# Transfer Learning with Deep Q-learning Networks

**By Line Kruse**
MA Cognitive Science
Aarhus University, Denmark

Code on github: https://github.com/LineKruse/Data-Science

***Abstract***

Deep Q-learning networks (DQN) have proven highly useful in typical reinforcement learning (RL) tasks, in which an artificial agent aims to maximize cumulative reward by learning from prior experience in a sequential decision-making paradigm. DQNs combine Q-learning with neural networks to approximate an optimal policy on a RL problem. The current project aimed to apply deep Q-learning algorithms to the Cartpole and Lunar Lander game environment, and investigate the extent to which transfer learning from the former improves performance on the latter. Transfer learning was assessed under four conditions, varying the amount of layers with pre-trained weights as well as the position of such transfer layers in the network architecture. Transfer learning was further compared between DQNs with two or three hidden layers. Results showed that transfer learning improved performance on the Lunar Lander task under certain circumstances. Best performance was obtained by a D3-DQN applying pre-trained weights in the second hidden layer and reinitializing weights in the third hidden layer. Thus, DQN models appear to be able to generalize experience from one domain to another, laying initial building blocks for improving speed of learning in RL problems as well as building more adaptable AI systems.

## 1.0. Introduction

Reinforcement Learning (RL) algorithms are a class of machine learning methods applied in continuous decision making tasks, where an artificial agent seeks to optimize control of an environment by learning from experience. In contrast to other types of machine learning, such as supervised learning, which aim to predict values or classes from labelled data, and unsupervised learning aiming to find relations and clusters in unlabelled data (Zychlinski, 2019), RL algorithms aim to maximize an agent's cumulative reward through sequential interaction with a continuous environment. By reinforcing actions with rewards or punishments, the agent learns to develop an optimal strategy, *a policy,* of successful behaviour (Surma, 2018).

### *1.1. The Reinforcement Learning Environment*

The environment of RL problems is characterized by a set of states, *S,* and a set of actions, *A.* In each state, $s \in S$, the agent chooses an action, $a \in A$, yielding a new state in which a new action is chosen, and so forth until the terminal state where the task is solved. Each action yields a reward, which can be positive, negative, or zero. Based on these rewards, the agent learns an optimal behaviour in difference scenarios (states) of the environment by deriving appropriate representations of the environment and generalize these from past to future experiences (Mnih, 2015).

A special property of RL is that the ability to achieve the goal requires the execution of a sequence of non-special actions (Zychlinski, 2019). That is, since the environment is continuous, actions that may not yield an immediate reward must be executed in order to reach the state in which a reward-yielding action can be performed. Hence, the agent must learn to act on the basis of potential future rewards by exploiting the fact that each state is a consequence of previous actions, and previous states therefore contain useful information for choosing future actions (Zychlinski, 2019). However, in many cases it will be ineffective, and even impossible, for an agent to remember all past states and incorporate them all into the decision making process. For this reason, RL problems are typically modelled as a Markov Decision Process (MDP). In an MDP each state is assumed to be dependent only on the prior state and the transition from here to the current state (Kim et al., 2019). This idea of sequential dependencies between states is crucial for learning optimal long-term strategies.

*1.2. Q-learning*

Q-learning is a prominent RL algorithm, providing a means for learning from experience in situations where the majority of random moves will cause failure (Patel, 2017). Learning is based on a Q-table containing a utility value, Q, for every possible action in every state of the environment. The Q-value is the cumulative reward received by a sequence of actions. In each state, the best possible action is the action that will maximize the Q-value across successive trials. That is, the Q-value of action *a* in state *s,* is dependent on the Q-values of the possible actions in later states. Formally, the choice strategy of the Q-learning agent is described by what is known as the Bellman equation (Bellman, 1957):

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a) \qquad \textit{Equation (1)}$$

where *r(s,a)* denotes the reward received by executing action *a* in state *s, s'* denotes the next state from choosing action *a,* and $\gamma \in (0,1)$ is a discount factor, determining the importance of long-term rewards to the decision process relative to current rewards. The Q-value obtained from the current state, *s,* and action, *a,* is thus the immediate reward received by performing action *a,* plus the maximum possible Q-value to be obtained from the following state, *s'.* The agent will receive the highest cumulative reward, by maximizing the Q-value on each trial. However, since the agent doesn't know the solution to the task, it will have to learn the Q-values from experience by updating the Q-value for the chosen action in each state according to the Bellman Equation. As the agent's experience accumulates it will become more and more certain about which sequence of actions yield larger long-term rewards. However, if the agent always chooses the action yielding the highest Q-value it will never explore new actions that may yield even higher rewards than what it has experienced so far. To avoid this, Q-learning uses an $\varepsilon$-greedy approach, in which choice of action is controlled by the parameter $\varepsilon$. In each state the agent will thus choose the action with the highest Q-value (exploitation) with a probability of *1- $\varepsilon$,* while it will choose a random action (exploration) with probability $\varepsilon$. Further, $\varepsilon$ is depreciated over time by a decay rate so that the agent will explore more on earlier trials, while exploitation of knowledge increases following more experience.

Calculating true Q-values is however only possible in very simple low-dimensional environments where the set of possible states is minimal. In most tasks the state space is continuous and the

number of possible state-pairs is too large to enable the calculation of all true Q-values. However, a nice property of the Bellman Equation and MDP is the recursive nature of the update function, which allows future rewards to propagate back to previous states (Patel, 2017). That is, without knowing the true Q-values, it is possible to approximate these from past action-state pairs and the associated rewards using an approximator function. Deep neural networks have proven highly efficient in this regard.

**2.0. Deep Neural Networks**

Neural Networks are a class of algorithms characterized by hierarchically organized layers of computing units, neural units, passing information through the network from the input layer to the output layer. Neural units resemble biological neurons by converting a complex input pattern into a single output. Each neuron in the network takes a weighted sum of its input, adds a bias term, and computes a new weighted sum as output. In the most basic architecture, a feedforward neural network, the output from each layer is passed directly as input to the next layer in the hierarchy. Deep Neural Networks are characterized by having one or more hidden layers, i.e., layers of neural units between the input and output layer. Passing information sequentially through hidden layers corresponds to dividing the task into sub-problems where the network builds up progressively more abstract representations of the input (Nielsen, 2015). An activation function is applied to the weighted sum of each neuron, determining the output of the neuron given the input. If the activation function is a step function, the neuron will simply turn “on” or “off” depending on a threshold. However, a type of linear activation function, such as the Rectified Linear Unit (ReLU), will often improve learning ability, since small changes in the input will not affect the output drastically (Nielsen, 2015). Learning occurs via error back-propagation, in which the error in the output of the network is propagated back through all layers and the weights and biases of the neural units are adjusted accordingly, to minimize error in future outputs. This process is controlled by a loss function and an optimizer. The loss function determines how to calculate the error in the output. The optimizer function determines how the weights and biases of the neurons should be adjusted according to the error (Gulli and Pal, 2017).

**3.0. Deep Q-learning**

Classical Q-learning algorithms are challenged when the state space of the task is too complex for the agent to learn a Q-value for every possible action-state pair. However, applying deep neural

networks allows the algorithm to approximate appropriate Q-values. A DQN takes the state of the environment as input, and outputs a Q-value for each possible action given the state. Since the networks have parameters, weights, that can be trained, we can replace the function $Q(s, a)$ with $Q(s, a; \theta)$, where $\theta$ represents the weights of the network, and we get a parameterized approximator function. The weights of the network can then be trained by minimizing error in the output. According to the Bellman equation (Eq. 1) the optimal action, *a,* in state, *s,* is the action that yields equality between the right-hand side and left-hand side of the equation. Thus, what the network should aim to minimize is the difference between these two. By applying the approximator function $Q(s, a; \theta)$ to the Bellman equation it is transformed into the Mean Squared Error (MSE) loss function with which the network can be trained (Mehta and Main, 2009):

$$Cost = \left[ Q(s, a; \theta) - \left( r(s, a) + \gamma \max_{a} Q(s', a; \theta) \right) \right]^2 \qquad \textit{Equation (2)}$$

Thus, the loss in each state is the mean of the squared difference between the current Q-value and the maximum Q-value possible from being in state *s'*. If the state is a terminal state (task solved), there will be no future rewards or states, and Q-value will be equal to the reward received, *r*. Similar to regular deep neural nets, the weights can then be updated over consecutive trials with an optimizer function and error back-propagation, until the weights converge on values consistently producing small errors.

