
Cardiff University

Computer Science and Informatics

CM3203 Final Year Project

# Reinforcement Learning for Autonomous Racing Using Deep-Q Learning and FastSLAM2

Final Report

---

Author:
Benjamin Loki Hughes

Supervisor:
Dr Kirill Sidorov

Moderator:
Dr Matthias Treder

# Contents

# Reinforcement Learning for Autonomous Racing Using Deep-Q Learning and FastSLAM2

Benjamin Loki Hughes

## 1 Abstract

**This project sought to use Deep-Q Learning; a Reinforcement Learning algorithm, to map observations to actions for autonomous racing, specifically to compete in the FS-AI 2019 competition. The model generated was able, after training on random, simple, tracks, consisting of spaced blue and yellow cones, as specified in the FS-AI rules, to complete a lap on an unseen track in a $2D$ simulation, following a good racing line, proving itself as a working proof of concept for such a system to be deployed in the FS-AI competition for autonomous racing. The model only assumes that cones within a 12 metre range would be detected, and triangulated with some accuracy, accounting for noise. Given previous work on the problem in creating a stereo system to do this, it is reasonable to assume this would be given in a real situation. The model successfully deploys fastSLAM2 to transform these measurements, along with odometry readings to create a probabilistic map of its environment, to be given to a Deep Neural Network to successfully dictate actions that allow for a track to be completed.**

## 2 Introduction

As autonomous cars become an increasing reality, with advancements in Machine Learning, sensor accuracy and computational power now being able to support the 'self driving car' vision, the Formula Student Competition [FS, ] has branched into this field. As such, the Cardiff Racing team now operates a FS-AI subgroup to tackle the problem.

As the setting for formula racing is much simpler, and more predictable than pedestrian roads, the problem is a simplified sub-case that a usual driverless car might have to tackle. For example, the car will not have to abide by road safety laws, or accommodate for pedestrians, traffic lights etc. The car simply needs to drive as fast as it can around a track indicated by cones.Formally, the car will need to drive 10 laps around a track specified by blue and yellow cones for the inner and outer track respectively. These cones are spaced at a maximum of 5 metres apart, to form a 6 metre track width [FS-AI, 2019].

As a team, thus far, we have a means of detecting cones, using a trained Cascade Detector [Zhu et al., 2006] and a Support Vector Machine [Suykens and Vandewalle, 1999] (SVM), and have a means for determining the distance of the cone to the car, using stereo vision to triangulate the cone in 3D space. Given the flat nature of the racetrack environment, the problem can be further simplified. Ignoring the vertical axis, the car may decide on its actions based on only a distance and angle in 2D space, as the $3^{rd}$ axis would be only of very slight variance, or none, in either case, unimportant for action formulation.

A survey of the current self driving car models can be explored in [Paden et al., 2016], they find, at a high level, an autonomous car must adhere to the following pipeline: first must some goal state must be decided, this can be done dynamically, or directly given. In our case, the goal is to perform a lap (Drive a circuit and return to its start position). Upon receiving a goal, or checkpoint, the agent must use *route planning* to calculate a route from its current pose to the goal pose, a survey of such techniques can be found [Bast et al., 2016],

where classical A* or Dijkstra methods break down due to the size of complexity of most maps in such cases, more sophisticated graph searches must be deployed, requiring a good estimation of the environment. Once a route is decided, this must be passed to a *behavioural decision maker* which will decide on high level actions (e.g change lane, overtake, follow road). This is then broken into motions by some *motion planning* algorithm to decide on the immediate action sequence (e.g break, accelerate, turn 30°). Finally, this action is actuated by the vehicle, which, following this, takes in further sensor readings to form its feedback loop. Indeed AMZ Driverless use this pipeline as their successful software implementation for FS-AI Germany [Kabzan et al., 2019].

The results of these algorithms are completely interpretable [Bibal and Frénay, 2016], and thus preferable for trust in commercial self-driving cars [Kaur and Rampersad, 2018]. However, given the racing environment, the same risks (running over a child etc) do not apply, and so the AI can favour performance over interoperability. Deep Learning techniques often wrestle with interoperability/ performance tradeoffs [Ishibuchi and Nojima, 2007] in such critical cases as self driving cars. In this project, I seek to exploit their accuracy for better performance in motion planning through to action formulation. This project seeks to use a Deep Learning approach to suggest the control, from the observations alone. This will take the place of route planning, behavioural decisions and the motion planning elements in producing a control $u$ at time $t$, given observations $z$ as input. The observations in this case are distances and angles to cones, detected using the system previously designed, described.

This project, then, explores the use of Reinforcement Learning for action formulation. Specifically, this project investigates the use of Deep-Q-Learning, developed by DeepMind [Mnih et al., 2015]. This is not the first attempt at using Deep Reinforcement Learning for autonomous driving [Wang et al., 2018], but as this is a constrained problem, it requires a specially designed solution.This technique poses the problem as a game, where the agent will learn how to obtain the best score, posed as a reward from the environment using a deep neural network. For this the agent needs a representation of its environment. Prior work has tried to do this from immediate sensor readings only, however, this representation does not account for noisy measurements and is confined by only the cones within range of the car. For this project, I also use SLAM, specifically FastSLAM2 [Montemerlo et al., 2002] to allow the car to build a representation of the map, as it discovers it, as well as understand its pose in the map, and use this as the input for its decision making and learning. This has proven much more successful.

This project, following from the phrasing of the problem, trains a Deep-Q algorithm in a 2D environment, assuming cones are detected and a distance and angle to the cone from the car is given to the car. Here I provide the necessary background into both Reinforcement Learning and SLAM, to appreciate the Deep-Q algorithm and FastSLAM2 algorithm, before my implementation and results.

# 3 Reinforcement Learning

Reinforcement Learning (RL) is a branch of Unsupervised Machine Learning, where optimal actions, given an environment are learned through reinforced behaviour [Sutton et al., 1998]. Behaviour, formally *policy* is learned through interacting with the environment and having its actions rewarded if they lead to some desirable outcome, or penalised otherwise. This is formalised as a reward cycle; An *agent* performs an *action* on an *environment* and in return, receives a *reward* and an *observation*. The goal of the learning process is to reinforce the optimal policy for performing actions on the agent's environment to obtain the maximum reward, the agent can then perform optimally, having been sufficiently trained.

