explored independently. In fact, for continuous state or action spaces, this is impossible; an infinite number of state-action pairs would take infinitely long to learn each value. To solve this, a function can approximate the true value of each state-action pair by correlating or grouping features. The function can then be trained in a finite amount of time, while still having a close-to-perfect estimation of values. This process is known as **function approximation**.

In the TD learning paper [4], it was proven that learning algorithms that take some advantage of its estimated values (ex. Temporal Difference learning) can converge more quickly and to a more accurate value than those that only train on the complete reward, such as the Monte Carlo method. In the Lunar Lander case, the Q value is trained on the estimated Q value of the next state, rather than the sum of rewards from that state to the end. This is known as **bootstrapping**, and is important because policies can be rewarded for transitioning to more advantageous states (i.e. those that have a higher chance of landing correctly), even if the reward for a given episode under the policy did not land correctly, and vice versa when landing in suboptimal states even though the episode ended with a positive outcome [3].

Finally, $\mathbf{Off-policy\ learning}$ is learning that can be used to evaluate policies besides the one used for training. Q learning, by definition, is a form of off-policy learning, because the policy used to take the action can differ from the one used to update the Q value. This is important for learning, because it allows the training policy to change when more optimal paths are discovered.

Although each of these three methods can lead to more optimal training and results, the combination of off-policy learning, function approximation, and bootstrapping often leads to divergence. This is known as the Deadly Triad [2], because the combination of all three can prove dangerous to the efficacy of the learning algorithm. To combat this, researchers have recognized that these concepts aren't completely binary; for example, in function used for function approximation can be a nonlinear neural network rather than a linear function, which can decouple states that were correlated in the linear function [3].

2.4 Deep Q Learning

As mentioned before, function approximation can help condense a large state space. Deep Q learning does function approximation with an artificial neural network (ANN) to determine the Q value for the state space.

Rather than update a tabular value of $Q(s, a)$, deep Q learning approximates the Q value as a neural network.Here, Q will be parameterized by a set of weights, labelled θ. The concept of a neural network is based on an individual unit called neurons. A neuron is comprised of two parts, a weighted combination of the input and a nonlinear function which zeros out values that don't score highly. Because this nonlinear function determines if the neuron plays an "active" role in subsequent layers, it is called an activation function. Several neurons are stacked into a vector called a layer, and several layers are combined sequentially to form the network.

Updating a neural network happens in two steps: forward and back propagation. Forward propagation, or prediction, takes the input and feeds it through each layer, generating the output. Back propagation takes the difference between the predicted output and the actual output and computes the gradient with respect to the weights. The gradient is then used to update the weights to lower the subsequent error.

Traditionally, in supervised learning, a neural network would learn on a pre-labeled training dataset, updating random pieces of the data in batches to prevent over-fitting a specific section of the data. However, in traditional Q learning, the values are added one iteration at a time, in the order they are decided. To solve this, implementations such as the Mnih et. al. paper create something called a replay memory, which stores the state, next state, action, and other important metadata for a given iteration in the episode. This memory can then be batched at random and used to update the Q network like in supervised learning algorithms.

2.5 Double Deep Q Learning

In the original Deep Q Learning update step, the same policy is used to both evaluate and update. When evaluating an incomplete problem, the policy is biased towards the already known actions. If that same policy is also evaluating the value of a state/action, it will also train the Q function to reward the current action. This causes certain actions to have disproportionally higher amount of information, and is known as overestimation.

To prevent this, Double Deep Q learning uses two Q functions, one for determining the actions, and another for grading the result of the actions. Generally speaking, this is represented as the same Q function, but with different weights θ and θ':

$$\Delta Q_{t+1}(s, a, \theta) = \alpha(R_t + \gamma \max_a Q_t(s', a, \theta') - Q_t(s, a, \theta))$$

In the double Q learning algorithm used in the paper, the weights θ were set to the values of θ' after a fixed number of iterations; this value is referred to as C (from the corresponding variable in the Mnih et. al. paper).

2.6 Lunar Lander

Lunar Lander is an OpenAI gym problem characterized by a lander, visually represented by a simplified lunar module, and a semi-randomly generated terrain. The terrain has a goal landing area in the lower center section, which then extends to the left and right with random curvature.

The Lunar Lander state is defined by $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}, leg_L, leg_R)$. x is the horizontal position of the Lunar Lander (-1 being the left wall, and 1.0 being the right), y is the vertical position (-1.0 being the bottom of the screen, 1.0 being the top), and $theta$ is the orientation in radians. The next three are the time-based derivatives of these three values which can have any real value. Finally, leg_L and leg_R are binary variables, equal to 1 if the left leg / right leg (respectively) is touching the ground, and 0 otherwise.

