University of Glasgow | School of Computing Science

Deep Reinforcement Learning based Scheduler for HVFT applications in IoT

Angelos Perivolaropoulos (2208128p)

April 26, 2020

ABSTRACT

This paper presents a Deep Reinforcement Learning based Scheduler for High Volume Flexible Time(HVFT) applications. The use case for this scheduler is large scale Internet of Things(IoT) networks that rely on high network performance. Scheduling is known to be a notoriously difficult problem and over the years there have been a lot of different machine learning approaches to solve it. My solution to it is based on Deep Reinforcement Learning or most specifically on the Deep Deterministic Policy Gradient(DDPG) algorithm. The scheduler converges in 30 to 50 steps, trains in 4 to 7 seconds, provides better performance the bigger the scale and always returns an optimal scheduling solution.

1. INTRODUCTION

In this paper, I will be presenting the results of a scheduler I created using the Deep Deterministic Policy Gradient Reinforcement Learning(DDPG) algorithm (Lillicrap et al. 2015). As will be seen in the literature survey below, most Deep Reinforcement Learning implementations include a variation of Deep Q-Learning (Yu and He 2019) which is an inferior algorithm to DDPG proven by formal methods in this paper. The goals of the scheduler is to serve scheduled High Volume Flexible Time(HVFT) traffic and minimize network traffic degradation. The scheduler needs to be able to scale for a large number of devices and be performant enough to handle little to no network disruption. When a device needs to run a HVFT application, it sends a request to the base station that runs the scheduler and the scheduler replies with a time for when the HVFT application should be run. The limitation is that HVFT application can only be scheduled at the start of the day. HVFT application can be defined with the following ways:

- A device transferring a large amount data to the Cloud or another device on the network.
- A device doing a large software update.
- A device downloading a large amount of data.

Most large scale Internet of Things(IoT) networks will need to run HVFT applications on a semi regular basis and having a scheduler find out the best time for them to be scheduled is really helpful for the productive and without issue operation of those networks.

The paper will be split into 4 sections:

1. Background and literature survey
2. The theory behind DDPG
3. The simulated environment
4. The results

For the Background and literature survey, I will be describing the background behind my idea and reviewing the progress of other Deep Reinforcement Learning applications in the academic world to explain my reasoning behind choosing DDPG as the machine learning algorithm I used to implement the scheduler. In the theory behind DDPG, I will be explaining step by step how the DDPG algorithm came to be and what mathematical foundation it's actually based on. For the simulated environment, I will be describing the network simulation I used to train the scheduler. And finally, in the results, I will be showing the evaluation of my findings.

2. BACKGROUND

The internet of things is becoming a bigger and bigger part of our lives. With computational power becoming cheaper and chip sizes becoming smaller, it's only a matter of time before most of our devices are connected together in one way or another. It is predicted that there will be an average of 6.58 connected devices per person by 2020 (Inc 2019). Currently, we see the emergence of devices like Amazon Echo becoming part of our homes and joining the smart revolution. But the IoT isn't just a concept implemented for commercial use, businesses from a variety of fields have started to use the IoT to improve their performance, monitoring, security and cost. There are multiple examples to demonstrate that fact, but the one that comes to mind is health care. IoT in a lot of ways has revolutionised health care in many developed countries. It has given doctors the ability to monitor their patients remotely, helped with rehabilitation and medication management, and overall improved the lives of both the staff and the patients (Nogueira 2019). It has also allowed hospitals and other businesses to work on scale that they were unable to do before. But scale can also be a big technical limitation.

2.1 IoT Scale

As mentioned above, scale can be very difficult to work with, but first we need to define what scale means in this case. IoT networks consist of multiple usually low powered devices that fulfil a simple role and require relatively small amounts of network activity and processing power. The latency of those devices is most of the times needed to be

very low, so that decisions can be taken quickly and without delay. Disruptions in the network system in some cases, like the health care example, can cause significant damage or even be fatal. Scale on devices like that, while not being trivial, can be achieved with relative ease because the capacity isn't that big for normal workload. However, there are situations where workload has to deviate from normal. This could be because of a software update, an irregular data processing job, a large transfer of IoT sensor data to the cloud or many other different reasons. If such a big increase to network activity happened spontaneously, it could seriously affect the network and put people's lives at risk. Thankfully, these changes to the normal workload are most of the times time flexible, which means they can be scheduled for a time that causes the least disruption. We call these High Volume Flexible Time (HVFT) applications (Chinchali et al. 2018). Considering the seriousness of the situation, the scheduling of HVFT applications needs to be as accurate as possible to cause the least disruption to network activity. Most schedulers would just try to find the off peak times and schedule those jobs then, but this might not be enough in our situation since the traffic can be unpredictable. My recommended solution is using a Reinforcement Learning technique to train and optimise the scheduler, so that the HVFT application throughput is maximized and disruption of the network is minimized.

2.2 Machine Learning Technique Applications

Machine Learning(ML) is a technological advancement that has changed the world as we know it. Difficult real world problems of the past became several factors easier with new cutting edge AI implementations. Optimisation is one of those problems that ML techniques can work better than algorithmic approaches since its applications usually involve predicting the future which ML is quite good at. In this case, we can use a Machine Learning method called Reinforcement Learning which is a known good approach for these types of problems. Reinforcement Learning is the area of ML that focuses on agents whose goal is to maximize their cumulative reward and they achieve that using a balance of Exploration (new actions) and Exploitation (known actions).

