ASSIGNMENT 4: APPROXIMATION BASED REINFORCEMENT LEARNING USING TILE CODING

Daniël Zee
s2063131

Noëlle Boer
s2505169

1 INTRODUCTION

For this assignment we studied the the use of function approximation in reinforcement learning using tile coding. We start by exploring why tile coding is an effective feature construction approach and eventually we will use this method as an approach to the Mountain Car problem.
Up until now we have used tabular representations of the state-action space to represent the approximate value function under a policy v_π. This was possible because in previous assignments we were always dealing with discrete state and action spaces. But this is not always the case for real world problems, as the state space could be continuous and therefore we cannot represent all possible states in a table. To address this we represent our approximation of v_π not as a table, but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$, giving us the approximate value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for a given state s. The number of weights is typically much less than the number of states in the state space, so changing one weight changes the estimated value of many states. This generalisation of the state space can be very beneficial for learning, as not all of the states will be visited as often, but we can still learn to approximate $\hat{v}$ by visiting neighboring states.
In order to approximate any continuous function, we first need to specify an explicit objective for prediction, called the prediction objective. In the tabular case such a measure of prediction quality was not necessary, because the learned values function could converge to the true value function exactly. Because we have more states than weights, updating the weights to make one state more accurate could decrease the estimate in another state. We therefore must define a state distribution $\mu(s)$, where $\sum_s \mu(s) = 1$ to describe the importance of the error for each state. The natural objective function becomes the difference between $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$, called the mean square value error, weighted over the state space by μ, denoted by $\overline{\text{VE}}$:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 . \tag{1}$$

The goal of our function approximation is therefore to minimize $\overline{\text{VE}}$. There are many methods to do this, but we will focus on stochastic gradient and semi-gradient methods. Stochastic gradient-descent (SGD) tries to reach this goal by adjusting the weight vector slightly in the direction of each new value we observe at every timestep t, after interacting with the true function by comparing it with our current estimation. We assume that each state S_t we try, appears with the same distribution μ. This gives the equation:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{2}$$

Where α is the stepsize, or learning rate, and $\nabla f(\mathbf{w})$ denotes the derivative vector, also called the gradient, of f with respect to $\mathbf{w}$. SGD is called stochastic because the update is done on only a single example, which might have been selected stochastically. Over many examples, after many timesteps, the overall effect minimizes $\overline{\text{VE}}$ and is guaranteed to converge to a local optimum under the standard stochastic approximation conditions stated by equation 2.7 in Sutton & Barto (2018).

2 TILE CODING

For this assignment we only study the cases where our approximate function $\hat{v}$ is a linear function of the weight vector $\mathbf{w}$, meaning that for every state s, there is a real-valued vector $\mathbf{x}(s) \doteq$

$(x_1(s), x_2(s), \dots, x_d(s))^\top$, with the same length as $\mathbf{w}$. This is also called the feature vector. The approximate state-value function is then calculated by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s). \tag{3}$$

From this equation we can deduce that the gradient of $\hat{v}$, given a state with respect to $\mathbf{w}$, is just equal to the feature vector at that state. We can therefore reduce equation 2 to the simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \mathbf{x}(S_t). \tag{4}$$

The question now arises how to represent our continuous state space in terms of features. One simple approach is called state aggregation, where we group together areas in our state space with one estimated value. The estimated value for that group would be its weight in $\mathbf{w}$ and the feature vector $\mathbf{x}(s)$ would be a binary string specifying the presence of the state in a group. Using this approach we have to find a balance between generalisation and resolution, as increasing the number of groups leads to higher precision of the estimates, but decreases the amount of generalisation. A way to address this is to represent a state with features that represent groupings in the state space that overlap, known as coarse coding. Tile coding is a form of coarse coding where we extend the idea of state aggregation to multiple partitions, called tilings. Each tiling can be seen as their own state aggregation grouping, using the same number of groups, but uniformly offset from the other tiling. The vector $\mathbf{x}(s)$ is therefore a binary string with one component for each tile in each tiling and the number of components that are set equal to 1 will be equal to the number of tilings for every state s.

2.1 METHODOLOGY

We first constructed a `TileCoder` class that is able to produce the binary feature vectors for points in a (multidimentional) continuous state space. Our implementation is a modified version of the implementation by MeepMoop on GitHub, with the only difference being that we use a uniform offsetting scheme between tilings. The class accepts as parameters the number of tiles in each dimension, the limits of the values in each dimension and the number of tilings. The resulting encoders for two one-dimensional configurations are shown in figure 1.