In order to control the Lunar Lander, the user must select one of 4 options: do nothing (0), fire the left orientation thruster (1), fire the main engine (2), and fire the right orientation thruster (3). The orientation thrusters incur -0.03 penalty per action, the main engine incurs -0.3 penalty per action, and doing nothing does not change the reward.

Moving from the top of the screen to the landing area at (0, 0) will result in a reward between 100 and 140 points. Straying from the center (i.e. $x = 0$) will be penalized, but a played can earn back the penalty by returning to the middle. Each lunar leg touching the ground is worth 10 points, and coming to a stop finishes the game with 100 points. However, if the vehicle does not land properly (either too quickly or in the wrong orientation), the game incurs a penalty of -100 points.

3 Methods

3.1 Starting with CartPole

In order to make sure the neural network was learning properly without running hundreds of iterations, it was first tested on the CartPole problem. The objective of this game is to balance a pole on a moving cart. This was due to its simplified state space and lack of high-risk, high-reward scenarios. In this problem, there are four states and two actions: $(x, \theta, \dot{x}, \dot{\theta})$ and

$(left, right)$, respectively. x and $\dot{x}$ represent the position and velocity of the cart, and θ and $\dot{\theta}$ are the angle and angular velocity of the pole, respectively.

Each frame the pole is "balanced" (i.e. $abs(\theta) < 15deg$) gives a reward of 1. The more the optimal the policy, the higher reward it'll get. This format is more forgiving than the Lunar Lander challenge, where landing with a suboptimal policy could penalize the agent 100 points or reward the agent 100 points depending on the velocity.

The biggest change in parameters from CartPole to Lunar Lander was the increase in the width of the hidden layers of the neural network. In the CartPole problem, I only used layers of size 12 and 8. In the Lunar Lander version, however, these both increased to 128. However, the number of layers (2) was kept the same, as having too many layers can result in overfitting.

3.2 Mnih et. al. implementation

The Double Deep Q network implementation in the paper most closely represents the paper from Mnih et. al and published in Nature. This learning algorithm, in addition to implementing the algorithms described above, added a few key features to ensure the convergence of their algorithm.

One drawback to the traditional Q-learning process is that multiple iterations are spent on the same episode. Because of function approximation, Q values of states outside of the episode's path could be affected. To solve this, Mnih et. al. used batched learning. Rather than do backpropagation on each successive iteration, Deep Q learning randomly chooses a batch of iterations from the "training set" of all iterations of all run episodes. This helps ensure the update step is taken from a variety of locations in the state space.

Second, the exploration-exploitation paradox was solved by using a decaying value of ϵ in ϵ-greedy exploration. The value starts at $\epsilon := 1$ (the agent is fully random) and linearly decays (i.e. a constant amount each iteration) to 0.1 (only 10 percent of actions are random) over the course of 1 million iterations. From that point, the epsilon value reamins constant. The value used in this paper is 100,000 because the total number of iterations is smaller.

ϵ was also kept during the test section in order to prevent overfitting. In the Mnih et. al. implementation, the value of ϵ was 0.05, whereas in the implementation here it was 0.1.

3.3 Differences from Mnih et. al. paper

Because the Mnih et. al. paper dealt with a visual input (the Atari screen), it made sense to use a convolutional neural network. It would have also been possible to utilize a CNN with the Lunar Lander, as the render could be used as the input rather than the observation vector. However, since the important components of the Lunar Lander state spaced are condensed to 8 components, using the full environment render would add unnecessary complexity. Instead, a fully connected network with 2 hidden layers (each of size 128), each with ReLU activation, was used.

The original Nature paper used 10 million iterations when training its network. Because the fully-connected ANN used in this implementation had fewer layers (2) and parameters (17920) than the convolutional network used in the Nature paper (5 and at least 70720, respectively [1]), fewer iterations were necessary. This meant, however, that many of the iteration-based hyperparameters (such as the original replay start count) also changed to reflect the overall lower iteration count. These variables included:

- C (the number of iterations between target updates)
- final exploration count (the number of iterations until the epsilon stops decaying)
- replay memory size (number of iterations the network can look back to do batch update)
- relay start size (number of iterations before the network starts learning)

For this paper's implementation, C is 50, final exploration count is 100000, replay memory size is 10000, and replay start size is 1000.

