Autonomous Lane Merging: A Comparison Between Reinforcement Learning Algorithms

Abstract

Despite the advancements of self-driving cars, autonomous on-ramp merging on highways still proposes difficulties. To solve this merge problem a simulation was set up in the Unity game engine and an agent was trained using two state of the art reinforcement learning algorithms, Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC), utilizing the Unity Learning Agents Toolkit ML-Agents. The two algorithms are compared to each other with respects to training speed, performance, stability and success rate. The robustness of the algorithms were tested by having the traffic (1) vary in speed, (2) vary in starting positions and (3) switch lanes. The agent had a similar performance with a success rate of 95% when employing either PPO or SAC. Both algorithms showed their advantages and disadvantages. PPO had a more stable performance and less variability in mean reward, while SAC was more sample efficient. Results show that reinforcement learning is an avenue worth pursuing to reach fully autonomous driving. Improvements to the results could still be made through hyperparameter tuning, more complex neural network setup and a more realistic simulation, further proving the advantage of reinforcement learning.

Key words: reinforcement learning; Proximal Policy Optimization; Soft Actor-Critic; autonomous car; lane merging; Unity

Pieter El Sharouni (5930499)

Supervisor: Natasha A. Alechina

Bachelor Thesis Artificial Intelligence

Department of Humanities, University Utrecht

Introduction

In late August 2018, Apple's automated driving project experienced a public-road collision for the first time. A vehicle rear-ended the automated vehicle when it came to a near-stop while waiting for an opportunity to merge onto a multi-lane highway (Dollar & Vahidi, 2019). Merging is a complex and challenging problem for driver assistance. Despite the advancement of automation levels, the implementation of autonomous driving for highway on-ramp merge still presents considerable challenges (Wang & Chan, 2017). In the near future, it is almost certain that self-driving cars will become part of everyday traffic. It is therefore pertinent that this merge-problem is solved and to eventually implement fully autonomous vehicles, which will presumably result in fewer collisions or other traffic issues.

Several modelling methods have been suggested to solve the autonomous on-ramp merging problem. One method was a slot-based merging algorithm, which defined a slot occupancy status, either free or occupied, based on the agents' speed, position, and driving behaviour in terms of acceleration and deceleration (Marinescu et al., 2012). A theory about driving rules and gap acceptance was modelled for a decision-making process for the merge problem on an urban expressway (Wang & Chan, 2017). These rule-based models are able to handle the merge-problem in theory but lack the flexibility necessary to adapt to new situations. Their ability to merge can only be successful in predefined rules and states, and may therefore fail when confronted with unforeseen scenarios.

Contrary to rule-based models, machine learning models can have the potential to deal with the complex situations of traffic without resorting to predefined rules or models. Particularly, reinforcement learning can learn optimal actions by itself through trail and error. A reinforcement agent observes its environment, interacts with it by predefined actions, and is rewarded or punished for these actions according to the goal state (Lin, 1993). By maximizing the reward, the agent will find the optimal policy to achieve its goal, and will therefore

successfully merge. Reinforcement learning algorithms already have had some success in this merge problem using deep Q-learning (Wang & Chan, 2017; P. Wang & Chan, 2018).

In the current study two modern reinforcement learning algorithms will be utilized. The reason for this is twofold, to evaluate the ability of reinforcement learning algorithms to solve the merge problem and to compare the two algorithms. The first algorithm is Proximal Policy Optimization (PPO), which was introduced by the OpenAI team in 2017 and has had huge success in very complex environments such as the MOBA Dota2, one of the most complex and popular E-sport games to date (*Proximal Policy Optimization*, 2017; Schulman et al., 2017). This algorithm will be compared to the Soft Actor-Critic (SAC), an off-policy actor-critic deep reinforcement learning algorithm based on the maximum entropy reinforcement learning framework. With the SAC algorithm the agent aims to maximize expected reward while also acting as randomly as possible. (Haarnoja et al., 2018). PPO and SAC were chosen because the problem consists of continuous action and decision space. Compared to other on-policy reinforcement learning algorithms, PPO seems to be easier to implement and is at least on par or outperforms similar reinforcement learning algorithms (Schulman et al., 2017; Wang et al., 2020). While SAC is not as easily implemented, by combining off-policy updates with a stable stochastic actor-critic formulation, it could outperform other on-policy and off-policy methods (Haarnoja et al., 2018).

Reinforcement learning

In a reinforcement learning problem, an agent generates its own training data by interacting with the environment based on its current policy. Therefore the data distributions of the observations and rewards are constantly changing as the agent learns, which can cause instability in the learning process (Lin, 1993). The environment is typically stated in the form of a Markov Decision Problem (MDP) with a set of agent states in the environment, S; a set of

actions of the agent, *A*; the probability of a transition at time *t* from state *s* to *s'* under action *a*, $\Pr(s_{t+1} = s' | s_t = s, a_t = a)$ and the immediate reward after the transition from *s* to *s'* with action *a*, $R_a(s, s')$. The goal of the reinforcement learning agent is to learn a policy $\pi : A \times S \rightarrow [0,1], \pi(a, s) = \Pr(a_t = a | s_t = s)$, which maximizes the expected cumulative reward. However, in the more complex environment of the simulation there are no discrete states actions or immediate rewards, therefore a Markov Decision Problem is too simplistic and deep reinforcement learning has to be used (Denardo, 1973).