### *3.1. Training the DQN-agent*

In contrast to supervised learning environments, in which neural networks are typically applied, the agent in RL tasks does not have labelled data with information on the right moves to take in different scenarios available to train on. Rather, the agent must learn from experience. This feature is implemented in DQNs as Experience Replay (Zhang and Sutton, 2017). With Experience Replay, training data is created on the fly from previous trials. In each step the agent stores its experience (state, action, reward and next state) in memory. When *b* experiences are recorded, the agent randomly samples *b* experiences from memory and train on these. The batch size, *b,* is a hyperparameter of the model. The experience-samples are drawn uniformly and at random to avoid training on a possibly skewed data distribution (e.g., only most recent steps), and to reduce correlation between experiences resulting from the order of observation (Mnih et al., 2015).

In sum, the DQN agent is composed of a deep neural network, which implements the Bellman equation from Q-learning as loss function. For each input state, the deep neural network will output a value for each of possible action to take, and the agent will choose the action with highest value (with $p$=1-$\varepsilon$). Based on past experiences the loss is calculated, and weights of the network are updated via error back-propagation according to an optimizer function. The updated weights can be conceptualized as the agent learning to pay attention to the features of the environment most useful for approximating the true Q-values (Mnih et al., 2015). The DQN requires four hyperparameters; $\gamma \in (0,1)$ of the Bellman equation (Eq. 1), determining the depreciation of future relative to immediate rewards, $\varepsilon \in (0,1)$, determining the extent to which the agent exploit current knowledge or explore new options, a decay rate of $\varepsilon$, $\varepsilon_{decay} \in (0,1)$, and a batch size, $b$, for experience replay.

*3.2. Transfer learning in DQN*

Deep Q-learning algorithms have shown remarkable results when applied to RL tasks with complex and high-dimensional state spaces. Mnih and colleagues (2015) demonstrated that the same DQN performed significantly better than all previously proposed algorithms, as well as professional human game-testers, on 49 Atari games. Further, the results clearly showed that the agent learned an optimal strategy on several of the games, such as first removing all blocks on one side of the screen in the Breakout game. However, the speed of learning remains a salient limitation of these models. Due to the large parameter space of DQNs (input dimensions x neurons x 2 (weight and bias)), convergence on parameter values is a long process, and increasingly so with increasing environment complexity. One solution to this problem is the use of transfer learning. Transfer learning refers to the process of transferring knowledge learned in one task domain to another task domain. In DQN's, this is typically done by transferring pre-trained weights from a DQN trained in one environment to a DQN learning another environment. These are then updated through training on the new task. Conceptually this represents the ability to use information learned on another task to guide behaviour on a new task. Scholars have demonstrated how transfer learning can reduce the need for equal data distributions in training and test data, reduce the need for large training datasets, and improve performance in domains where data may update regularly (Pan and Yang, 2009). Asawa and colleagues (2017) implemented transfer learning from the task PuckWorld to Snake and found that this model both improved performance and reduced oscillations in performance, compared to a model with no prior knowledge. Intuitively, the successfulness of transfer learning depends on the extent to which the two tasks are similar, and hence, how relevant information from

one is for the other. An interesting line of research, however, is the investigation of how similar two tasks need be in order to facilitate effective transfer learning. It has for instance been demonstrated that successful transfer learning could be obtained between two very different tasks (CartPole and MountainCar) using sparse coding (Ammar et al., 2012). This work is highly relevant given the great effort in the field of AI to develop so called Artificial General Intelligence (AGI), where AI systems can learn efficiently and act appropriately in a broad domain of tasks.

The current study aimed to assess the performance of a DQN algorithm on two different OpenAI gym task environments, Cartpole and Lunar Lander, and to investigate the extent to which transfer learning from the Cartpole game can improve performance on the Lunar Lander task. While the tasks are conceptually very different, good performance on both tasks is dependent on some sort of physical balancing of a virtual agent. Success of transfer learning was quantified both by whether it improved the agent's score and speed of learning. Additionally, evidence suggests that transfer of weights in the earlier layers of the network may be more efficient, than of later layers, since later layers tend to capture more task-specific features of the environment (Asawa et al., 2017). However, this project gave as input to the network the pixels of the game image, while the current project employ position-vectors of the agent. Hence, transfer was tested both in early and later layers in order to establish whether a similar difference can be found with this type of input.

## 4.0. Methods

### *4.1. Learning environments*

The two task environments were obtained from the OpenAI gym. In the Cartpole task (Figure 1B), the agent is to balance a vertical pole fixed to a cart by one joint. The cart can move horizontally along a frictionless track, and the agent can move the cart to the left or right by applying a force of +1 or -1. The goal of the task is to prevent the pole from falling over. The agent receives a reward of +1 for every time step the pole remains upright. The reward of the agent can range from $[-\infty, \infty]$. The task is considered solved when the agent has an average reward of 195 over 100 consecutive trials. An episode ends if the pole falls more than 15 degrees from vertical, or if the cart is moved more than 2.4 units away from the centre. The Cartpole game has a continuous observation space of 4, representing the position of the cart, velocity of the cart, angle of the pole, and rotation rate of the pole. The task has a discrete action space of 2, representing a move of one unit to the right or left.

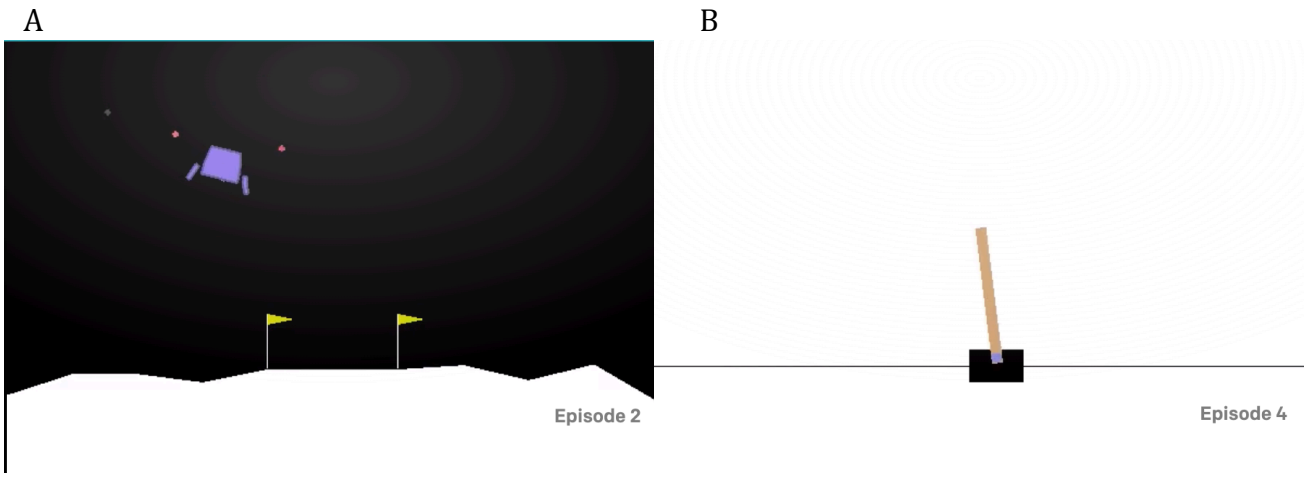


*Figure (1).* Game environments from OpenAI gym. A: Lunar Lander. B: Cartpole.

In the Lunar Lander game (Figure 1A) the agent is controlling a virtual spacecraft with the goal of landing it safely on a landing pad always positioned at coordinates (0,0). The spacecraft has two legs on which it must land. The agent controls the spacecraft by firing either the left, right, or main engine. Landing safely on the landing pad yields a reward between 100 and 140, depending on the precision with which it is on the centre of the pad. Fuel capacity is infinite. Using the main engine gives a reward of -0.3 per state. A reward of +10 is provided for each time the craft has leg contact with the landing pad. An episode finishes when the craft crashes or lands on the pad, yielding a reward of -100 or 100, respectively. The task is considered solved when the agent has an average reward of 200 over 100 consecutive trials. The Lunar Lander game has a continuous observation space of 8, representing the horizontal and vertical position, the horizontal and vertical velocity, angle and angular velocity, and left and right leg contact. The environment has a discrete action space of 4, referring to no action, fire left engine, fire right engine, and fire main engine.

