Dare to Discover: The Effect of the Exploration Strategy on an Agent's Performance

Thijs Nieuwdorp
s4210654

Bachelor Artificial Intelligence
Social Sciences Faculty
Radboud University Nijmegen

Supervisor: Dr. M.A.J. van Gerven

June 4, 2017

Contents

Chapter 1

Introduction

1.1 Reinforcement Learning

Reinforcement is a phenomenon everyone deals with on a daily basis. Most of our actions turn out to be either beneficial or disadvantageous. This either reinforces the behaviour which caused taking the action, or discourages that behaviour respectively. For example, a kid that climbs a tree carelessly and falls out of the tree as a result, will be more careful when attempting such behaviour next time. This field of behaviourist psychology has become the foundation for a field of research within Machine Learning, namely Reinforcement Learning. This field focuses on maximising the reward an agent receives in an environment by altering its behaviour.

Reinforcement Learning algorithms usually work in a similar manner. An agent performs actions (either single or a sequence) on an environment and in turn gets rewarded or punished for performing those actions, as illustrated in figure 1.1. This reinforcement signal, or return, is fed back to the agent. The agent in turn improves the policy that is used to pick actions. This improvement results in a better reward from the environment. The parts that make up a reinforcement learning algorithm are typically the policy (which maps states to actions), a reward function (returns a numerical evaluation of the current state, represents what is good and bad for the agent now), a value function (returns a numerical evaluation of the distant future from this state, represents long-term goals), and optionally a model of the environment. The focus in this experiment lies on a class of model-free Policy Gradient techniques called REINFORCE, as described by Williams (1992).

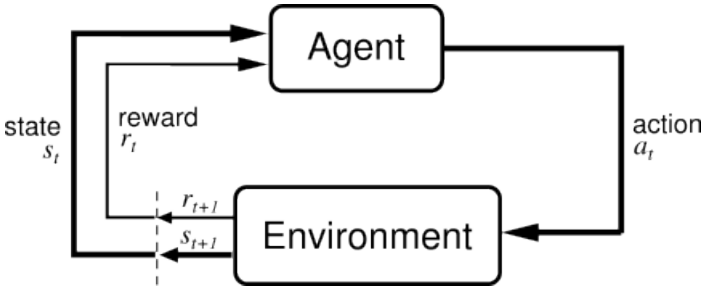

Figure 1.1: The reinforcement learning loop.

1.2 Exploration versus Exploitation

The trade-off between the Exploration of options with unknown outcome and Exploitation of proven successful options is a subject which is widely discussed in general. Uotila et al. (2009) look at the financial performanceas a result of exploration and exploitation of several companies on the American stock market (S&P 500). Hoang and Rothaermel (2010) discuss exploration and exploitation in the context of the Research and Development domain. On the topic of Reinforcement Learning this is no different. Audibert et al. (2009) explore the trade off in the context of multi-armed bandits. This is also stated by Kaelbling et al. (1996) in their survey on Reinforcement Learning. The trade-off is as follows: once an agent has taught itself a profitable course of action, should it pursue this proven path, or explore alternative options in hopes of finding even better actions? An agent that never explores may forever be performing sub optimally, yet an agent that only explores never fully utilises its experience. In order to strike a balance between these two extremes, several strategies have been proposed. These strategies range from full exploration to full exploitation in differing quantities. The strategies that will be compared are:

- Random: Full exploration.
- Greedy: Full exploitation.
- ϵ-greedy: Exploit, except for a certain amount of the cases, in those cases explore. This amount of cases is declared with ϵ.
- Decaying ϵ-greedy: Works like epsilon greedy, only with an epsilon value dat decays over time.
- Softmax: Full exploration, except instead of sampling the actions from a uniform distribution, sample from a multinomial distribution with a bias for preferable actions.

What is the effect of these exploration strategies on the convergence performance of a Reinforcement Learning agent? In order to explain this effect this thesis aims to compare these strategies, which differ on the exploration-exploitation spectrum. The differences between strategies will be measured using the time until convergence, the required amount of iterations until convergence, and the average score over the last 100 iterations.

It is hypothesised that the softmax strategy performs the best out of these strategies. While other strategies sample from the action space as a uniform distribution, softmax has a preference for actions it thinks perform best in the current state which means it samples actions from the action space as a multinomial distribution instead of a uniform one. This allows it to keep exploring until the policy gets more certain that specific actions are promising.

Chapter 2

Methods

2.1 Policy Gradient and REINFORCE

The essence of Policy Gradient methods is quite simple: adjust the policy in the direction that makes it better. In order to find the proper direction for policy adjustment the gradient of the parameters with respect to the score function is calculated. Williams (1992) describes a general class of associative reinforcement learning algorithms he calls REINFORCE algorithms which are used in this experiment.