## 3.1 Bellman Optimality Equation

In RL, the agent must act on its environment in states, where states are uniquely identifiable permutations of the environment. The value of a state is defined as the expected reward obtainable from a state, where $r_t$ is the local reward obtained at step t, multiplied by the dampening factor $\gamma$ between

0 and 1 to dictate the rewards importance as $\gamma$ is decayed over time, far future rewards are worth much less than immediate rewards, with a spectrum in between [Lapan, 2018].

$$V(s) = \mathbf{E}[\sum_{t=0}^{\infty} r_t \gamma^t] \tag{1}$$

If an action $a_i$ is performed in state $V_0$ from a deterministic set of actions $A$, then the Value will be given as $V_0(a = a_i) = r_i + \gamma V_i$. If an agent is to perform optimally, it must choose an action $a$ such that the resulting value is the maximum possible value. As such, value should be computed as

$$\max_{a \in A}(r_a + \gamma V_a) \tag{2}$$

Thus we get the Bellman optimality equation, by iterating over the expected values of performing action $a$ over every state $s$ and calculating it's value with the Bellman equation, then returning the action that got to the best valued state.

$$V_0 = \max_{a \in A} \mathbf{E}_{s \sim S}[r_{s,a} + \gamma V_s] \tag{3}$$

As well as defining the value of a state, we can define the reward from performing action $a$ on state $s$ as a Q value $Q_{s,a}$ in terms of $V_i$.

$$Q_{s,a} = \mathbf{E}_{s' \sim S}[r_{s,a} + \gamma V_{s'}] \tag{4}$$

## 3.2 Q-Learning

This gives rise to the RL family of Q - Learning, algorithms concerned with learning Q values for state/action pairs. A classical example of such an algorithm would be the value iteration method, where every action/state pair value is iterated over for the highest Q value. However, this includes iterating over states that are of obvious no value to the agent, or are not feasible. Instead, we can update Q values through experiencing the environment. Tabular Q-Learning addresses this problem by instead sampling the environment for state, action, reward, new state tuples $(s, a, r, s')$ and learning from these experiences. The algorithm is then the following:

1. Start with empty Q table
2. Act with the environment to gain $(s, a, r, s')$ tuples
3. Make Bellman update
4. Check convergence conditions
5. repeat from 2 until convergence conditions are satisfied

Here the convergence conditions are met if the Q values are being updated with a value below some convergence threshold. Q values are also not directly updated, they are updated using a *blending* technique, to ensure for smoother learning, with a learning rate $\alpha$. They are thus updated with the following equation

$$Q_{s,a} \leftarrow (1 - \alpha)Q_{s,a} + \alpha(r + \gamma max_{a' \in A} Q_{s',a'}) \tag{5}$$

For sampling actions, some method must be used for deciding which actions to take. This form must balance *exploration vs exploitation*. In other words, the agent must exploit its trained knowledge of the environment whilst allowing itself to explore new actions. For this project a random action is chosen with probability $\epsilon$ o, and $(1 - \epsilon)$ of exploiting it's model for an action. With $\epsilon$ being decayed at every iteration, the agent will begin by exploring, then as the model is trained, exploit progressively more. This is known as the epsilon-greedy method.

## 3.3 Deep-Q-Learning

This solves the problem raised with the iteration method concerning it's need to iterate through impossible states, but in representing Q values in a state/action table, this algorithm will not extend beyond simple environments, and is not feasible for large or infinite state spaces, or indeed actions. To account for this, rather than a table, the agent must learn a function that maps state and action to a Q value. To achieve this, we use a Neural Network (NN) to approximate this transition. Though the general algorithm is in fact an unsupervised learning algorithm, we periodically grab training data to be used in the supervised machine learning problem of training this NN to approximate this function. Where its output is trained to give the Q value for every available action, from observed $(s, a, r, s')$ tuples.

### 3.3.1 Neural Networks

A Neural Network is an algorithm that very loosely resembles a biological neural network. They are used to approximate non-linear mappings, for classification or regression problems. They consist of layers of neuron models that sum up their inputs, multiply them by some learned weight, add a learned bias and put them through some non linear function. See below, the equation for a neuron's outputs, on the $j^{\text{th}}$ neuron on the $l^{\text{th}}$ layer, where $\sigma$ is some non-linear function, such as the $Sigmoid$, or $ReLU$. function.

$$a_j^l = \sigma(\sum w_j k^l, a_k^{l-1} + b_j^l) \tag{6}$$

Over several, in the case of deep-learning, many, layers, they have been shown to perform classifications and predictions very accurately in various non-linear domains. Each layer then represents more abstracted and nuanced representations. So data is classified into more abstract ideas the deeper the network, until the output layer gives an output of the probability of every classification, the maximum of which can be taken to be the classification.

To train a neural network, the network will, at every iteration of learning produce an output after putting the data through the current network. This output can then be compared to the correct output to calculate the model's accuracy, and loss. The simplest of which is a mean square error function, which can be taken at every output, or over $n$ outputs. Where $y$ is the result, and $\hat{y}$ is the true classification.

$$\mathbf{L} = \frac{1}{n}\sum_{i=1}^{n}(y^i - \hat{y})^2 \tag{7}$$

Weights and biases are then optimised through calculating their partial derivative, with respect to the loss function, this is essentially how much each parameter impacts the loss. Then, exploiting the chain rule to propagate the loss through the network, we can use gradient descent for calculating more optimal values using the following update equation, for a given weight $\theta$ and learning rate $\alpha$, typically 0.01.

$$\theta_j \leftarrow \theta_j - \alpha\frac{\delta}{\delta\theta_j}J(\theta) \tag{8}$$

The network should converge, this way, on locally optimal weights and matrices to approximate some non-linear map that minimises loss and gives the greatest accuracy. It can then be used to obtain a classification of arbitrary samples.

