Reinforcement learning for supply chain optimization

Anonymous Authors

Abstract—**In this paper we investigate the performance of two reinforcement learning (RL) agents within a supply chain optimization environment. In section III we model the environment as a markov decision process (MDP) where during each step it needs to be decided how many products should be produced in a factory and how many products should be shipped to different warehouses. We then design three different agents based on a static (ς, Q)-policy, the approximate SARSA- and the REINFORCE algorithm which we describe in section IV. Here we pay special attention to different feature mapping functions that are used to model the value of state and state-action pairs respectively. We then evaluate the agents performance by exposing them to different scenarios of our environment that we initialize by defining different parameter settings. We find that both the approximate SARSA and the REINFORCE algorithm can outperform the static (ς, Q) agent and that the REINFORCE agent tends to perform best in the long run.**

I. Introduction

In the following paper we implement three different reinforcement learning agents and evaluate their performance within a supply chain environment. All the code used to create the results presented in this paper can be found here [1].

The supply chain optimization environment represents a problem faced by companies whose supply chain consists of a factory and multiple warehouses (so called hub-and-spoke networks as described by [4]). The main decision that needs to be made in these settings is how many products should be produced in the factory and how much stock should be build up in the warehouses. Seasonal demand can further complicate the decision problem since it might require the companies to start building up stock early so that high demands in the future can be satisfied (e.g. think about christmas where the stock of eggnog needs to be build up during november and december to be able to satisfy the high demand levels during the holidays). This requirement for mid- to long-term planning indicates that the problem can be formulated as a multi-step decision problem and further exhibits a weak signal because the reward for an early build-up of stock will only be realized in the future when the actual demand reaches its maximum.

Even for small supply chain networks we find that due to the curses of dimensionality, as described by [2], the state- and action-spaces of the respective decision problems become unfeasebly large. Therefore we turn to function-approximation and policy-search methods of reinforcement learning that are less affected by these problems. In our case we chose approximate SARSA and the REINFORCE algorithm as a basis for the agents.

[1]Our code is available here:https://uploadfiles.io/je31f

II. Background and Related Work

The supply chain optimization problem presented in this paper will be modeled as a markov decision process. Furthermore we rely on the approximate SARSA and REINFORCE algorithms to design agents capable of finding a good strategy to navigate within the environment.

MDPs are multi-step stochastic decision problems that rely on the markov property which implies that the transition probability $P(s_{n+1} \mid s_n)$ between two states s_{n+1} and s_n only relies on the current state s_n. We describe the MDP similar to [1] and [2] as an eight-tuple $(S, D, A, \mathcal{X}, T, Q, r, \gamma)$ consisting of a state-space S, a random environment process D, an action space A, a set of feasible actions $\mathcal{X}$, a transition function T, transition probabilities Q, a one-step reward function r and some discount factor γ.

The designed agents are based on the blueprints for the approximate SARSA and REINFORCE algorithm from [3]. We compare the performance of these reinforcement learning algorithms with an agent that acts according to a fixed heuristic based on the (ς, Q)-Policy [2] as described by [5]. (ς, Q)-Policies induce a simple replenishment strategy where the stock of a warehouse will be replenished by some amount Q as soon as it falls under a threshold ς. A basic visualization of the (ς, Q)-Policy is depicted in Figure 1.

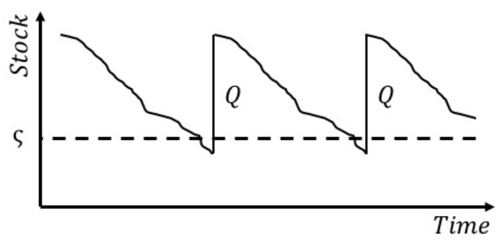


Fig. 1: Schematic visualization of a (ς, Q)-Policy for a single warehouse.