2.3 Background Survey

I reviewed a variety of papers to gain the appropriate knowledge needed to tackle that issue and see what has already been done in the academic world. The papers I focused on reading were based on Deep Reinforcement Learning applications, similar to the one I am attempting to do.

Starting with the Cellular Network Traffic Scheduling With Deep Reinforcement Learning (Chinchali et al. 2018) by Sandeep Chinchali. This is the paper I based most of my research on since, like I mentioned above, it tackles a very similar problem to the one I am dealing with. The paper is about the process of developing a scheduler for HVFT (High Volume Time Flexible) applications in Cellular Networks. The goal of the scheduler is to achieve the following goals better than the current default scheduler: "Gracefully interact with other application classes" without causing issues to things like video streaming or web browsing, "Maximize scheduled HVFT traffic" and "Generalize across cells" since there are thousands of cells it is impossible to fine tune for each one individually. The authors manage to create such a scheduler using a Deep Reinforcement Learning algorithm and managed to double the HVFT traffic with minimal compared to equivalent heuristic schedulers. This paper did a lot of things well. The results are very well defined as well as the problem and the explanation of the logic behind the algorithm. The graphs are well made and informative and give a good explanation of the results. My favourite part about the paper is the fact that they used real data for their experiments instead of relying on produced data like a lot of similar papers in the field. There are some aspects that maybe could be done a bit better, most notably the selection of the Benchmark Controllers but overall it is a very informative paper. The results show that the Reinforcement Learning model significantly outperforms the Long Short Term Memory(LSTM) Neural Networks (Sainath et al. 2015) and the Heuristic scheduler.

S. Chinchali (Chinchali et al. 2018) presented a Cellular Network Traffic Scheduler that used Deep Reinforcement Learning to find out the best policy for least congestion and max throughput for High Volume Flexible Time(HVFT) applications. H. Mao (Mao et al. 2016) shows a similar use case but instead of Cellular Networks, he created a scheduler for Large Scale Multi-Resource Clusters using similar methodology. His paper is about creating a multi resource cluster scheduler to resource manage high capacity clusters. The name of the scheduler is DeepRM. They way they managed to do it is by a Deep Reinforcement Learning algorithm that represents policies as a deep neural network and uses methods like Policy Gradient Descent (Silver et al. 2014) to maximise the cumulative discounted award. The end result is that DeepRM beats the Shortest Job First(SJF) agent as well as the agent based on Tetris by a significant margin while having quite a small training time per iteration. I really liked this paper from a learning standpoint. It explained the application of the algorithm really well and the use case was very clear. The algorithms that it was compared to were also quite relevant which made the results even more valid. I had a few gripes as well though, the way the data was generated was left quite vague which raised a few questions if the training was done properly. The results prove that the DeepRM scheduler is better than the SJF and the Tetris-like one but it also proves that these kinds of algorithms can work in large scale systems.

I. Arel (Arel et al. 2010) took a similar problem but grounded in real life and built a system for Network Traffic Signal Control which technically is also a form of scheduler but for vehicles instead. His paper describes the process of creating a multi agent system to control the the traffic signals that is an improvement over the current approach that is used on the road today. The authors attempt it by using a Q-Learning Reinforcement Learning algorithm which gets refined with a neural network (Huang et al. 2016). The algorithm it is being compared to is a Longest-Queue-First(LQF) algorithm that is being used currently. The result is equivalent performance for low relative traffic arrival rate but the RL algorithm pulls ahead for higher relative traffic arrival by a considerable margin. This was an okay paper overall, the explanation of the process behind training the ML model was quite informative and straight forward, but I'm not sure if the algorithm used was the most appropriate one. On one hand, Q-Learning is a good approach when there is ambiguity which model to use and there's a lack of information about the overall state but the argument for using it here

is because "a model based solution adds unnecessary complexity". I don't think that's a valid reason for not trying them out instead since Q-Learning is most of the times inferior to model based approaches. Other than that, I believe the paper achieves what it's trying to do quite well. The paper's conclusion is that the Reinforcement Learning approach performs the same way for low relative arrival rates but outperforms the LQF algorithm for higher relative arrival rates.

All 3 papers used a Deep Reinforcement Learning algorithm with (Mao et al. 2016) and (Arel et al. 2010) using a very similar Q-Learning approach. We can see that DRL is a very useful method for optimizing scheduling systems. Next, we have 2 more papers that focus on more complicated applications of relatively the same algorithms.