4 Results

The first subsection will cover the results of the optimal Q algorithm trained for 1500 iterations. The next three sections will cover each of the three hyperparameter changes.

4.1 Results on Lunar Lander project

This training run used hyperparmeters discussed in the Methods section, as well as the optimal hyperparameters found in the experiments (0.99 γ, 1.0 initial / 0.1 final ϵ, replay memory size 10000). Although some of the experiments were originally run to 1000 iterations, the final training was extended to 1500 in order to fully capture the jump between an expected value of 100 to an expected value of > 200.

The training curve starts out with rewards on par with a random agent. Around 500 and 600 frames, the average value of each episode becomes positive. This is where the lander first discovers positive rewards from landing. At around 1000 frames, the final landing strategy is discovered, and the average training reward jumps to over 200. This is when the lander learns to reach the goal quickly enough to get to the landing point efficiently. The test curve's average value (221.71) is notably lower than the median value (255.30), meaning that there are occasional outliers over the course of several episodes. From observing the landing sequences, these outliers are due to landing a little too quickly, or landing on uneven ground.

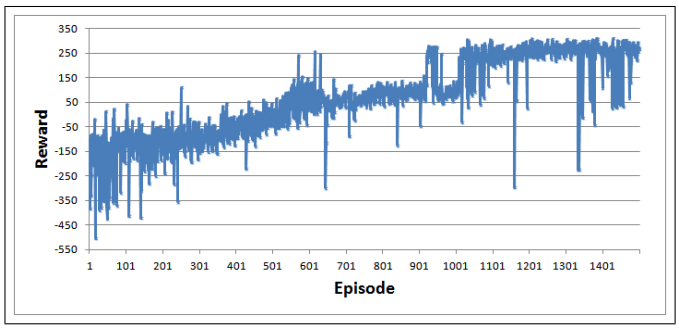

Figure 1: Reward per episode for 1500 training episodes using the optimal hyperparameters.

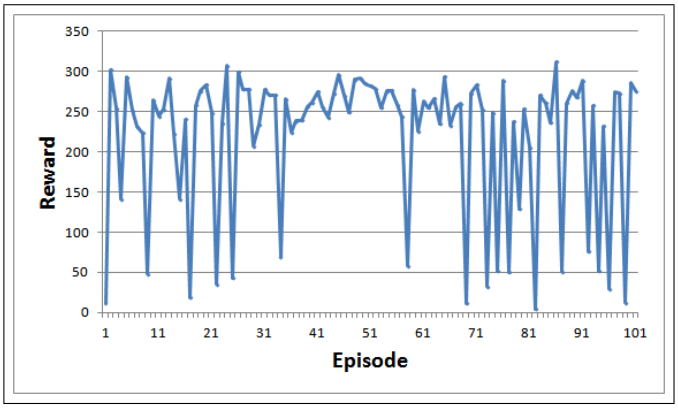

Figure 2: Reward per episode for 100 test episodes on the model after training for 1500 episodes.

4.2 Experiment 1: Changes in Gamma

In this experiment, γ is defined as the discount used when determining the value of the next state by the equation:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(R_{s,a} + \gamma \max_a Q(s',a))$$

Indirectly, the value of γ affects the horizon of the problem, or how many steps ahead the Q value considers for a given state. If $\gamma := 0$, then the Q value of future states are ignored, and if $\gamma := 1$, then all states from the current state to the end state are considered.

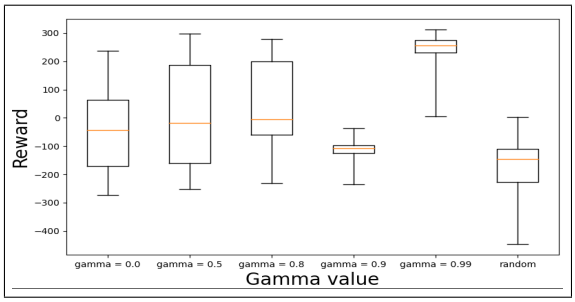

Figure 3: Reward distribution over 100 test episodes for each value of gamma after training for 1500 episodes. On the far right is the performance of a random agent. $\gamma = 0.999$ is excluded here because the results were too far off the scale with an average final reward of -810.1949

As expected, the highest value of γ (0.999) resulted in the agent moving away from the ground, receiving a worse-than-random average score of -810.1949 and a min/max of -4185 and -331. This is because, although landing correctly can give the agent a high reward, the chance of getting a penalty is much higher than the chance of reward, especially at larger velocities. This caused the Q-learning algorithms with high gamma to disincentivize falling quickly. However, modules with high enough gamma, the agent is disincentivized to the point of not exploring landing options, thereby not seeing the potential reward.