*4.2. Network architecture and Q-learning hyperparameters*

The DQN was built in *Keras* (Chollet et al., 2015) using the Tensorflow backend. Code was adapted from Shiva Verma (2020). Two types of network architectures were constructed. The first network had two hidden fully-connected layers with 150 and 120 neural units, respectively. The dimensions of the input layer was equal to the state space of the task environment, while the number of neural units in the output layer was equal to the action space. The second network had an additional hidden fully-connected layer composed of 120 neural units. In both networks the *ReLU*

activation function was applied to all hidden layers, which outputs the weighted sum if positive and otherwise negative. For the output layers, a *linear* activation function was applied, allowing the output Q-value estimate to take any real value. The linear function solves the regression problem of estimating the Q-value for each state-action pair (Gulli and Pair, 2017). The MSE loss function was applied to both networks, and weights were updated using the *adam* optimizer. Adam is a learning algorithm based on stochastic gradient descent (SGD). SGD seeks to find the global minimum of the cost function, i.e., the set of weights and biases that makes the cost as small as possible. This is obtained by calculating the partial derivative of each parameter, indicating the local shape of the cost function. The partial derivative represents how a small change to each parameter affects performance, and the algorithm then changes the weights and biases according to the effect on the error. This process is iteratively performed until reaching the minimum of the cost function (Kriegeskorte, 2015). The hyperparameter, *a,* is a learning rate determining the size of the steps taken down the slope of the cost function in each iteration. The SGD algorithm relies on error back-propagation working back through all layers to calculate the partial derivatives of each parameter (Kriegeskorte and Golan, 2019). The algorithm is stochastic as it computes an *estimate* of the gradient, averaging over the partial derivatives on a randomly sampled mini-batch of training input. The *adam* optimizer extends the SGD by applying an adaptive learning rate which can adapt separately for each parameter over the course of learning (Zeiler, 2012).

The networks were trained on 1 epoch and a mini-batch size of 32. Hyperparameters of the DQN was set to: $\gamma$=0.99, $\varepsilon$=1, $\varepsilon_{decay}$=0.996 ($\varepsilon_{min}$=0.01), $b$=*64,* and $a$=0.01.

*4.3. Training conditions*

Performance of the DQN on each of the two task environments were compared to performance of an agent following a random policy. In the random policy condition the agent randomly chose one of the four possible actions in each state, received a reward, and moved to the next state. Hence, this agent was expected to perform at chance level.

Transfer learning from the Cartpole to the Lunar Lander game was investigated in four conditions: 1) a DQN with two hidden layers where the second layer was retrained, 2) a DQN with three hidden layers where the second and third layer were retrained, 3) a DQN with three hidden layers where the second layer was retrained, and the third layer reinitialized, and 4) a DQN with three hidden

layers where the second layer was reinitialized and the third layer retrained. In all four conditions the first layer and the output layer were reinitialized. Retraining layers refers to transferring weights from the Cartpole game and initialize new training with these. Reinitializing refers to randomly setting initial weights of the network (Asawa et al., 2017). For all conditions, an identical model was run on the Cartpole game, the Lunar Lander game with no transfer (bDQN), and the Lunar Lander with transfer (tDQN). All models were run 8 times in each condition for 200 episodes. The Cartpole game was run with a maximum step size of 1000 per episode, while the Lunar Lander task was run with a maximum step size of 3000.

## 5.0. Results

### *5.1. General performance of the DQN*

A deep Q-learning model with both two hidden layers (2D-DQN) and three hidden layers (3D-DQN) increased performance on both the Cartpole and the Lunar Lander game compared to a random policy (Figure 2). Hence, a DQN agent seemed to learn valuable information about the task environments facilitating better-than-chance performance. Further, results showed an increase in performance over time in both tasks, suggesting that the DQN learned from experience in an appropriate manner. The agent appeared to learn faster in the Cartpole game compared to Lunar Lander. In the former case, convergence on a strategy periodically yielding the threshold reward (195) occured in about 50 episodes, while on the latter, convergence on a strategy periodically yielding the threshold reward (200), occurred after 100+ episodes. In both conditions however, the agent seemed to converge on a sub-optimal strategy, exhibiting large oscillations in performance over the remaining (~200) episodes. The 2D-DQN performed superior to the 3D-DQN on the Cartpole task (Figure 3). For the Lunar Lander game, the 3D-DQN exhibited better mean performance across all training episodes (Figure 3A), while the 2D-DQN exhibited higher average performance on the last 100 episodes (Figure 3B).

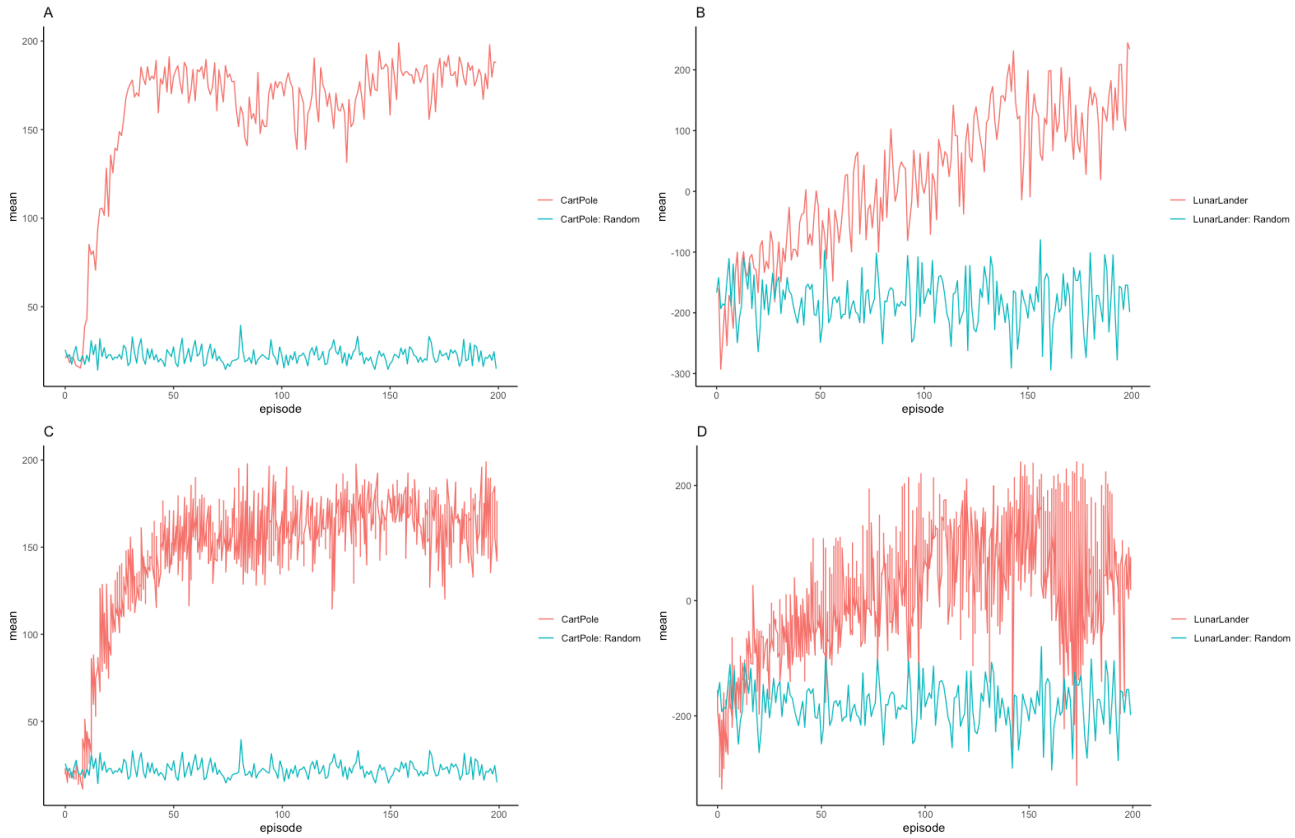


*Figure (2).* Comparison of DQN performance to random policy agent. x-axis = episodes. y-axis = average reward across 8 simulations. A: 2D-DQN on Cartpole game. B: 2D-DQN on Lunar Lander game. C: 3D-DQN on Cartpole game. D: 3D-DQN on Lunar Lander game. Red line in A reflect mean of 8 simulations. Red line on B, C, & D reflect mean of 24 simulations.

