**Implementing the Cascade Reinforcement Learning Algorithm**

Anthony Testa, Kevin Galvan Cuesta, Samuel Young

Case Western Reserve University

Artificial Intelligence: Sequential Decision Making CSDS 496

Soumya Ray

12/08/21

**Implementing CSI**

This paper explores the properties of the Cascading Algorithm as a solution to the Inverse Reinforcement Learning (IRL) problem. The IRL problem refers to the inverse of traditional reinforcement learning: generate a reward function given an optimal policy for a Markov Decision Process (MDP). It is more accurate to replace “optimal” with “an expert policy that is proven to be optimal” because the application of CSI is to find the reward function of an optimally performing agent. This paper demonstrates the two main steps of CSI, the classifier and regressor, through a simple domain and explores the effectiveness and efficiency of the algorithm through research extensions, such as different regressors and domain types.

**Significance and Applications**

As a whole, IRL is an important method to train other agents to perform at the same level as a given expert policy, without knowing or learning the exact reward function of the given policy. [1] It allows for the discovery of a policy's goals without imitating every action, including irrelevant ones, as Imitation Learning does [2]. IRL is unique in that it generates a reward function that will achieve near optimal performance when used by the expert policy. In other words, this method does not directly learn the expert policy and instead learns a sufficiently approximate reward that can be used by other RL methods to generate a near optimal policy. This technique increases the malleability of how an MDP is solved by allowing the utilization of different RL or Approximate Dynamic Programming (ADP) methods for different domains given the reward function generated by an IRL method.

Common challenges faced through IRL algorithms are scalability, data scarcity, and wasted efficiency by repeatedly solving MDPs in order to obtain the optimal policies of intermediate reward functions [1]. This last challenge of wasted efficiency can be drastically reduced using the Cascade Algorithm (CSI). Rather than continuously solving the MDP every time a possible reward function is generated, CSI generates samples from several iterations of the expert policy and non-expert policies. These samples are used to define a score function q(s,a), and by taking the argmax for each state, a new policy can be defined that mirrors the expert policy.

(1)

This method provides an efficiency advantage over other IRL algorithms. The CSI also tackles the challenges of scalability and data scarcity through the steps used to determine this score function and create a reward function from the function: the classifier and regressor. These two steps will be explained in depth later; the point here is that the wide range of developed classifiers and regressors allows for high adaptability in the exact process of the algorithm to create more accurate and near expert results depending on the domain. It could be said that the CSI functions as a skeleton and the most appropriate tools are inserted into the designated slots to achieve the best results.

The Cascade Algorithm can be applied to most IRL and IRL-based AL problems. Specifically, CSI is useful for problems where the expert policy is a human agent. The agent decides what is the best action to take at a given state and acts accordingly, while also taking some wrong or unnecessary actions. Data is collected from the player and fed into a CSI, which outputs a reward function tailored to the player’s goals. A policy generated from this reward function would then be able to learn how to accomplish these goals with near expert value. In other words, the algorithm does not merely perform supervised learning to emulate the player. It instead learns why the player behaves a certain way and understands how to satisfy those reasons itself.

One application for this type of learning is self-driving cars. The goal is not to copy the exact actions of the driver; this would lead to a crash if the algorithm does not have the same initial state or failure if there is a different destination. The purpose of the algorithm is to learn how the driver drives, and what rules dictate the acceptable, and beyond that optimal, behaviors on the road. Optimality can be defined differently depending on the domain. For one example of this application, the highway driving simulator, optimality was dependent on speed and collision avoidance. The example consisted of a player car that could switch between three lanes, go off-road on either side, and switch between three speeds [1]. The performance of CSI was compared to the SCIRL algorithm, and matched its value.



Fig. 1: Graph of the average performance over 100 runs of the highway simulator problem. CSI narrowly maintains the best performance. The number of samples has a drastic improvement up to 100, but beyond this the improvement decreases exponentially until it reaches an asymptotic upper bound[1].

An important note from the research paper is that this performance for the CSI was achieved using standard inputs for the classifier and regressor, meanwhile the SCIRL utlilized “handmade features and a custom classifier”[1]. This once again shows the adaptability of CSI.

