Improved Search Heuristics via Imitation Learning

Adeem Jassani
MSc in Applied Computing
adeem.jassani@mail.utoronto.ca

Mahesh Jayasankar
MSc in Applied Computing
mahesh.jayasankar@mail.utoronto.ca

Naveen Thangavelu
MSc in Applied Computing
naveen.thangavelu@mail.utoronto.ca

1 Abstract

Our proposal focuses on learning effective heuristics for minimizing search costs in tree-based navigation, an essential technique for motion planning in robotics. To this effect, we examine SaIL, an algorithmic framework for learning heuristics adaptively to the information obtained during the search[1]. Firstly, we modify the loss function used in SaIL to encourage admissibility. Secondly, we experiment with the network architecture and the input state representation. We use an image patch of the area around the current search node as a feature in combination with a CNN-based architecture[1]. We investigate the potential improvements obtained through these methods and evaluate it against the baselines suggested in the paper.

2 Introduction

Formulation of robotic path planning as Graph search

The problem of robotic path planing can be formulated as a graph, where vertices V represent physical robot configurations and edges E represent potentially valid movements of the robot between these configurations. Thus, path planing is equivalent to computing a path from the start vertex v_s to the goal vertex v_g. It is important to note that firstly, all vertices in this graph are not valid as the corresponding physical configurations may have obstacles for example walls and secondly, the robot can only check the validity of its current neighbors. Therefore, there is no explicit graph representation and an implicit graph is used which is represented by a successor function $Succ(v)$ which returns a list of outgoing edges and child vertices for a vertex $v \in V$ and an evaluator function called $Eval(e, \phi)$ which computes the validity of these edges where ϕ is the information sensed from the environment used by the function. Thus, the graph is constructed dynamically during the search by repeatedly expanding vertices using $Succ(v)$ and checking the validity using $Eval(e, \phi)$. Since it is typically expensive to query $Eval(e, \phi)$ (requires complex geometric intersection operations by the robot), it is important to minimize these queries.

Search as Sequential Decision Making under Uncertainty

The information available at the current iteration can be represented as a concatenation of three lists: $\{\mathcal{O}, \mathcal{C}, \mathcal{I}\}$ where $\mathcal{O}$ is the set of candidate vertices to be expanded, $\mathcal{C}$ is the set of vertices already expanded and $\mathcal{I}$ is the set of invalid vertices found till the current iteration. The general search algorithm can thus be formulated as first selecting a vertex v from the candidate vertices $\mathcal{O}$ by invoking $Select(\mathcal{O})$ following which, the lists are updated accordingly. The algorithm terminates when v_g is reached i.e., when v_g is appended to $\mathcal{O}$.
The task can be formulated as of selecting the appropriate node for expanding as an MDP where the state $s_t = \{\mathcal{O}, \mathcal{C}, \mathcal{I}\}$ and action a_t is the vertex $v \in \mathcal{O}$ to expand. Each state transition is an expand

[1]The source code with our changes can be found here, https://github.com/naveentvelu/SaIL

operation where the state transition function $P(s_{t+1} \mid s_t, a_t)$ is determined by the underlying world ϕ sampled from the prior $P(\phi)$. **Thus, the objective is to learn a heuristic policy (a vertex selection strategy) that minimises the total expansions (all expansions are unit cost) while reaching the goal**.

For the current problem statement, the underlying world ϕ is a grid with interspersed obstacles and therefore path planning is equivalent to finding a path from v_s to v_g with minimum transitions while avoiding obstacles. It is also important to note that a prior distribution $P(\phi)$ is used to sample ϕ because the policy specific to the given distribution only (for certain kinds of obstacles) is learned.

3 Related Work

SaIL(Search as Imitation Learning [1])
Learning such a policy using model-free Q learning requires lots of data and is slow to converge. Instead, the policy is learnt by imitating a *clairvoyant oracle*, an algorithm which has access to the full state ϕ and can compute the optimal path efficiently. The algorithm is required to be efficient enough so that the training samples can be collected in reasonable time. This can be achieved for the current problem statement (grid with obstacles) as the optimal path can be computed from every vertex to the goal state efficiently using backward Dijkstra given the full grid ϕ.