In deep reinforcement learning, the observations collected by the agent are fed into a neural network, which outputs an action based on its current policy. At the end of an episode, the agent earns a reward based on its actions and states which acts as a feedback mechanism to the neural network, which will either increase of decrease the likelihood of the actions given a certain state. Reinforcement learning does suffer from high sensitivity to hyperparameter tuning such as the learning rate and initialization. If the learning rate is too high, the current policy will be updated with the possibility of pushing the policy network into a region of the parameter space where its next batch of data is collected under suboptimal policy, creating fewer possibilities of recovery. PPO deals with these problems by being easy to implementation and is easily tuned. PPO is an on-policy gradient method, meaning that unlike popular Q-learning methods such as Deep Q-Networks that learn from stored offline data, PPO learns directly from the experiences it gathers in its environment. Contrary to this, SAC is an off-policy method, meaning its agent learns from stored data the agent encountered in past episodes that are stored in a buffer. Therefore SAC utilizes the experience from previous episodes, while in on-policy methods the agent is limited to the current batch of experience. Before the performance of the algorithms are compared, both algorithms will first be explained to give an idea how they work and how they differ. First the PPO algorithm will be explained, which is defined by two parts, the policy gradient loss and the trust region.

Policy Gradient Methods

In Proximal Policy Optimization methods the policy gradient loss (1) is defined first, which allows the increase of actions that yield a positive result and the decrease the actions that yield a negative reward.

$$L^{PG}(\theta) = \hat{\mathrm{E}}_t[\log \pi_\theta(\alpha_t|s_t)\hat{\mathrm{A}}_t] \quad (1)$$

Here, $\pi_\theta(\alpha_t|s_t)$ is the action α_t taken by the current policy π_θ at time t, given state s_t with policy parameters θ. The loss function $L^{PG}(\theta)$ is the estimate reward of the action at time t, $\hat{\mathrm{E}}_t$, and a second term, the advantage function $\hat{\mathrm{A}}_t$ which estimates the relative value of the selected action at time t. The advantage function consist in two parts, the discounted sum of rewards and the baseline estimate. The discounted sum of rewards (2) is a weighted sum of all the rewards the agent receives during each timestep in the current episode.

$$\sum_{k=0}^{\infty} \gamma^k r_{t+k} \quad (2)$$

The discount factor γ, often between 0.9 and 0.99, indicates the prioritization of rewards in close proximity above rewards set further ahead. The sum of all the rewards from time t until $t + \infty$ is calculated. The rewards are multiplied by the discount factor that corresponds with the amount of timesteps ahead. All the rewards are known due to the fact that the advantage function is calculated after the episode sequence is collected from the environment.

The second part of the advantage function is the baseline function, which is a neural network. The baseline function calculates an estimate of the discounted return from its current position. It estimates the expected reward at the end of the episode, based on previous

experiences. The input of the neural network are the states and the output of the neural network is the predicted discounted sum of rewards. Because the discounted sum of rewards is calculated at the end, the prediction from the baseline function can be compared to the actual discounted sum of rewards. The neural network is adjusted to correspond to the actual reward, creating a supervised learning problem. The advantage function is calculated by subtracting the baseline estimate from the actual discounted reward. The final optimization objective is calculated by multiplying the probability of the policy actions with this advantage function. If the actions taken by the agent resulted in a higher than average return, the probability of the actions are increased in the future.

Trust Region Methods

One of the problems of gradient descent is that the parameters can be updated excessively, creating a suboptimal policy from which the agent will gather suboptimal data. Trust Region Policy Optimization (TRPO) ensures that when updating the policy, it does not stray too far from the old policy.

$$maximize_\theta \ \hat{\mathrm{E}}_t \left[\frac{\pi_\theta(\alpha_t|s_t)}{\pi_{\theta old}(\alpha_t|s_t)}\right] \hat{\mathrm{A}}_t$$
$$subject\ to\ \hat{\mathrm{E}}_t\big[KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)]\big] \leq \delta \tag{3}$$

Because the policy is stochastic and not deterministic, the actions chosen by the policy are denoted as $\pi_\theta(.|s_t)$. To ensure the policy is not updated excessively, a KL constraint is added to the optimization objective, limiting the amount of alteration to the policy. However, the KL constraint does add limitations to the optimization process and can sometimes result in

undesirable training behavior. PPO includes this extra constraint directly in the optimization objective to alleviate this issue (Schulman et al., 2017).