**Cascade Algorithm**

**Domain**

The domain refers simply to the state-action map of the MDP which is solved through the utilized reinforcement learning algorithm. In the above application, the domain was the highway space and the actions available in that space. For this report, the domain is a simple grid space and the actions are moving from one grid square to an adjacent square. Certain squares contain events that incur a reward. Policy iteration is utilized to create an expert policy for this gridspace. Two datasets are generated from the MDP.

The first is [1], which is the action chosen for an input state chosen by the deterministic expert policy, essentially the optimal action for any given state. The expert policy is run multiple times from random initial states in order to collect data that spans the majority of the domain. This dataset is the input for the classifier.

The second is [1], which is a collection of data points that show what states can be transitioned to from state through action . The purpose of this dataset is to understand the dynamics of the entire state space; therefore, using the expert policy would limit exploration to only the optimal actions. Instead, this data set is generated through random policies in order to explore the possible actions in every state. For a state action pair, the number of data points transitioning to over all possible states are proportional to the transition function of the expert policy. This quality will allow the score function generated by the classifier to be used to invert the Bellman Equation without explicitly knowing the transition function.

**Classifier**

The classifier is a machine learning model that associates an action with an inputed state. Specifically, CSI uses a score function-based multi-class classifier (SFMC), which forms a score function that associates a value between the action state pair. The classifier learns from the data set; the score for a pair, is positively correlated to the number of datapoints containing this pair over all data points for this state . A “classifier policy” can then be generated by utilizing the score function as the state action value function, and taking the greedy choice as shown in (1). This only knows optimal choices cased on the provided pairs. While it may work truly optimally for a simple domain such as the grid space, it would falter under a larger, more stochastic domain where there is not always one optimal action per state.

**Regressor**

The components discovered so far could provide a solution for the IRL problem; however, they ignore the transition function of the original MDP. Including the transition function provides a solution much nearer the expert policy [1]. Another way of interpreting this statement is that the previously discovered score function serves as the optimal action function of , and thus it can be used in the Bellman Equation

(2)

The reward function can then be found by inverting (2).

(3)

As the transition function is unknown, this equation cannot be solved directly. Instead, is utilized to create a collection of data points, each corresponding to the reward of a state given the state itself, the action chosen, the state it transitions to, gamma, and the next action chosen by the classifier policy.

{ (4)

These rewards are sampled in relation to the transition function P(·|,), thus they accurately (within an error margin) represent the reward across the state space. A regressor of some form is used to consolidate these points into an approximation of , often a least-squares regressor. This reward function can be used by a RL algorithm to generate a near expert policy. The purpose of this paper is only to prove the reward function generated closely approximates the original reward function of the domain and how to improve the accuracy of .

**Involvement**

As this is a relatively simple algorithm, there are few components to implement. Sam Young created the domain and expert policy and generated the necessary data sets. Anthony Testa chose the SFCM and regressor and implemented each. Both worked together to debug as well as improve the algorithm by altering the domain, classifier, and regressor.

**Experiments**

**Domain Choice**

The domain chosen for this paper underwent several revisions. Originally, the team hoped to find a library that could handle fast calculation of an expert policy for an intermediate sized state action space(several hundred states, 10s of actions per state). Such a domain would create an adequate simulation for the type of expert policy CSI would be useful for solving. This type refers to human driven expert policies for complex situations such as driving as referred to above. Such a library that seemed capable of handling this computation was CRAAM (C++ Robust and Approximate MDP) [3] and its python extension RAAM [4]. The code provided several ways to form an MDP including samples and transition matrices. The policies could be solved through value or policy iteration. The ultimate downfall of this decision was that the code was four years old at least, and the most recent version did not match the white pages. As a result, obtaining results or interpreting the outputted policy was unnecessarily difficult and confusing. A benchmark included with the CRAAM library output an expert policy for an MDP as originally desired, however, the team was unable to discern how it was accomplished. Although the team began testing the classifier with this policy, it was decided to abandon this domain in favor of one that could be more easily controlled and changed.