Instead of a learned Q-function that attempts to estimate the true value function, REINFORCE uses the return r as an unbiased sample of Q for the given policy, state, and action performed. A baseline b can be subtracted from this return to reduce variance. This is then multiplied with the policy gradient and the learning rate to calculate the parameter update:

$$\theta \leftarrow \theta + \alpha(r - b)\nabla_\theta log\pi_\theta(s_t, a_t)$$

θ = Policy parameterisation

α = Learning rate factor

r = Reinforcement signal, the cumulative reward from that state on

b = the baseline, which ideally represents the expected average return from that state for all actions. However, in this experiment the baseline is set to a constant 0 to keep the implementation simple

$\nabla_\theta log\pi_\theta(s_t, a_t)$ = Gradient of log(action distribution given state s_t according to the policy).

This parameter update is iterated for each time step in an episode. Because this experiment's environment deals with delayed reward, an additional step must be made to circumvent the credit assignment problem. This problem occurs when actions early on still have an impact on a reward that is given later on. An episode must be rolled out, so that discounted rewards can be calculated for the entire episode. After that the parameters can be updated for every time step using these discounted rewards. The code for discounting of rewards can be seen in appendix A.2.

2.2 Strategies

A comparison will be made between several strategies that decide which actions the agent will take and whether it explores unknown alternatives or not. All of these strategies have a different ratio between exploration (picking an action of which the agent does not know whether it will perform well or not) and exploitation (picking the action of which the agent thinks it performs best).

2.2.1 Random

This strategy represents pure exploration and will sample actions from a uniform distribution that represents the action space. Though the agent may gather experience about the environment, this strategy forces that it is never used. Because of this feature, this strategy will perform as a baseline to compare whether an agent learns at all, or not.

2.2.2 Greedy

The greedy strategy always picks the action the policy thinks performs best, even when this policy has not converged yet. Therefore, this strategy embodies pure exploitation, and is the opposite of the Random strategy. This means it has a very high chance of picking an action, or a sequence of actions, that performs sub-optimally and sticking to it which prevents policy improvement. In the code and visualisations this strategy is called ARGMAX. The action selection can be described as:

$$a_t = argmax_{a \in A}(ER_t(a))$$

with $ER_t(a) =$ the expected return of action a at time step t.

2.2.3 Epsilon-greedy

In order to constantly stimulate exploration, yet still pick actions known to perform well, this strategy performs exploration in a constant amount of cases, denoted by ϵ. This triggers the agent to sample other actions which may be beneficial, yet still stick to proven strategies. However, because this ϵ is a constant, even when the agent has converged it will explore and pick sub-optimal actions. The agent picks a greedy approach, as explained above, with probability $1 - \epsilon$ and picks an exploratory approach, like the Random strategy explained above, with probability ϵ.

2.2.4 Decaying epsilon-greedy

This strategy is based on the ϵ-greedy strategy, however the ϵ value decays over time. In practice this means that the strategy starts out with a high ϵ, and thus a high exploration rate. Over time this ϵ grows ever smaller until it fades, optimally as the policy has converged, so that an optimal policy can be executed without having to take further (possibly sub optimal) exploratory actions. This strategy functions the same as normal ϵ-greedy, however the epsilon value changes as a function of time, according to γ^t with γ as a function of time.

2.2.5 Softmax

When exploring, ϵ-greedy samples an action from a uniform distribution, which means it could pick an unsatisfactory action just as likely as an action which is preferable. In a Softmax policy the distribution between actions is not uniform, but is biased towards promising actions.

The Softmax strategy calculates the probability of taking an action as follows for the output layer:

$$P(a) = \frac{e^{x_j^T}}{\sum_{k=1}^{K} e^{x_k^T}}$$

where:

x = the net input (connected nodes multiplied by their weight) for output node j

K = the amount of output nodes

T = the temperature parameter which can be tweaked to impact exploratory behaviour

In this experiment T will be set to a constant 1 which allows the use of the built-in TensorFlow tf.nn.softmax function which unfortunately, at the time of the experiment, does not support temperature.