## *2. Transfer learning*

### *5.2.1. Condition 1: Transfer in the second of two hidden layers*

The first condition applied a 2D-DQN, in which the first layer was reinitialized while the second was retrained on the Lunar Lander game from weights trained on the Cartpole game. Results indicated a slight improvement in average performance across the 8 simulations in the transfer compared to no-transfer condition. Generally, the tDQN achieved higher scores across the 200 episodes than the bDQN (Figure 3A). The tDQN did not reduce oscillations in performance (Figure 4A). Both the tDQN and bDQN were able to solve the task within 200 episodes (Figure 6A).

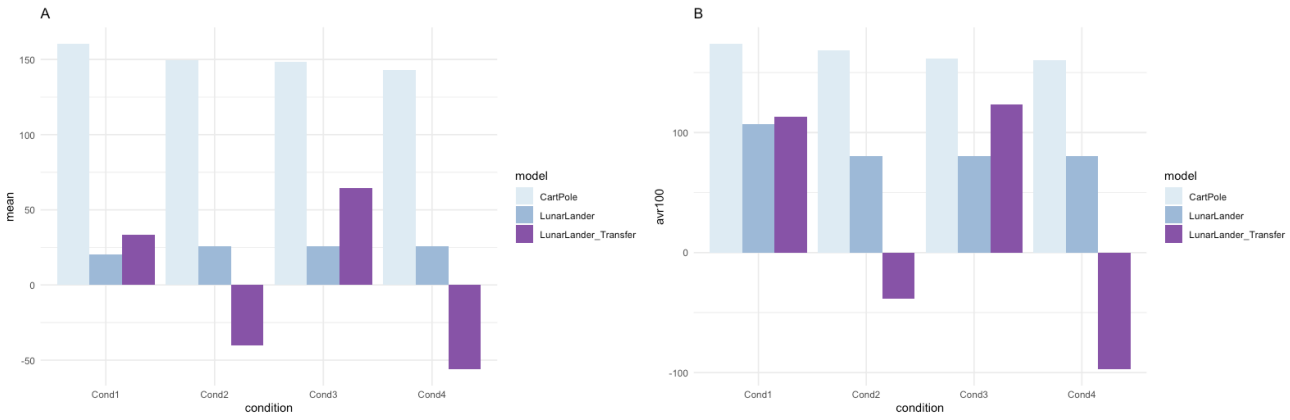


*Figure (3).* A: Mean reward across all 200 episodes (averaged across 8 simulations). B: Average reward on the last 100 episodes (averaged across 8 simulations).

### *5.2.2. Condition 2: Transfer in the second and third of three hidden layers*

The second condition applied a 3D-DQN, where weights were transferred from the Cartpole game to the Lunar Lander task in both the second and third layer. Results indicated that the tDQN reduced performance compared to the bDQN. (Figure 3). While performance of the two models were highly similar in early episodes (~100), the bDQN consistently achieved larger rewards on the remaining training episodes (Figure 4B). As evident from Figure 5B the best performance on the last 100 episodes was achieved by the tDQN, however, this performance was largely inconsistent, as it also produced poorest performance. The tDQN was able to solve the task in less than 200 episodes in two of eight cases, while the bDQN was able to solve it once (Figure 6B). Hence, while transfer learning periodically enhanced performance, results suggest no general positive effect of transfer learning in this condition.

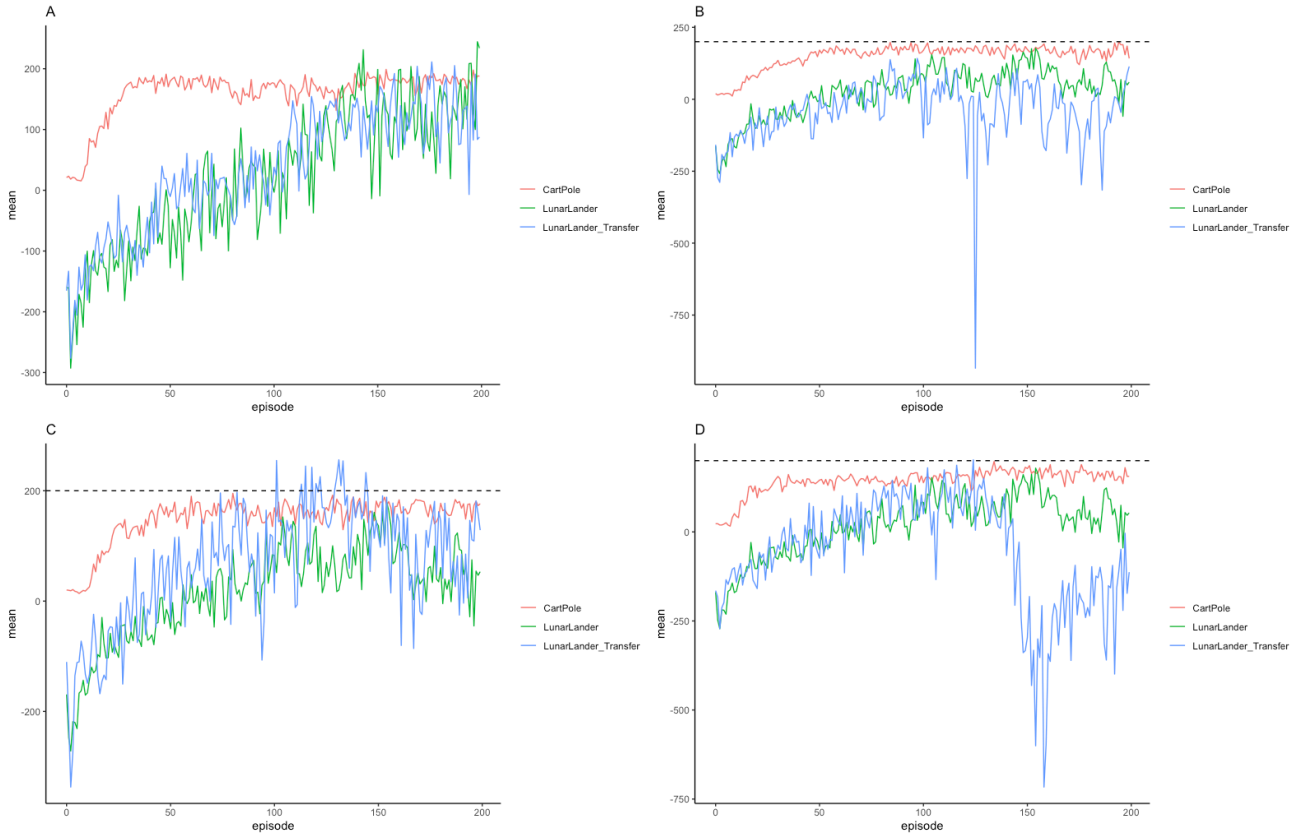


*Figure (4).* Reward over training episodes. x-axis = episode. y-axis = mean reward across 8 simulations. Dotted line = threshold for solved task. A: condition 1. B: condition 2. C: condition 3. D: condition 4.

### *5.2.3. Condition 3: Transfer in the second of three hidden layers*

Condition 3 applied a 3D-DQN, where weights were retrained in only the second hidden layer, while the first and third layer were reinitialized. In general, the tDQN improved overall performance compared to the bDQN. (Figure 3). Specifically, the tDQN agent exhibited a slightly steeper learning curve in beginning of training (~100 episodes), and was, to some extent, able to maintain it's supremacy (Figure 4C). While the two models maintained equivalent stability in performance in the first 150 episodes, stability in performance of the tDQN decreased drastically in the last 50 episodes, in which it achieved both the best and worst performance of the two models (Figure 5C). Hence, the tDQN model increased oscillations in performance slightly, while maintaining the best overall performance. Similar to the bDQN, the tDQN was able to solve the task within 200 training episodes once (Figure 6C).