Further exploration of possible domains led to the discovery of a simple domain that only required numpy [5]. This simple domain was a grid space with twelve states and actions in all cardinal directions where applicable. The main issues were that the reward was only based on the state instead of the state action pair, and that any increase in state size made it impossible for value iteration to achieve a solution with a reasonable error margin. The first issue was solved by adjusting how the reward was decided and the second was solved by implementing policy iteration instead of value iteration. These solutions allowed for the creation of a simple 4 by 4 grid domain with the following reward features.



Fig 2: The grid space domain. Agent is a dirt-biker trying to get home. Moving through the mud hole has high negative reward, medium negative reward for adjacent squares. Hitting the deer has a high negative reward and terminates the policy. Reaching home has a high positive reward.



Fig. 3: Although a very simple domain, the MDP becomes complex looking for only one state and three actions. The transition probability is 90% for any action to desired state s’, and the remaining 10% is divided equally among the other possible states to be transitioned to. [6]

Initially, was collected from twenty iterations of the expert policy starting from random initial states, and was collected from 100 iterations of random policies starting at different states. The 20 iterations proved to be insufficient to map the score function for every state, so the iterations of the expert policy were increased to 100 as well.

**Classifier**

The classifier chosen for this paper is a standard SFMC classifier, utilizing a Support Vector Machine to determine the decision boundary for classification. This SVM is imported from scikit [7] and uses as the kernel coefficient with a radial basis function kernel. is used as the training data, with the states being features and the action taken in each state being the values to predict on. The probability of possible outcomes for each action given an input state is then used to create the score function. This may be the shortcoming of our reward values ultimately, as this more closely represents the probability of a path rather than the value of the path. This will alter the scale of our results, but should still provide values. Given an input state the probability that every possible action is chosen at that state is then predicted by the SVM, with the max being the action predicted. While a relatively simple process, some issues arose. The base implementation of the classifier assumed all states utilized all possible actions, which was untrue and skewed the score function. This problem was solved by implementing a check to see if a state action pair could generate a score, and if not, was the action possible at that state. The values predicted by the SVM are also not tracked in the scikit library, making prediction more difficult to interpret. This necessitated the use of a map to track which features were being chosen, and to assign their score function values. Once this process is complete and these fixes implemented, the score function and classifier policy are generated according to equation (1).

**Regression**

The regressor chosen for this paper is a simple least-squares regression function, otherwise known as a linear regressor [7]. Once the score function is generated, reward values are created through equation (4). The results are input to a Dataframe for easy regression. The Dataframe points consist of the current state, current action, and reward value. These parameters are split into features and values. The features are the current state and action (encoded as a number according to the action map from the classifier) and the values are the calculated reward values at each point. These lists are then fit by a linear regressor from scikit to generate the reward function .

**Results**

The support vector machine used was able to successfully choose an action to perform given a state every single time, with an value of 1.0. This suggests potential overfitting, however is not of real concern, as what is really of interest is the reward function. The linear regression had an value of just 0.15 on average, suggesting as noted above that the actual reward values were skewed by the score functions inability to reflect the real world value of taking a specific path.If we look at the difference in reward values between predicted and actual, we can see that the score function is outputting rewards that are frequently not close to that of the actual state.

Fig. 4: difference in reward values generated by equation (4) and actual reward

Average difference 0.27372

These results suggest that the chosen CSI classifier or regressor were a bad configuration for this domain. It was expected that the linear regression would generate a poorly fitting function as many states have a real reward of zero, creating a flat reward graph with spikes at several states. In practice, this led to a problem of multicollinearity. Since the domain space was small in size, this should be expected. Unfortunately, standard linear regressors are unable to deal with divide-by-zero errors and will default to dropping the observation entirely if this arises. Moreover, the failure of the score function suggests an issue with the classifier. However, the classifier is able to create a policy that behaves optimally, so the issue may not be the classifier itself but the team’s interpretation of what the score function is. The score function, by the team’s understanding, is giving a value to the association between an action and an input state. This association was found using the prediction probability method in scikit, which calculates the probability of an action occurring at a certain state. While the reward values and the real rewards do not have to be exactly the same, the reward values generated from the state function and optimal classifier policy should be proportional, yet these results show a random distribution. 