Starting with Xiangping Bu (Bu, Rao, and Xu 2009) attempting to tackle a notoriously difficult issue in Systems Engineering, Auto-configuration of Online Web Systems. This problem is really hard because of the vast amount of variables which make up the State and Action space. X. Bu (Bu, Rao, and Xu 2009) uses a complex adaptive initialization policy along with a variation of Q-Learning to speed up the convergence rate and improve the accuracy of the algorithm. In his paper a new approach for online web system auto-configuration is discussed. Auto-configuration in web system is a notoriously difficult problem due to the vast amount of variables that can be configured. The authors find a really smart way to use Reinforcement Learning with efficient heuristic initialization policies which speed up the convergence process by a significant margin. The end result is an almost 5 times performance increase in a lot of cases compared to the trial and error method which is conventionally used. They used TPC-W benchmark (Menascé 2002) to evaluate their results and tried a variety of different approaches with the conclusion being that a Q-Learning Reinforcement Learning algorithm with an adaptive init policy and online learning yielded the best results. I was very pleased with this paper and learned a lot from it. It does a great job of explaining the process of reaching the final algorithm and that process if logical and data driven. From a machine learning perspective, this is a great paper that successfully tackles a hard problem but from a networking perspective, it fails in a lot of ways which is actually briefly acknowledged in the paper itself. First of all, the evaluation benchmark TPC-W is an unrealistic scenario in real world applications. The results seem too over fitted for the benchmark unlike typical workloads which would have a lot of variety in the requests. Nevertheless, it lays a very good foundation for future approaches to improve on that method and apply it in a real world scenario. The results show that we could potentially see a 5 times increase in response time after convergence of the algorithm which is a very significant value. Some criticisms that I have is that I believe the scenarios used to evaluate the algorithm were not based on real world scenarios and the algorithm was over fitted for the TPC-W benchmark. We know that since some of the configuration were objectively bad for real workloads like the KeepAlive Timeout config which was unnaturally high since TPC-W uses long term TCP connections which isn't something that usually happens in real web systems. Of course this is just a proof of concept so it is very likely that the method could work with real data as well, but we can't know that until those experiments are done.

On the other hand, Guanjie Zheng (Zheng et al. 2018) tries out a similarly daunting task of creating a News Recommendation Framework. In this case, the author uses a lot of cutting edge Reinforcement Learning techniques like predicting the future rewards and using improved Exploration-Exploration approaches to allow their framework to surpass all other conventional recommendation systems in both accuracy and diversity of choices. His paper introduces a method for News Recommendation using Deep Reinforcement Learning. What the paper does different than other similar ones in the subject is that it uses an algorithm that attempts to predict the future reward of an action rather than just utilizing the current reward. This allows the algorithm to more efficiently select news articles based on the user's preference and user activeness. The algorithm proposed is based on a Deep Q-Learning Network (DQN) (Silver et al. 2014) that we've seen in previous papers as well. The difference is that the exploration-exploitation method is Dueling Bandit Gradient Descent (Yue and Joachims 2009) which takes advantage of two different Q Neural Networks to select the next action selected. Another difference is that it tries to predict the future reward that a choice with give and takes it into account when deciding which article to recommend. The result is the final algorithm significantly outperforms other cutting edge algorithm both in offline and online scenarios. This is the paper with the most recent publication date that I have reviewed here and we can definitely see that with how much more advanced the algorithms used are. For example, we see how outdated the e-greedy method for exploration and exploitation is, something that is heavily used in almost all other Deep Q-Learning application papers I've reviewed so far. Also, trying to calculate the future reward is something none of the other publications have attempted. Overall, I'm quite impressed with this paper, the experiments were clear and the thought process behind the final algorithm makes sense and is easy to follow. The paper proves that by using cutting edge machine learning algorithm there is significant improvement in performance compared to previous recommendation solutions. As far as improvements, There's not much that I would do differently than what the authors have done since the paper is quite complete. Maybe more focus could be given into the diversity of the actions of each algorithm evaluated. There is some discussion being done at the end saying that the HLinUCB algorithm, that is conventionally used for news recommendation, is surprising close in diversity to the final algorithm and beats all previous iterations of it. I would do some more research into comparing the 2 algorithms to see if there's any room for improvement in that area.

(Bu, Rao, and Xu 2009) and (Zheng et al. 2018) both use an advanced variation of the Q-Learning algorithm to achieve their goals. These papers helped me a lot to understand and formulate what the best algorithm to use for my use case would be, and like I mentioned above, I decided to go with DDPG.

3. SCHEDULING ALGORITHM

For the scheduling algorithm I used the Deep Deterministic Policy Gradient(DDPG) algorithm. This is a Policy Gradient (Sutton et al. 2000) Reinforcement Learning algorithm that uses the Actor-Critic (Konda and Tsitsiklis 2000) paradigm with a Replay Buffer cache. This algorithm was invented as an improvement over previous iterations of

Policy Gradient and is proven to be particularly good for scheduling tasks that have a constant action space. To explain how the algorithm works, it needs to be broken down to the 3 algorithms it was derived from:

1. Deep Q-Learning (Yu and He 2019)
2. Policy Gradient (Sutton et al. 2000)
3. Actor-Critic (Konda and Tsitsiklis 2000)

3.1 Deep Q-Learning

Starting with the one that the other 2 are based of, Deep Q-Learning, is the fundamental algorithm that most modern Reinforcement Learning methods are based of. Before moving to Deep Q-Learning, we need to explain Q-Learning first. Q-Learning is a really simple Reinforcement Learning algorithm that predicts actions based on a Q-table which includes all possible states of the environment and a number for each possible action from that state. After an action is taken, the Q-table is update based on the Bellman equation (Beard, Saridis, and Wen 1997):

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma * max_a(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{1}$$

where

Q = the Q-table
s_t = the current state
s_{t+1} = the next state
a_t = the current action
α = the learning rate ($0 < a \leq 1$)
γ = the discount factor

It is pretty obvious that this algorithm doesn't scale very well for more complicated state and action spaces since the Q-table becomes too big and unmanageable, so the solution to that found in Deep Q-Learning is using a neural network instead of a Q-table. Neural networks are really good at approximating results, so the idea here is instead of building a Q-table, it's more efficient to create a neural network that takes the state as input and returns Q-values for each action like the figure below.