2.3 Strategy Parameters

2.3.1 Epsilon comparison

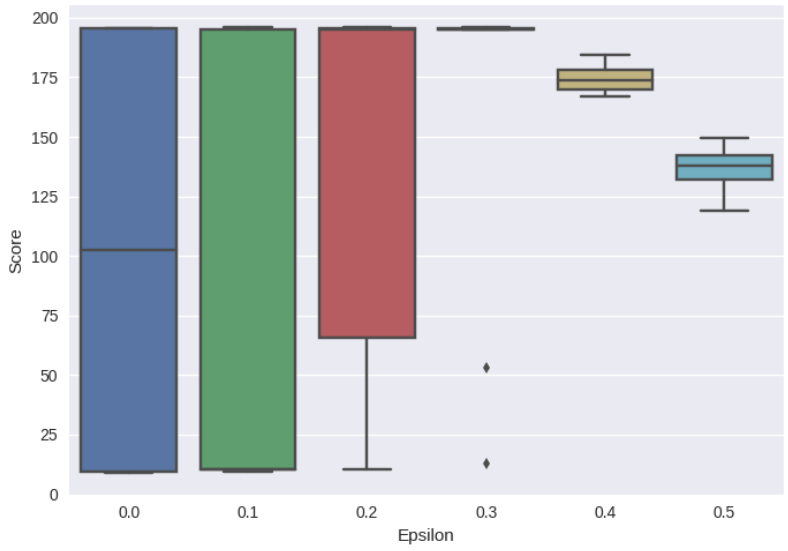

Figure 2.1: Different epsilon values and their average score over the last 100 iterations.

Plotted in figure 2.1 is the average score of the last 100 iterations as a function of epsilon. There was a maximum of 1000 iterations and each epsilon value ran for 20 trials. Only $\epsilon = 0.3$ passed almost consistently. This epsilon value was picked for the experiment.

2.3.2 Epsilon decay rate comparison

Plotted in figures 2.3, 2.4, and 2.2 is the performance of different epsilon decay rates for the decaying ϵ-greedy strategy. The higher decay rates pass the task consistently as can bee seen by their scores in figure 2.3. The decay rate 0.99 has the best performance in iterations (figure 2.2) and in time until convergence (figure 2.4) in addition to passing the task for all trials. This is the value that will be used in this experiment. These results were gathered over 20 trials with a maximum of 5000 iterations. The resulting decay rate creates the epsilon plot shown in figure 2.5

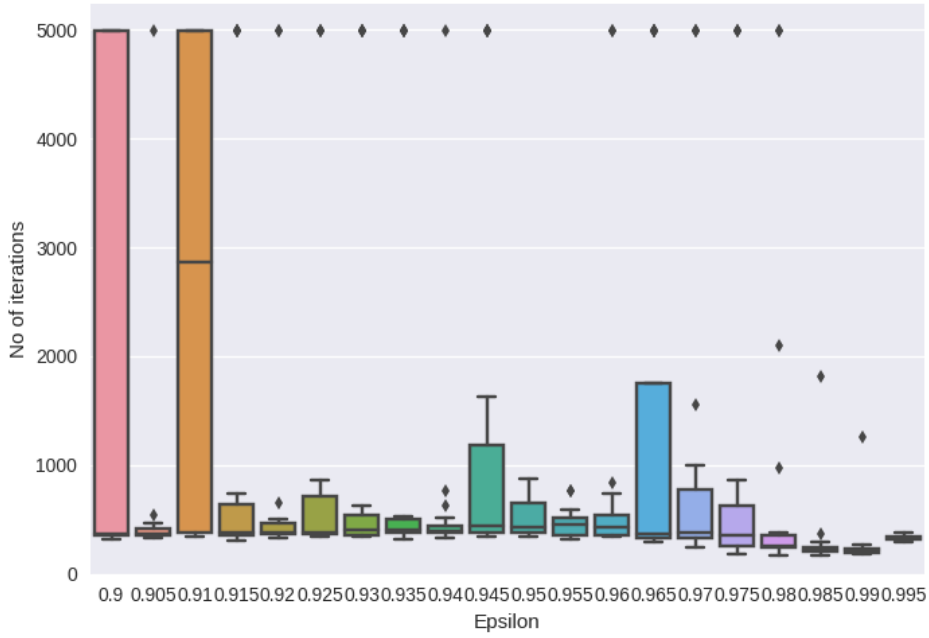

Figure 2.2: Different epsilon decay rates and the amount of iterations before convergence.