Fig: 5. Rewards derived from linear regressor versus actual rewards

Average difference 0.07031

The results of the reward function reflect the above issue. Almost no state action pairs reflect the correct reward, with some even having a negative reward that should be negative and vice versa. Upon reflection, the probability function may not be a score generated by the classifier, but how the classifier relates the sample data points () to each other. Another method may be possible to generate the score function values that the team cannot see. The research extensions aim to check this conclusion by swapping out the classifiers and regressors to see if the correlation improves.

**Involvement**

The experiments to decide and tweak the chosen domain were performed by Sam Young. The implementation and refinement of the classifier and regressor were performed by Anthony Testa. Kevin Cuesta aided both in debugging and interpreting the results created from the base implementation[1].

**Classifier Machine Exploration**

Anthony Testa and Sam Young

In the SCI algorithm, both the classifier and the regressor are highly customizable. In this research extension, this paper explores other classifiers than SVM.

**Gaussian Naive Bayes**

The Gaussian Naive Bayes classifier predicts the likelihood of the “attributes” occurring for a class label. In this case, the attributes are the desired state and action and the class label is the data points.

(5)

This equation holds two assumptions: the attributes are independent of each other, and the probability is not affected by any latent attributes [8].

Similarly to the SVM, the class label is predicted by taking the maximum probability of the class itself multiplied by equation (5). What makes this classifier Gaussian is the probabilities of each attribute given the class label is assumed to be on a Gaussian distribution.

(6)

Running with this classifier yields.



Fig. 6: Difference between reward values and the real rewards for a data sample set using Gaussian Naive Bayes

Average difference 0.071482

On the surface, the Gaussian Naive Bayes classifier works much better than the SVM with a lower average difference between estimated and real reward values. Almost no differences are greater than one, meaning no estimated reward was given an incorrect reward parity. More points have false positive rewards, but this could be a random effect. What this distribution does show is that the method used by SVMs to map the values of the state action pairs was not the issue. Assuming the distribution was Gaussian rather than uniform did not impact the spread of values in any significant way. As the Guassian classifier suffers the same randomness, the unidentified issue in how the team generates the score function is likely our main point of failure.

**GridSearch(CV)**

GridSearch is an exhaustive classifier that does not perform a single score calculator, but multiple over all possible selected hyperparameters and chooses the best one. This is performed by first applying K-fold Cross Validation. The samples from the are split into k partitions, in one iteration a single partition is isolated for testing, and the other k-1 partitions are used for training the classifier. With the test partition serving as a base, each iteration refines the best values of the hyperparameters before feeding them into GridSearch to obtain a score function.



Fig. 7: Difference between reward values and the real rewards for a data sample set using SearchGridCV

Average difference 0.154464

The reward values generated through SearchGridCV have both the highest number of accurate rewards as well as the largest range of incorrect reward values. The team expected that this classifier would work better due to the grid space nature of the domain, and the results both support and disprove that idea. An interesting note is the almost parabolic shape of the error. The shape starts at zero, curves down to an average of -1, then curves back up to zero at point 22. The error then creates another parabola in the opposite positive direction. This shape may be correlated to the segment-like nature of the value generation for this classifier. The default is five partitions, and the third corresponds almost perfectly to the middle srtech of accurate rewards. However, it is more likely the score function issue is skewing these results, so these conclusions cannot be validated.

Now through the third classifier, there is little more to report on how the values are constructed as there appears to be a core issue in the score value calculation. From the results found, Gaussian Naive Bayes appears to generate the best results from this small state space. A revaluation after correcting the methodology from which the score function is extracted would allow for a more accurate interpretation

**Regression Alternatives**

Kevin Galvan Cuesta

The authors’ work shows the use of a simple least-squares regressor to create a generalization of the reward function, RC, from the data points in {((sj , aj), rj)j} over the whole state-action space. In section 6.1, they explicitly note that this regression technique is an unduly rough estimate especially as the state-action space increases. As we are by no means looking to overfit the data, alternative estimation techniques have been ubiquitous in data science research to find smoother estimates.

