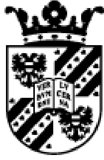

university of groningen / faculty of science and engineering

The Lottery Ticket Hypothesis in Deep Reinforcement Learning

Bachelor's Project Thesis

Lennard R. Bornemann, s3576981, l.r.bornemann@student.rug.nl
Supervisors: Dr. M. Sabatelli

Abstract: Previous research suggests the existence of small sub-networks within convolutional neural networks which perform just as well as their original non-pruned counterpart. Such implication has wide reaching impact, as it suggests that any neural network can produce such sub-networks, known as winning lottery tickets. The existence of such winning tickets in all fields of machine learning is a promising idea, as it could allow for the spacial and computational reduction of any machine learning task. This paper aims to investigate the existence of such winning lottery tickets within the field of deep reinforcement learning, specifically within deep Q-networks.

1 Introduction

The lottery ticket hypothesis (Frankle and Carbin, 2018) encompasses the idea that small sub-networks can be found within large neural networks which can perform on a similar level. This sub-network can be found by randomly initialising a network and proceeding to train and prune it until a considerable number of weights have been removed without impacting the performance, leaving the so called winning ticket of the network. The implications of this hypothesis are promising, as it could lead to an improvement of neural network training performances as such winning tickets might be found quickly, influence better network design through learning from the structures of the winning tickets and possibly being able to transfer a winning ticket mask to other similar networks as well as providing a general better theoretical understanding of neural networks. However, the original paper of Frankle and Carbin (2018) focused on fully connected and convolutional neural networks, which are certainly useful for a variety of tasks, but the utility of the lottery ticket hypothesis might be as important in other branches of artificial intelligence.

One such branch is reinforcement learning, in which an agent learns an optimal behaviour within an environment through repeated rewards and penalties based on its actions. The agent therefore learns by attempting to maximise its cumulative future reward or Q-function. Usually, within value-based reinforcement learning methods, the Q-function is estimated and used to determine an agent's policy, meaning which actions should be taken. Such a Q-function is normally represented as a Q-table consisting of values and their corresponding state and action pairs. However, such value-based methods can run into the problem presented by the curse of dimensionality (Verleysen and François, 2005). This can be overcome by using deep Q-network methods instead, which approximate Q-values normally stored in a Q-table. This will be elaborated upon in the methods section.

Since the Q-table can be approximated by a neural network, it stands to reason that this network should adhere to the lottery ticket hypothesis, meaning that a smaller, equally efficient network should be nested within the larger parent network. The presence of such a winning ticket inside of deep Q-networks, would be promising, as it would help to further solidify the lottery ticket phenomenon as a general feature of neural networks rather than an aspect of only supervised learning.

This paper will attempt to answer whether the lottery ticket hypothesis holds in the field of deep reinforcement learning, specifically within deep SARSA(Zhao, Wang, Shao, and Zhu (2016)) and

deep Q-networks(Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, et al. (2015)), two approaches of using Q-networks.

2 Related Work

Inspiration for this paper was drawn from multiple previous works in similar fields. Yu, Edunov, Tian, and Morcos (2019) wrote about the validity of the lottery ticket hypothesis within reinforcement learning and natural language processing. Their aim was to expand the lottery ticket hypothesis beyond the scope of Frankle and Carbin (2018). They covered natural language processing with the use of a recurrent LSTM (Long short term model), Hochreiter and Schmidhuber (1997), machine translation using transformer models, classic control problems such as the problems tackled in this paper and also ATARI games, Bellemare, Naddaf, Veness, and Bowling (2013), which are popular test environments within machine learning. Their research served to be encouraging, as they were able to conclude that the lottery ticket hypothesis does hold true in the areas they explored.

Similarly, Vischer, Lange, and Sprekeler (2021) investigated the existence of lottery tickets amongst a variety of deep reinforcement learning problems with a main focus on task representation and the underlying mechanisms of the lottery ticket effect, which gave insight into implementing deep reinforcement learning algorithms such as the deep Q-network.