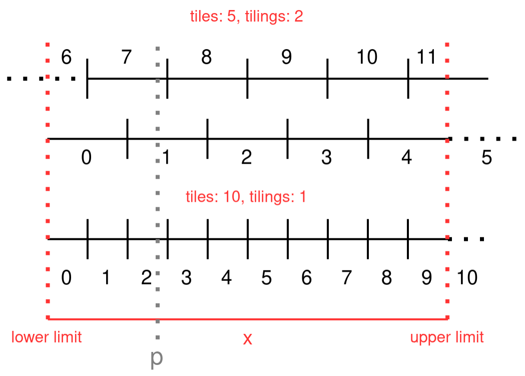

Figure 1: Visualisation of two configurations of a tile coder with a total of 10 bins with the tiles numbered in the order of the resulting feature vector.

Because of the offsets, all tilings will have one tile extra compared to the specified number of tiles to ensure that every point belongs to a tile in each tiling. The total number of tiles are therefore $n\_tiles = tilings * (tiles + 1)^{dims}$. We can see in figure 1 that both configurations divide the domain of x in an equal number of separated areas, when combining all the tilings. We will call this the number of bins with $n\_bins = tilings * tiles$. The number of bins specifies the resolution we have in a dimension. After initialisation we can provide the `TileCoder` class with a point in the

continuous space and it will return an array with the indices of the active tiles (instead of returning the full binary vector). Point p in figure 1 would for example return $\{1,7\}$ and $\{2\}$ for the top and bottom configurations respectively.
Using our tile coding implementation we performed two experiments where we analysed the efficiency of different tiling configuration on simple function approximation problems using SGD. We constructed a `FuncApproxAgent` class that accepts as parameters: the tiling specifications mentioned before and the learning rate α used in the SGD update rule specified in formula 2. The learning rate is internally divided by the number of tilings, to ensure that the rate of learning is independent of the amount of tilings used. We initialise the `TileCoder` and initialise the weight vector $\mathbf{w}$ to zero. The `FuncApproxAgent.get` method can then be used to calculate the current function approximation for a state following equation 3 and the `FuncApproxAgent.update` method implements the SGD update.
For the first experiment we use two tiling configurations to approximate the one-dimensional target function $y = \sin(x * 2\pi)$, where the values for x ranges from 0 to 1. Both `FuncApproxAgent` classes are configured for a resolution of 20 bins, one with 20 tiles using 1 tiling and the other with 10 tiles using 2 tilings. We then generate 200 random values in the range of x, get the real values from the target function, and use them to update both approximators. Afterwards we evaluate our function approximations at every value in the range of x in steps of 0.01 and plot the function curves together with the target function. We now train both approximators with 200 new random values and do this evaluation again.
For the second experiment we use multiple titling configurations to approximate the two-dimensional target function $z = \sin(x * 2\pi) + \cos(x * 2\pi)$, where both x and y range from 0 to 1. The first two configurations are the same as the first experiment but now we also test 2 tiles with 10 tilings and 14 tiles with 2 tilings. We generate random point for x and y to update our approximators and after every 100 updates we again evaluate them at every value in the range of x and y in steps of 0.01 to calculate the RMSE of our approximators compared to the target function. We plot the RMSE of all approximators against the number of update timesteps. For all experiments we used $\alpha = 0.1$.

2.2 Results

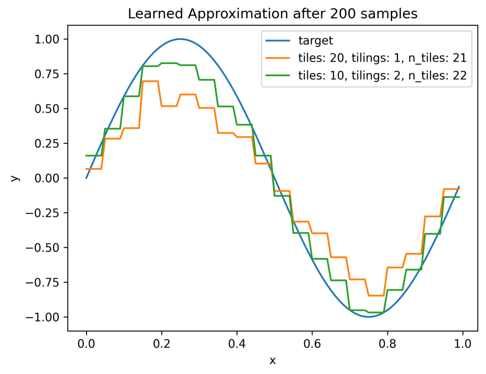

Figure 2: Learned function approximation for $y = \sin(x * 2\pi)$ after 200 updates.

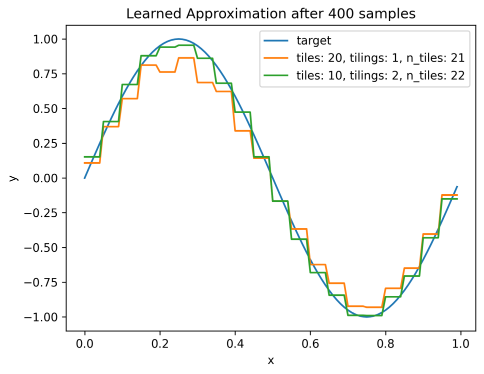

Figure 3: Learned function approximation for $y = \sin(x * 2\pi)$ after 400 updates.