PPO

Let $r_t(\theta)$ determine the ratio between the new updated policy and the previous old policy. This ratio will be greater than 1 if the action is more likely now than under the old policy. For more readability this ratio $r_t(\theta)$ can be multiplied with the advantage function (4).

$$L^{CPI}(\theta) = \hat{\mathrm{E}}_t\left[\frac{\pi_\theta(\alpha_t|s_t)}{\pi_{\theta old}(\alpha_t|s_t)}\hat{\mathrm{A}}_t\right] = \hat{\mathrm{E}}_t\left[r_t(\theta)\hat{\mathrm{A}}_t\right] \quad (4)$$

Without a constraint, maximizing $L^{CPI}(\theta)$ would lead to excessively large policy updates. Hence the objective function can be modified (5), to penalize changes to the policy that move $r_t(\theta)$ away from 1,

$$L^{CLIP}(\theta) = \hat{\mathrm{E}}_t\left[\min\left(r_t(\theta)\hat{\mathrm{A}}_t, \mathrm{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\,\hat{\mathrm{A}}_t\right)\right] \quad (5)$$

where ε is a hyperparameter, say ε = 0.2. The objective function that PPO optimizes is an expectation operator, $\hat{\mathrm{E}}_t$, computed over batches of experiences. This expectation operator is taken over the minimum of two terms. The first of these terms is $r_t(\theta)\hat{\mathrm{A}}_t$, which ensure the policy takes actions that yield a high positive advantage over the baseline. The second term, $\mathrm{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{\mathrm{A}}_t)]$ modifies the objective by clipping the probability ratio, which removes the possibility for r_t to move away from the interval $[1-\varepsilon, 1+\varepsilon]$. The advantage estimate can be both positive and negative, which changes the effect of the min operator. As seen in Figure 1, the probability ratio is clipped whether the advantage is positive or negative.

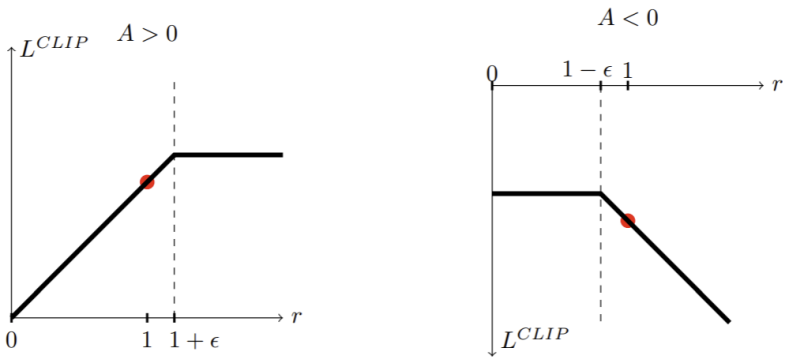

Figure 1. the effect of the advantage function on the clipping functionality, taken from Schulman et al. (2017).

The pseudocode for the PPO algorithm is shown in Figure 2. Each iteration, each of the *N* agents, in this case only 1, collects *T* timesteps of experience data. These experiences are collected and gradient descent is run on the policy network for each batch of experience for *K* epochs, using the clipped PPO objective, updating the policy (Schulman et al., 2017).

Algorithm 1 PPO

---

1. **for** iteration = 1, 2,…. **do**
2. **for** actor = 1, 2,…, *N* **do**
3. Run policy $\pi_{\theta old}$ in environment for *T* timesteps
4. Compute advantage estimates $\tilde{A}_1,\ldots,\tilde{A}_T$
5. **end for**
6. Optimize surrogate *L* wrt θ, with *K* epochs and minibatch size $M \leq NT$
7. $\theta_{old} \leftarrow \theta$
8. **end for**

Figure 2. The pseudocode of the PPO algorithm (Schulman et al., 2017).

Soft Actor-Critic

Soft Actor-Critic is an off-policy algorithm with a central feature in entropy regularization. The SAC policy tries to maximize the balance between the expected return and entropy, a measure of randomness in the policy. This is similar to the exploration-exploitation trade-off, which may increase learning rate later on, but may also prevent the policy from converging earlier.

To explain Soft Actor-Critic, the entropy-regularized reinforcement learning setting is introduced first.

Entropy can be seen as the amount of randomness of a variable. If x is a random variable then P is the probability function. The entropy H of x is calculated from this distribution P (6).

$$H(P) = \underset{x\sim P}{\mathrm{E}}[-\log P(x)] \tag{6}$$

In entropy-regularized reinforcement learning, the agent is awarded a bonus every timestep in in line with the entropy of the policy at the corresponding timestep, changing the reinforcement learning problem (7)

$$\pi^* = \arg\max_{\pi} \underset{\tau\sim\pi}{\mathrm{E}}[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\big(\pi(.|s_t)\big)\right)] \tag{7}$$

where τ is the trajectory sampled from policy π, $\alpha > 0$ is the temperature parameter that controls the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy π^*. The discount factor γ is introduced to ensure that the expected sum of rewards is discounted by how far off in the future the rewards are obtained. The optimal policy can be calculated by maximizing the expected reward of the policy. The expected reward is the sum of the discount factors at time t multiplied by the reward of the transition from s_t to s_{t+1} given action a_t, in addition to the entropy. With this the value functions that determine the expected return of a policy given a state or a state-action pair can be defined. V^{π} (8) gives the expected return given starting position s and policy π, which includes the entropy bonus from every timestep.