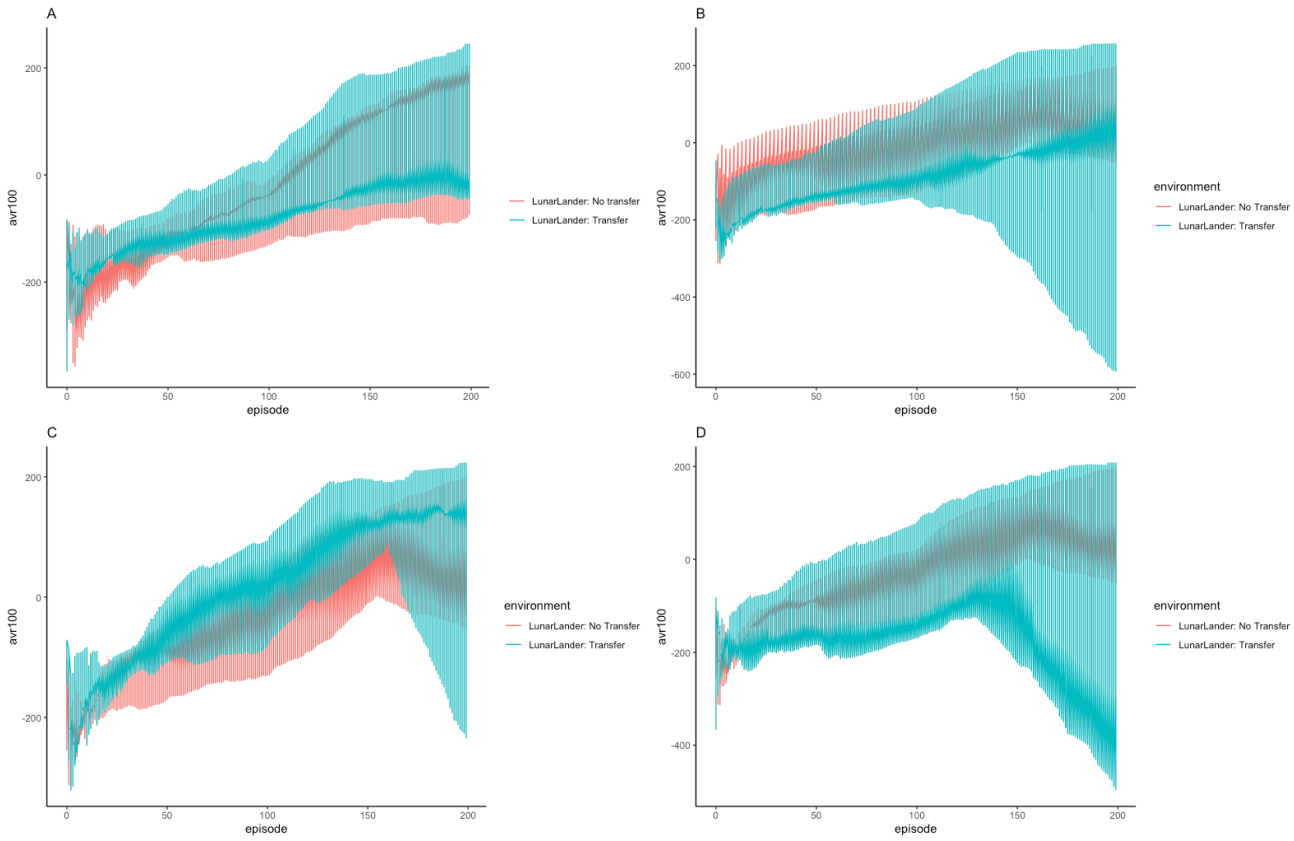


*Figure (5). Average score over last 100 episodes (if episode>100, then average over ~N episodes). x-axis = episode. y-axis = mean of the average score of last 100 episodes across 8 simulations. A: condition 1, B: condition 2, C: condition 3, D: condition 4.*

### *5.2.4. Condition 4: Transfer in the third of three hidden layers*

Condition 4 applied a 3D-DQN, of which the third layer was retrained, while the first and second layer were reinitialized. The tDQN did not improve overall performance on the Lunar Lander task, compared to a bDQN (Figure 3). While the tDQN appeared to perform equivalently to the bDQN through most episodes (~140 episodes), performance dropped drastically by the end of training (Figure 4D). Hence, the tDQN exhibited lower average performance both across all training episodes as well as by the end of training. It is evident that the tDQN exhibited larger oscillations in performance than the bDQN throughout training (Figure 5D). The best performing agents generally received higher rewards in the transfer condition compared to no transfer, however, similar to the bDQN, the tDQN was able to solve the task within 200 training episodes once (Figure 6D).

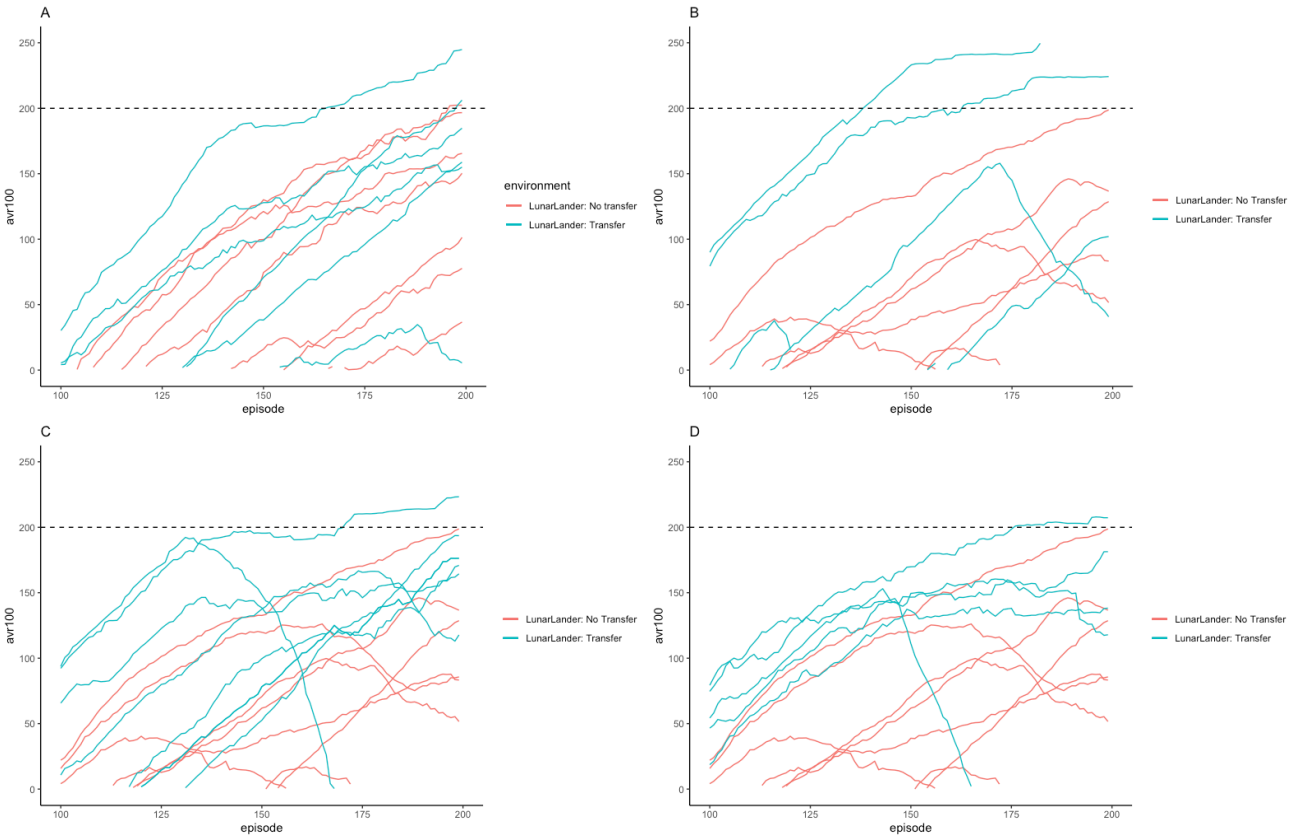


*Figure (6).* Average reward on last 100 episodes for each individual simulation run. x-axis = episode 100-200. y-axis = average reward on last 100 episodes. Dotted line: threshold for task solved (=200). A: condition 1. B: condition 2. C: condition 3. D: condition 4.