Therefore, the policy is learnt by learning $Q(s, a)$ to imitate $Q^{COR}(v, \phi)$ where action a is the selected vertex v and s is the concatenated list defined above. $Q^{COR}(v, \phi)$ is the cost of the optimal policy computed by the oracle using the complete information ϕ. Since, the dimensionality of the state space is very high, the state space is featurized and a parametric $Q_\theta(\hat{s}, a)$ is learned where $\hat{s}$ is the feature vector obtained from s_t i.e., $\hat{s} = f(s)$. A neural network is used as the parametric Q_θ with the state feature $\hat{s_t}$ and the action a in the input layer and $Q_\theta(\hat{s}, a)$ as the single neuron in the output layer. The neural net is learned by minimising the squared error between $Q_\theta(\hat{s}, a)$ and $Q^{COR}(v, \phi)$. The learned policy then is simply the action which minimises the learned Q value for the current state.

As SaIL does not explicitly try to minimize the admissibility violations, we **modify the loss function by adding the admissibility constraint** to the existing loss function.

The original SaIL framework uses a collection of linear features that represent the state of the search as inputs to a neural network to obtain the heuristic. **We propose a modification to this heuristic-prediction model by including an image-patch input.**

4 Methodology

4.1 Admissibility Regularization

In SaIL, the heuristic is learnt through a neural network architecture that doesn't account for the monotonicity and admissibility requirements of a heuristic function. We incorporated the admissibility property by adding regularization term on the loss function during optimization, inspired by the conservative loss regularization techniques in Conservative Q-Learning [2].

The *clairvoyant oracle planner* used in SaIL, computes the optimal cost-to-go from any vertex to the goal state by employing *backward Djikstra's algorithm* and stores in the look up table $Q^{COR}(v, \phi)$ for every vertex $v \in V$ in the world ϕ using dynamic programming.

$$\hat{\theta}_{i+1} = \arg\min_{\theta \in \Theta} \mathbb{E}_{(s_t, a_t, Q^{COR}) \sim \mathcal{D}} \left[(Q_\theta(s_t, a_t) - Q^{COR}(v, \phi))^2 \right] \tag{1}$$

The above equation tries to learn a parameterized function $Q_\theta(s_t, a_t)$ which approximates $Q^{COR}(v, \phi)$. The above loss function does not explicitly minimize the admissibility violations that incur using the current heuristic policy.

$$\hat{\theta}_{i+1} = \arg\min_{\theta \in \Theta} \mathbb{E}_{(s_t, a_t, Q^{COR}) \sim \mathcal{D}} \left[(Q_\theta(s_t, a_t) - Q^{COR}(v, \phi))^2 \right] + \lambda ReLU \left[Q_\theta(s_t, a_t) - Q^{COR}(v, \phi) \right] \tag{2}$$

where λ is the regularization parameter and

$$ReLU(x) = \max(x, 0)$$

We add the admissibility constraint to the loss function as a regularization term to explicitly reduce the admissibility violations by penalising only the over-estimations of the current heuristic policy.

4.2 Image Patch

In SaIL, a 17-dimensional input vector is passed as the input to the neural network, which includes the search features (coordinates of the vertex, coordinates of the goal, depth of the search tree, etc.) and the environment features (nearest obstacle derived from the Invalid list). As these features alone might be insufficient to compute the best heuristics, we include a patch-based input feature that will provide additional information. This patch-input is a 3-layer encoding of a square region around the current search point. Each layer contains binary data that inform whether the point is explored, unexplored or an obstacle. The dimensions of this patch feature is therefore $3 \times p \times p$, where p is the size of the patch. For symmetry, p should be kept as an odd number.