Morcos, Yu, Paganini, and Tian (2019) delve into transfer learning using the lottery ticket hypothesis within convolutional neural networks, similar to those used by Frankle and Carbin (2018). Whilst their focus lies more on transferring lottery tickets to similar data sets, they provided a good insight on different optimisers, which served as inspiration to use the ADAM optimiser, Kingma and Ba (2014), in this paper. Their findings suggest that winning lottery tickets can be transferred across similar datasets and optimisers, which is supported also by the research of Sabatelli, Kestemont, and Geurts (2020) which investigated the transferability of winning lottery tickets obtained from datasets in the natural image domain to datasets in non-natural image domains.

Frankle, Dziugaite, Roy, and Carbin (2019) extended the original work of Frankle and Carbin (2018) by investigating the stability and performance of the lottery ticket hypothesis in deeper networks. The original lottery ticket hypothesis paper by Frankle and Carbin (2018) looked at supervised learning networks such as Lenet and Conv-6, which have up to 6 different layers, whereas Frankle et al. (2019) used Resnet-18 and VGG-19 networks, also supervised learning networks, which respectively have 18 and 19 layers. Their research found that iterative magnitude pruning, which will be used in this paper, sometimes fails to find winning tickets in such deep networks. Deep networks equivalent to theirs will not be used here, but their findings nonetheless influenced the decision for the pruning process used later on.

Further inspiration about the pruning process was drawn from Malach, Yehudai, Shalev-Schwartz, and Shamir (2020), whose paper focuses on the importance of pruning in the lottery ticket hypothesis as it draws comparisons between weight initialisation and weight pruning when trying to find winning tickets.

3 Methods

This paper used `python 3.9` to implement the experiment alongside various helpful libraries. These include `numpy`, `tensorflow` Abadi et al. (2015), `keras` Chollet et al. (2015), `gym` and `csv`. Their uses are explained throughout the methods section.

3.1 Environments

In order to investigate the lottery ticket hypothesis within deep reinforcement learning, the first thing that is needed is an environment which allows the different approaches to be trained and evaluated effectively. For this purpose, the OpenAi Gym (Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, and Zaremba (2016)) is used. The OpenAi Gym is a toolkit filled with a number of tasks specifically designed for developing and evaluating a wide variety of reinforcement learning algorithms. For the purpose of this paper, the `CartPole-v0` and `Acrobot-v1` environments will be used as their action spaces are similarly discrete but their tasks are slightly different from each other.

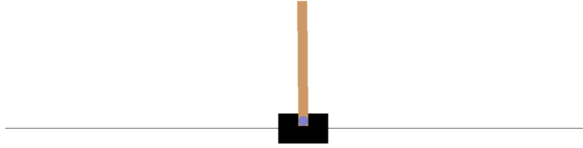

Figure 3.1: The Cartpole-v0 environment.

In the `CartPole-v0` environment, shown in Figure 3.1 the agent has to balance a pole on a moving cart. The pole is attached at its lower end and will tip to either side if the cart is not moved. The state space of this environment includes the position of the cart on track, the angle of the pole, the velocity of the cart and the rate of change of the angle, as described originally by Barto, Sutton, and Anderson (1983). The possible actions in this environment consist of two choices; to move right or to move left. By moving in a correct fashion, the pole will stay balanced on top of the cart. By keeping the pole balanced for a longer amount of time, the reward will be higher, increasing by a reward of 1 per time step, with a max of 200.

The `Acrobot-v1` environment, shown in Figure 3.2 consists of two poles, originally described by Sutton (1995). One is connected to a central point around which it can move. The second pole is attached to the first via a joint and can swing freely. Moving the second pole therefore causes the first to swing, but the first poles movement also impacts the movement of the second. The goal is to swing any part of the two poles above a threshold line located at the top of the environment. The state space of this environment includes the angles of the first and second pole, as well as their velocities. The possible actions consists once again of left or right movement of the first pole, or no movement at all. The difference from the `CartPole-v0` environment however comes from the fact that the poles will sometimes counteract the movement of the other, adding difficulty to the task of reaching the threshold line. The reward of this environment

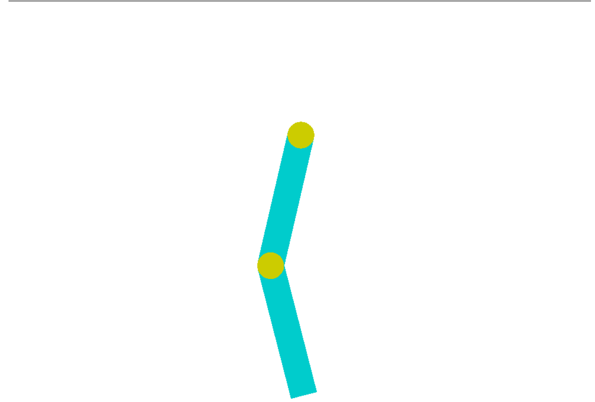

Figure 3.2: The Acrobot-v1 environment.

decreases with each time step it takes by -1 to reach the threshold line with a minimum reward of -500, at which an episode ends. A score closer to 0 is therefore desirable.