At 0.9, the agent developed a policy against using its orientation thrusters, instead letting itself drift towards the edge. Likely, when using thrusters, landing at an angle proved safer than coming straight down. Potentially due to function approximation, drifting to the side for a safe landing meant often going off screen. The mean value of this policy was -114.48, with standard deviation of 34.8 , which is comparable to the random policy average of -180 (with standard deviation 102.3).

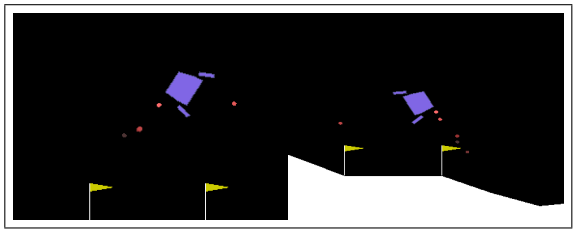

Figure 4: At low values of γ (displayed are results at $\gamma = 0$, the agent favors a swinging motion to save fuel and drift towards the center

Interestingly enough, for low values of γ (< 0.9), a unique strategy emerged. Because the relative cost of the main engine versus the orientation engines mattered more, and because the lander is rewarded for moving towards center, the agent opted to swing in an arc while descending. At the height of the swing, the orientation engines were pointed directly downward, allowing the agent to use it like the main engine. The large fluctuation in landing angles caused the lander to crash more often, resulting in an overall higher variance. The average final reward of each value of γ was -42 for $\gamma = 0.0$, -3.2 for $\gamma = 0.5$, and 42 for $\gamma = 0.8$.

The most unexpected part was that, even with low values of γ (0.1), the policy considered firing the orientation thrusters to stay afloat more optimal than doing nothing. The eventual 10^2 order of magnitude reward on landing would be less significant the immediate 10^{-3} order of magnitude penalty of firing of the orientation thrusters within 5 or 6 frames. From empirical tests, the average frame count of an episode when running a random policy is between 90 and 100, so the majority of frames wouldn't consider the end reward.

The best explanation is the involvement of function approximation. Even if relatively few frames considered the landing error, the landing error was large enough that it was a factor when computing the gradients on each update. Overall, the high variance of results from this policy made it too inconsistent for the final solution, but it was nevertheless an intriguing phenomenon worth exploring.

4.3 Experiment 2: Initial and Final Epsilon

In the background section, a proposed solution to the exploration-exploitation dilemma was to have an ϵ-greedy policy that chose a random action some percent of the time. The Mnih et. al. section discussed further expanded this by having ϵ decay from 1.0 to 0.1 over a series of iterations. However, given the differences between the operating spaces, the optimal values of initial and final ϵ could be different. In this experiment, the initial value of ϵ was tested over a range of values from 0.3 to 1.0, and the final value from 0.05 to 0.9. Each of the two variables were tested in isolation, with the other variable set to the default values from the paper.

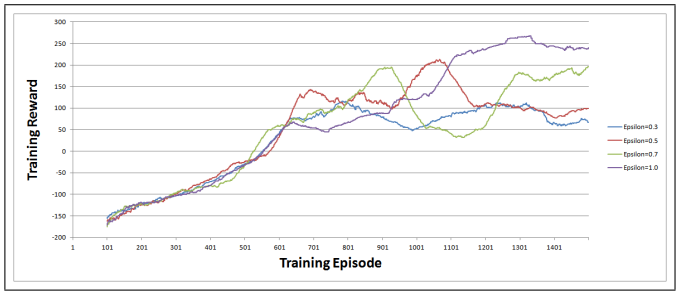

Figure 5: Average training error of past 100 episodes during 1500 episodes of training for different values of **initial** epsilon.

For low initial ϵ (0.3, 0.5), the training error per 100 episodes ended at 67.1 and 99.05 average reward, respectively. At an initial ϵ of 0.7, the final training reward was 197.03, and at 1.0, the final training reward was 238.40. At the extremes of 0.3 and 1.0 initial ϵ, the peak average training reward was 115.82 and 267.56, respectively. This shows that, without initial exploration, the training algorithm will more likely converge to a local optimum than find a maximum one.