This patch can be visualized as a coloring of the surroundings, with different colors given for different type of tiles on the grid. An analogy can be drawn to image data, and hence we term this input feature as an image-patch feature.

A sketch (Figure 1) is provided that describes the architecture of the neural network model. The existing linear features are passed through a two-layer Multi-Layer Perceptron (ReLU activation) of layer sizes 100 and 50. The image-patch input is fed into a two-layer CNN (ReLU activation) of Kernel sizes 3, Padding value 2, and channel sizes 6 and 10. The output shape of this CNN is dependent on the size of the image patch. This intermediate layer is flattened and fed into a Linear layer (with ReLU activation) that outputs a vector of size 50.

The two outputs are concatenated into a single linear input, which is then passed through a final linear layer to obtain the heuristic value.

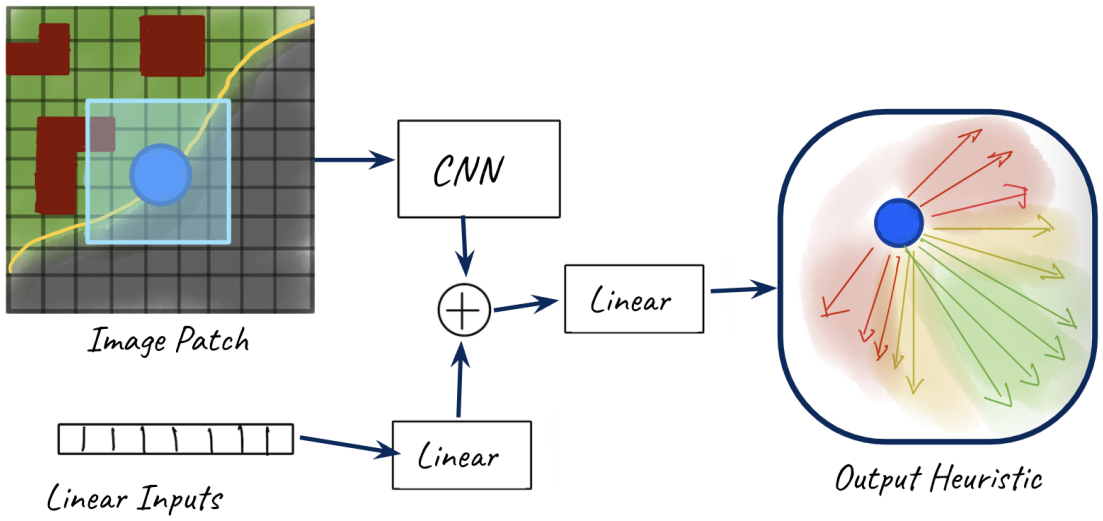

Figure 1: Image-Patch Input architecture

4.3 Code Improvements

4.3.1 Modernization and Standardization

1. **Ported to Python 3**: Since the SaIL repository was written in Python 2, we updated the code so that it is compatible with Python 3. This involved updating the imported libraries to the latest Python 3 versions and modifying the portions of code that referenced deprecated behaviour.
2. **Tensorflow to Pytorch**: The original implementation of the learning heuristic was in Tensorflow 1(which is deprecated), we provide an equivalent implementation in Pytorch upon which our proposed modifications to the architecture were also implemented.
3. **Quick-Start Readme**: We have documented all the steps required to install and get the code running, and have summarized these steps in the Quick-Start section of our Readme.
4. **Reproducibility**: The code in the authors' repository was not reproducible, however, we have added random seeds for the relevant libraries to make the code reproducible.

4.3.2 Code Optimizations

1. **Optimized oracle generation**: On finding that the oracle generator step of the pipeline was slow: 1h for 1 environment type, we optimized it using multiprocessing and achieved significant speedup without additional overhead. We tested our optimization by running the oracle generation step on the comps0 cluster (comps0.cs.toronto.edu) with 4 and 10 workers and achieved a 4X and 10X speedup respectively, showing that the achieved speedup is proportional to the number of cores.
2. **Optimized rollouts**: On profiling the training code, we found out that the data collection step involving rollouts takes a lot more time than actually training the network. We, therefore, optimise by doing the rollouts from the example environments in parallel using multiprocessing. We observe a speedup slightly lesser than the number of cores which shows that there is some overhead for the parallelization. However, due to concerns over high memory usage, we have not used this optimization in the generation of our results.