The neural network usually is a Convolution Neural Network(CNN) (Lin and Shen 2018) and is initialized with random coefficients at the beginning and it gets updated using back-propagation and mini-batches stochastic gradient descent. The model uses the following Loss function:

$$Loss = \frac{1}{2}(r + max_a(Q(s_{t+1}, a_{t+1}; \theta_{t+1})) - Q(s_t, a_t; \theta_t))^2 \tag{2}$$

where

Q = the Neural Network
θ_t = the current state coefficients
θ_{t+1} = the next state coefficients
s_t = the current state
s_{t+1} = the next state
a_t = the current action
a_{t+1} = the next action

Deep Q-Learning can be further optimised by using Experience Replay, a concept that we will discuss further later, which is basically a cache that can be used to sample mini-batches to improve training.

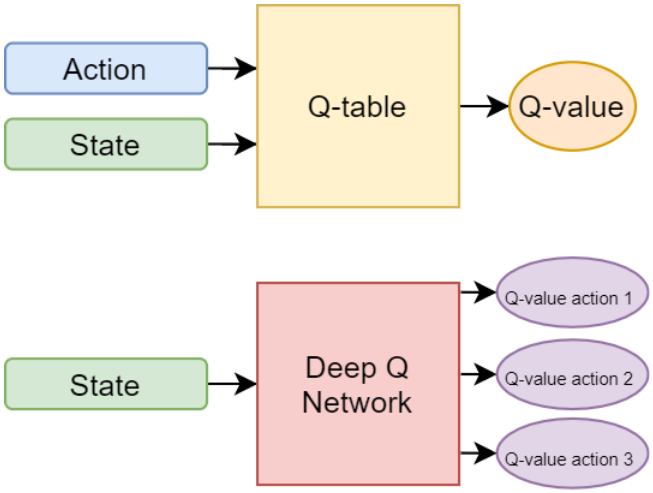

Figure 1: The difference between Q-Learning and Deep Q-Learning

3.2 Policy Gradient

A policy is a function that maps a state to an action that provides the highest reward. All of Reinforcement Learning is based on the idea of optimising the coefficients of a policy to find that optimal policy which maximises the expected reward. Q-Learning works by trying to improve the policy with optimising the estimates of different actions. This is an implicit way of improving a policy and a lot of the times is quite inefficient. It makes sense that a way of optimising a policy directly would be better and more efficient. This is where we have the Policy Gradient algorithms. Policy Gradient methods are better than Deep Q-Learning in a multitude of ways (Kapoor n.d.):

1. As mentioned above PG algorithms explicitly improve a policy which makes them more efficient.
2. Q-Learning is deterministic which means it is impossible to learn any stochastic policies while PG algorithms can.
3. Q-Learning can't solve problems that have a continuous action space which includes the problem described in this paper, while PG algorithms can.

Now to explain how Policy Gradient methods work, the idea is to directly optimise the coefficients of the policy using Gradient Descent. For a policy $\pi_\theta(s_t) = a_t$, we write down a function that takes the coefficients θ and returns an expected reward:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r(\tau)] \tag{3}$$

where

θ = the coefficients of the policy
r = the reward function based on the trajectory
τ = the current trajectory

In this case, the goal is to find the coefficients θ to maximize J and since it's a maximization problem, we can do that using gradient descent. The update function for θ is:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{4}$$

Calculating $\nabla J(\theta_t)$ is difficult and would require a lot of integral calculations. To simplify that, according to the **Policy Gradient Theorem**:

$$\nabla J(\theta) = \nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}[r(\tau) \nabla \log \pi_\theta(\tau)] \tag{5}$$

which comes from:

$$\begin{aligned} \nabla \mathbb{E}_{\pi_\theta}[r(\tau)] &= \int \nabla \pi_\theta(\tau) r(\tau)\, dr \\ &= \int \pi_\theta(\tau) \nabla \log \pi_\theta(\tau) r(\tau)\, dr \\ &= \mathbb{E}_{\pi_\theta}[r(\tau) \nabla \log \pi_\theta(\tau)] \end{aligned} \tag{6}$$

By expanding the definition of $\pi_\theta(\tau)$ we get:

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \tag{7}$$

which then makes:

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[(r(\tau) \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \right) \right] \tag{8}$$

To be able to calculate the expectation, we would need find the average of a large amount of sampled trajectories using Markov Chain Monte-Carlo(MCMC) (Carlo 2004). And with that, we have the basis for all Policy Gradient algorithms which can be used with good success. However there's still quite a lot of variance considering we are still taking the trajectory into account from $r(\tau)$. We can replace that with the expected discounted return G_t since past rewards shouldn't be affecting the optimization of the coefficients.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... \tag{9}$$

This gives us the final equation:

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} G_t \nabla \log \pi_\theta(a_t|s_t) \right] \tag{10}$$

This is the **REINFORCE** Policy Gradient algorithm.