## 6.0. Discussion

The current study aimed to investigate the extent to which deep Q-learning agents can benefit from experience in the Cartpole game when learning the Lunar Lander game. General performance of the DQN on the two tasks was compared to that of an agent following a random policy. Transfer learning from Cartpole to Lunar Lander environment was investigated in four conditions under varying network architectures and experience transfer.

Results showed that a DQN agent was able to successfully learn from its experience and achieve good performance on both the Cartpole and Lunar Lander game. A 2D-DQN performed better than a 3D-DQN on the Cartpole game, suggesting that the more complex model may have overfitted the training data. Interestingly, for the Lunar Lander game a 3D-DQN achieved better average

performance across all training episodes, while a 2D-DQN exhibited superior average performance on the last 100 episodes. This indicates that while the 3D-DQN achieved the largest rewards of the two models, performance was highly unstable. Further, this could suggest that the 3D-DQN learned faster than the 2D-DQN, while the 2D-DQN was better able to maintain high performance by the end of training. This seems to be supported in Figure 4, where the 3D-DQN achieves a consistent average score above 0 in about 80 episodes, while the 2D-DQN do so in about 130 episodes. These results suggest that the 3D-DQN may have overfitted to the Lunar Lander environment by adding too many parameters, allowing the network to represent more detailed examples of the environment. Note that direct comparison of performance on the Lunar Lander game between the D2-DQN and D3-DQN should be done with caution, as visualizations of the former is based on 8 simulations, while the latter is based on 24 simulations (8 for each of the three D3-DQN conditions).

The implementation of transfer learning improved performance on the Lunar Lander game in two of four conditions. While applying pre-trained weights in the second hidden layer of a 2D-DQN improved performance (condition 1), applying the same procedure in the second and third layer of a 3D-DQN greatly impaired performance (condition 2). This suggest that transferring weights from the last hidden layer may only be beneficial in more simple models, where less detailed information about the environment is captured in the source model (trained on Cartpole). Results additionally showed a positive effect of transfer learning on performance in a 3D-DQN where weights were transferred in the second of three hidden layers. In fact, this model exhibited the best performance on the Lunar Lander tasks of all tested models. These results support evidence from Asawa et al. (2017) suggesting that earlier layers may contain more general information about the environment more likely to generalize to other domains. Further, this is not a general effect of using pre-trained weights in just one of three hidden layers, since the 3D-DQN with transfer in the third layer performed poorest of all models. Hence, it appears that information stored in the last layer of the source model may be more task-specific and not readily generalizable from the Cartpole to the Lunar Lander game.

In general, the current results suggest that transfer learning might be most effective when preceding a reinitialized layer. While performance on the Lunar Lander game decreased when adding a third layer to the DQN compared to a 2D-DQN, the use of transfer learning in the second layer mitigated

this effect and exhibited performance superior to all other models. Further, of the two conditions in which transfer learning had a positive effect, the effect was larger when adding a reinitialized layer after the transfer layer. Hence, while a more complex model in itself reduced performance compared to a 2D-DQN, it improved performance when using transfer learning. This could indicate that transfer learning may reduce overfitting of more complex DQNs. By encompassing information from two different environments, the weights of the network might be more generalizable to different states of the Lunar Lander environment. However, while exhibiting slightly smaller transfer effects, the simpler model showed more stable performance than the successful 3D-tDQN. Importantly, the results reported here might be highly task specific, and future research is needed to establish the extent to which these results generalize to other game environments.

### *5.1. Limitations and improvements*

The current study has several limitations on which interpretations of results should be based. First, performance of each DQN model is averaged over 8 simulations, which is far from enough to establish general behaviour, given the amount of randomness in the models resulting from random weight initialization and the $\varepsilon$-parameter of Q-learning. More simulations should be run to average over the random noise in the models and decrease performance variation. However, due to limited processing power, this requirement could currently not be satisfied. Second, DQN-agents were trained on only 200 episodes, which didn't seem to be enough time to converge on an optimal strategy. While exhibiting good performance, most agents did not solve the task. Hence, including more training episodes could potentially reveal interesting differences between transfer- and no-transfer DQNs, regarding how quickly, and how often, they converge on the optimal strategy. Third, the implementation of transfer learning in the current study did not include retraining weights in the first hidden layer. This is problematic if early layers are indeed the most generalizable, and should be addressed in further work. Fourth, in line with the notion above, the fact that the models did not tend to converge on an optimal strategy for the Cartpole game, prior to the weights being transferred to train on the Lunar Lander game, may have impaired a potentially larger effect of transfer learning. That is, the weights being transferred may still constitute counter-productive information, which could potentially make learning more difficult in the new environment. Ideally, weights should be transferred only from models, which have clearly converged on an optimal policy in the source environment. Fifth, performance was not compared between models with different hyperparameters of neither the Q-learning algorithm nor the neural network, and no conclusions can

be drawn regarding the effect of hyperparameters on transfer learning. Lastly, the DQN models employed here did not rely on a target network. In Q-learning the value of executing an action in current state is computed from the values of performing actions in successive states, hence the values on both side of the update equation (Eq. 1) are updated simultaneously (Kim et al., 2019). This often causes great instability in results as it conceptually implies that the goal changes every time the agent performs an action. Target networks are typically implemented in DQNs to reduce this sort of correlation between action values and target values (Mnih et al., 2015). Essentially, target networks are a direct copy of the action values, however, rather than being updated in each time step, it is adjusted according to the action values after *C* time steps, where *C* is a hyperparameter of the model (Mnih et al., 2015). The large oscillations observed in the current study may in part result from the lack of a target network. Kim and colleagues (2019) have however problematized the use of target networks, arguing that they don't resemble real online reinforcement learning by delaying the updating of value functions, and that they reduce learning speed and add complexity to the models by doubling the required memory capacity of the model. Hence, target networks present a trade-off between stable performance and fast, "human-like" learning. Alternatively, Kim and his colleagues recently proposed DeepMellow, a softmax operator (using Mellowmax) that can be incorporated into the Bellman equation replacing the max operator. They showed that a DeepMellow model both increased stability and learning speed compared to a DQN using a target network.

A possible improvement to the current models is the use of a differential learning rate. For instance, von Csefalvay (2019) suggested applying a lower learning rate on early transfer layers, and a higher learning rate on the later reinitialized layers. This way weights in the reinitialized layer will be more influenced by the new environment, than will weights in earlier layers, facilitating faster learning from new task-specific information, while generalizable experience from the source task is maintained and less influenced by the new environment. Further, the models may improve by inputting pixels of the game image to the model. This procedure would allow the implementation of convolutional layers in the network, which uses kernels (or feature maps) to extract increasingly complex visual features of the image. While such a model would have a much larger parameter-space, it could potentially facilitate transfer learning to a greater extent, as input dimensions could be kept constant across game environments. Such deep Q-learning convolutional neural networks (CNN) might thus be more generalizable across tasks. However, the improvement of such a model

compared to the DQNs implemented here, is likely to be dependent on the extent to which the two tasks are *visually* similar in contrast to similarity in more physical state properties as used here. That is, if two tasks have very different colour schemes, a deep Q CNN could potentially experience problems with transfer learning.

In conclusion, transfer learning with DQN from the Cartpole game was able to improve performance on the Lunar Lander game under certain circumstances, in particular when preceding a reinitialized fully-connected layer. Contrary, when applied to the last hidden layers of the network, performance decreased drastically. This suggests that early layers of the DQN may contain more generalizable information, while later layers represent more task-specific features of the environment. Further, successful transfer learning indicated the potential to reduce over-fitting of complex models.