3.2 Reinforcement learning

According to Sutton and Barto (2018), the aim of reinforcement learning is to maximise the reward function, given by equation 3.1, of a problem by learning an optimal policy. The reward function returns a scalar value r which is used in later equations. In equation 3.1, s represents a state and a represents an action. Equation 3.2 displays the representation of an environment, consisting of the set of all possible states $\mathcal{S}$, the set of all possible actions $\mathcal{A}$, the transition function $\mathcal{P}$ and the reward r.

$$\Re(s, a, s_{t+1}) \tag{3.1}$$

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle \tag{3.2}$$

A policy, π, can be seen as the strategy an agent uses to reach a goal in an environment, shown in equation 3.2. An optimal policy can be found using different value functions. The first is the state-value function, $V^{\pi}(s)$, shown in equation 3.4, which gives

the utility value of a state as determined by the discounted cumulative reward, see equation 3.3 in which γ is the discount factor.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.3}$$

$$V^\pi(s) = \mathbb{E}[G_t \mid s_t = s, \pi] \tag{3.4}$$

The second value function is the state-action value function, $Q(s, a)$, shown in equation 3.5, gives the utility value of a state-action pair, also determined by the discounted cumulative reward G_t.

$$Q^\pi(s, a) = \mathbb{E}[G_t \mid s_t = s, a_t = a, \pi] \tag{3.5}$$

3.2.1 Dynamic programming

Introduced by Bellman (1952), dynamic programming (DP) refers to a set of algorithms that are able to find optimal policies by using the Bellman equations of value functions 3.4 and 3.5, shown in equation 3.6 and 3.7, given a model of the environment $p(s_{t+1}|s, a)$.

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s_{t+1}} p(s_{t+1}|s, a)[r + \gamma V^\pi(s_{t+1})] \tag{3.6}$$

$$Q^\pi(s, a) = \sum_{s_{t+1}} p(s_{t+1}|s, a) \big[r + \gamma \sum_{a_{t+1}} \pi(a_{t+1}|s_{t+1}) Q^\pi(s_{t+1}, a_{t+1})\big] \tag{3.7}$$

Optimal value functions are maximised versions, returning the highest value given a policy, shown in equations 3.8 and 3.9.

$$V^*(s) = \max_\pi V^\pi(s) \tag{3.8}$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{3.9}$$

Such, the Bellman optimality equations that result are shown in equation 3.10 and 3.11.

$$V^*(s) = \max_a \sum_{s_{t+1}} p(s_{t+1}|s, a)[r + \gamma V^*(s_{t+1})] \tag{3.10}$$

$$Q^*(s, a) = \sum_{s_{t+1}} p(s_{t+1}|s, a)[r + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})] \tag{3.11}$$

Whilst the Bellman optimality equations are able to find the optimal policy mathematically, dynamic programming uses generalised policy iteration to find such an optimal policy. Generalised policy iteration is a combination of policy evaluation and policy improvement steps. Policy evaluation uses equations 3.6 and 3.7 to determine the utility values of the current policy. Policy improvement uses equation 3.12 to compare policies and chose a new policy, π'.

$$\pi' = \arg\max_{a \in A} Q^\pi(s, a) \tag{3.12}$$

3.2.2 Monte Carlo methods

Dynamic programming has limitations in that the environment, equation 3.2, is known. Once parts of the environment are unknown, Monte Carlo (MC) methods can be used to overcome a lack of environment information through gathering experience, which is done by sampling states, actions and rewards from the environment. MC can be used to learn both $V^\pi(s)$ and $Q^\pi(s, a)$. The value of each state in an environment can be updated using the MC update rule shown in equation 3.13, in which $\alpha \in [0, 1]$ represents the learning rate and G_t is the discounted cumulative reward from equation 3.3. Similarly, the value of each state action pair can be updated using equation 3.14.