As a replacement to the regressor, we offer a few alternatives to solve this problem. The following regression alternatives were created using the scikit machine learning models [7]. Each regression used the features and values created by the classifier to fit the model. Particular differences in implementation are described below. At the end of each regression section, a graph showing the differences between the real and estimated rewards is presented.

**Ridge Regression**

Ridge regression offers us a way to deal with multicollinearity. Simply put, the smaller the size of the model, the more likely we are to come across multicollinearity as samples are increasingly identical. As our model is small in size, linear regression may exclude particular observations as a result of a divide-by-zero error. Thus, we should expect higher variance in the final output of the regression as it can ignore a significant portion of the data set.

To fix this problem, ridge regression first includes the standardization of observations in our dataset. It computes this by first subtracting the mean and then dividing by the standard deviation to center 0. In practice, this can be solved in a multitude of ways. For example, increasing the observation subspace will decrease the likelihood of a collinear sample. Doing so by further dividing and specifying the Markov Decision Process into smaller parts, in our case increasing the number of possible states [9].

As in ordinary least squares, we compute the coefficients via the standard form:

During ridge regression we simply add the identity matrix across the center of the matrix multiplied by a small factor, *k*.

From this step, we may proceed as normally in solving the original least squares regression with in place of where *R* is the correlation from *X’X*. This provides an effective way to deal with multicollinearity by equivalently shifting each observation along a different axis. To avoid large changes in estimation, k is usually kept small. The literature states that a value of about 0.3 is sufficient to offset multicollinearity without substantially affecting the regression [9].

Once the regression has completed, the standardization is undone to provide the results in our desired space.

In ridge regressions, the hyperparameter alpha determines the balance between minimizing the residual-square-sum or the coefficient-square-sum. After testing multiple values of this coefficient, we found little to no difference between the results. Thus, we left this value at 1.0 [7].



Fig. 8: Difference between reward function values and the real rewards for a data sample set using Ridge Regression

**Bayesian Linear Regression**

The Bayesian Linear Regression, sometimes called Bayesian Ridge Regression, builds off many of the concepts previously presented. To compute this regression, the process first involves ensuring that each requirement of the ridge is met (standardization). From the generated coefficients, we sample predictions such that the expectation for each regression coefficient is 0 and that its variance is 1/k. We can do so by sampling from a normal distribution given the information about our data and coefficients.

Its normal form is:

Where *o2* is the variance or standard deviation squared. Through this model, the regression is not trying to estimate best candidates for the model parameters and instead is attempting to discover the posterior distribution, as stated before, from the features and values of the original model. Thus, by Bayes rule, it is equivalent to the prior multiplied by the likelihood of the observation, dividing by a normalizing constant.

Finally, to return to our desired space and create predictions from this form of regression, we simply need some way to estimate continuous values for rewards given the Bayesian model. Instead of assuming that the output coefficients are single points as done in previous models, the Bayesian regression model assumes that each coefficient is being drawn from a distribution of possible coefficients and thus selects the one with most likeliness, or where *X* are the features and *y* is the predicted rewards.



Fig. 9: Difference between reward function values and the real rewards for a data sample set using Bayesian Regression

**Gaussian Process Regression**

The Gaussian Process Regression takes a different approach to the general linear models and fits remarkably close with the Bayesian Ridge Regression process. This model warranted particular attention as it is another approach (in addition to the normal Ridge Regression) that works well in smaller data sets.

The process is quite similar to the Bayesian Linear Regression in that it estimates the posterior distribution using the following form:

It should be noted that this distribution is not an estimation about the parameters, but instead calculates the probability distribution of all the *functions* that may be admissible. Thus, the final estimations in the regression are the highest likeliness estimates from the function distributions.

Our implementation made use of standard Gaussian procedure techniques. We include both the dot-product kernel, which is a simple dot product of the input matrices, and a white noise kernel, which was designed by the scikit team to filter noise in the data that is independent and identically distributed [7].

By adding together these two kernels, we still obtain a valid kernel, i.e. since both the dot product kernel and the white kernel are positive semi-definite, the resulting sum of both kernels is another valid, positive semi-definite kernel.