However, we still have a lot of variance in the sampled trajectories and it would be a good idea to stabilize them using a baseline that wouldn't interfere with the policy coefficients. The baseline that's mostly used for the REINFORCE algorithm is the State Value function which takes the current state returns the expected returns of the policy:

$$V(s) = \mathbb{E}_{\pi_\theta}[G_t | S_t = s] \tag{11}$$

This makes the new REINFORCE algorithm which is called REINFORCE with Baseline:

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} (G_t - V(s)) \nabla \log \pi_\theta(a_t|s_t) \right] \tag{12}$$

3.3 Actor-Critic

Actor-Critic algorithms are the natural progression of the REINFORCE with Baseline Policy Gradient algorithm. The idea here is that since we need to find a good $V(s)$ baseline, we can use Gradient Descent on it to calculate the best coefficients for it, which we will call ω. We then need to calculate the gradients, one for $V^\omega(s)$ which we will call the critic, and one for our original algorithm which we will call the actor. First, we need to rewrite the discounted return in regards to the State Value function with coefficients ω and we will also be approximating the discounted return to only 1 step to simplify the calculations:

$$G_t \approx R_{t+1} + \gamma V^\omega(S_{t+1}) \tag{13}$$

This gives us two gradients to calculate:

$$\nabla \mathbb{E}_{\pi}[r(\tau)] = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T} \left(R_{t+1} + \gamma V^\omega(S_{t+1}) - V^\omega(S_{t+1}) \right) \nabla \log \pi(a_t|s_t) \right] \tag{14}$$

Which is the **Actor** gradient.

$$\nabla J(\omega) = R_{t+1} + \gamma V^\omega(S_{t+1}) - V^\omega(S_{t+1}) \tag{15}$$

Which is the **Critic** gradient.

The critic will continuously optimise the coefficients of the baseline that is used for the actor, which provides increased efficiency compared to the REINFORCE with baseline algorithm.

3.4 Deep Deterministic Policy Gradient

Here is where everything comes together. The DDPG algorithm can be seen a combination of the Actor Critic Policy Gradient algorithm mentioned above, along with an approach similar to Q-Learning called Q-function. The goal here is to find the optimal action-value function $Q^*(s, a)$ where we can calculate the optimal action using:

$$a^*(s) = \arg\max_a (Q^*(s, a)) \tag{16}$$

Now we can think of Q as a neural network approximator with coefficients θ and we can use the Bellman equation that was also used in Q-Learning previously to get to:

$$Q_\theta(s_t, a_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) \right] \tag{17}$$

where

Q_θ = the neural network with coefficients θ
s_t = the current state
s_{t+1} = the next state
a_t = the current action
γ = the discount factor

Similarly with Deep Q-Learning, we can use the mean-squared Bellman error (MSBE) as a loss function to train the network:

$$Loss = \mathbb{E}\left[\left(r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t) \right)^2 \right] \tag{18}$$

This looks very similar to the loss function in Deep Q-Learning but there are some extra things that make DDPG much better (OpenAI n.d.).

1. DDPG uses a Replay Buffer which is a cache that stores previous experiences that the model can sample from. The buffer should contain a variety of both new and old experience as to not overfit the model.

2. DDPG also uses a target network. This means that 2 neural networks are created for training and both are trying to optimise the parameters θ, but the target network's parameters are slightly different because it has a time delay, and which we will call θ_{targ}. The target network is updated after every main network update using polyak averaging based on the hyperparameter $0 < \rho < 1$:

$$\theta_{targ} = \rho\theta_{targ} + (1 - \rho)\theta \tag{19}$$

3. DDPG uses noise added to the actions, in this case use the Ornstein–Uhlenbeck (OU) (Schöbel and Zhu 1999) noise, for exploration vs exploitation.

The target network has its own policy $\mu_{\phi_{targ}}(s)$. By taking the Replay buffer and the target network into account, our new Loss becomes:

$$Loss = \mathbb{E}_{s,a,r\sim D}[(r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_{\theta_{targ}}(s_t, \mu_{\phi_{targ}}(s_{t+1}))^2] \tag{20}$$

Using that loss we can train the Q^θ neural network and use it as our **Critic** network. Next we need to train the policy $\mu_\phi(s)$. This can be done using gradient ascent on:

$$J(\theta) = \mathbb{E}_{s\sim D}\left[Q_\theta(s, \mu_\phi(s))\right] \tag{21}$$

Which will give us the **Actor** network. And that concludes how DDPG works.

4. ENVIRONMENT

Since creating a physical IoT network to use for training the scheduler would have been inefficient and quite expensive, I created a virtual simulation environment instead. The environment would be based on OpenAI gym (Brockman et al. 2016) and have the same interface.

4.1 Network Traffic Simulation

The environment would simulate internet traffic from 30 virtual devices based on an On-Off distribution (Chandrasekaran 2009). The way it works is by viewing each device on the network as a two state channel. As long as a channel is on, it continues to send network traffic until it gets turned off again. Each device has a different traffic profile, with some devices doing heavy network loads while other having lighter loads. Depending on which state each channel is at, there is a probability that it can be switched. This probability is based on the time of the day as a device is most likely to be used during the day time rather than night time.