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{3.13}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[G_t - Q(s_t, a_t)] \tag{3.14}$$

Equations 3.13 and 3.14 follow the current policy, π, until a learning episode terminates, at which point, much like in DP, the policy is evaluated. MC therefore follows the ideas of DP as it cycles through policy evaluation and policy improvement, which is handled in a similar way to equation 3.12.

3.2.3 Temporal difference learning

A core concept of reinforcement learning is that of temporal difference (TD) learning, which combines ideas of Monte Carlo (MC) methods and dynamic programming (DP). TD does not have to wait until the end of an episode to update its value estimates but rather only until the next time step, as at step $t+1$ a target for learning, the TD-target, already exists. The update rule for TD learning is shown in equation 3.15.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (3.15)$$

This equation, 3.15, is similar to the MC update rule in equation 3.13 except that G_t is replaced by the TD-target; an observed reward r_t and a discounted guess of a future state value, $V(s_{t+1})$, as $V^{\pi}(s)$ is unknown. The entire section behind α in equation 3.15 is known as a TD-error and is important for later on and off-policy techniques.

3.2.4 Q-learning

Q-learning is a popular algorithm that builds on TD learning, introduced by Watkins and Dayan (1992). It is an off-policy version of TD learning that is able to converge to the optimal state-action pair value function $Q^*(s,a)$ from equation 3.9. An off-policy algorithm performs independently from an agent's actions, managing to figure out an optimal policy regardless of the agent's behaviour. In the case of Q-learning this is done by creating the TD-error using $\max_{a \in A}$ as this will ignore the current policy to choose the best next action, meaning that a Q-learning agent is always learning in a greedy way. The update rule of Q-learning is shown in equation 3.16.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1},a) - Q(s_t,a_t)] \quad (3.16)$$

3.2.5 SARSA

SARSA, short for state action reward state action, is another popular algorithm that builds on TD learning, introduced by Rummery and Niranjan (1994). It is an on-policy version of TD learning that is usually able to converge to a near optimal state-action pair value function. An on-policy algorithm attempts to evaluate and improve the policy which has led it to the current action. In the case of SARSA this is done by creating a TD-error without using a max value, unlike Q-learning, as well as taking the next action into account. The update rule for SARSA is shown in equation 3.17. It is strikingly similar to equation 3.16, however SARSA learns depending on the next state and action pair which may or may not be a greedy approach.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \quad (3.17)$$

3.3 Deep Reinforcement Learning

3.3.1 Deep Q-networks

Using equation 3.16 an agent can create a large table, known as a Q-table, of information listing which actions in which states would give the highest rewards, however such a table quickly becomes too large to handle for complex problems. This is where deep Q-networks can be used. A deep Q-network adds a neural network to the Q-learning process to replace the Q-table.

A problem with deep Q-learning is that the target, the final reward of the agent, is nonstationary, as it changes with depending on yet unknown Q-values. To counteract this problem, a target network can be used to calculate the target value. This target network should have the same architecture as the function approximator network for the Q-table but its parameters should be frozen. The Q-table network then updates the target network every I iterations, where I is a predetermined hyperparameter. The agent's experiences also need to be saved in order to calculate loss values, the difference between the target and predicted Q-values. Such a marriage of Q-learning and neural networks was introduced as a deep Q-network by Mnih et al. (2015).

$$\mathcal{L}(\theta) = \mathbb{E}_{<s_t,a_t,r_t,s_{t+1}>\sim U(D)}[(r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1},a;\theta^-) - Q(s_t,a_t;\theta))^2] \quad (3.18)$$

The loss function for a DQN neural network, θ, is shown in equation 3.18, in which the expected values for s_t, a_t, r_t and s_{t+1} are sampled from the experience replay buffer, $U(D)$, and used to calculate the loss between the target network, θ^- and the prediction network, θ.

The Q-network that is used here consists of a flattened input layer, two dense layers of 24 nodes and a rectified linear unit activation function and a final dense output layer with 2 nodes, corresponding to the movement options in the environments used and a linear activation function. Details on the hyperparameters can be found in table 3.1.