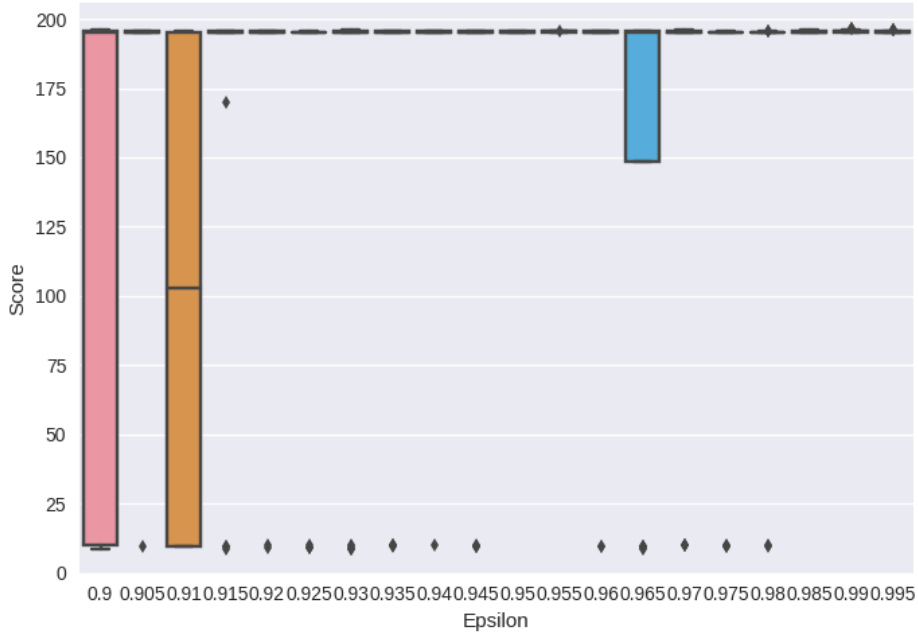

Figure 2.3: Different epsilon decay rates and the scores achieved with them.

2.4 Language, tool kits, and implementation

The implementation is done in python with the TensorFlow API, which is described by Abadi et al. (2016). This implements a single layer neural net agent that represent an associative policy. OpenAI gym, as

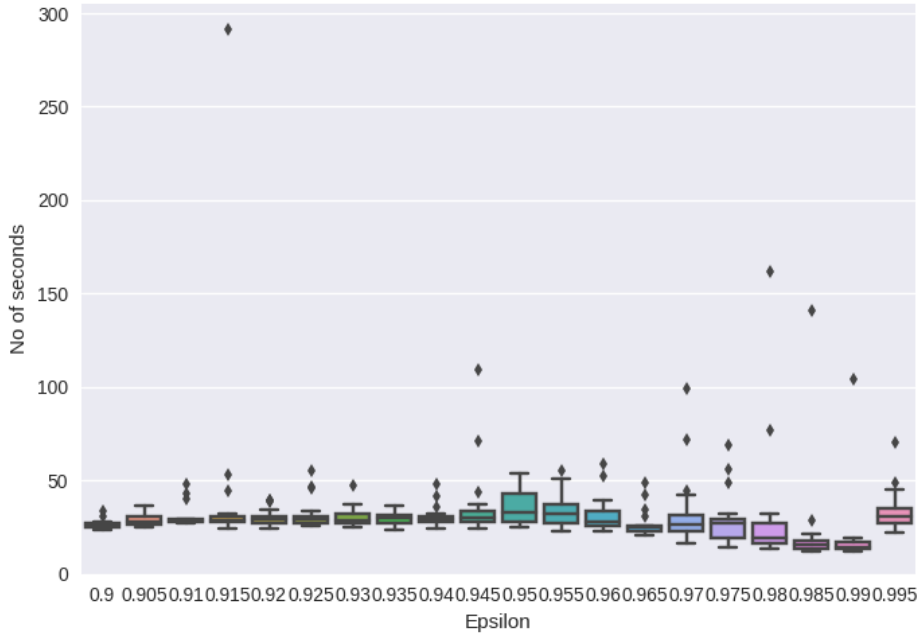

Figure 2.4: Different epsilon decay rates and the time passed until either convergence, or passing the task.

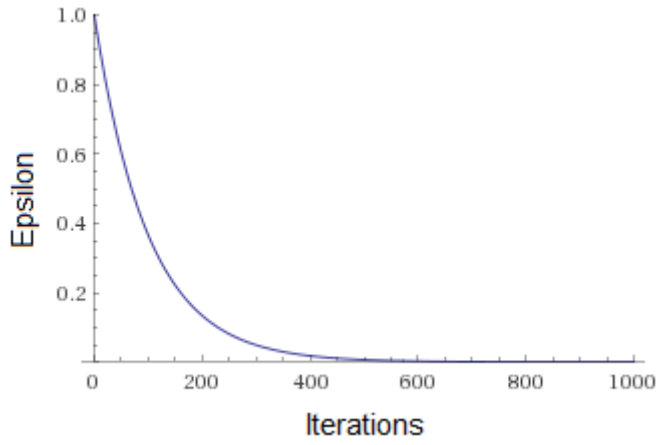

Figure 2.5: The epsilon plot resulting from an epsilon decay rate of .99

described by Brockman et al. (2016), is used to provide the environment on which the agent acts. The gym's CartPole environment (illustrated in figure 2.6) will be used for the benchmark. This environment corresponds with the pole balancing problem as described by Barto et al. (1983). In this environment 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. This environment represents a simple Markov Decision Process on which the REINFORCE algorithms are applicable. In order to properly visualise the results the Seaborn[1] library is used.