### 3.3.2 The Deep-Q Algorithm

The full algorithm, then, is very similar to the tabular method, but using a neural network to learn how to map states to Q values for state/ action pairs. There is another nuance to this algorithm, which is its use of a *replay buffer*. The agent's experiences as $(s, a, r, s')$ tuples are stored in a fixed size buffer for training. However, when the neural network trains on these examples, they are not all taken for training. For improved data distribution, to conform with the notion of *independent and identically distributed (i.i.d)*, the replay buffer is sampled randomly to form a batch of tuples. These are used as training data for the NN in episodes. Thus the algorithm, as presented by Deep-Mind is as follows [Mnih et al., 2015]:

1. Initialise parameters for $Q_{s,a}$ and $\hat{Q}_{s,a}$ with random weights, $\epsilon \leftarrow 1.0$, empty replay buffer
2. select random action $a$ with probability $\epsilon$ , otherwise $a = argmax_a Q_{s,a}$
3. Perform action $a$ and observe the the reward $r$ and the next state $s'$ to be stored in the replay buffer
4. For sampled tuples in the reaply buffer, observe $r$ if the episode ended, else observe
$$y = r + \gamma \max_{a' \in A} \hat{Q}_{s',a'}$$
5. Calculate the Loss and update the network using Gradient Descent for $Q_{s,a}$
6. Every N steps, copy weights from training network to agent network
7. Repeat from step 2 until convergence criteria are met

It is this algorithm that I use for my project, the specifics of which I discuss under 'Implementation'.

# 4 SLAM

The *Simultaneous Localisation and Mapping* (SLAM) problem is a fundamental problem in robotics. This problem must be solved when an agent's state is not given to it, nor is a map of its environment [Thrun et al., 2005] . The agent most must use its controls $u$ at every time-step and sensor measurements $z$ at every time-step to estimate its pose (position and orientation) $x_t$ and a representation of its environment, a map $m$. A representation of the map is important for additional tasks, such as path planning, as well as being needed to rectify the agent's pose estimation. Representations are usually landmark-oriented. That is, positions of heterogeneous landmarks are correlated to form a map representation.
To complicate things, all measurements, intrinsic or extrinsic for the agent carry uncertainty, that is, measurements are noisy, that is imperfect, or altered by the environment. The SLAM problem is thus probabilistic, given that nothing can be calculated with certainty if every measurement is noisy. Thus the SLAM problem's difficulties arise from [Stachniss, 2013]:

1. Both agent pose and map are unknown
2. Errors the map and pose estimation are correlated
3. The mapping between observations and landmarks are unknown, where incorrect data association (attributing sensor to landmarks) can have bad consequences in the SLAM estimations

Formally, the SLAM problem translates to estimating the posterior of the pose and map, given observations $z_{1:t}$ and controls $u_{1:t}$ :

$$p(x_t, m|z_{1:t}, u_{1:t}) \tag{9}$$

This formalisation is known as the *online* SLAM problem, which differs subtly from the *full* SLAM problem, in that the posterior is over the current pose and map, and not the pose at every time-step:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t}) \tag{10}$$

The current pose of the agent can be written probabilistically as a posterior given the previous pose and the command given. This is the motion model; how the robot's pose is updated $p(x_t|x_{t-1}, u_t)$ Similarly sensor observations may also be represented as a posterior given the pose of the agent. This is the measurement model, and is expressed as $p(z_t|x_t)$

Using these models, the SLAM posterior can be computed recursively using the Bayes filter. Where we apply Bayes rule to the SLAM posterior, and adhere to the Markov Assumption, the assumption that given an agent's current state is the only state that affects sensor readings, future, nor past states supply additional information. The resulting Bayes filter is presented here, see [Siegwart et al., 2011] for full derivation.:

$$Bel(x) = \eta P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) Bel(x_{t-1}) dx_{t-1} \tag{11}$$

Most SLAM algorithms, are based around estimating this posterior. The most common of which is the Extended Kalman Filter.

## 4.1 Extended Kalman Filters

The EKF algorithm citeeinicke1999robust represents the SLAM posterior as a high dimensional Gaussian function, where the mean $\mu$ represents the most likely pose of the agent, and landmarks, and the covariance $\Sigma$ represents the correlations between all variables, where N is the number of landmarks.

$$p(x_t, m|u_t, z_t) = \mathcal{N}(x_t; \mu\Sigma_t)$$

$$x_t = \{s_t, \theta_1, \ldots, \theta_N\}$$

$$\mu_t = \{\mu_{s,t}, \mu_{\theta,1,t}, \ldots, \mu_{\theta N,t}\}$$

$$\Sigma_t = \begin{bmatrix} \Sigma_{st,t} & \Sigma_{st\theta 1,t} & \ldots & \Sigma_{st\theta N,t} \\ \Sigma_{\theta 1,st,t} & \Sigma_{\theta 1,t} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{\theta N,st,t} & & \ldots & \Sigma_{\theta N,t} \end{bmatrix}$$

For a 2d environment, the mean $\mu$ will be of dimension $2N + 3$, where $N$ is the number of Landmarks, represented by $(x, y)$ and the 3 spaces are for the agent's pose as $(x, y, \theta)$, similarly, $\Sigma$ will be of size $2N + 3$ by $2N + 3$.