Fig. 10: Difference between reward function values and the real rewards for a data sample set using Gaussian Process Regression

**Results**

The best performers among the four regression models were the Gaussian Process regression and the Ridge regression. Since both of these measures were designed with collinearity and small datasets in mind, the results closely matched our expectations.

Overall, the Ridge regression performed best with an average R^2 value of 0.2986. Through the shifting procedure, we were able to avoid close to all instances of a divide-by-zero error. An analysis from the data shows that this problem was highly prevalent. In some experiments, our regular linear regression had almost 53% of the observations removed as a result of multicollinearity. Thus, the resulting linear regression could not accurately estimate the coefficients with the entirety of the dataset. Instead, the ridge regression was able to predict with a large majority of the data, only excluding at most 1 or 2 observations. In many cases, none were dropped.

Second, the Gaussian Procedure regression performed only slightly worse than the Ridge regression model. As the procedure has historically done well in small data sets with high levels of multicollinearity, we did expect it to perform well. A variety of factors could have influenced this result. First, is the inclusion of divide-by-zero observations, and second is the non-parametric nature of the model. By exchanging the parametric models with their probabilistic distributions, we were able to smooth the model. The Bayesian model, while performing a similar procedure, did so on a much smaller subsection of the data. Moreover, we were able to reduce noise in the data given our kernel. Possible alternatives to this kind of kernels are a subject of future research.

Lastly, our Bayesian regression model performed the worst by far. Its average R^2 was only a meager 0.0722, thus significantly worse than even the linear regression model. We attribute this error to the use of a simple linear model as opposed to a shifted model as in ridge regression. Overall, this model had many similarities to the ridge regression and gaussian process regression, but had no means of evaluating observations with collinearity and divide-by-zero errors. Thus, a significantly smaller proportion of the observations were used in the final regression.

Taken together, these results show that the Cascaded Supervised Learning technique is vulnerable to multicollinearity and small state spaces. To solve this, we’ve recommended alternative methods for increasing the size of the state space in the introduction as offered by NCSS [9]. However, if the learning task is still suited to a small state space, two regressions, Ridge and Gaussian Process show promising results. In comparison to other standard regression methods, these are able to successfully deal with multicollinearity, thus preserving a higher proportion of the observations to be used in estimation.

**References**

Klein E., Piot B., Geist M., Pietquin O. (2013) A Cascaded Supervised Learning Approach to Inverse Reinforcement Learning. In: Blockeel H., Kersting K., Nijssen S., Železný F. (eds) Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2013. Lecture Notes in Computer Science, vol 8188. Springer, Berlin, Heidelberg. <https://doi.org/10.1007/978-3-642-40988-2_1> [1]

B. Piot, M. Geist and O. Pietquin, "Bridging the Gap Between Imitation Learning and Inverse Reinforcement Learning," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 8, pp. 1814-1826, Aug. 2017, doi: 10.1109/TNNLS.2016.2543000. [2]

Petrik, M. (2018). *CRAAM*. Bitbucket. Retrieved December 8, 2021, from <https://bitbucket.org/marekpetrik/craam/src/master/> [3]

Petrik, M. (2017). *RAAM*. Github. Retrieved December 8, 2021, from <https://github.com/marekpetrik/RAAM> [4]

St-Amant, F. (2021). *How To Code The Value Iteration Algorithm For Reinforcement Learning*. Medium. Retrieved December 8, 2021, from <https://towardsdatascience.com/how-to-code-the-value-iteration-algorithm-for-reinforcement-learning-8fb806e117d1> [5]

Kirsch A. (2017). *BlackHC*. Github. Retrieved December 9, 2021, from <https://github.com/BlackHC/mdp> [6]

Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. “Scikit-learn: Machine Learning in Python” in *Journal of Machine Learning Research,* vol. 12, pp. 2825--2830, 2011 [7]

A. H. Jahromi and M. Taheri, "A non-parametric mixture of Gaussian naive Bayes classifiers based on local independent features," *2017 Artificial Intelligence and Signal Processing Conference (AISP)*, 2017, pp. 209-212, doi: 10.1109/AISP.2017.8324083. [8]

“Ridge Regression” *NCSS Statistical Software Manual,* 2018, pp. 335-355. [9]