## References

Ammar, H. B., Tuyls, K., Taylor, M. E., Driessens, K., & Weiss, G. (2012, June). Reinforcement learning transfer via sparse coding. In *Proceedings of the 11th international conference on autonomous agents and multiagent systems* (Vol. 1, pp. 383-390). International Foundation for Autonomous Agents and Multiagent Systems Richland, SC.

Asawa, C., Elamri, C., & Pan, D. (2017) Using Transfer Learning Between Games to Improve Deep Reinforcement Learning Performance and Stability.

Bellman, R. (1957). A Markovian decision process. *Journal of mathematics and mechanics*, 679-684.

von Csefalvay, C. (2019, March 17). Transfer learning: the do's and don'ts [blog post]. Retrieved from: https://medium.com/starschema-blog/transfer-learning-the-dos-and-donts-165729d66625

Chollet, F., & others. (2015). Keras. GitHub. Retrieved from https://github.com/fchollet/keras

Gulli, A., & Pal, S. (2017). *Deep learning with Keras*. Packt Publishing Ltd.

Kim, S., Asadi, K., Littman, M., & Konidaris, G. (2019, August). Deepmellow: removing the need for a target network in deep Q-learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI* (pp. 10-16).

Kriegeskorte, N. (2015). Deep Neural Networks: A New Framework for Modeling Biological Vision and Brain Information Processing. Annual Review of Vision Science, 1, 417-446.

Kriegeskorte, N., & Golan, T. (2019). Neural network models and deep learning. Current Biology, 29, R231-R236.

Mehta, P., & Meyn, S. (2009, December). Q-learning and Pontryagin's minimum principle. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference* (pp. 3598-3605). IEEE.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533.

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). San Francisco, CA, USA:: Determination press.

Pan, S. J., & Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, *22*(10), 1345-1359.

Patel, Y. (2017, July 30). Reinforcement learning w/Keras + OpenAI: DQNs [blog post]. Retrieved from: https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c

Surma, G. (2018, September 26). Cartpole – Introduction to Reinforcement Learning (DQN – Deep Q-learning) [blog post]. Retrieved from: https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288

Verma, S (2020). Github repository: "OpenAIgym". Retrieved from: https://github.com/shivaverma/OpenAIGym

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, S., & Sutton, R. S. (2017). A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*.

Zychlinski, S. (2019, January 9). Qrash Course: Reinforcement learning 101 & Deep Q Networks in 10 minutes [blog post]. Retrieved from: https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677

# Appendix

Content:

1) DQN code for Cartpole environment
2) DQN code for Lunar Lander environment
3) DQN code with transfer learning from Cartpole to Lunar Lander environment

# Code 1

DQN for Cartpole environment

```
# A pole is attached by an un-actuated joint to a cart,
# which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart.
# The pendulum starts upright, and the goal is to prevent it from falling over.
# A reward of +1 is provided for every timestep that the pole remains upright.
# The episode ends when the pole is more than 15 degrees from vertical,
# or the cart moves more than 2.4 units from the center.


import gym
import random
from keras import Sequential
from collections import deque
from keras.layers import Dense
from keras.optimizers import adam
import matplotlib.pyplot as plt
import numpy as np
from numpy import savetxt
import os
from keras.layers import Dense, Dropout, Flatten, Input
from keras.layers import Conv1D, Conv2D,MaxPooling2D
from numpy import array
from keras.models import Model

env = gym.make('CartPole-v0')
env.seed(0)
np.random.seed(0)


class DQN:

    """ Implementation of deep q learning algorithm """

    def __init__(self, action_space, state_space):

        self.action_space = action_space
        self.state_space = state_space
        self.epsilon = 1
        self.gamma = .95
        self.batch_size = 64
        self.epsilon_min = .01
        self.epsilon_decay = .995
        self.lr = 0.001
        self.memory = deque(maxlen=10000)
        self.model = self.build_model()

    X = (env.observation_space.shape[0],64)
    X = np.expand_dims(X, axis=2)

    def build_model(self):
        model = Sequential()
        model.add(Dense(150, input_dim=self.state_space, activation='relu'))
        model.add(Dense(120, activation='relu'))
        model.add(Dense(120, activation='relu'))
        model.add(Dense(self.action_space, activation='linear'))
        model.compile(loss='mse', optimizer=adam(lr=self.lr))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):

        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_space)
        act_values = self.model.predict(state)
```

```
        return np.argmax(act_values[0])

    def replay(self):

        if len(self.memory) < self.batch_size:
            return

        minibatch = random.sample(self.memory, self.batch_size)
        states = np.array([i[0] for i in minibatch])
        actions = np.array([i[1] for i in minibatch])
        rewards = np.array([i[2] for i in minibatch])
        next_states = np.array([i[3] for i in minibatch])
        dones = np.array([i[4] for i in minibatch])

        states = np.squeeze(states)
        next_states = np.squeeze(next_states)

        targets = rewards + self.gamma*(np.amax(self.model.predict_on_batch(next_states), axis=1))*(1-dones)
        targets_full = self.model.predict_on_batch(states)

        ind = np.array([i for i in range(self.batch_size)])
        targets_full[[ind], [actions]] = targets

        self.model.fit(states, targets_full, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay


def train_dqn(episode):

    loss = []
    episodes = []
    agent = DQN(env.action_space.n, env.observation_space.shape[0])
    for e in range(episode):
        episode = e
        state = env.reset()
        state = np.reshape(state, (1, 4))
        score = 0
        max_steps = 1000
        for i in range(max_steps):
            env.render()
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            score += reward
            next_state = np.reshape(next_state, (1, 4))
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            agent.replay()
            if done:
                print("episode: {}/{}, score: {}".format(e, episode, score))
                break
        loss.append(score)
        episodes.append(episode)

    #Save csv file with loss per episode
    os.chdir('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning')
    data = np.column_stack((loss, episodes))
    savetxt('lossCartPole_8_7.csv', data, delimiter=',')

    # serialize model to JSON
    model = agent.model
    model_json = model.to_json()
    os.chdir('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning')
    with open("modelCartPole.json", "w") as json_file:
        json_file.write(model_json)
    # serialize weights to HDF5
```

```
    model.save_weights("modelCartPole.h5")
    print("Saved model to disk")

    return loss


def random_policy(episode, step):
    loss = []
    episodes = []
    for i_episode in range(episode):
        env.reset()
        episode = i_episode
        score = 0
        print(i_episode)
        for t in range(step):
            env.render()
            action = env.action_space.sample()
            state, reward, done, info = env.step(action)
            score += reward
            if done:
                #print("Episode finished after {} timesteps".format(t+1))
                break
            #print("Starting next episode")
        loss.append(score)
        episodes.append(episode)

    os.chdir('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning')
    data = np.column_stack((loss, episodes))
    savetxt('lossCartPole_7_1.csv', data, delimiter=',')


if __name__ == '__main__':

    episode = 200
    loss = train_dqn(episode) #run DQN model
    #loss = random_policy(episode, 1000) #run random policy
    plt.plot([i+1 for i in range(0, episode, 2)], loss[::2])
    plt.show()

import Code2pdf
from Code2pdf.code2pdf import Code2pdf
ifile,ofile,size = "test.py", "test.pdf", "A4"
pdf = Code2pdf(ifile, ofile, size)  # create the Code2pdf object
pdf.init_print()    # call print method to print pdf
```