The EKF is an extension of the Kalman Filter citewelch1995introduction, where the KF is limited to only linear motion and sensor models, the EKF can take a non-linear function for motion and sensor models, though is more accurate the more linear they are, due to linearising the covariance. This is naturally more powerful, and more usable for real world scenarios. The motion model in the EKF then is a function $h(x_{t-1}, u_t)$ with covariance $P_t$, and the sensor model is a function $g(x_t)$ with covariance $R_t$. The EKF algorithm for updating the mean $\mu$, being the estimated pose and landmark locations, and covariance $\Sigma$ is:

**Algorithm 1:** Extended Kalman Filter

**Input:** $\mu$, $\Sigma$, $z$, $u$

**Result:** updated $\mu$ and $\Sigma$

$$\hat{\mu}_t = h(\mu_{t-1}, u_t)$$
$$\Sigma_t = \Sigma_{t-1} + P_t$$
$$G_x = \nabla_x tg(x_t, n_t)$$
$$\Sigma_t^- = G_x \Sigma_t^- G_x^T + R_t$$
$$K_t = \Sigma_t^- G_x^T Z_t^-$$
$$\mu_t = \mu_t^- + K_t(z_t - \hat{z}_n t)$$
$$\Sigma = (I - K_t G_t)\Sigma_t^-$$

For a complete explanation and derivation see [Thrun et al., 2005], For this report the following will suffice as an explanation. In line (1) the predicted belief $\mu$ is calculated through applying the motion model $h$ to the current belief $\mu$ with control $u$. The covariance is then updated with the new covariance of the motion model in lines (2-3). The EKF then calculates a Kalman Gain, this specifies the degree to which new measurements will be incorporated. Line 6 then manipulates the mean, in proportion to the Kalman Gain $K_t$ and the deviation of the measurement $z$ from the predicted measurement $\hat{z}$. The new covariance matrix is updated similarly to fully update the mean and the covariance with the new measurement.

The EKF suffers mostly from the quadratic complexity in the number of landmarks, as well as sensitivity to data association, as it does not have a mechanism for correcting bad data association. Meaning the model is easily poisoned if data association is ambiguous in an environment, for example, an environment of homogeneous landmarks.

## 4.2 FastSLAM2

FastSLAM2 is an example of a particle filter, specifically, it is an implementation of a Rao-Blackwellized particle filter [Grisetti et al., 2007]. Particles hold probabilistic hypothesis of the state; pose and all landmark positions, contrast to parameterised model, like the EKF, which estimates a Gaussian [Montemerlo et al., 2002]. It takes advantage of the sparsity in the needed updates in a SLAM model; due to each update only changing a small number of variables, the fully correlated model only comes to be after a large amount of observations, before this, landmarks are not correlated. Similarly, commands only constrain the pose of the robot from its last pose, and thus can be treated separately from landmark locations.

FastSLAM2 calculates a subtly different posterior to the standard SLAM posterior. Where the posterior is usually over the *pose* $p(x_t, m|z^t, u^t)$, in FastSLAM2 it is over the *path* $p(s^t, m|z^t, u^t)$. With this, following the Markov assumption, given knowledge of the path, thse observation of a landmark will not provide information of any other landmark. Meaning, landmark positions can be calculated independently. With this, the SLAM posterior can be factored into a path posterior, and N landmark posteriors. Which follows from the structure of the SLAM problem exactly.

FastSLAM2 uses a particle filter to estimate the path posterior, holding for every path hypothesis $N$ landmark posteriors, conditioned on that path alone, which are estimated using EKFs. Every EKF then tracks only a single landmark position, conditioned only on a single robot path. Using $M$ particles, and $N$ landmarks, there are $N \times M$ EKFs of low dimensions, tackling the dimensionality problem of using an EKF to hold a probabilistic model of the the pose and every landmark. Each particle is in the form:

$$X_t^m = \langle x^{t,m}, \mu_{1,t}^m, \Sigma_{1,t}^m, \ldots, \mu_{N,t}^m, \Sigma_{N,t}^m \rangle \qquad (12)$$

The FastSLAM2 algorithm follows these steps:

1. Sample a new pose for each particle, given a control

2. Update every Landmark EKF for the observed landmarks

3. Calculate importance weights for every particle

4. Draw new, unweighted particles using importance resampling

The specifics of every step can be found [Montemerlo et al., 2002], however, for the purpose of this report, it is enough to understand the SLAM problem, the basic steps, and that the result is a set of particles, carrying hypothesis about the agent's path. Each one holding the estimate of the path, and an EFK for every landmark, carrying a mean and covariance of their positions correlated with the path. The most likely of which is taken to be the path and map.
It is important to note that this algorithm does not suffer from scaling with $N$ landmarks as and EFK does, as it grows linearly. It also has a mechanism for filtering bad data associations, as they should only be considered by a small number of particles, which will not corroborate future observations , they are filtered out of the model.

# 5 Implementation

For this project I built a *Pygame* environment in Python, based heavily on [rasmaxim, 2018]. This is the basic environment in which the agent trains. I train the agent using the Deep-Q-Learniing algorithm, using the FastSLAM2 algorithm from [Sakai, 2019] to build a representation of the map. The map is made up of cones which form a random racetrack, these cones are represented as a point, assuming that in reality, this would be the centre of mass for the cones, from which the stereo vision system would measure the distance and angle to, this is what is assumed to be given to the car for all cones within its sensor range. The code for this project can be found: https://gitlab.cs.cf.ac.uk/c1628696/reinforcement-learning-for-ai-racing

Using Pygame and python made producing a screen with objects very fast, as importing images, resizing, producing shapes and dynamically changing the environment is made easier with its inbuilt functions, it also allowed me to manually control the car for testing much faster than had I have built every part from scratch. I did not employ any physics at this point, that will wait until later experiments.

### 5.0.1 Random Track Building

For the trained model to be robust to all types of tracks, the agent had to train on randomised tracks, this way there is no bias in the shape of the track, and the car learns to drive given a range of turns and lengths. For this I needed a way to randomise tracks. For this I employed the following technique, following [Maciel, 2013]:

1. Create a random number $n$ between 10 and 30

2. Create $n$ random $(x, y)$ points

3. Define a non-convex-hull of points This is a shape that only includes the outer most points [Gobor, 2017]

4. For every pair of points in non-convex-hull, create a point at a random distance along the line created by the two points, between the two points

5. Perturb the point by some random distance, with a $0.5$ chance of being $+$ or $-$ along the norm Note this step can be skipped for a *simple track* as is shown in my results.