Furthermore we use several different stepsize formulars from [2] that are used to update some of the agents parameters such as the learning rate α. The idea of specific stepsize rules is to allow the agent to go through a longer learning phase in the beginning with higher learning rates and then converge in the long term to stepsizes close to 0. Among others we use a so called search-then-converge stepsize rule as presented in [2]

[2]In the literature this policy is usually called (ς, Q)-Policy. We chose a slightly different name to avoid notation conflicts in our model description.

where the learning rate in the n^{th} iteration is calculated by

$$\alpha_n = \alpha_0 \frac{\frac{b}{n} + a}{\frac{b}{n} + a + n^{\beta}} \tag{1}$$

with some parameters α_0, b, a and β that can be set by the user based on the required behavior of the stepsize.

III. The Environment

We will model the environment as a MDP that we regard over $N \in \mathbb{Z}$ periods $t = 0, 1, ..., N$. The environment consists of one factory (indexed by 0) and up to $k \in \mathbb{N}$ warehouses (indexed by $j = 1, ..., k$) where in each period it needs to be decided how many units of a product (e.g. butter) should be produced and how many should be shipped to the individual warehouses. A representation of a network with 5 warehouses is depicted in Figure 2. The state space

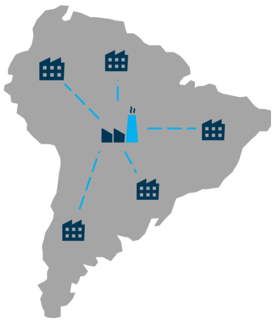


Fig. 2: Example of a supply chain network with $k = 5$ warehouses $(s_1, .., s_5)$ and the factory (s_0).

consists of $k+1$ elements describing the current stock levels of the factory and the warehouses $(s_j, j = 0, ..., k)$, where each stock level is limited by some capacity $c_j \in \mathbb{N}$. Note that this implies that the factory itself can also build up stock. Furthermore we assume an environment process with states $d \in D \in \{0, ..., d_{max}+1\}^k, d_{max} \in \mathbb{N}$ describing the (stochastic) demand d_j at each warehouse with transition probabilities $q_d(d')$ that are unknown by the agent. We assume a discretized version of a sinus-function and some shocks ϵ where the demand follows a model

$$d_{i,t} = \left\lceil \frac{d_{max}}{2} \sin\left(\frac{2\pi(t+2i)}{12}\right) + \frac{d_{max}}{2} + \epsilon_{i,t} \right\rceil \tag{2}$$

with $\lceil \cdot \rceil$ the ceiling function and $P(\epsilon_{i,t} = 0) = P(\epsilon_{i,t} = 1) = 0.5$. We will include $d_{last}, d_{hist} \in D$ in our state space where for some time t we have the tuple $\tilde{s} = (s, d_{last}, d_{hist}) \in \tilde{S} := S \times D^2$ with $s = s_t$, $d_{last} = d_{t-1}$ and $d_{hist} = d_{t-2}$. Note that this implies that the demand d_t of period t is not observed by the agent. This means, that the demand in a period t will be realized after observing the stock levels s_t and can only be observed in the next period $t+1$. The reason we add the old demands to the state space is to allow the agent to have a simple understanding of the demand history to be able to gather a basic understanding of the demands movement. In each period the agent can now set the factories production level for the next period $a_0 \in \{0, .., \rho_{max}\}$ (with a maximum production of $\rho_{max} \in \mathbb{N}$) aswell as the number of products shipped to each location $a_j \in \mathbb{N}^k$ that is naturally limited by the current storage level in the factory ($\sum_{j=1}^k a_j \leq s_0$). Based on this information and the demand d we can now describe the state transitions $(\tilde{s}, d, a) \to T(\tilde{s}, d, a)$ by

$$\begin{aligned} T(\tilde{s}, d, a) := (\, &\min\{s_0 + a_0 - \sum_{j=1}^k a_j, c_0\}, \\ &\min\{s_1 + a_1 - d_1, c_1\}, \\ &... \\ &\min\{s_k + a_k - d_k, c_k\}, \\ &d, \\ &d_{last}). \end{aligned} \tag{3}$$