[1] https://github.com/mwaskom/seaborn

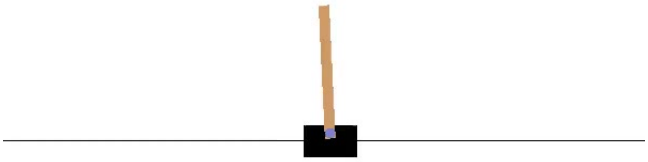

Figure 2.6: The Cartpole environment

2.4.1 Policy representation

The policy is represented by a single layer perceptron, as shown in appendix A.1. The input layer contains the same amount of neurons as the number of observations received from the environment. As the CartPole environment is being used the input layer receives the following variables:

- The position of the cart on the track
- Angle of the pole with the vertical axis
- Cart velocity
- Rate of change of the angle

This input layer feeds a fully connected output layer, which is a layer with softmax activation the size of the action space of this environment. In this case this layer has a size of 2 nodes (either add or subtract 1 force to the cart, pushing it left or right). This policy thus maps the observations of the environment to a probability distribution for the action space.

2.4.2 Training

The following algorithm is looped until the policy has converged, or reached the maximum number of allowed iterations:

1. An episode is rolled out until the cart crashes, or the maximum number of steps is reached (which is in this case 500 steps). For every step the state, action, and resulting reward is captured in a buffer. Depending on the strategy, the agent takes different actions:
    - Random: The agent samples a random action with uniform probability from the environment's action space, meaning all options are sampled from a uniform distribution.
    - Greedy/Argmax: The agent picks the action with the maximum expected return.
    - ϵ-greedy: The agent picks the action with maximum expected return for $1 - \epsilon$ of the cases. For ϵ of the cases, it samples a random action from a distribution representing the action space with uniform probability instead. The ϵ value for this experiment will be 0.3, for an explanation, see section 2.3.1.
    - Decaying ϵ-greedy: Does the same as ϵ-greedy, however, the epsilon value starts out near 1, and decays over time according to γ^{x} where x represents the iteration the agent is in. For γ 0.99 is used. This is explained in section 2.3.2.

    - Softmax: The agent samples its actions according to the multinomial distribution that is provided by the softmax policy neural network.
2. The reward buffer is discounted in order to deal with delayed reward (or the blame attribution problem). For details see appendix A.2.
3. The discounted rewards, taken actions and input states are used to calculate the gradients for the network using TensorFlow's built in $compute\_gradients()$ as explained by Abadi et al. (2016).
4. These gradients are then applied to the trainable variables in the policy network using the AdamOptimizer in TensorFlow which works in accordance with the description by Kingma and Ba (2014). This is a stochastic gradient based optimiser which utilises estimates of lower order momentum for faster convergence.

After the agent converges, or hits the iteration cap, this method is repeated for the next trial. This is done 30 times for every strategy. The code behind this algorithm is shown in appendix A.3.

2.4.3 Test setup

Every strategy runs for 30 trials. Each trial runs for up to 1000 iterations with a maximum of 500 steps in the environment per iteration. The number of iterations is limited to 1000 because this is a reasonable time in which the agent can converge. If the agent has not converged in this amount of iterations, chances are it will never converge, therefore it is capped to prevent the experiment from taking too long. Reaching a >195 average score over the latest 100 iterations means the agent passes the test in this environment. For every trial three things are measured:

- Average score over the last 100 episodes. This is used to discern whether the agent has passed the task.
- Amount of required iterations to converge.
- Number of seconds the agent takes to either finish the task, or reach the iteration limit.

These measures will be compared to each other in order to conclude whether there is a change in performance caused by learning strategy or not. The code behind this main loop is shown in appendix A.4.