6. For every pair of points in the track, linearly place cones at a pre-defined distance along the line made by the two points

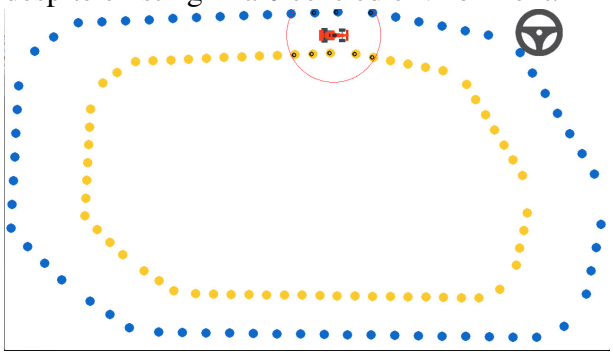
My implementation has a function for creating an outer track as defined above, using the scikitlearn function for defining the non-convex-hull, then takes every cone in this track and creates an inner track by copying this cone and moving it by a fixed amount $\theta$ between $0$ and $1$ towards the centre with the following movement equation.

$$x' = (\theta \times (x - x_{center})) + x_{center} \tag{13}$$

$$y' = (\theta \times (y - y_{center})) + y_{center} \tag{14}$$

I use $\theta = 0.75$ as my distance. I use the same technique for defining checkpoints for my RL algorithm between the outer track and inner track with $\theta = 0.5$.

As a consequence of how FastSLAM2 defines particle creation, the car must start its estimated position at $[x, y, \theta] = [0, 0, 0]$ and so the track had to start at $(0, 0)$. This is a problem when randomly generating tracks, as well as representing this in a game world where $(0, 0)$ is in the top left corner and not the centre. To overcome this, the track is generated in a normal coordinate space, where $(0, 0)$ is in the centre, it is then moved into the screen centre by adding $[W/2, H/2]$ to every point. Then moved over to the car, by subtracting the starting midpoint coordinates, so the whole track is shifted to make the car's position the start point, for SLAM to be performed on it. Then, when everything it blited to the screen, it goes through a function that displays it by subtracting the vector added to move it to the car. This way everything is displayed in the game space, despite existing in a 0 centred environment.



### 5.0.2 Reward Function

During training, It is important that rewards are fed frequently to the agent. This ensures that rewards are not sparse, making training difficult, for example, one approach here would have been to reward the agent inversely proportionally to the time it takes to finish the lap, to encourage speed. However it would have taken the agent the whole lap to see this reward, which, given random actions, would have been probabilistically unlikely in any given episode. To reward the car frequently, I create checkpoints along the track. A checkpoint is placed between every inner cone / outer cone pair. The reward function then rewards the agent for being within range of a checkpoint, to gain a reward from that checkpoint. Once a reward is claimed from a checkpoint, it cannot be claimed again, ensuring the agent doesn't stay in place for optimal rewards. If the agent is not in range of any checkpoint it is penalised, as it is now outside the track. The reward for each reward is multiplied by $\frac{1}{\delta t}$ where $\delta t$ is the time since the last checkpoint, to reward speed. The reward function is the following, where the variable $r$ refers to cone $i$ being within the car's sensor range, and the variable $c$ referring to the reward from cone $i$ having been claimed:

$$r = \begin{cases} 0 & \text{if } cone_i \in cones(r \ \& \ c) \\ (N - i) \times \frac{1}{\delta t} & \text{if } cone_i \in cones(r \ \& \ not \ c) \\ -100 & \text{otherwise} \end{cases} \tag{15}$$

This performs better than previous work, which had only rewarded being inside the cone barriers and velocity. This reward function rewards speed, travelling in the right direction, and staying within the cones.

### 5.0.3 Deep-Q-Learning implementation

The Deep-Q algorithm I implemented was initially taken from [Kim, 2018] and edited, incorporating a suggestion in [Lapan, 2018], of transferring the network on every $N$ steps, where in my implementation $N = 10$, to allow for smoother training. Then altering hyper-parameters following experimentation.

To implement the neural network, I used the *Keras* library with a *Tensorflow* backend. This allowed for rapid prototyping, as it heavily abstracts from the particulars of a neural network, but is highly optimised. The network used, after experimenting with hyperparameters, is a sequential Neural Network with 4 hidden layers, of 600 neurons, with a $RelU$ activation function, using standard Stochastic Gradient Descent to optimise it with a 0.001 learning rate and 10 epochs. It takes, as an input, a $1280 \times 700$ sized $2D$ bitmap, generated by the agent using SLAM2 on its sensor inputs.

The Algorithm lets the agent play $10,000$ episodes of $10,000$ frames unless the car reaches a final state in this time (Finishes a lap or drives outside the cones). This, I found was enough time for the car to drive through a randomly generated track. In each

episode, the car chooses an action with a probability of $1 - \epsilon$, with a decaying rate of $0.995$, from the neural network, taking the $max(Q_{state,action)}$ action, or a random action. The simulation then performs that action and feeds back a $(s, a, r, s')$ tuple, where $s$ and $s'$ are SLAM generated bitmaps of the state, where no cone is represented by 0, yellow cones are represented as 1s, blue cones are represented as 2s and the car is represented by a 3. These 10,000 frames are put into a memory queue of length 20000, where 100 samples are taken off randomly in every episode to train the network on. The environment is reset every episode, and should the car complete a lap, a new track is generated for robust training. Every 10 episodes, the network is transferred to the agent.

The model is saved every 1000 episodes incase the simulation fails, as it is a long training procedure, and can then be used by car to make decisions in new environments, it can also, continue to store $(s, a, r, s')$ tuple to be trained offline, if required.

#### 5.0.4 FastSLAM2 implementation

To give the most meaningful information forward to the Deep-Q-Learning algorithm, I used FastSLAM2 to turn sensor measurements and commands into a map representation, including every seen cone location and the car's relative position in that map. For this, I used Atsushi Sakai's implementation found here [Sakai, 2019]. This implementation serves as a reminder for its usefulness as well as a well structured code base for easy manipulation and understanding. I shan't repeat how the algorithm works from the background I shall only mention my alterations, the sensor and motion models, and the ability to capture different landmarks; blue and yellow cones, in the sensor model.