$$V^{\pi}(s) = \underset{\tau\sim\pi}{\mathrm{E}}[\sum_{t=0}^{\infty}\gamma^{t}\left(R(s_t,a_t,s_{t+1}) + \alpha H\big(\pi(.|s_t)\big)\right)|s_0 = s]. \tag{8}$$

Q^{π} (9) gives the expected return if the starting position is s, an arbitrary action a is taken, but then all actions taken are according to policy π. In addition to this, the entropy bonuses are added from every timestep except the first.

$$Q^{\pi}(s,a) = \underset{\tau\sim\pi}{\mathrm{E}}[\sum_{t=0}^{\infty}\gamma^{t}\left(R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^{t}H\big(\pi(.|s_t)\big)\right)|s_0 = s, a_0 = a] \tag{9}$$

Given these definitions, $V^{\pi}$and Q^{π} can be joined (10).

$$V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}}[Q^{\pi}(s,a)] + \alpha H\big(\pi(.|s)\big) \tag{10}$$

The Bellman equation (11) for Q^{π} writes the value of a decision problem at a certain point in time in terms of the payoff from some initial choice in addition to the value of the remaining decision problem that results from those initial choices.

$$\begin{aligned} Q^{\pi}(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') + \alpha H\big(\pi(.|s')\big)\right)\right] \\ &= \underset{s'\sim P}{\mathrm{E}}[R(s,a,s') + \gamma V^{\pi}(s')]. \end{aligned} \tag{11}$$

Here $s'\sim P$ is shorthand for $s'\sim P(.|s,a)$, indicating that the next state s' is sampled from the environment's transition rules. $a\sim\pi$ is shorthand for $a\sim\pi(.|s)$, meaning that the action is

sampled from the policy rules. Therefore the Bellman equation is the expected reward of a state transition plus the value of the state transitioned towards.

Soft actor-critic learns both a policy and two Q-functions, $Q_{\phi 1}$ and $Q_{\phi 2}$, at the same time. Q-functions are approximators for the optimal action-value function. With the definition of entropy, the Bellman equation can be rewritten (12).

$$Q^{\pi}(s,a) = \underset{\substack{s'\sim P \\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') + \alpha \log \pi\left(a'|s'\right)\right)\right] \tag{12}$$

Because Q^{π} is an expectation over future states which comes from the replay buffer and future actions, it can be approximated with samples and rewritten (13).

$$Q^{\pi}(s,a) \approx r + \gamma(Q^{\pi}(s',a')) - \alpha \log \pi(\tilde{a}'|s')), \quad \tilde{a}' \sim \pi(.|s') \tag{13}$$

Instead of a', ã' is used to indicate that the next actions have to be sampled 'fresh' from the policy, whereas r and s' come from the replay buffer. This replay buffer should ensure that SAC is more sample efficient compared to PPO, as more experiences can be saved in the buffer. Sample efficiency entails how much experience the agent needs in order to reach a certain level of performance. The Mean Squared Bellman Error (14) is calculated to indicate how close a Q-function comes to satisfying the Bellman equation. SAC utilizes a clipped double-Q method, and takes the minimum Q-value between the two Q approximators.

$$L(\theta_i, D) = \underset{(s,a,r,s',d)\sim D}{\mathrm{E}}\left[\left(Q_{\theta i}(s,a) - y(r,s',d)\right)^2\right] \tag{14}$$

Here *D* is the replay buffer and *d* the done signal with a target function (15).

$$y(r, s', d) = r + \gamma(1 - d)(\min_{j=1,2} Q_{\phi targ,j}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')), \quad \tilde{a}' \sim \pi_\theta(.|s') \qquad (15)$$

In each state the policy will try to maximize the expected future return in addition to the expected future entropy, therefore maximizing V^π (16).

$$\begin{aligned} V^\pi(s) &= \mathop{\mathrm{E}}_{a\sim\pi}[Q^\pi(s,a)] + \alpha H(\pi(.|s)) \\ &= \mathop{\mathrm{E}}_{a\sim\pi}[Q^\pi(s,a) + \alpha \log \pi(a|s)] \end{aligned} \qquad (16)$$

SAC utilizes entropy regularization to train a stochastic policy and while exploring in an on-policy way. The exploration-exploitation balance is regularized through coefficient α. A higher α corresponds with more exploration, and a lower coefficient corresponds to more exploitation. The pseudocode for the Soft Actor-Critic algorithm can be seen in Figure 3.