3.3.2 Deep SARSA-network

Like Q-learning, a SARSA agent would fill a Q-table with relevant information, using equation 3.17, and use it to execute its actions, which presents the same problem of dimensionality as mentioned above, leading to a similar solution of using a Q-network instead of a Q-table. This creates a deep SARSA agent, inspired by Xu, Cao, Chen, Li, Zhang, and Lai (2018) and Zhao et al. (2016).

$$\mathcal{L}(\theta) = \mathbb{E}_{<s_t,a_t,r_t,s_{t+1},a_{t+1}>\sim U(D)}[(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}; \theta) - Q(s_t, a_t; \theta))^2] \quad (3.19)$$

The loss function for a deep SARSA-network, θ, is shown in equation 3.19. Much like in the DQN loss function, the values for s_t, a_t, r_t, s_{t+1} and a_{t+1} are sampled from an experience replay buffer $U(D)$.

The Q-network that is used for this agent is the same as that of the deep Q-network agent, in that it consists of a flattened input layer, two dense layers of 24 nodes and rectified linear unit activation functions and a final output layer with 2 nodes and a linear activation function. Details of the hyperparameters can be found in table 3.1

3.4 Pruning

For the experiment's pruning, an iterative pruning method is used. In iterative pruning, an agent and its corresponding network is fully trained before a number of weights are pruned. The remaining weights are then saved and loaded onto a new agent and network sharing the same architecture as the old version and fully trained again. This process is repeated until a desired sparsity of weights is achieved.

The pruning itself is achieved via a minimum weight pruning method. The paper by Molchanov, Tyree, Karras, Aila, and Kautz (2016) gave extensive insight into the inner workings of pruning neural networks and inspiration for the implementation of this pruning method.

The pseudocode of algorithm 3.1 displays the general implementation of minimum weight pruning. This algorithm sorts the indexes of the weights and biases by their absolute values, before setting the values of entities with indexes within the threshold, to zero. The threshold in this case is calculated by multiplying the number of weights or biases (lengths of index arrays) by the current desired sparsity.

The pseudocode of algorithm 3.2 shows the general process of pruning a model. Here it is important to note that the for loop goes up in steps of 2.

Algorithm 3.1 Minimum weight pruning

Require: $\mathbf{w}$ = :Weight matrix
Require: $\mathbf{b}$ = :Bias matrix
Require: s = Sparsity
  idx = store sorted indexes of $\mathbf{w}$
  num = get number of indexes
  $toP = num*s$
  $idxP$ = store indexes to be pruned ($idx[0 : toP]$)
  $\mathbf{w}[idxP] = 0$ set identified weights to 0
  repeat above steps for $\mathbf{b}$
  return $\mathbf{w}$
  return $\mathbf{b}$

Algorithm 3.2 Prune weights of model x

Require: m :A model
Require: s :A sparsity
  $\mathbf{w}$ = Weights of model m
  $\mathbf{wN} \leftarrow$ [] List of new weights
  for i in range length of $\mathbf{w}$ in steps of 2 **do**
    $wVals, bVals$ = MinWPrune($\mathbf{w}[i], \mathbf{w}[i+1], s$)
    append $wVals$ to $\mathbf{wN}$
    append $bVals$ to $\mathbf{wN}$
  end for
  set weights of model m to $\mathbf{wN}$
  compile model m
  return m

Algorithm 3.3 Reinitialise weights

Require: tm :Target model (to be reinitialised)
Require: bw :Baseline model weights
  tw = weights of target model
  for i in range length of tw in steps of 2 **do**
    x, y = coordinates of non-zero weights in $tw[i]$
    $tw[i][x, y] = bw[i][x, y]$
  end for
  set weights of tm to the new tw
  compile tm
  return tm

This is because the weights of a `keras` model are saved alongside their biases. This becomes apparent when looking at **w**, as the for loop accesses both **w**[i] and **w**[i+1] at the same time. The function MinWPrune refers to algorithm 3.1. Compilation in this case means that the loss function, optimiser and metrics for the model tm are defined.