The ability to represent different colour cones comes from the labbeling process in the cited code base. Here, when the measurement is compared against the current model; *update_with_observation*, it invokes either *update_landmark* which updates the landmark's EKF using Cholesky Decomposition [Nino-Ruiz et al., 2018], or creates a new landmark with *add_new_lm*. When a landmark is created, I label it, instead of with a single number to later index it by, a tuple, that includes its index as well as a label, $0 = yellow$, $1 = blue$. This can can later be used to differentiate between blue and yellow cones when building a bitmap of the map. This is also given the cone detection tool in place being able to feed this additional information, having been trained to recognise blue and yellow cones separately.

#### 5.0.5 Sensor Model

The sensor model I use gives the car a distance $x$ and an angle $\theta$ from its centre to the cone's centre, if it is within the sensor range. It is reasonable to assume this is given with good accuracy, following from previous work done in the FS-AI team for recognising cones, in stereo vision, to also triangulate it's position in space relative to the camera. In my simulation then, it suffices to give this information to the car, instead of simulating some complex situation where this is employed, and so the motion model is as follows; for every yellow, and then blue cone in the separate list of cones, made by the track generation algorithm, I measure the distance and angle between it and the car:

$$\begin{aligned} dx &= x_{cone} - x_{car} \\ dy &= y_{cone} - y_{car} \\ \theta &= \tan \frac{dy}{dx} - \theta_{car} \end{aligned} \tag{16}$$

If this is within the sensor range (in the model I use a sensor range of 30) which translates into 10 metres, in reality the car has a larger range than this, so this is a safe assumption, then it is added to the list of $[dist, \theta]$ fed to SLAM as an observation. In the simulation this is added to a noise matrix in the form:

$$\begin{bmatrix} \epsilon & 0 \\ 0 & \epsilon \end{bmatrix} \tag{17}$$

Where $\epsilon$ is a random number between 0 and 1. This is to simulate natural noise in real sensor measurements.

#### 5.0.6 Ackerman Motion Model

As a motion model, I chose a simplified version of the Ackerman Kinematic Model; the bicycle model [Snider et al., 2009], given its simplicity, this

was easy to implement in a $2D$ simulation. A control is given by two attributes $(\phi, a)$ where $\phi$ is the steering angle and a is the acceleration. This is realistic and appropriate for this purpose. Velocity is then calculated, simply, as

$$v' = v + a \times \delta t \tag{18}$$

The turning radius is calculated as $L/\tan\phi$ And the angular velocity $\omega$ is calculated as $v/r$. The car is thus updated by the following equations:

$$x' = xcos(\phi) - ysin(\phi) \tag{19}$$

$$y' = ysin(\phi) + ycos(\phi) \tag{20}$$

$$\theta' = \theta(\omega) \times \delta t \tag{21}$$

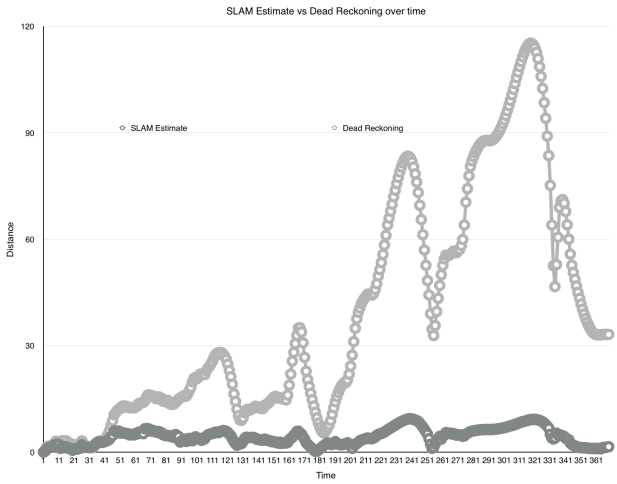
The cars position is now $(x', y', \theta')$. And is multiplied by a random noise matrix similar to the sensor model to model the noise in the mapping between commands and movement due to odometry inaccuracies, slipping, skidding etc.

# 6 Results

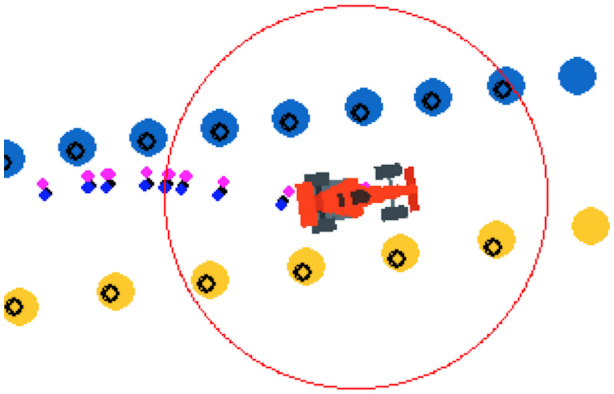
## 6.1 SLAM

The SLAM2 implementation, discussed, works excellently for estimating both the car's pose and the positions of cones. Below is an image of the scatter produced by the particle filter, where each blue spot is a pose hypothesis. The mean of which converge accurately to the car's true pose.

To further emphasise the need for SLAM, the figure below shows the Dead Reckoning over time as well as the SLAM estimated pose over time, as a distance from the true pose. Note that the dips in distance are accounted for by landmark observations, which are probabilistically correlated with the pose. It can be seen that with few, to no, observations, pose uncertainty grows for the SLAM model, while the Dead Reckoning is not dependant on these, dips can also be accounted for during certain turns where the Dead Reckoning comes closer to the true pose by happenstance. During the simulation, path estimates are shown as blue spots, while pink spots show the Dead Reckoning; the position that would be taken, had SLAM not accounted for the noise, in the simulation this is simulated noise, in reality this would occur naturally from slipping etc. The difference in Dead Reckoning estimations to the SLAM estimations can be seen visually in all simulated laps.