Both SAC and PPO implement a Long Short-Term Memory (LSTM). A LSTM is a Recurrent Neural Network (RNN) that is capable of learning order dependency in sequence prediction problems. However, a problem does occur in RNN's due to the scalar weight recurrence between each of its recurrently connected blocks. It will eventually become either infinite if the scalar weight is greater than 1 or becomes 0 if the scalar weight is smaller than 1. LSTM deals with the exploding or imploding scalar weight by replacing the hidden units with a LSTM memory cell and adding another connection from every cell called the cell state (Hochreiter & Schmidhuber, 1997). A LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These memory blocks contain one or more recurrently connected memory cells and three gates, an input gate, which determines if the cell is updated, an output gate which determines if the memory of the cell is erased and a forget gate, which

determines if the information of the current cell state is made visible. These gates provide continuous directions to read, write or reset the memory cell (Graves & Schmidhuber, 2005).

Algorithm 2 Soft Actor-Critic

---

1. Input: initial policy parameters θ, Q-function parameters $\phi_1,\ \phi_2$, empty replay buffer *D*
2. Set target parameters equal to main parameters $\phi_{targ,1} \leftarrow \phi_1,\ \phi_{targ,2} \leftarrow \phi_2$
3. **repeat**
4. Observe state *s* and select action $a \sim \pi_\theta(.|s)$
5. Execute a in the environment
6. Observe next state *s'*, reward *r*, and done signal *d* to indicate whether *s'* is terminal
7. Store (*s, a, r, s', d)* in replay buffer *D*
8. If *s'* is terminal, reset environment state
9. **if** it's time to update **then**
10. **for** *j* in range(number of updates) **do**
11. Randomly sample a batch of transitions, *B* = {(*s, a, r, s', d*)} from *D*
12. Compute targets for the Q-functions:

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi targ,i}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \tilde{a}' \sim \pi_\theta(.|s')$$

13. Update Q-functions by one step of gradient descent using:

$$\nabla_{\phi i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_{\theta i}(s,a) - y(r,s',d))^2 \quad \text{for } I = 1, 2$$

14. Update policy by one step of gradient descent using

$$\nabla_{\phi i} \frac{1}{|B|} \sum_{s\in B} (\min_{i=1,2} Q_{\theta i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)),$$

Where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(.|s)$ which is differentiable wrt θ via the reparameterization trick.

15. Update target networks with

$$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$$

16. **end for**
17. **end for**
18. **until** convergence

Figure 3. The pseudocode for the SAC algorithm, taken from https://spinningup.openai.com/en/latest/algorithms/sac.html.

Methodology

The Agent

The goal of the agent is to merge with the ongoing traffic and reach its destination at the end of the road as soon and as efficiently as possible. The criteria for a successful merge are (1) no collision between the agent and the other vehicles or the side of the road; (2) no stopping at the side line, although slowing down is acceptable; (3) reaching the end of the road at the top of the screen. If a merge is successful, the agent is rewarded at the end of the episode. The agent is also rewarded by going through checkpoints placed along its path and is rewarded proportionally to its speed. If a merge is unsuccessful, the agent is punished. A small punishment is also added for each decision the agent makes, to ensure it makes the most efficient decisions possible, which should eventually lead to straight driving. Other studies implementing algorithms for lane merging have had success rates between 90% and 100%, with most achieving a success rate of around 95% (Hu et al., 2020; Triest et al., 2020). Therefore, for the agent to be considered successful, at least 95% of the episodes need to be completed successfully. Although this still seems low for a simple simulation, the setup and methods used are also simple compared to other studies in terms of neural network and LSTM. However, this allows the current study to focus on the comparison of the two algorithms rather than the success rate.

While most technologically advanced cars use a plethora of sensors such as LiDAR, camera's, digital maps and Differential Global Positioning System to detect their surroundings (Eckelmann et al., 2017), the agent in the simulation only has two sets or rays. One of these sets of rays is for road detection, to indicate how far it is from the sides of the lane. The other set of rays is to detect other traffic on the road, in indicate the distance to the other cars around it. Additionally, the agent knows its own speed and direction, as this effect how the car controls, so it can determine how to control the car depending on its speed.

Simulation setup

To solve the merge-problem, a simulation was set up for the reinforcement learning agent to learn how to merge successfully into traffic with either SAC or PPO. The tapered acceleration lane, as seen on the left in Figure 4, was utilized as this is assessed as the most difficult highway merge situation (Hussain et al., 2018). The simulations are made in Unity, a game engine with a built-in physics engine, which allows for realistic car controls such as wheel spinning and collision with the terrain, wheel suspension, damper, slipping and more. The ML-Agents package will be used to assist with the neural network, which enables the use of the TensorFlow library, an open source library to develop and train machine learning models.

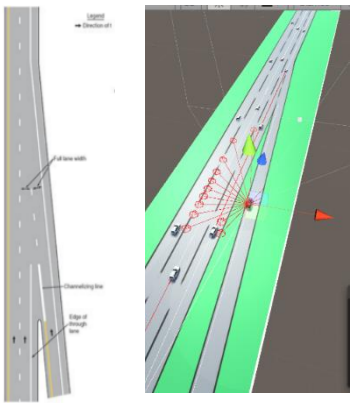

Figure 4. The tapered highway example on the left taken from Hussain et al. (2018) and the simulation on the right, made in Unity.