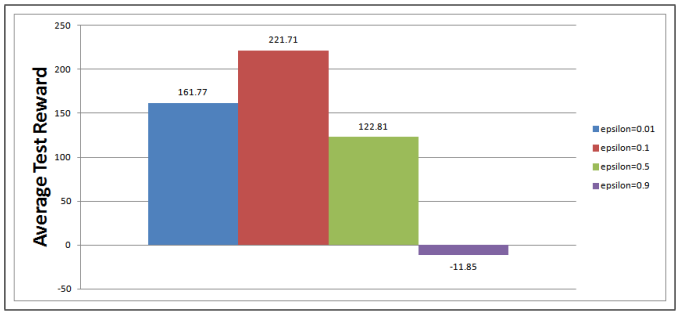

Figure 6: Average test error after 1500 episodes of training for different values of **final** ϵ. Test error was used instead of training error due to high final ϵ runs being essentially random exploration.

On the other end, the final value of epsilon should be small, but not too small. Figure 6 compares the results of having a final ϵ of 0.01, 0.1, 0.5, and 0.9. As expected, after Q-learning with an ϵ value of 0.9 (i.e. little exploration), the trained model performs poorly (although significantly better than the random agent reward of -180). However, the chart also shows that too small a final ϵ (0.01) leads to inadequate exploration of final values (compare the final average training reward of 181 at $\epsilon = 0.01$ vs. 241 at $\epsilon = 0.1$) and some overfitting (the test average reward of 161 is notably smaller than the training average reward of 181 for $\epsilon = 0.01$).

4.4 Experiment 3: Replay Memory size

The replay memory parameter is the number of previous iterations stored for use in batched learning. The idea was to prevent learning on overestimated values, and instead train on a wide range of data. It turns out that the size of the replay memory does have some effect on the learning curve. While it makes sense that too small a replay memory reopens the overestimation issue from before, too large a replay memory size means the batch update is less likely to use newer, more optimal policy runs.

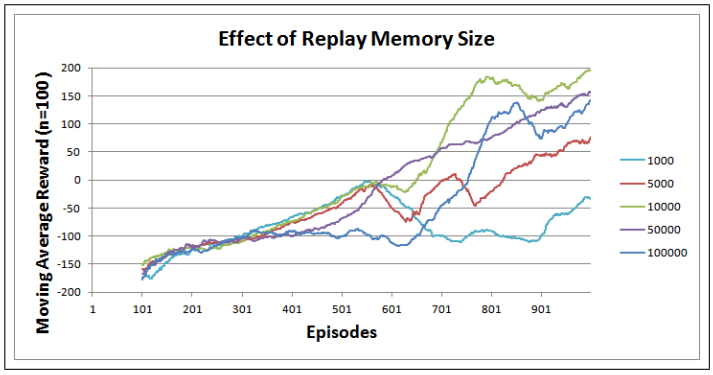

Figure 7: Moving average (n=100) of the reward per episode over 1000 training episodes. Each series is a different replay memory size. As seen here, a replay memory of 10000 learns more quickly than the larger replay sizes (50000, 100000) and retains memory and utilize successful episodes more than the smaller replay memory sizes (1000, 5000).

The paper originally put the memory at 1000000 ; in this experiment, the values ranged from 1000 to 100000. The peak values for each memory size (denoted as $< replay\_memory\_size >, < reward >$) are: 1000, 208.23; 5000, 270.8525; 10000, 290.4245; 50000, 290.113; 100000, 287.501. The final average test value for each replay memory size was as follows: 1000, -31.173; 5000, 78.486; 10000, 196.112; 50000, 157.615; 100000, 143.039.

As expected, too low a value of replay did not converge to the optimal reward because of overestimation, and high values of replay memory did not converge as quickly. Overall, the replay memory size of 10000 scored the highest peak reward and highest average training reward at the end of the run.

5 Conclusion

The end goal of this experiment was to train the LunarLander to score above a 200 for an average of 100 episodes. This goal was achieved through Deep Q Learning with hyperparameter tuning. Additionally, each of the three experiments showed a significant effect of the hyperparameters on the training and/or the final result.

References

[1] Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David et. al. "Human level control through deep reinforcement learning,". 26 February 2015. [retrieved 28 Jun 2020]

[2] Sutton, Richard S. and Barto, Andrew G. "Reinforcement Learning: An Introduction". 2020. [retrieved 28 Jun 2020]

[3] van Hasselt, Hado et. al. "Reinforcement Learning and the Deadly Triad". 6 Dec 2018. [retrieved 6 Jun 2020]

[4] Sutton, Richard S. "Learning to Predict by the Methods of Temporal Differences". 1988. [retrieved 6 Jun 2020]