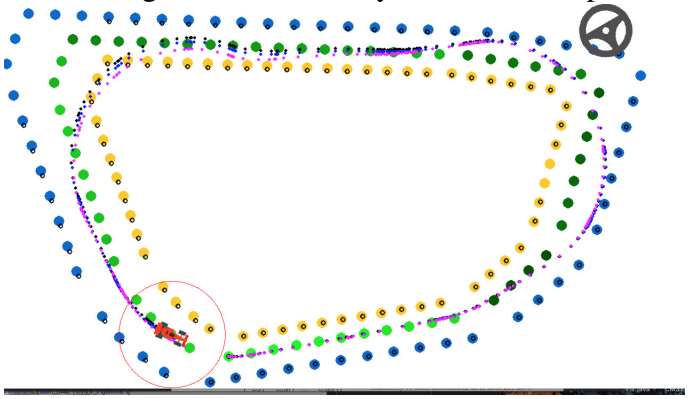



Landmark estimates can also be seen as black circles over the cones, accurately predicting the cone locations, the circles here are from the most likely particle, thus the most probabilistic position, from the model. Shown below is an early shot, where the discovered cones have a black circle over their location to represent the SLAM estimate of their respective positions. As these positions are then held, irrespective of where the car is, or is detecting at that moment, such landmark estimates are kept as a map for the Deep-Q Learning Algorithm.

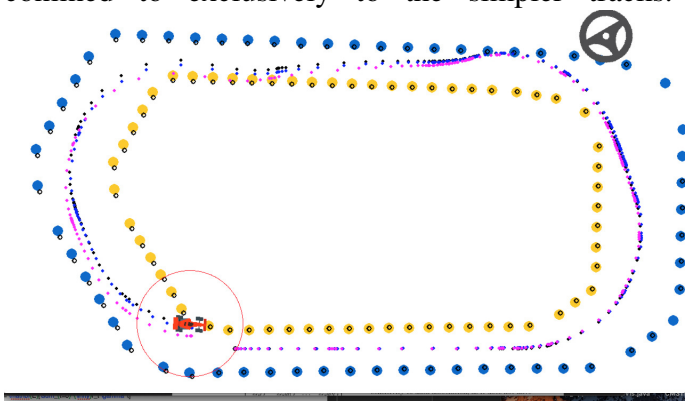


## 6.2 Reinforcement Learning

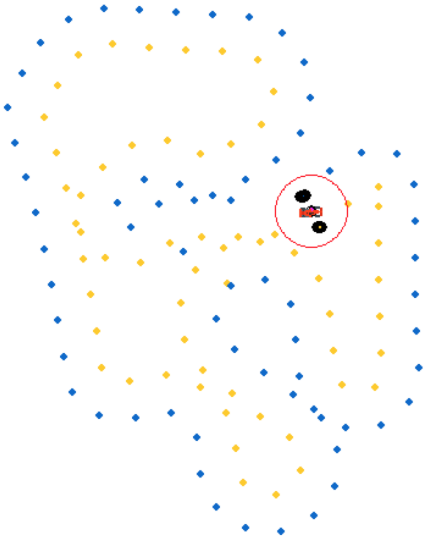
After 10 hours of training on a MacBook Pro 2015 15”, the car showed promising results. Using the green checkpoints shown below, on with algorithm described, with the simplified random track generation. The car’s path can be seen by the black path, while the SLAM path is in blue, and the Dead Reckoning in pink. The car’s racing line should also be noted in both these examples as a good racing line [Kuhn, 2017], accelerating in tactically relevant points.



This model was able to generalise to unknown tracks as well, as shown, but was confined to exclusively to the simpler tracks.



Later, a custom track generation algorithm was developed in the team to represent more complex tracks. However the model could not generalise a complex track, where the track layout become more ambiguous, an example of which is seen below.



Future work will focus on better track representations to handle such cases. But the results of training on simpler tracks are promising enough to count this as a successful proof of concept, where further parameter tuning of the model, and cost function should converge on a more robust agent.

# 7 Conclusions and future work

This project sought to use Deep-Q Learning; a Reinforcement Learning algorithm, to map observations to actions for autonomous racing, specifically to compete in the FS-AI 2019 competition. The model generated was able, after training on random, simple, tracks, consisting of spaced blue and yellow cones, as specified in the FS-AI rules, to complete a lap on an unseen track in a $2D$ simulation, following a good racing line, proving itself as a working proof of concept for such a system to be deployed in the FS-AI competition for autonomous racing. The model only assumes that cones within a 12 metre range would be detected, and triangulated with some accuracy, accounting for noise. Given previous work on the problem in creating a stereo system to do this, it is reasonable to assume this would be given in a real situation. The

model successfully deploys fastSLAM2 to transform these measurements, along with odometry readings to create a probabilistic map of its environment, to be given to a Deep Neural Network to successfully dictate actions that allow for a track to be completed. Future work should focus on the following areas

1. Producing a vector field of the agent's decisions in all locations of an un-seen track, to better understand its current limitations.

2. Creating a better map representation from the SLAM landmark estimations. A suggestion is to calculate the track boundaries mathematically and use these as the input for the NN, rather that a bitmap of cones.

3. Continue improving the reward function to generalise better. More emphasis should be placed on accelerating, as well as staying close to the racing line. A suggestion of an improved function would be to calculate the racing line numerically [Kuhn, 2017] whilst in the simulation and reward the agent based on how close it is to the racing map

4. Including physical limitations in the simulated environment will train an agent more realistically, work is currently being done by the Cardiff Autonomous Racing (CAR) team for producing such a simulation using the Unreal Engine for capturing inertia, air resistance etc as well as having a better, more accurate model of the car to train. Training in this environment will make for a far more robust agent.

5. Better hyper-parameters could be found for the Neural Network through taking a large buffer during the training phase, storing this and using it as a training set, whilst using a graph search method to find the optimal hyper-parameters [Bergstra et al., 2013] for the network. This network could then be trained in the usual fashion using Deep-Q Learning.

6. CAR is currently producing a $10^{th}$ scale test environment for the competition algorithms, immediate work should be to port this model to this car for true performance testing. This project proved as a proof of concept to pursue this technique further for the FS-AI competition, I believe the above suggestions will produce a much more robust model for competing.