In figure 2 we can see that our function approximations look like a stepwise ladder, which is to be expected as we only have a resolution of 20 bins. We see that when using two tilings, our approximations are closer to the target function compared to using only 1 tiling. This can be explained by the fact that multiple tilings overlap, so updating the approximation for one point will not only update the points that belong to the same bin, but also the points belonging to neighboring bins, as they share one of the tiles. This results in faster learning using the same learning rate and better learning less frequently areas of the domain. Figure 3 shows that after 400 updates both approximations are already a lot closer to each other. So given enough updates even with only a single tiling, which is basically just state aggregation, we do reach the best possible approximation function given our resolution.

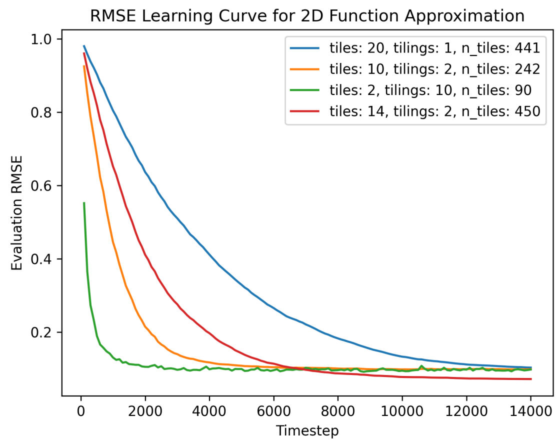

Figure 4: RMSE for the learned function approximation for $z = \sin(x * 2\pi) + \cos(x * 2\pi)$ evaluated after every 100 update timesteps.

Figure 4 shows again that a higher amount of tilings leads to faster learning given the same resolution, even in higher dimensions. An important observation is that in higher dimensions the number of total tiles dramatically decreases using more tilings, as only the number of tiles in each tiling gets raised to the power of the number of dimensions. This results in better performance using less memory. We do however see that the learning curve for 2 tiles with 10 tilings is less smooth compared to the rest. This is because the tilings have a lot of overlap so every update has an impact on the approximations of a relatively large section of the domain, which can temporarily throw some approximations off. The last configuration, using 14 tiles with 2 tilings, has about the same number of total tiles compared to the first configuration and shows that given a fixed memory size, we can achieve better resolution of our approximations, and therefore a lower RMSE, using more tilings.

3 Episodic Semi-gradient SARSA

Up until now we have always had access to the value function we are trying to approximate to perform the SGD update in formula 2. But in a real world reinforcement learning setting we usually cannot perform this exact update, because $v_\pi(S_t)$ is unknown. In the case of the SARSA algorithm, which we studied in a previous assignment, we only have access to bootstrapping targets using $\hat{v}$. In this case we can approximate $v_\pi(S_t)$ by substituting it with the TD(0) target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{5}$$

This modification does have an important effect on our gradient descent method. For true gradient descent it is required that the target is independent of $\mathbf{w}_t$, but our bootstrapping target does depend on the current value of $\mathbf{w}_t$, which implies that it will be biased and will not produce a true gradient-descent method. It includes only a part of the gradient and we therefore call this a semi-gradient method. Semi-gradient methods do not converge as robustly as gradient methods, but they will in the linear case.
For the semi-gradient SARSA algorthm, we extend this update rule to state-action values by approximating not the value function but the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$. Since we will still be using linear function approximation we can therefore derive the following update rule for one-step semi-gradient SARSA:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\right] \mathbf{x}(S_t). \tag{6}$$

where R_{t+1} is the reward received by the environment at timestep $t+1$ and γ is the discount rate. The full pseudocode of the episodic semi-gradient SARSA algorithm can be found on page 244 in Sutton & Barto (2018), with the only difference compared to regular SARSA being that we use value-function weights instead of an Q-table.

3.1 Methodology

To test the Episodic Semi-gradient SARSA we use the classic control Mountain Car environment from Gym. This is an environment with a continuous state and a discrete action space. The goal in this environment is learning how to get the car to the top of the right hill. The observation space of the environment consists of 2 variables. First we define the position along the x-axis m with $-1.2 \leq m \leq 0.6$. We also define the velocity of the car v, with $-0.07 \leq v \leq 0.07$
The three actions we can take are: accelerate to left, don't accelerate and accelerate to the right. For each timestep the agent gets -1 reward and an episode ends when the car has reached the top of the hill.
We constructed a `TiledSARSAAgent` class that implements the initialisation, action selection and update steps of the semi-gradient SARSA algorithm. We again initialise the `TileCoder` and initialise the weight vector $\mathbf{w}$ to zero. We also divide the learning rate by the number of tilings. The `TiledSARSAAgent.select_action` method implements the following ϵ-greedy action policy:

$$\pi_{\epsilon-greedy}(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_{b\in\mathcal{A}} Q(s,b) \\ \frac{\epsilon}{(|\mathcal{A}|-1)}, & \text{otherwise} \end{cases} \tag{7}$$

Finally the `TiledSARSAAgent.update` method implements the update rule in formula 6.
We wrote an experiment function that implements the remaining steps of the algorithm that runs a `TiledSARSAAgent` instance for a specified number of episodes, keeping track cumalitive rewards of each episode. The function does this for a specified number of repetitions and finally averages the cumalivitve rewards of each episode over all repetitions. We use this to plot the learning curves for multiple tiling configurations. We tested three configurations using only 1 tiling with 5, 10 and 20 tiles and the three configurations with a total of 20 bins, which we also used in the previous experiment. We ran our experiment function for 10 repetitions of 700 episodes using $\alpha = 0.1$, $\gamma = 1.0$ and $\epsilon = 0.1$. The learning curves were smoothed with a smoothing window of 31.