The reward within each period consists of the revenue from sold products at a fix price p less production costs $\kappa_{pr} a_0$, storage costs $\sum_{j=0}^k \kappa_{st,j} \max\{s_j, 0\}$, penalty costs $\kappa_{pe} \sum_{j=1}^k \min\{s_j, 0\}$ and transportation costs $\sum_{j=1}^k \kappa_{tr,j} \lceil a_j / \zeta_j \rceil$. We let $p, \kappa_{pr}, \kappa_{pe}, \kappa_{st,j}, \kappa_{tr,j} \in \mathbb{R}_+$ and $\zeta_j \in \mathbb{N}$. We can now define the one-step reward function by

$$\begin{aligned} r(\tilde{s}, d, a) :=& p \sum_{j=1}^k d_i - \kappa_{pr} a_0 - \sum_{j=0}^k \kappa_{st,j} \max\{s_j, 0\} \\ &+ \kappa_{pe} \sum_{j=1}^k \min\{s_j, 0\} - \sum_{j=1}^k \kappa_{tr,j} \left\lceil \frac{a_j}{\zeta_j} \right\rceil. \end{aligned} \tag{4}$$

The terminal reward after period $t = N$ is 0 meaning that we will not account for remaining positive or negative stock. Furthermore we chose a discounting factor $\gamma \in \mathbb{R}_+$ that can be interpreted e.g. as a result of inflation.
Based on this model description we can now formulate the task as a finite horizon markov decision process that is subject to a varying environment. Thereby we receive the tuple $(\tilde{S}, D, A, \mathcal{X}, T, Q, r, \gamma)$ with

Definition 1.

1. $\tilde{S} \times D := \prod_{j=0}^k \{s_j \in \mathbb{Z} \mid s_j \leq c_j\} \times D^3, c_j \in \mathbb{N}$ the state space consisting of elements $(\tilde{s}, d) = ((s, d_{last}, d_{hist}), d)$;
2. $A := \{0, ..., \rho_{max}\} \times \mathbb{N}_0^k$ the action space consisting of elements $a = (a_0, ..., a_k)$;
3. $\mathcal{X}(\tilde{s}) := \{0, ..., \rho_{max}\} \times \{a \in \mathbb{N}_0^k \mid \sum_{j=1}^k a_j \leq s_0\}$, the set of all feasible actions in a state $\tilde{s}$ and $\mathcal{X} := \{(\tilde{s}, d, a) \in \tilde{S} \times D \times A \mid a \in \mathcal{X}(\tilde{s})\}$;
4. $T : \tilde{S} \times D \times A \to S$ the transition function defined by Eq. (3);
5. $Q : \mathcal{X} \times \tilde{S} \times D \to [0, 1]$ the transition probabilities with $Q(\tilde{s}', d' \mid \tilde{s}, d, a) := q_d(d')$ for $\tilde{s}' = T(\tilde{s}, d, a)$ and 0 otherwise;
6. $r : \tilde{S} \times D \times A \to \mathbb{R}$ the one-step reward function as described in Eq. (4);
7. $\gamma \in \mathbb{R}_+$ the discounting factor.

For the calculation of the expected discounted revenue for each tuple $(\tilde{s}, d)$ we obtain the value function V_n with

$$
\begin{aligned}
V_n(\tilde{s}, d) = \max_{a \in \mathcal{X}(\tilde{s})} &\{r(\tilde{s}, d, a) \\
&+ \gamma \sum_{d' \in D} q_d(d') V_{n+1}(T(\tilde{s}, d, a), d')\}, \\
&\tilde{s} \in \tilde{S}, d \in D, 0 \le n \le N-1.
\end{aligned} \tag{5}
$$

IV. The Agent

In this paper we evaluate the performance of three agents based on different algorithms: a heuristic based on the (ς, Q)-Policy that we use as the baseline for performance evaluation, an approximate SARSA algorithm and an implementation of the REINFORCE algorithm.