Chapter 3

Results

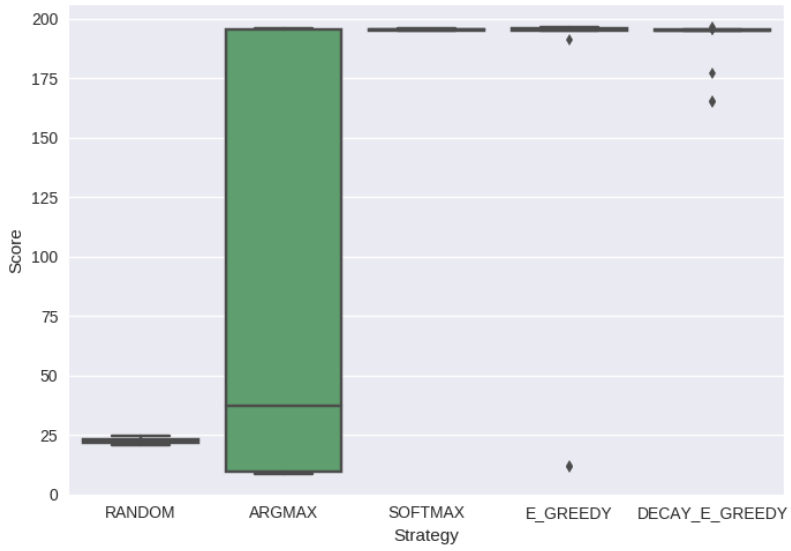

Figure 3.1: The mean of the score the agent received for the last 100 iterations, with 30 trials per strategy. This number has to be larger than 195 in order to pass the task. When this score is not achieved, the agent has reached the iteration cap and performed 1000 iterations without passing the task.

The first performance measure was the score provided by the environment. The task was passed when the average score over the last 100 iterations was larger than 195. As the agent was able to run in the environment for up to a reward of 500, sometimes the averages rose above 195. The Random strategy never passed the task, scoring between 20-25 consistently. The Argmax passed the task in less than 50% of the cases. Most of the time the Argmax strategy does not allow the agent's policy to converge. Both ϵ-greedy and decaying ϵ-greedy had some trials in which the agent did not pass the task, although it did pass the task most of the time. Softmax passed the task consistently. This is shown in figure 3.1.

The second performance measure is the amount of iterations before convergence is reached. The agent was allowed a maximum of 1000 iterations. The Random strategy never passed the task, it always took 1000 iterations, as this was the maximum amount of iterations it was allowed to run. This is also true for most of the trials for Argmax. The trials for Argmax that did pass the task took at least 450 iterations

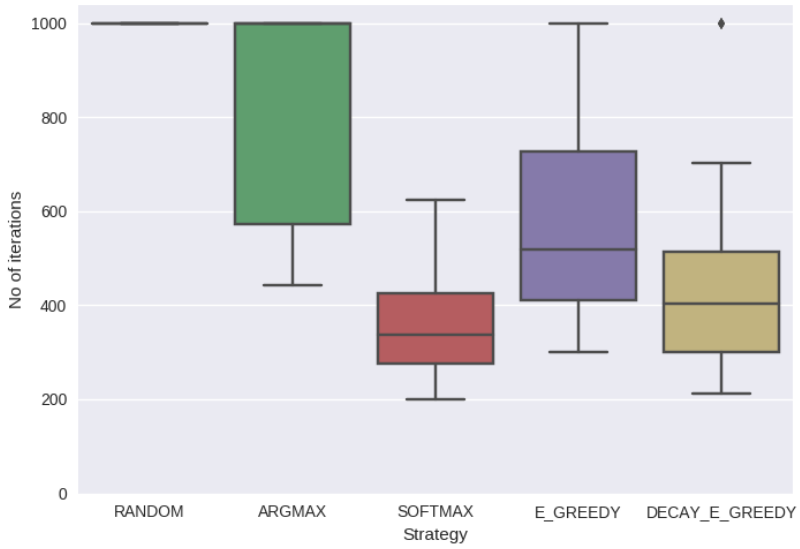

Figure 3.2: The number of iterations the agent ran to complete the task for every trial. The agent is allowed a maximum of 1000 trials to complete the task, which is not enough for some strategies.

before convergence. Again, both ϵ-greedy and decaying ϵ-greedy had a small amount of cases that reached the limit of 1000 iterations. Except for these outliers the decaying ϵ-greedy strategy converges in fewer iterations than ϵ-greedy. These results are visualised in figure 3.2.