To do that, I created a 24x2 matrix that includes all the D_{on} and D_{off} probabilities for every hour of the day. To make sure my results were resembling a real life scenario, I made sure that the D_{on} probability was significantly higher during the day time, with the peak being around 2PM, while the D_{off} probability remained on relatively similar levels throughout the day with a smaller dip during night time.

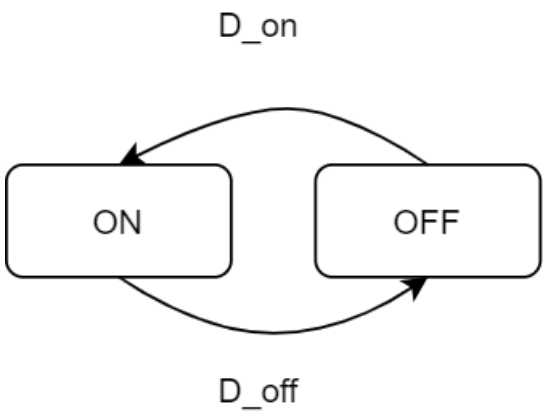

Figure 2: Showing the state changes of a channel

The way the simulation works is by starting with every device being off, then for every simulated minute, a random number would be sampled for each device and depending on the device's state if the number sampled is greater than the D_{state} probability of this hour, then the device would switch states. Depending on the device, a different network traffic load would be sent to the network per minute, as long as the device stays in an on state. The simulator would then create 2 timeseries, one with the total number of bandwidth per minute and one with the total number of devices per minute.

4.2 Gym Environment

At the initialization stage of the environment, an array with the sizes of all the jobs that are to be scheduled needs to be provided, an array with times each job would take, and the total devices and bandwidth limit of the network. For every step, an action would need to be given, that action would be a 1D array with the schedule time for each job. The environment would then simulate the whole day and add the job traffic to the 2 simulated timeseries. It would then return the bandwidth timeseries as a 1D array for the state of the environment and this reward:

$$r = \frac{\sum_{i=0}^{1440} 0.8\left(\max((b_i - B), 0)\right) + 0.2\left(\max((d_i - D), 0)(B/D)\right)}{1440}$$

where

B = the bandwidth limit of the network in KB
b_i = the bandwidth of the network at the i minute
D = the total devices limit of the network
d_i = the total number of on devices of the network at the i minute

5. EVALUATION

I implemented the DDPG scheduler in python using Google's Tensorflow (Abadi et al. 2016). I also used a decaying ϵ variable to scale the OU noise added to the action for Exploration/Exploitation. The hyperparameters used are:

- Actor Learning Rate = 0.00005
- Critic Learning Rate = 0.001
- $\tau = 0.001$
- $0.01 < \epsilon < 1$
- Exploration Decay = 1/60
- Mini Batch Size = 30
- $\gamma = 0.99$

I trained the scheduler with 100 step epochs on a variety of instances of the environment. The first instance I evaluated was trying to schedule 7 HVFT applications with all requesting a fraction of the total bandwidth limit to run continuously for an hour. I trained the model with 4 different fractions, the fractions I tested were [0.2, 0.25, 0.3. 0.35]. This is the results I got from this run are in figure 3.

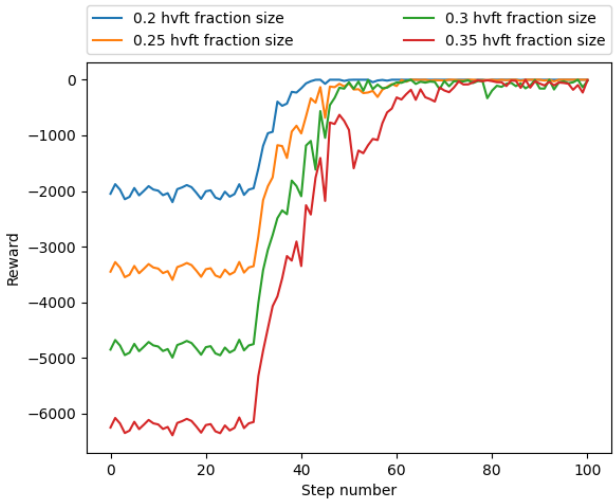

Figure 3: Rewards for training the scheduler for 6 HVFT apps

As expected, the easiest problems are converging first while the harder ones are taking a little bit longer. There is also a big bump on the rewards when the mini batch size is exceeded on step 30 which makes sense since the initial state is random with a lot of noise. The training was also done really fast, with the hardest problem taking only 4.5 seconds to train which can be found on figure 4.

Then, I tried making the problem even harder by training it on an environment with 7 HVFT applications and keeping the same fractions. The results can be found on figure 5.

We can see that the scheduler was having a bit of trouble converging for the harder instances of the environment but still manages to get good results. It is possible that there is no perfect solution of the problem at this point and the scheduler is trying to find the best possible solution instead.

The next run was even harder with the scheduler learning to schedule 8 HVFT applications with again the same fractions. The results are on figure 6.