The simulation was created in 3D, which allows the car to be subjected to gravity, drag and wheel collision. The car has front-wheel driving, a weight of 1500 kg and a maximum speed of 100 km/h as is the case for Dutch highways since 2020 (Rijkswaterstaat, 2020). To determine the position of the car with regards to the road, 9 rays are used with a length of 20 meters. To determine the positions of the other cars, 15 rays are used with a length of 70 meters. These, along with the current speed and direction of the car are the input vectors of the neural network. The input vectors are stacked 3 times so the car can determine the change in position of the other cars. A neural net of 3 layers each with 128 nodes was trained with a learning rate of 0.0004. In addition to this a LSTM of 32 cells is utilized to assist in avoiding cars as this

will process the sequences of data, which allows the agent to determine how fast the oncoming cars are going and if the agent needs to accelerate or decelerate.

Comparison

The two algorithms are compared to each other in terms of performance, learning rate, and driving efficiency. To compare the two algorithms, data is gathered by having the agents complete 200 episodes. Performance is determined by the overall success rate of the agents while the learning rate is determined by the total amount of hours and total amount of timesteps it took for the agent to reach a certain level of performance. Efficiency is determined by both the total amount of steering and average timesteps per episode. The total amount of steering should indicate how much unnecessary actions the agent took and is calculated by adding the absolute values of the steering input.

To test the robustness of both algorithms, the other vehicles have random starting positions at the start of the highway and speeds varying between 90 and 110 km/h. Sensors are added to all the vehicles to determine if there is a car next to them. If there isn't, they have a random chance to merge on to the empty lane. The cars also have sensors to indicate whether or not a car is in front or behind them. If there is a car in front, the vehicle will slow down. If a car is behind the vehicle, it will speed up. If there is a car both in front and behind, the vehicle will ensure that equal distance is kept between the cars.

Results

The results show that both algorithms performed similarly, as can be seen in Table 1. The agent is able to merge onto the highway with at least a success rate of 95%. If cars are close by when the agent tries to merge it will either slow down or speed up, depending on how close the agent is to the main highway and depending on the proximity to the other cars, which is desirable.

Both algorithms continued until they either performed 4 million timesteps or spent 24 hours training.

The maximum reward for an episode is 62.4, therefore an average of 60 is considered decent. While SAC needed fewer timesteps to have a decent performance compared to PPO, about 750.000 steps compared to 1.2 million steps, it took about 10 hours compared to the 4 hours of training for PPO to reach a decent average reward. PPO had a more stable increase per timestep than SAC and less variability in performance once a high success rate was achieved, indicated by the sudden drop in performance at the 1.3 million timestep mark. While the amount of timesteps per episode are similar for both algorithms, meaning both agents completed the episode in comparably the same amount of actions, the SAC agent had almost double the amount of steering per episode. The performances of both algorithms can be seen in Figures 5 and 6.

Table 1

Algorithm Performance

Algorithm	**PPO**	**SAC**
Success rate	97%	95%
Timesteps	4 million	1.5 million
Timesteps till 60 reward	1.2 million	750.000
Training hours	16	24
Time till 60 reward	4 hours	8 hours
Average timesteps per episode	154	165
Average steering per episode	47.3	91.6

Note. PPO and SAC compared in terms of sample efficiency, learning rate and performance.

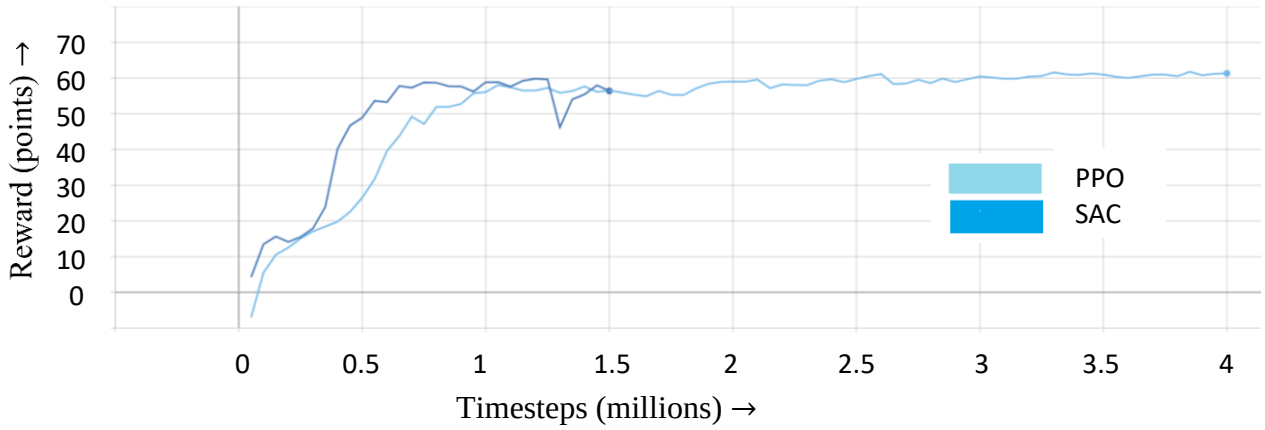

Figure 5. The average episode reward for each algorithm against actions taken in timesteps for both PPO and SAC. Maximum reward is 62.4