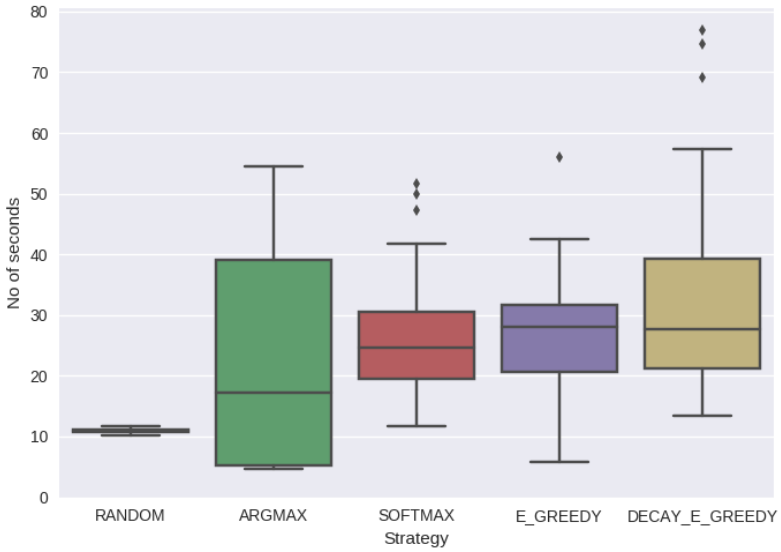

Figure 3.3: The number of seconds the agent took to either complete the task, or hit the iteration cap per trial.

The last measure for performance that was used was the time it took until the agent converged, or until the agent reached the iteration limit. The Random strategy never converged, and took the least amount of time, which means it failed the quickest. Argmax's performance varies greatly. As this strategy sometimes

prevented the agent from learning a good policy it failed quickly. In some cases the agent did pass, and when it did, it took a long time. Because of this the amount of time Argmax took varies greatly. The ϵ-greedy and decaying ϵ-greedy average at the same amount of time required until convergence, though decaying ϵ-greedy had a positive skew, whereas ϵ-greedy had a negative skew. This means ϵ-greedy had more outliers towards a lower amount of time required, whereas decaying ϵ-greedy had more outliers towards a higher amount of time required. The Softmax strategy required the least amount of time before convergence. This is plotted in figure 3.3.

Chapter 4

Discussion

As discussed earlier, the trade-off between exploration and exploitation is studied in several fields, for Reinforcement Learning this is no different. Never exploring may result in not finding better courses of action, yet not exploiting experience will lead to picking suboptimal actions. What are the effects of strategies that differ on the exploration-exploitation spectrum on the agent's performance?

The most extreme strategies on the spectrum perform poorly. The Random strategy, which exclusively explores, never converges, and has a very poor average score of 25, while an average of 195 is required to pass the task. The other extreme, the greedy (or argmax) strategy never explores and starts picking the actions it thinks are best from the beginning. Because of this the agent remains stuck in sub optimal policies, as it may not find better alternatives. The other strategies pass the task almost every time. The ϵ-greedy, which has a constant exploration rate takes the most amount of iterations to converge. This may be caused by the fact that even when it has gathered enough experience in the environment it is still forced to explore, causing it to never become fully optimal. In changing environments a constant exploration rate may prove useful, however in this setting that is not the case and leads to sub optimal behaviour. Which leads us to the next strategy, decaying ϵ-greedy. This strategy has a few outlying results that do not converge. It is suspected that in the cases of the outliers the agent was unable to explore courses of action that allowed it to converge in time for the phase where the agent fully exploits this experience. Apart from those outliers, this strategy performs really well, which is caused by the fact it uses its experience optimally later in time. Because this strategy starts exploring aggressively early on, yet decays its ϵ until it becomes a greedy strategy later on in time, it operates more optimally. The last and quickest converging strategy is the softmax strategy, which has no outliers. This strategy uses, on average, the least amount of both iterations and time to converge. Likely this is caused by the fact it uses the information of the return signal in a more efficient way, because in the case of exploration it does not sample the action space uniformly, but prefers actions of which it thinks they perform better.

These results confirm the hypothesis that strategies indeed have an effect on the convergence performance of a Reinforcement Learning Agent. From these strategies softmax has the most positive effect on the convergence performance of a Reinforcement Learning agent. However, since this was a simple task, using a single layer perceptron, more complex tasks and policies may yield different results. In future work these options could be explored. For example the "FrozenLake8x8-v0" in the openAI Gym has both a larger observation and action space. Even more complex environments like "BipedalWalkerHardcore-v2" which requires the agent to move a two-legged entity as optimally as possible could also be a possible

extension. Additionally, the softmax function can be expanded to allow for the use of temperature, which could improve convergence performance. Adding a baseline could improve convergence speed drastically as well, according to Williams (1992), although this applies to REINFORCE in general, and is not exclusive to a strategy. As these environments get more complex, the complexity of neural network architectures to represent the policy could increase as well.

This conclusion is in line with existing literature, such as findings by Kaelbling et al. (1996). They describe several ways to handle the exploration versus exploitation trade-off in order to improve learning abilities of a reinforcement learning agent, suggesting this impacts the convergence performance.