The (ς, Q)-Policy based agent is not smart in the way that it does not learn over time. Due to the popularity of the (ς, Q)-Policy in practice we use it as a baseline for performance evaluation of the other agents. In the heuristic we iterate over $s_1, ..., s_k$ and replenish the respective warehouses by some amount $a_i = Q_i$ if the current stock is below a level ς_i and there is still stock left in the factory s_0. At the end we set the production level for the next period to Q_0 if $s_0 - \sum_{i=1}^{k} a_i < \varsigma_0$ and 0 otherwise. The thresholds ς and replenishment levels Q need to be set by the user when initializing the agent.

Approximate SARSA uses a linear approximation $Q_\theta(\tilde{s}, a) = \theta^T \phi(\tilde{s}, a)$ of the Q-function. We chose this method as it solves the problem of exponentially growing state- and action- spaces that is likely to occur in multidimensional environments. Furthermore it allows us to use our knowledge of the environment (especially the structure of the reward function $r(\tilde{s}, d, a)$) to design $\phi(\tilde{s}, a)$ such that it preserves some of the MDPs structure. The algorithm is implemented as depicted in [3] where we use an ϵ-Greedy strategy to select the action and a search-then-converge stepsize rule to update the learning rate α and the probability of chosing a random action ϵ as defined in (3).

One of the crucial tasks when designing the approximate SARSA agent is the model for $\phi : \tilde{S} \times A \to \mathbb{R}^p, p \in \mathbb{N}$. This can be understood similar to feature engineering where we use some state s and some action a to compute p features that should be capable of describing $Q(s, a)$. In our case we create over 15 different types of features that can be binary as well as continuous. One of the main ideas is to get a rough estimate of the next demand and state by

$$
\begin{aligned}
\hat{d}(\tilde{s}) &= d_{last} + (d_{last} - d_{hist}) = 2d_{last} - d_{hist}, \\
\hat{s}(\tilde{s}, a) &= T(\tilde{s}, \hat{d}(\tilde{s}), a)_{(d, d_{last})},
\end{aligned} \tag{6}
$$

where $T(\cdot)_{(d, d_{last})}$ is the transition function excluding its last two elements. This way the agent can get a basic understanding of rewards and penalties associated with $\hat{s}$. Among others we use the expected penalty costs, the expected reward and the squared difference of the stock levels s from their $25\%, 50\%$ and 75% maximum capacity. Furthermore we include the respective rewards and costs for two scenarios where $\hat{d}^+ = \hat{d}(\tilde{s}) + 1$ and $\hat{d}^- = \hat{d}(\tilde{s}) - 1$. While designing the agent we ended up with two different functions ϕ and ϕ' that affect the agents behavior. Both functions have been tested and are included in our code. We encourage the reader to try out both. A full list of all features can be found in the appendix.

When testing the approximate SARSA algorihm we found that for some environments the parameters θ would increase until computations became numerically unstable. To avoid this issue we restrict the temporal difference

$$
\delta_n = r(\tilde{s_n}, d_n, a_n) + \gamma \theta_n^T \phi(\tilde{s}_{n+1}, a_{n+1}) - \theta_n^T \phi(\tilde{s}_n, a_n) \tag{7}
$$

that is used to update θ_n within the intervall $[-1\text{e}100, +1\text{e}100]$. Furthermore we initialize the agent with a very small stepsize α. **REINFORCE** is based on a parametrized policy for which the expected reward has to be maximized. We assume our policy to be parametrized as a softmax function for multiple actions. The softmax function assigns a probability for each action and thus behaves as a stochastic policy which can, however, converge to a deterministic one. We discretize the action space such that we only have three available actions per location (for each warehouse and the factory). For a warehouse, for example, these possible actions are an order of 0,1 or 2 full trucks from the factory. For a given state, some actions might not be available. Since a projection of an unfeasible action to the closest feasible action is not possible in our environment, we used an alternative approach which only updates parameters for available actions. This leads to the following policy parametrization:

$$
\begin{aligned}
&p(a = a^{(i)}|s) = \pi_\Theta(a = a^{(i)}|s) = \\
&= \frac{e^{\phi(s)^T w_{a^{(i)}}} \cdot f(a^{(i)}|s)}{\sum_{j=1}^{n_a} e^{\phi(s)^T w_{a^{(j)}}} \cdot f(a^{(j)}|s)} := \sigma_i(s)
\end{aligned} \tag{8}
$$

where

$$
f(a^{(j)}|s) = \begin{cases} 1 & \text{if } a^{(j)} \text{ is allowed in state s} \\ 0 & \text{otherwise} \end{cases} \tag{9}
$$

As we allow 3 different actions per entity, the number of possible actions is $n_a = 3^{(k+1)}$. The feature map $\phi(s) : \mathbb{Z}^{k+1} \to \mathbb{R}^d$ maps the state into a space of dimension d in which an optimal action is easily seperable from the other ones. We tried several different options for $\phi(s)$ including first and second order terms and radial basis functions. We always include a bias term as well. The parameter matrix Θ is assembled in the following way:

$$
\Theta = \left(w_1 | w_2 | ... | w_{n_a}\right) \in \mathbb{R}^{d \times n_a} \tag{10}
$$

where $w_i \in \mathbb{R}^d$ (including a bias). We initialized Θ to zeros in order to start with equal probabilities for each action.

A gradient ascent method is used to find the parameters that maximize the expected reward following this policy. The gradient for our parametrization evaluates to:

$$
\nabla_{w_j} ln(\pi_\Theta(a^{(i)}, s)) D_t = \begin{cases} (1 - \sigma_j(s)) \phi(s) D_t & \text{if } i = j \\ -\sigma_j(s) \phi(s) D_t & \text{if } i \ne j \end{cases}
$$

The full derivation for this gradient is shown in the Appendix. A key parameter for this algorithm is the learning rate which we left constant, but close to zero since a small learning rate allows a lot of exploration which is needed for all parameters to adapt.

V. Results and Discussion

A. Performance of your Agent in your Environment

To test the algorithms, two main test were done. All of the tests were computed using 10000 episodes of 24 steps each in which our demand distributions make a whole cycle, allowing it to have the full seasonal demand.
The first test is a simple test, having only the factory and one warehouse, with no storage cost in the factory and small cost of production, transportation and storage in the warehouse. The objective of this test was to see if the agents can learn that even if producing, sending and storing is a bad decision in short term, because there is a penalty cost that accumulates for not sending the goods to the warehouse, the overall strategy should be to do it anyway. But because there is a cost of production, the optimal strategy would be to not build as much as we can but to build enough to be able to satisfy the season of high demand but no more than that.

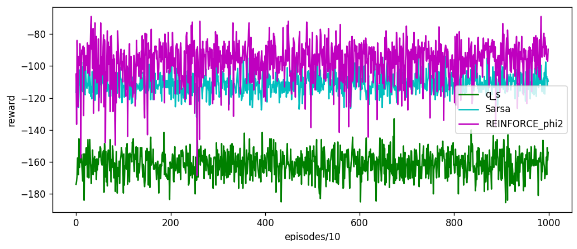


Fig. 3: Rewards over time for the three agents in the first test.

The graph seen in Figure 3 shows that both the approximate SARSA algorithm and the REINFORCE can learn this task, with the REINFORCE algorithm having slightly better results and with notable better results that the (ς, Q)-Policy based agent, who sends and produces goods before they are needed and ends up getting a higher penalty. This shows that the basis of our environment can be learned by our agents even if they are only getting weak signals.
The second test is a harder environment. In this environment the number of warehouses went up to 3, each with different properties. The first warehouse has a high storage and transportation cost, the second warehouse has no storage cost but high transportation cost and the third one no transportation nor storage cost. Furthermore, because of how the production and the demand are balanced there are not enough goods to satisfy all the demand, the agent need to learn everything he learned in the simple scenario but also to prioritize some warehouses before others. This second graphs seen in Figure 4 shows the (ς, Q)-Policy based agent who begins with better results thanks to his simple yet good solution, but after all the training the REINFORCE algorithm manages to learn a better strategy. As shown in Figure 5, the storage levels in the differents stores for the (ς, Q)-Policy strategy are pretty similar while on the REINFORCE we have the warehouse 1 who has at beginning almost zero stock due to its high storage cost, and when the demand is too high the first two warehouses are forfeit to the third one, because of its zero cost of storage and transportation. The SARSA on the other hand does not manage to learn