# 8 Reflection

As discussed, I view this project as an overall success. It would have been preferable to have finished with a more robust model, capable of driving the more complex tracks, but I understand this to be the next phase of this project now, given the learning curve and infrastructure needed to produce this first stage, it is reasonable to have produced the work I have, in my opinion.

I had not understood SLAM properly until this project, meaning much time had to go into understanding SLAM, including EKFs and particle filters to truly understand the FastSLAM2 algorithm, to the point I could implement functions myself, as well as adjust others to purpose, this consumed production time. Especially given my weaker math background, though I pride myself on learning what's needed when needed, I cannot pretend I didn't spend many evenings trying to understand some of the math prerequisites for this.This was difficult, though was overcome in time, resorting to a white board often to imagine it's workings better.

I had previously understood Neural Networks and had a good machine learning background, though had not done Reinforcement Learning, and thus had to learn both traditional methods and the newer Deep Learning methods. This was a fun learning curve that involved building simple projects to overcome simple problems, such as cart-pole and Pac-Man. This was a natural addition to my Machine Learning knowledge and was not too difficult to understand or implement.

Building the environment took very little time, though in hindsight, more time should have been given to produce better random-tracks, given the impressive complexity of tracks later developed, more time should have been given to produce a more nuanced environment. The environment also does not represent any physical forces, which are

things I planned on including, though didn't due to time restrictions and wanting to move onto the algorithmic side of the project. This is a shame, and is work I plan on completing in the near future.

My initial plan laid out a very reasonable and achievable timescale, more effort should have been made to stick to this plan. I naturally work in sprints, neglecting work for sometime, then working through days and nights other times. I learn again from this project that I should pace myself through a project and work consistently to meet milestones on time and keep on top of the project.

Overall I am pleased with my personal progression and achievements in this project and mark the results as successful, given all limitations; time, background and skill.

# 9 Acknowledgements

With thanks to: Dr Kirill Sidorov for support during the project academically and personally, and David Buchanan who designed an improved random track algorithm for training on, as well as everyone who calmed me down whilst stressing about this project during the year.

# References

[Bast et al., 2016] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2016). Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer.

[Bergstra et al., 2013] Bergstra, J., Yamins, D., and Cox, D. D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures.

[Bibal and Frénay, 2016] Bibal, A. and Frénay, B. (2016). Interpretability of machine learning models and representations: an introduction. In *Proceedings of the ESANN*.

[FS, ] FS. Formula student. https://www.imeche.org/events/formula-student.

[FS-AI, 2019] FS-AI (2019). 2019 formula student – artificial intelligence (fs-ai) rules. https://www.imeche.org/events/formula-student/team-information/rules.

[Gobor, 2017] Gobor, Z. (2017). Finding the convex or non-convex hull of a random number of vertices simple task? MECHEDU.

[Grisetti et al., 2007] Grisetti, G., Stachniss, C., Burgard, W., et al. (2007). Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, 23(1):34.

[Ishibuchi and Nojima, 2007] Ishibuchi, H. and Nojima, Y. (2007). Analysis of interpretability-accuracy tradeoff of fuzzy systems by multiobjective fuzzy genetics-based machine learning. *International Journal of Approximate Reasoning*, 44(1):4–31.

[Kabzan et al., 2019] Kabzan, J., Valls, M. d. l. I., Reijgwart, V., Hendrikx, H. F. C., Ehmke, C., Prajapat, M., Bühler, A., Gosala, N., Gupta, M., Sivanesan, R., et al. (2019). Amz driverless: The full autonomous racing system. *arXiv preprint arXiv:1905.05150*.

[Kaur and Rampersad, 2018] Kaur, K. and Rampersad, G. (2018). Trust in driverless cars: Investigating key factors influencing the adoption of driverless cars. *Journal of Engineering and Technology Management*, 48:87–96.

[Kim, 2018] Kim, K. (2018). deep-q-learning. `https://github.com/keon/deep-q-learning`.

[Kuhn, 2017] Kuhn, P. W. (2017). Methodology for the numerical calculation of racing lines and the virtual assessment of driving behavior for training circuits for the automobile industry. *Transportation research procedia*, 25:1416–1429.

[Lapan, 2018] Lapan, M. (2018). *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd.

[Maciel, 2013] Maciel, G. (2013). How to generate procedural racetracks. `http://blog.meltinglogic.com/2013/12/how-to-generate-procedural-racetracks/`.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[Montemerlo et al., 2002] Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., et al. (2002). Fastslam: A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai*, 593598.

[Nino-Ruiz et al., 2018] Nino-Ruiz, E. D., Sandu, A., and Deng, X. (2018). An ensemble kalman filter implementation based on modified

cholesky decomposition for inverse covariance matrix estimation. *SIAM Journal on Scientific Computing*, 40(2):A867–A886.

[Paden et al., 2016] Paden, B., Čáp, M., Yong, S. Z., Yershov, D., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55.

[rasmaxim, 2018] rasmaxim (2018). pygame-car-tutorial. `https://github.com/rasmaxim/pygame-car-tutorial`.

[Sakai, 2019] Sakai, A. (2019). Fastslam2. `https://github.com/AtsushiSakai/PythonRobotics/tree/master/SLAM/FastSLAM2`.

[Siegwart et al., 2011] Siegwart, R., Nourbakhsh, I. R., and Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.

[Snider et al., 2009] Snider, J. M. et al. (2009). Automatic steering methods for autonomous automobile path tracking. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*.

[Stachniss, 2013] Stachniss, C. (2013). Slam lecture series for university of freiburg, germany. Available at: https://www.youtube.com/watch?v=V9qQc5X7O0k.

[Sutton et al., 1998] Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

[Suykens and Vandewalle, 1999] Suykens, J. A. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300.

[Thrun et al., 2005] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. MIT press.

[Wang et al., 2018] Wang, S., Jia, D., and Weng, X. (2018). Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*.

[Zhu et al., 2006] Zhu, Q., Yeh, M.-C., Cheng, K.-T., and Avidan, S. (2006). Fast human detection using a cascade of histograms of oriented gradients. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1491–1498. IEEE.