# Code 2

DQN for Lunar Lander environment

```
# Landing pad is always at coordinates (0,0). Coordinates are the first
# two numbers in state vector. Reward for moving from the top of the screen
# to landing pad and zero speed is about 100..140 points. If lander moves
# away from landing pad it loses reward back. Episode finishes if the lander
# crashes or comes to rest, receiving additional -100 or +100 points.
# Each leg ground contact is +10. Firing main engine is -0.3 points each frame.
# Solved is 200 points. Landing outside landing pad is possible. Fuel is
# infinite, so an agent can learn to fly and then land on its first attempt.
# Four discrete actions available: do nothing, fire left orientation engine,
# fire main engine, fire right orientation engine.


import gym
import random
from keras import Sequential
from collections import deque
from keras.layers import Dense, Input
from keras.optimizers import adam
import matplotlib.pyplot as plt
from keras.activations import relu, linear
import os
from keras.models import model_from_json, Model
from numpy import savetxt

import numpy as np
env = gym.make('LunarLander-v2')
env.seed(0)
np.random.seed(0)


class DQN:

    """ Implementation of deep q learning algorithm """

    def __init__(self, action_space, state_space):

        self.action_space = action_space
        self.state_space = state_space
        self.epsilon = 1.0
        self.gamma = .99
        self.batch_size = 64
        self.epsilon_min = .01
        self.lr = 0.001
        self.epsilon_decay = .996
        self.memory = deque(maxlen=1000000)
        self.model = self.build_model()

    def build_model(self):

        model = Sequential()
        model.add(Dense(150, input_dim=self.state_space, activation='relu'))
        model.add(Dense(120, activation='relu'))
        model.add(Dense(120, activation='relu'))
        model.add(Dense(self.action_space, activation='linear'))
        model.compile(loss='mse', optimizer=adam(lr=self.lr))
        return model



    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):

        if np.random.rand() <= self.epsilon:
```

```
            return random.randrange(self.action_space)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])

    def replay(self):

        if len(self.memory) < self.batch_size:
            return

        minibatch = random.sample(self.memory, self.batch_size)
        states = np.array([i[0] for i in minibatch])
        actions = np.array([i[1] for i in minibatch])
        rewards = np.array([i[2] for i in minibatch])
        next_states = np.array([i[3] for i in minibatch])
        dones = np.array([i[4] for i in minibatch])

        states = np.squeeze(states)
        next_states = np.squeeze(next_states)

        targets = rewards + self.gamma*(np.amax(self.model.predict_on_batch(next_states), axis=1))*(1-dones)
        targets_full = self.model.predict_on_batch(states)
        ind = np.array([i for i in range(self.batch_size)])
        targets_full[[ind], [actions]] = targets

        self.model.fit(states, targets_full, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay


def train_dqn(episode):

    loss = []
    episodes = []
    agent = DQN(env.action_space.n, env.observation_space.shape[0])
    for e in range(episode):
        episode = e
        state = env.reset()
        state = np.reshape(state, (1, 8))
        score = 0
        max_steps = 3000
        for i in range(max_steps):
            action = agent.act(state)
            env.render()
            next_state, reward, done, _ = env.step(action)
            score += reward
            next_state = np.reshape(next_state, (1, 8))
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            agent.replay()
            if done:
                print("episode: {}/{}, score: {}".format(e, episode, score))
                break
        loss.append(score)
        episodes.append(episode)

        # Average score of last 100 episode
        is_solved = np.mean(loss[-100:])
        if is_solved > 200:
            print('\n Task Completed! \n')
            break
        print("Average over last 100 episode: {0:.2f} \n".format(is_solved))

    #Save csv file with loss per episode
    os.chdir('/Dropbox/ExamProject - DQN Transfer Learning/')
    data = np.column_stack((loss, episodes))
    savetxt('lossLunarLander_7_8.csv', data, delimiter=',')
```

```
133
134     return loss
135
136 def random_policy(episode, step):
137     loss = []
138     episodes = []
139     for i_episode in range(episode):
140         env.reset()
141         episode = i_episode
142         score = 0
143         print(episode)
144         for t in range(step):
145             env.render()
146             action = env.action_space.sample()
147             state, reward, done, info = env.step(action)
148             score += reward
149             if done:
150                 #print("Episode finished after {} timesteps".format(t+1))
151                 break
152             #print("Starting next episode")
153         loss.append(score)
154         episodes.append(episode)
155
156     os.chdir('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning')
157     data = np.column_stack((loss, episodes))
158     savetxt('lossLunarLander_7_1.csv', data, delimiter=',')
159
160
161 if __name__ == '__main__':
162
163     print(env.observation_space)
164     print(env.action_space)
165     episodes = 200
166     loss = train_dqn(episodes)
167     #loss = random_policy(episodes, 1000)
168     plt.plot([i+1 for i in range(0, len(loss), 2)], loss[::2])
        plt.show()
```

# Code 3

DQN for transfer learning from Cartpole to Lunar Lander environment

```
# Landing pad is always at coordinates (0,0). Coordinates are the first
# two numbers in state vector. Reward for moving from the top of the screen
# to landing pad and zero speed is about 100..140 points. If lander moves
# away from landing pad it loses reward back. Episode finishes if the lander
# crashes or comes to rest, receiving additional -100 or +100 points.
# Each leg ground contact is +10. Firing main engine is -0.3 points each frame.
# Solved is 200 points. Landing outside landing pad is possible. Fuel is
# infinite, so an agent can learn to fly and then land on its first attempt.
# Four discrete actions available: do nothing, fire left orientation engine,
# fire main engine, fire right orientation engine.


import gym
import random
from keras import Sequential
from collections import deque
from keras.layers import Dense, Input
from keras.optimizers import adam
import matplotlib.pyplot as plt
from keras.activations import relu, linear
import os
from keras.models import model_from_json, Model
from numpy import savetxt

import numpy as np
env = gym.make('LunarLander-v2')
env.seed(0)
np.random.seed(0)


class DQN:

    """ Implementation of deep q learning algorithm """

    def __init__(self, action_space, state_space):

        self.action_space = action_space
        self.state_space = state_space
        self.epsilon = 1.0
        self.gamma = .99
        self.batch_size = 64
        self.epsilon_min = .01
        self.lr = 0.001
        self.epsilon_decay = .996
        self.memory = deque(maxlen=1000000)
        self.model = self.build_model()

    def build_model(self):

        #Layers with the same name as in the CartPole model will get the transferred weights
        model = Sequential()
        model.add(Dense(150, input_dim=self.state_space, activation=relu, name='newDense1'))
        model.add(Dense(120, activation='relu', name='dense_2'))
        #model.add(Dense(120, activation='relu', name='dense_3'))
        model.add(Dense(self.action_space, activation=linear, name='newDense3'))
        model.load_weights('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning/modelCartPole.h5', by_name=True)
        model.compile(loss='mse', optimizer=adam(lr=self.lr))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):

        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_space)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])

    def replay(self):

        if len(self.memory) < self.batch_size:
            return

        minibatch = random.sample(self.memory, self.batch_size)
        states = np.array([i[0] for i in minibatch])
```

```
        actions = np.array([i[1] for i in minibatch])
        rewards = np.array([i[2] for i in minibatch])
        next_states = np.array([i[3] for i in minibatch])
        dones = np.array([i[4] for i in minibatch])

        states = np.squeeze(states)
        next_states = np.squeeze(next_states)

        targets = rewards + self.gamma*(np.amax(self.model.predict_on_batch(next_states), axis=1))*(1-dones)
        targets_full = self.model.predict_on_batch(states)
        ind = np.array([i for i in range(self.batch_size)])
        targets_full[[ind], [actions]] = targets

        self.model.fit(states, targets_full, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay


def train_dqn(episode):

    loss = []
    episodes = []
    agent = DQN(env.action_space.n, env.observation_space.shape[0])
    for e in range(episode):
        episode = e
        state = env.reset()
        state = np.reshape(state, (1, 8))
        score = 0
        max_steps = 3000
        for i in range(max_steps):
            action = agent.act(state)
            env.render()
            next_state, reward, done, _ = env.step(action)
            score += reward
            next_state = np.reshape(next_state, (1, 8))
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            agent.replay()
            if done:
                print("episode: {}/{}, score: {}".format(e, episode, score))
                break
        loss.append(score)
        episodes.append(episode)

        # Average score of last 100 episode
        is_solved = np.mean(loss[-100:])
        if is_solved > 200:
            print('\n Task Completed! \n')
            break
        print("Average over last 100 episode: {0:.2f} \n".format(is_solved))

    #Save csv file with loss per episode
    os.chdir('/Users/lineelgaard/Documents/Data Science/ExamProject - DQN Transfer Learning')
    data = np.column_stack((loss, episodes))
    savetxt('lossTransfer_CPtoLL_6_2.csv', data, delimiter=',')

    return loss


if __name__ == '__main__':

    print(env.observation_space)
    print(env.action_space)
    episodes = 200
    loss = train_dqn(episodes)
    plt.plot([i+1 for i in range(0, len(loss), 2)], loss[::2])
    plt.show()
```