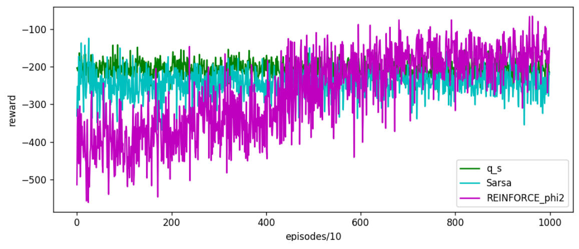
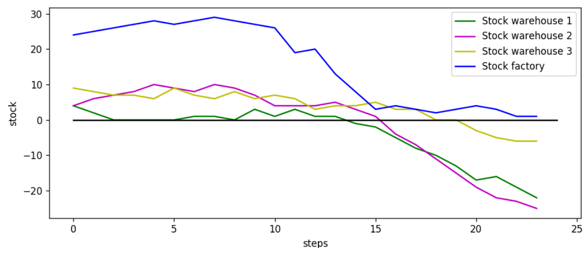
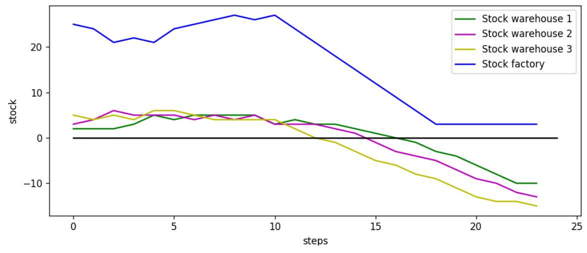


Fig. 4: Rewards over time for the three agents in the second test.



(a) Stocks for the (ς, Q)-Policy based agent.



(b) Stocks for the REINFORCE agent.

Fig. 5: Stocks in the second test

the right strategy, building less stock during the low demand and not prioritizing early warehouse n3. This comes from the fact that the SARSA algorithm is really dependant on his set of features and can only learn strategies that could be obtained with a right combination of weights for this features. And because this weight can't be positive and negative at the same time, a simple feature like ”storage level” can't tell the algorithm how much to store but only that storing is good or bad. So a balance between the features needs to be reached, where the agent can say that producing is good unless there is too much. This balancing of feature is hard to learn and provoques bad results in too complicated scenarios. This might be solved with a different combination of features but for this particulars parameters none was found.
Other test also showed a small issue for the REINFORCE learning. Because the actions provided are limited to sending 0, 1 or 2 full trucks, there is a problem in some scenarios if the maximum posible production is notably lower than the truck capacity. This issue could be solved by chanching the posible actions of the REINFORCE algorithm but require some understanding of scenario and is not ideal. The readers are encouraged to create their own scenarios in *testing.py* to see the different performances of the agents. To compare their performances or recreate the graphs seen in this paper *graph_creation.py* needs to be run.

B. Performance of your Agent in the ALife Environment

Our implementation of the approximate SARSA algorithm uses a very specific value function approximation that depends strongly on assumptions about the underlying environment. Since this environment has no structural similarity with the ALife[3] environment it can not be used for testing. To adapt the algorithm to the ALife environment it would be necessary to create new features and design a different function ϕ to be used within the existing agent.
In general, the REINFORCE approach is very nicely applicable to the Alife environment as was shown in the lab. Allowing multiple possible actions can also be a reasonable extension. Our implementation, however, is not directly transferable, since we use additional information about which actions are possible which is not available in the Alife environment.