3.5 Reinitialisation

As described by Frankle and Carbin (2018), in order to find the winning ticket, weights should be re-initialised to their original values after pruning. This can be done by comparing the old and new weights. The new weights will have many which are set to 0, also called a mask. The indexes of this mask can be found and used to copy and replace only the weights which are not set to 0 by the mask, allowing for relatively easy re-initialisation.

The pseudocode of algorithm 3.3 shows the process of reinitialising the weights of a model to those of the original network. In order to be able to do this, it is important to save those original weights, as shown in line 5 of algorithm 3.4.

3.6 Experiment

The pseudocode in algorithm 3.4 shows how the main file of the experiment was implemented. As shown, it starts with initialisation of the desired environment and agent. In the case of this experiment, the sparsity values, stored in the list spr, range from 0 to 0.9 in steps of 0.1, except for the last two values which are 0.95 and 0.99. This list of sparsity values was decided upon after multiple test runs, which consistently showed little change in performance until around 90% of the weights were cut, which is the reason for a half step from 90% to nearly all of the weights, 99%.

Algorithm 3.4 Find winning ticket

  create environment
  build model bm (baseline model)
  build agent ba with bm
  train agent ba
  save weights of bm
  spr = list of sparsities
  for i in range length of spr **do**
    build temp model tmp
    load weights of bm onto tmp
    $tmp = \text{ModelPrune}(tmp, spr[i])$
    $tmp = \text{Reinitialise}(tmp, bm)$
    build temp agent ta with tmp
    train ta
    test ta
    save weights of tmp
    record performance
  end for

In this pseudocode, 3.4, the ModelPrune function refers to algorithm 3.2 and the Reinitialise function refers to algorithm 3.3. The for loop runs until the last desired sparsity value is achieved. Since the weights of every iteration are saved, the performance drop off can be inspected and used to determine at which point a winning lottery ticket is present.

For all model compilations used in this experiment, the Adam optimiser is used. This optimisation algorithm, introduced by Kingma and Ba (2014) is a widely used and popular optimiser as it is able to quickly produce good results. An optimiser serves the purpose of updating the weights of networks.

The following table presents the hyperparameters that were used in this paper.

Table 3.1: Hyperparameters

Name	Value	Description
nb_steps	5000	number of training steps
nb_episodes	100	number of testing episodes
node_num	24	number of nodes per layer
nb_steps_warmup	100	number of warm up steps
lr	1e-3	learning rate of Adam
memory	50000	replay buffer size
discount	0.99	discount factor γ

These hyperparameters were set depending on a variety of sources and prior experimentation.

3.6.1 Tickets

As mentioned by Frankle and Carbin (2018), a winning ticket is found not only when it performs highly even with high pruning, but also if it performs better than a random ticket. A normal lottery ticket can be defined as a function of x, as seen in equation 3.20, in which the mask m and the weights θ are taken into account. Such normal tickets can become winning tickets when the above mentioned criteria is satisfied.

$$f(x; m \odot \theta_0) \tag{3.20}$$

In which m is a mask found via the previously described minimum weight pruning and θ_0 is the original set of initialised weights.

To serve as a comparison, a random ticket is produced, however instead of using the previously described minimum weight pruning, it's weights are pruned randomly according to the desired sparsity. This way a random mask m_{Ra} is created. The weights of the random ticket are also not reinitialised to the original network weights θ_0, but instead they are reinitialised to random values, giving the random ticket random weights θ_{Ra}.

$$f(x; m_{Ra} \odot \theta_{Ra}) \tag{3.21}$$

4 The results

Results for this experiment were collected by running the experiment 5 times for each algorithm and environment. The performance data was collected each time an agent was tested in the corresponding environment. It is important to note that the small dotted line at $y = 200$ and $y = 0$ represent the maximum scores in the environments.

In order to statistically analyse the results, two tests were used. The first is the Shapiro-Wilk test for normality, which determines if data is normally distributed. This test is necessary to confirm that the data collected is not normally distributed, in order to confirm that the second test, the unpaired Wilcoxon rank sum test, should be used. This test can be performed on abnormally distributed data of two groups to determine if the data is statistically different. Whilst visual inspection is a good indication of whether a lottery ticket is a winning one, these statistical tests can help in their interpretation.