Appendix A

Appendix

A.1 Policy network

The definition of the single layer net that defines the policy, using TensorFlow-Slim. [1]

```
'''
# Define the policy network
'''
state_in = tf.placeholder(shape=[None, input_size], dtype=tf.float32)
output = slim.fully_connected(state_in, output_size,
                              activation_fn=tf.nn.softmax,
                              biases_initializer=None)
```

A.2 Discounted rewards

This function computes discounted reward backwards in time, because in this environment actions leading up to a rewarding time step have contributed to this reward as well. The gamma value used here is 0.99.

```
def discount_rewards(r):
    """ take 1D float array of rewards and compute discounted reward """
    discounted_r = np.zeros_like(r)
    running_add = 0
    for t in reversed(range(0, r.size)):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

A.3 Training procedure

The policy net training operation in the TensorFlow graph.

[1] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

```
'''
# Training procedure.
'''
# Learning rate (lr) is set to 0.05, and is constant for all strategies
optimizer = tf.train.AdamOptimizer(learning_rate=lr)
# One-hot encoded vector of the picked action
action_holder = tf.placeholder(dtype=tf.int32)
return_holder = tf.placeholder(dtype=tf.float32)
# To improve time to convergence the baseline can be set to average expected
# return from the current state
baseline = tf.constant(value=0, dtype=tf.float32)

# The network's predicted probability of the performed action
network_prediction = tf.reduce_sum(tf.multiply(output,
    tf.one_hot(action_holder, env.action_space.n)), axis=1)
log_probabilities = tf.log(network_prediction)
# We need to maximize the function below, so minimize its negative
# (the optimizer only supports minimize)
loss = -tf.reduce_sum(lr *
    tf.multiply(log_probabilities, (return_holder - baseline)))
# Function below calculates the gradients of trainable variable given
# the above loss function as function to minimize
optimize = optimizer.minimize(loss)
# Above operation requires state_in = s, action_holder = a,
# and reward_holder = r
```

A.4 Experiment loop

This is the code that runs the experiment itself.

```
for strategy in Strategy: # Where Strategy is an Enum with all strategies
    for trial in range(0, num_trials):
        # Launch a clean TensorFlow session
        with tf.Session() as sess:
            sess.run(init)  # Initialize graph and variables
            total_reward = []

            for i in range(max_episodes):
                s = env.reset()  # Reset the environment
                running_reward = 0
                ep_history = []
                for step in range(max_steps):
                    # Ask the policy for an action distribution for
```

```
# the current state
net_a_dist = sess.run(output,
    feed_dict={state_in: [s]}).flatten()
# Choose an action (depends on exploration strategy)
if strategy == Strategy.ARGMAX:
    a = np.argmax(net_a_dist)
elif strategy == Strategy.SOFTMAX:
    # Softmax policy
    a = np.random.choice([0, 1], p=net_a_dist)
elif strategy == Strategy.E_GREEDY:
    # epsilon-greedy
    if np.random.uniform(0, 1) < epsilon:
        a = env.action_space.sample()
    else:
        a = np.argmax(net_a_dist)
elif strategy == Strategy.DECAY_E_GREEDY:
    # decaying epsilon-greedy
    if np.random.uniform(0, 1) < np.power(epsilon_decay, i):
        a = env.action_space.sample()
    else:
        a = np.argmax(net_a_dist)
else:
    # Random
    a = env.action_space.sample()

s_new, r, d, _ = env.step(a)
ep_history.append([s, a, r, s_new])
s = s_new
running_reward += r
if d:
    # Update the network using discounted reward as
    # estimate for the Q-value
    ep_history = np.array(ep_history)
    ep_history[:, 2] = discount_rewards(ep_history[:, 2])

    feed_dict = {return_holder: ep_history[:, 2],
                 action_holder: ep_history[:, 1],
                 state_in: np.vstack(ep_history[:, 0])}
    grads = sess.run(optimize, feed_dict=feed_dict)
    total_reward.append(running_reward)
    break
```

```
            # Update buffer with number of iterations, running time,
            # and average score over last 100 episodes to see if
            # this iteration has passed the task.
            results[trial, strategy.value, :] = [i,
                time.time() - start_time, np.mean(total_reward[-100:])]
```

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876–1902, 2009.

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*, SMC-13(5):834–846, 1983. ISSN 21682909. doi: 10.1109/TSMC.1983.6313077.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

HA Hoang and Frank T Rothaermel. Leveraging internal and external experience: exploration, exploitation, and r&d project performance. *Strategic Management Journal*, 31(7):734–758, 2010.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

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

Juha Uotila, Markku Maula, Thomas Keil, and Shaker A Zahra. Exploration, exploitation, and financial performance: analysis of s&p 500 corporations. *Strategic Management Journal*, 30(2):221–231, 2009.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.