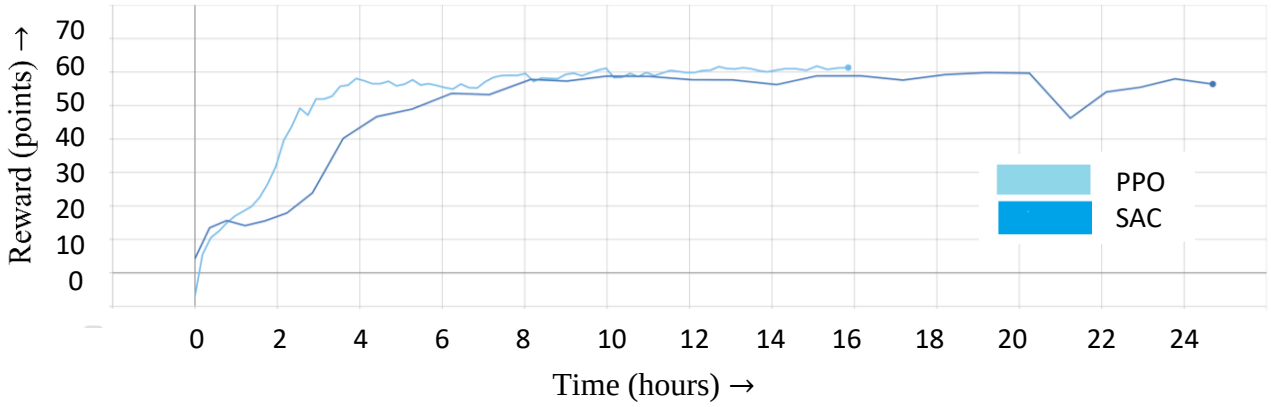

Figure 6. The average episode rewards for each algorithm against time in h ours for both PPO and SAC. Maximum reward is 62.4

Discussion

In the current study the PPO and SAC algorithms are compared to each other by having an agent learn how to merge onto a high traffic highway. In terms of overall performance, both SAC and PPO performed similarly, having a success rate of 95% and 97% respectively. In terms of learning rate, the SAC algorithm needed fewer timesteps and fewer episodes than PPO to reach this success rate, indicative of its sample efficiency. However, it took SAC longer to process those episodes than PPO, most likely due to the off-policy updates of SAC. With this in mind, SAC should be utilized over PPO in problems where sample availability is limited as it learns from all previous experiences. PPO does train faster but requires more data to be successful.

In terms of driving efficiency, the SAC agent took more timesteps to complete an episode and had more total amount of steering. PPO took about 150 timesteps while SAC took 165 timesteps to successfully merge, indicating that SAC took more inefficient actions compared to PPO. There was also a large difference between the total amount of steering done by the agents. Thus the SAC agent took more unnecessary actions and worse actions on average, making it suboptimal compared to PPO in this regard.

As both algorithms are state of the art reinforcement learning algorithms, both performed as expected. A high success rate was achieved by both even though a very simple setup was utilized in terms of both the neural network and the Long Short-Term Memory. Both algorithms have been used in more complex problems and therefore, with more processing power, hyperparameter tuning and optimization, a higher success rate should be possible. The sample efficiency of SAC is evident as expected while PPO has a more stable performance. Both algorithms have their advantages and disadvantages which are expected when considering their way of learning.

However, the problem of overfitting does emerge with both SAC and PPO as this also occurs in most machine learning problems. The agent might be able to merge the lane with a high success rate, but this does not mean that the car is able to drive in many different scenarios. The agent does not learn the rules of the road, it merely has sensors as input and estimates which outputs will result in the highest reward. This can however be abetted by implementing the agent in multiple environment in parallel. With parallel environments the agent is put in different environments simultaneously, so it can learn how to drive in all scenarios. This should enable the agent to be put in all environments separately and drive safely.

When utilizing simulations to analyze ways to solve a problem such as the highway merge problem, there are a few caveats. The simulation is an oversimplification compared to the real world. A major problem in autonomous driving is computer vision, which is mostly

disregarded in this simulation. Weather conditions and damages to the road can make it complicated for artificial intelligence to recognize their surroundings, while in the current simulations these conditions are always perfect, which results in a much simpler problem to solve. Future simulation studies should include these conditions such as weather and road decay to simulate more realistic road conditions. Preferably a simulation would be set up where these conditions are met, such as modern video games. Games such as Grand Theft Auto simulate traffic in a more realistic way and have more realistic settings such as changing weather conditions which could aid in analyzing how a real car would learn under such conditions.