3.2 Results

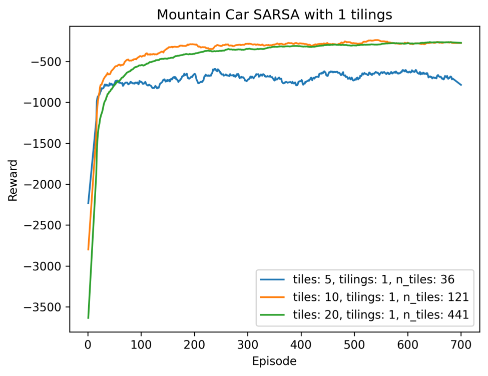

Figure 5: Cumulative reward per episode over 10 repetitions in the Mountain Car environment using Semi-gradient SARSA with tile-coding with 1 tiling.

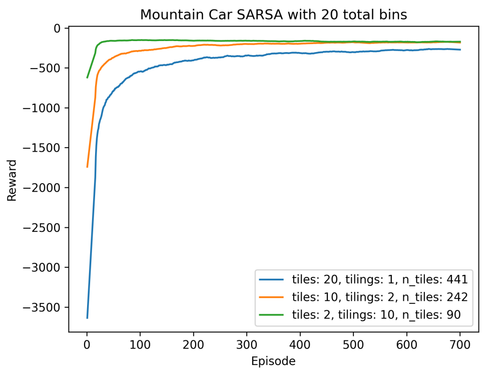

Figure 6: Cumulative reward per episode over 10 repetitions in the Mountain Car environment using Semi-gradient SARSA with tile-coding with 20 total bins.

Figure 5 shows the performance when using only one tiling, which was the same as state aggregation. We can see that a lower number of tiles results in higher rewards at earlier episode, which makes sense because we have a higher degree of generalisation. We do however see that with only 5 tiles we do not reach the optimal episode reward for the policy, indicating that the resolution of our function approximation is too low to capture all the necessary information to approximate the true value function. 10 tiles does seem to be enough and reaches the same cumulative rewards on as with 20 tiles. Figure 6 shows results that are in line with what we found in figure 4, where a higher amount of tilings with the same number of bins leads to faster learning and therefore better rewards at earlier episodes. The big difference here compared to using only 1 tiling is that the increase in generalisation does not negatively impact the resolution of our approximation and the convergence of the learned value function. So when using a higher number of tilings, we use less memory space and converge to the optimal episode reward earlier, assuming that the total number of bins is high enough.

4 Discussion

We have seen that more tilings using the same number of total bins always results in better results during our experiments. An important observation however is, that all the functions we have approximated have been fully deterministic. An interesting topic for further research would be to test different tiling configurations on a stochastic environment to see how the increased generalisation from the tilings impacts the learning in those cases.
Another important thing to note is that we have only used uniform offsets for our tiling. Figure 9.11 in Sutton & Barto (2018) shows that this results in a diagonal bias and does not create the spherical tiling around a point that would be optimal in a two-dimensional space. Choosing an asymmetrical offset would improve the tile-coding and this results in a better accuracy and therefore a higher reward. Therefore an asymmetrical tiling offset is preferred. The difference this would make compared to uniform offset would also be an interesting point for further study.
It would also be interesting to compare semi-gradient SARSA using tile-coding with other types of reinforcement learning to see how good this algorithm performs in relation to other algorithms on the mountain car learning environment.

5 Conclusion

Linear function approximation using tile coding has been shown to be an effective way to approximate a continuous value function that is easy to understand and implement. By using multiple overlapping tilings we can generalise our state space and achieve linear space complexity compared to exponential using state aggregation, while achieving the same resolution. In choosing an effective tiling scheme to use on a reinforcement learning problem it is important to first analyse the required resolution in each dimension to capture enough information to achieve a close enough approximation of the true value function and then increase the number of tilings, while keeping this minimum resolution in order to increase generalisation and speed up the learning process.

REFERENCES

Mountain Car gymnasium. https://gymnasium.farama.org/environments/classic_control/mountain_car/. Accessed: 2024-05-19.

MeepMoop. Tile coding. https://github.com/MeepMoop/tilecoding.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.