Here the scheduler is managing fine for the first 3 problems, but is having some trouble finding a perfect solution to the hardest problem. By looking at the actions taken

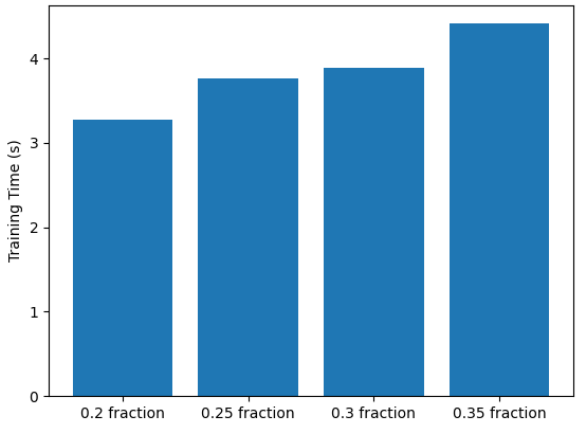

Figure 4: Time taken to train for 6 HVFT apps

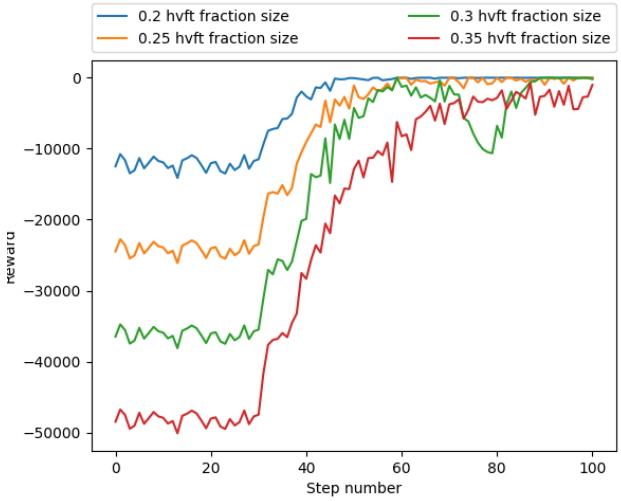

Figure 5: Rewards for training the scheduler for 7 HVFT apps

here, we can see that we've reached the point that there isn't a perfect solution as there exists no hour block where the internet traffic goes below 0.65 of the bandwidth limit. The scheduler seems to have found a pretty good solution regardless considering the rewards ends up being more than -2000 which is only around 1.5% above the network limit.

The next step of the evaluation is testing a heavily scaled up version of the environment. This time the number of devices on the network is increased to 150 from 20 and the HVFT apps to be scheduled has also increased to 30. The fractions have also been decreased to compensate. The new fractions array is [0.06, 0.08, 0.1, 0.13] and the results can be found on figure 7.

Again the easier problems converge quite quickly but the hardest problem is trying to find the optimal non perfect solution and doesn't converge on reward 0. This is still a pretty good result considering that the fraction used is quite high for 30 HVFT applications. Comparatively with the previous problems, the scheduler should have had a harder time

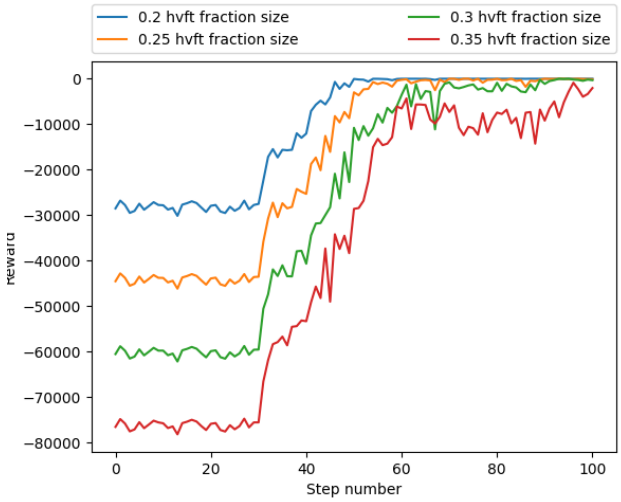

Figure 6: Rewards for training the scheduler for 8 HVFT apps

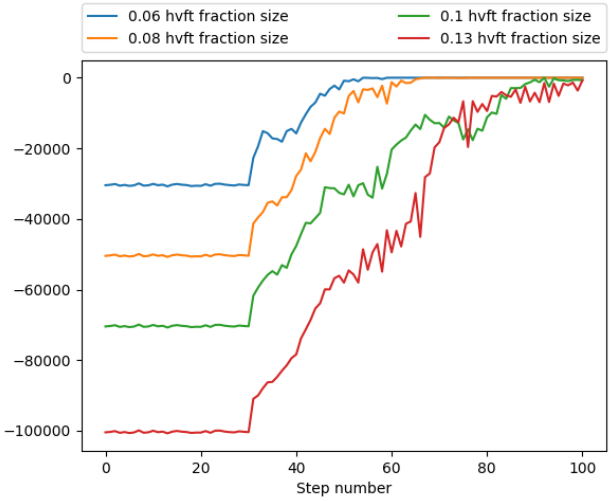

Figure 7: Rewards for training the scheduler for 30 HVFT apps on a scaled environment

finding a solution here. However, the fact that there are way more devices on the network means there's less variance from the On-Off distribution which helps the model recognise patterns. This means the scheduler should have an easier time scheduling the more device are on the network. The training times were also quite fast here too with the highest for the most difficult problem only being around 7 seconds. The times plot can be found on figure 8.