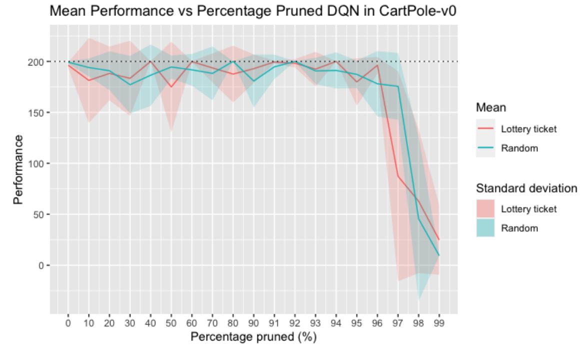

Figure 4.1: Mean performance ±Standard deviation of the DQN agent in the CartPole-v0 environment.

The first agent environment combination is that of a DQN agent acting inside of the CartPole-v0 environment, the performance of which can be seen in Figure 4.1. The graph shows that some variation in performance does occur but in general all 5 of the test runs seemed to manage to perform at a high level until around 90% of their weights had been pruned. The Shapiro-Wilks test for the lottery ticket data resulted in a p-value of $4.223e^{-6}$, which is much lower than the statistically significant p-value of 0.05. In the context of the Shapiro-Wilks test this means that the data is highly unlikely to be normally distributed, meaning that the Wilcoxon rank sum test can be used. The results of this test are a p-value of 0.6614. In this case the p-value is above the significant 0.05, but not quite a 1. This suggests that the mean performances of the lottery ticket and the random approach are slightly different but not enough so to be significant. Visually, it is also quite clear that the difference between the lottery ticket and the random ticket is not very large and that the lottery ticket does not necessarily seem to outperform the random ticket. It therefore does not categorise as a winning ticket.

The next agent environment combination is that of a DQN agent acting inside of the Acrobot-v1 environment, shown in Figure 4.2. Similarly to performance in Figure 4.1, all 5 of the runs seemed

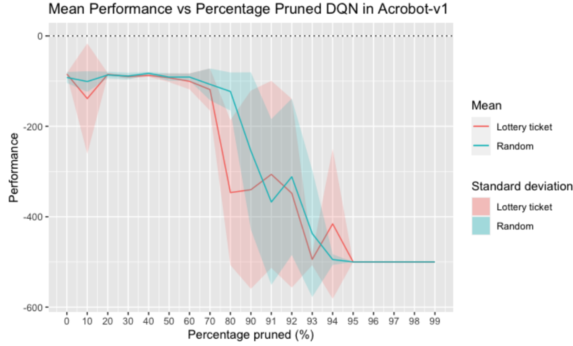

Figure 4.2: Mean performance ±Standard deviation of the DQN agent in the Acrobot-v1 environment.

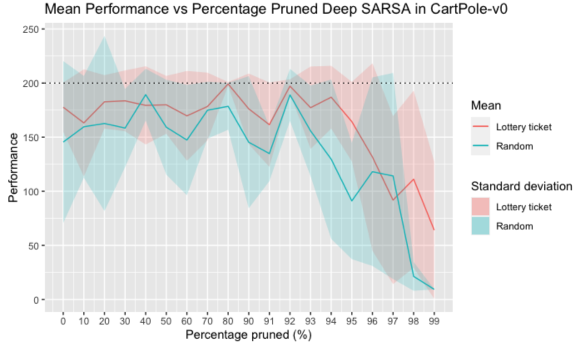

Figure 4.3: Mean performance ±Standard deviation of the Deep SARSA agent in the CartPole-v0 environment.

to perform well with low pruning, which continues until about 80% of the weights have been pruned. At this point performance drops, earlier than in the CartPole-v0 environment. This is most likely caused by the nature of the environment being more difficult in nature than the CartPole-v0 environment, meaning that cutting out too many weights damaged the performance, as some important weights were removed. Performing the Shapiro-Wilk test on the lottery ticket data gives a p-value of 0.001295, which is below 0.05, suggesting that the data is not normally distributed and the Wilcoxon rank sum test can be used. This test resulted in a p-value of 0.8713, which is above 0.05, meaning that the mean performances have a difference to them but not one that is statistically significant. Visually it once again becomes clear that the lottery ticket did not outperform the random ticket, even falling behind its performance at around 80%. The lottery ticket in this case can therefore once again not be categorised as a winning ticket.