VI. Conclusion and Future Work

The REINFORCE approach is certainly a viable option for this environment as it performs well in each scenario. Of course, the algorithm does have some downsides. As a gradient ascend algorithm it can easily get stuck in local optima which can easily arise in this most likely non-convex setting. Additional drawbacks come from the softmax parametrization which requires parameters for each action. This requires a lot of training and can lead to difficulties for actions that are rarely feasible. Moreover, the actions to replenish 0, 1 or 2 trucks are treated as completely separate, dropping the possibility to make use of their natural order. Another parametrization that would make use of this order would be e.g. a Gaussian setting. Unfortunately, our implementation of using independent Gaussians to parametrize the actions for each entity individually failed to show reasonable learning and was therefore dropped from this report.

This section summarizes the paper: Your environment and agent, its strength and its weaknesses. Also remark about what would be the next steps you would take if you or someone else were to continue/extend this project. Note that for the initial submission you are limited strictly to 4 pages (double column), *not including references*. An extra page will be allowed for final submission (after the initial reviews).

VII. Appendix

Derivation of the gradient for REINFORCE:

$$\nabla_{w_j} J(\Theta) = \nabla_{w_j} ln(\pi_\Theta(a(t) = a^{(i)}, s(t)))D_t$$

where for $i = j$

$$\begin{aligned}
&\nabla_{w_j} ln(\pi_\Theta(a(t) = a^{(i)}, s)) = \\
&= \nabla_{w_j}(\phi(s)^T w_{a^{(i)}}) + \nabla_{w_j} ln(f(a^{(i)}|s)) + \\
&-\nabla_{w_j} ln(\sum_{l=1}^{n_a} e^{\phi(s)^T w_{a^{(l)}}} \cdot f(a^{(l)}|s)) = \\
&= \phi(s) + 0 - \frac{e^{\phi(s)^T w_{a^{(j)}}} \cdot f(a^{(j)}|s)}{\sum_{l=1}^{n_a} e^{\phi(s)^T w_{a^{(l)}}} \cdot f(a^{(l)}|s)} \cdot \phi(s) = \\
&= (1 - \sigma_j(s)) \cdot \phi(s)
\end{aligned} \tag{11}$$

[3]https://github.com/jmread/alife

and for $i \neq j$:

$$\begin{aligned}
&\nabla_{w_j} ln(\pi_\Theta(a(t) = a^{(i)}, s)) = \\
&= \nabla_{w_j}(\phi(s)^T w_{a^{(i)}}) + \nabla_{w_j} ln(f(a^{(i)}|s)) + \\
&-\nabla_{w_j} ln(\sum_{l=1}^{n_a} e^{\phi(s)^T w_{a^{(l)}}} \cdot f(a^{(l)}|s)) = \\
&= 0 + 0 - \frac{e^{\phi(s)^T w_{a^{(j)}}} \cdot f(a^{(j)}|s)}{\sum_{l=1}^{n_a} e^{\phi(s)^T w_{a^{(l)}}} \cdot f(a^{(l)}|s)} \cdot \phi(s) = \\
&= -\sigma_j(s) \cdot \phi(s)
\end{aligned} \tag{12}$$

References

[1] L. N. Moritz. Target Value Criterion in Markov Decision Processes, *Karlsruhe, KIT, Diss.*, 2014.

[2] W. B. Powell, Approximate Dynamic Programming: Solving the curses of dimensionality, *John Wiley & Sons*, 2007.

[3] J. Read. Lecture IX - Deep reinforcement learning. *INF581 Advanced Topics in Artificial Intelligence*, 2018.

[4] K. Furmans, D. Arnold, Materialfluss in Logistiksystemen, *Springer Berlin Heidelberg*, 2009.

[5] H. Tempelmeier, Inventory management in supply networks: problems, models, solutions, *Books on Demand*, 2011.