6. CONCLUSIONS

This paper presented a Reinforcement Learning Scheduler for HVFT applications in IoT based on the Deep Deterministic Policy Gradient(DDPG) algorithm. The scheduler was trained using an environment that simulated an IoT network and produced network traffic using the On-Off distribution. The results showed that the scheduler trains really fast with the longest training time being only 7 seconds. The scheduler also showed good results by either converging to the best solution or, in the case that one didn't exist, finding an acceptable solution with minimal network disruption. The scheduler was also found to perform better the bigger the scale of the system which proves the original hypothesis that it could be applied to a large IoT network like a hospital or a factory.

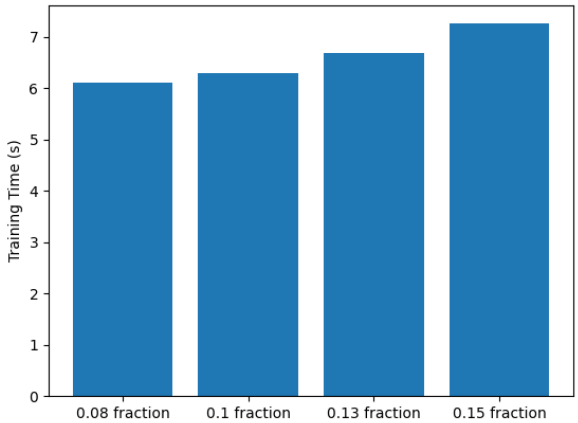

Figure 8: Time taken to train for 30 HVFT apps on a scaled environment

Future work would include running the scheduler in a real IoT environment and training it with real data to validate the results. Improvements can also be made on the algorithm itself by starting the training with a pre-trained policy and a Replay Buffer that includes real user data.

References

Abadi, Martın et al. (2016). "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283.

Arel, Itamar et al. (2010). "Reinforcement learning-based multi-agent system for network traffic signal control". In: *IET Intelligent Transport Systems* 4.2, pp. 128–135.

Beard, Randal W, George N Saridis, and John T Wen (1997). "Galerkin approximations of the generalized Hamilton-Jacobi-Bellman equation". In: *Automatica* 33.12, pp. 2159–2177.

Brockman, Greg et al. (2016). "Openai gym". In: *arXiv preprint arXiv:1606.01540*.

Bu, Xiangping, Jia Rao, and Cheng-Zhong Xu (2009). "A reinforcement learning approach to online web systems auto-configuration". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, pp. 2–11.

Carlo, Chain Monte (2004). "Markov chain monte carlo and gibbs sampling". In: *Lecture notes for EEB* 581.

Chandrasekaran, Balakrishnan (2009). "Survey of network traffic models". In: *Waschington University in St. Louis CSE* 567.

Chinchali, Sandeep et al. (2018). "Cellular network traffic scheduling with deep reinforcement learning". In: *Thirty-Second AAAI Conference on Artificial Intelligence*.

Huang, Gao et al. (2016). "Deep networks with stochastic depth". In: *European conference on computer vision*. Springer, pp. 646–661.

Inc, Statista. Statista (2019). *Number of network connected devices per person around the world from 2003 to 2020.* URL: https://www.statista.com/statistics/678739/forecast-on-connected-devices-per-person/.

Kapoor, Sanyam (n.d.). *Policy Gradients in a Nutshell*. URL: https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d.

Konda, Vijay R and John N Tsitsiklis (2000). "Actor-critic algorithms". In: *Advances in neural information processing systems*, pp. 1008–1014.

Lillicrap, Timothy P. et al. (2015). *Continuous control with deep reinforcement learning*. arXiv: 1509.02971 [cs.LG].

Lin, Guifang and Wei Shen (2018). "Research on convolutional neural network based on improved Relu piecewise activation function". In: *Procedia computer science* 131, pp. 977–984.

Mao, Hongzi et al. (2016). "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, pp. 50–56.

Menascé, Daniel A (2002). "TPC-W: A benchmark for e-commerce". In: *IEEE Internet Computing* 6.3, pp. 83–87.

Nogueira Vitor Carnaz, Gonçalo (2019). *An Overview of IoT and Healthcare*.

OpenAI (n.d.). *Deep Deterministic Policy Gradient*. URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

Sainath, Tara N et al. (2015). "Convolutional, long short-term memory, fully connected deep neural networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 4580–4584.

Schöbel, Rainer and Jianwei Zhu (1999). "Stochastic volatility with an Ornstein–Uhlenbeck process: an extension". In: *Review of Finance* 3.1, pp. 23–46.

Silver, David et al. (2014). "Deterministic policy gradient algorithms". In:

Sutton, Richard S et al. (2000). "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*, pp. 1057–1063.

Yu, F Richard and Ying He (2019). "Reinforcement Learning and Deep Reinforcement Learning". In: *Deep Reinforcement Learning for Wireless Networks*. Springer, pp. 15–19.

Yue, Yisong and Thorsten Joachims (2009). "Interactively optimizing information retrieval systems as a dueling bandits problem". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 1201–1208.

Zheng, Guanjie et al. (2018). "DRN: A deep reinforcement learning framework for news recommendation". In: *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, pp. 167–176.