Conclusion

The current study shows that PPO and SAC can both be used in the merge problem with a reasonably high success rate. While SAC showed more sample efficiency than PPO, SAC took longer to process the same amount of data, performed less optimally and less stable compared to PPO. Therefore SAC could be preferred over PPO when data availability is limited as SAC will utilize all past data, while PPO will only train with the most recent batch of experiences. The algorithms did not perform perfectly as they still had a small error in performance. This lack of performance can be attributed to the lack of processing power and a more simplified version of the learning process.

Nevertheless, the results show that PPO, SAC and reinforcement learning in general can be used in the merge problem. Due to the flexibility of reinforcement learning algorithms compared to rule-based methods, it is an approach worth pursuing that could eventually lead to fully autonomous vehicles. Future studies training such algorithms should utilize more realistic simulations for better representation and generalization of results, while also utilizing parallel simulations, so the same agent can learn different driving and merging scenarios.

References

Denardo, E. V. (1973). A Markov Decision Problem. In T. C. Hu & S. M. Robinson (Eds.), *Mathematical Programming* (pp. 33–68). Academic Press. https://doi.org/10.1016/B978-0-12-358350-5.50005-1

Dollar, R. A., & Vahidi, A. (2019). Automated Vehicles in Hazardous Merging Traffic: A Chance-Constrained Approach ∗∗This research was supported by an award from the U.S. Department of Energy Vehicle Technologies Office (Project No. DE-EE0008232). *IFAC-PapersOnLine*, *52*(5), 218–223. https://doi.org/10.1016/j.ifacol.2019.09.035

Eckelmann, S., Trautmann, T., Ußler, H., Reichelt, B., & Michler, O. (2017). V2V-Communication, LiDAR System and Positioning Sensors for Future Fusion Algorithms in Connected Vehicles. *Transportation Research Procedia*, *27*, 69–76. https://doi.org/10.1016/j.trpro.2017.12.032

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM networks. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, *4*, 2047–2052. https://doi.org/10.1109/IJCNN.2005.1556215

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *ArXiv:1801.01290 [Cs, Stat]*. http://arxiv.org/abs/1801.01290

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-term Memory. *Neural Computation*, *9*, 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hu, Y., Nakhaei, A., Tomizuka, M., & Fujimura, K. (2020). Interaction-aware Decision Making with Adaptive Strategies under Merging Scenarios. *ArXiv:1904.06025 [Cs, Stat]*. http://arxiv.org/abs/1904.06025

Hussain, S., Shahian Jahromi, B., Karakas, B., & Cetin, S. (2018). Highway Lane Merge for Autonomous Vehicles Without an Acceleration Area using Optimal Model Predictive Control. *World Journal of Research and Review*, *6*. https://doi.org/10.31871/WJRR.6.3.20

Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks*. 168.

Marinescu, D., Čurn, J., Bouroche, M., & Cahill, V. (2012). On-ramp traffic merging using cooperative intelligent vehicles: A slot-based approach. *2012 15th International IEEE Conference on Intelligent Transportation Systems*, 900–906. https://doi.org/10.1109/ITSC.2012.6338779

Proximal Policy Optimization. (2017, July 20). OpenAI. https://openai.com/blog/openai-baselines-ppo/

Rijkswaterstaat, Rijkswaterstaat, & Rijkswaterstaat. (n.d.). *Maximumsnelheid* [Webpagina]. Retrieved December 1, 2020, from https://www.rijkswaterstaat.nl/wegen/wetten-regels-en-vergunningen/verkeerswetten/maximumsnelheid/index.aspx

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017). Trust Region Policy Optimization. *ArXiv:1502.05477 [Cs]*. http://arxiv.org/abs/1502.05477

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *ArXiv:1707.06347 [Cs]*. http://arxiv.org/abs/1707.06347

Triest, S., Villaflor, A., & Dolan, J. (n.d.). *Learning Highway Ramp Merging Via Reinforcement Learning with Temporally-Extended Actions*. 6.

Wang, C., Zhang, Q., Tian, Q., Li, S., Wang, X., Lane, D., Petillot, Y., & Wang, S. (2020). Learning Mobile Manipulation through Deep Reinforcement Learning. *Sensors*, *20*, 939. https://doi.org/10.3390/s20030939

Wang, P., & Chan, C. (2017). Formulation of deep reinforcement learning architecture toward autonomous driving for on-ramp merge. *2017 IEEE 20th International*

Conference on Intelligent Transportation Systems (ITSC), 1–6. https://doi.org/10.1109/ITSC.2017.8317735

Wang, Pin, & Chan, C.-Y. (2018). Autonomous Ramp Merge Maneuver Based on Reinforcement Learning with Continuous Action Space. *ArXiv:1803.09203 [Cs]*. http://arxiv.org/abs/1803.09203