The third agent environment combination is that of the Deep SARSA agent acting inside of the CartPole-v0 environment, displayed in Figure 4.3. The performance is comparable to that of the DQN agent in Figure 4.1, as it too manages to score highly up until around 90% of its weights are pruned, although the overall performance doesn't reach as consistently high scores as the DQN agent. The Shapiro-Wilk test is once again performed, resulting in a p-value of 0.0006986. This value is lower than 0.05 suggesting that the data is not normally distributed and the Wilcoxon rank sum test can be used. This test resulted in a p-value of 0.01982, which is lower than 0.05, suggesting that while the mean performances could be considered statistically significantly different. Visual inspection backs this up, as the mean performances seem to follow similar trends, but with the lottery ticket outperforming the random ticket at nearly all sparsities. This lottery ticket could therefore be considered as a winning one.

Interestingly, whilst there is deviation in the performances of the agents, the general trend that can be observed is that of successful performance until a high percentage of weights is pruned. This is true for all of the three agent environment combinations.

5 Conclusions

This paper aimed to investigate whether the lottery ticket hypothesis holds beyond the original scope of supervised image classification by delving into the domain of deep reinforcement learning. Both deep Q-network agents and deep SARSA agents were used to broaden the scope within deep reinforcement learning. As described in the results section, a general trend seemed to form, in which both agents managed to perform at a high level in the different environments, even after having a large percentage of weights pruned. However the lottery tickets of the DQN agent did not show a significantly better performance than their random ticket counterparts. In their paper, Frankle and Carbin (2018) state that they managed to create networks with 10-20% of the original network size while still per-

forming as or nearly as well as the original network, while outperforming a random counterpart. Whilst the aspect of performance is true for both the Deep Q-Network and the Deep SARSA agents that were examined in this paper, the lottery tickets of the DQN agent can not be classified as winning tickets due to them not outperforming the random tickets. Interestingly however, the lottery ticket of the SARSA agent did seem to outperform a random ticket, whilst also being pruned to a much smaller size, meaning that the SARSA agent does seem to produce a winning ticket.

This leads to the conclusion that the lottery ticket hypothesis might hold true in deep reinforcement learning, for some of the methods that were explored in this paper. However it is likely that more work is needed to find winning tickets in the DQN agent, as these should outperform a random pruning approach in high pruning sparsities. This might be achieved by more advanced pruning techniques such as mutual information pruning or taylor expansion pruning, which are mentioned by Molchanov et al. (2016).

Further improvements to this research could be made by investigating yet larger deep reinforcement learning networks and solving more difficult tasks. Whilst the results of this paper point towards the lottery ticket hypothesis holding true for deep reinforcement learning, it would be interesting to see it applied and used for cutting edge tasks such as autonomous vehicles.

Another interesting future work would be further investigation into the interweaving of the lottery ticket hypothesis in deep reinforcement learning and transfer learning, as this could have some promising real life applications.

References

Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 2013.

Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38 (8):716, 1952.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. URL `https://gym.openai.com`. Software available from gym.openai.com.

Francois Chollet et al. Keras, 2015. URL `https://github.com/fchollet/keras`.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*, 2019.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

Ari S Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773*, 2019.

Gavin A Rummery and Mahesan Niranjan. *Online Q-learning using connectionist systems*, volume 37. Citeseer, 1994.

Matthia Sabatelli, Mike Kestemont, and Pierre Geurts. On the transferability of winning tickets in non-natural image datasets. *arXiv preprint arXiv:2005.05232*, 2020.

Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, 8, 1995.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

Marc Aurel Vischer, Robert Tjarko Lange, and Henning Sprekeler. On lottery tickets and minimal task representations in deep reinforcement learning. *arXiv preprint arXiv:2105.01648*, 2021.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Zhi-xiong Xu, Lei Cao, Xi-liang Chen, Chen-xi Li, Yong-liang Zhang, and Jun Lai. Deep reinforcement learning with sarsa and q-learning: A hybrid approach. *IEICE TRANSACTIONS on Information and Systems*, 101(9):2315–2322, 2018.

Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. *arXiv preprint arXiv:1906.02768*, 2019.

Dongbin Zhao, Haitao Wang, Kun Shao, and Yuanheng Zhu. Deep reinforcement learning with experience replay based on sarsa. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE, 2016.