5 Evaluation

Our modifications to the SaIL algorithm are being compared against Manhattan distance heuristic, Euclidean distance heuristic and Baseline SaIL.

The source paper compared performances across a number of different environments: Alternating Gaps, Single Bugtrap, Shifting Gaps, Forest, Bugtrap+Forest, Gaps+Forest, Mazes, Multiple Bugtraps. These environments are to be also used in the evaluation of our proposed method.

5.1 Baseline Results Reproduction

We have reproduced the results for 7 out of the 8 environments in table 1 with the same settings as mentioned in the SaIL paper. We also include the results for the naive Euclidean and Manhattan heuristics for comparison with heuristic weights 1 and 91. A weight of 1 implies that the cost and heuristic are weighted equally whereas a weight of 91 implies a greedy strategy with respect to the heuristic.

We observe that baseline SaIL is able to outperform the naive heuristics in 4 out of the 7 environments and performs competitively in the other 3 environments. The greedy Manhattan heuristic gives the best performance for these 3 environments but performs poorly in most of the other 4 environments. Secondly, we observe that not all environments are of similar difficulty. We observe Alternating gaps and Forest have low average expansions; Mazes, Single bugtrap, Shifting gaps and Bugtrap forest have moderate average expansions and finally, the Multiple bugtraps environments has the highest average expansions.

Environments	Euc (1)	Euc (91)	Man (1)	Man (91)	SaIL
Alternating gaps	16832.32	5288.04	3308.79	6231.5	**368.58**
Forest	12782.78	328.6	460.19	**321.02**	364.16
Mazes	12206.8	1026.25	4148.84	**952.52**	1128.53
Single bugtrap	14166.93	1151.74	1309.72	**1097.8**	1458.5
Shifting gaps	13068.15	2577.79	1606.05	2979.76	**933.49**
Bugtrap forest	18598.15	2378.52	3707.33	2006.79	**1183.6**
Multiple bugtraps	19232.25	2743.01	3763.44	2538.26	**2310.82**

Table 1: Reproduced Results for SaIL Baseline (seed value 1). Table shows average expansions on different environments for the test sets (best in bold). Planning parameters : map size: 200 × 200,Ttrain = 1100, Ttest = 20000. Data sizes are: train(200), test(100), validation(70). SaIL parameters are - k: 50, β_0 = 0.7. Training iterations for SaIL N: 15 iterations.

Lambda	Percentage of Admissibility Violations	Average Expansions
0 (No regularization)	25.97 ± 19.15	356.60 ± 12.22
1	26.47 ± 14.00	352.754 ± 11.50
10	19.62 ± 7.54	365.76 ± 23.84
10^2	8.98 ± 6.72	367.65 ± 40.56
10^3	4.59 ± 4.65	367.49 ± 8.57
10^4	$\mathbf{1.76 \pm 1.33}$	452.82 ± 115.26

Table 2: Comparison across different regularizations (Averaged across seeds [1, 2, 3, 4, 5]) Planning parameters: map size: 200 × 200,Ttrain = 1100, Ttest = 20000. Data sizes are: train(100), test(100), validation(70). SaIL parameters are - k: 50, β_0 = 0.7. Training iterations for SaIL N: 15 iterations.

We note that the results may not exactly match those of the original paper, not only due to the lack of seeding but also as a consequence of changes in the code architecture upon migration to Python 3.

5.2 Admissibility Regularization

We experimented with the Forest environment provided in the SaIL paper by varying different regularization parameter values from Equation 2 to encourage the admissibility of the heuristic policy. We ran each setting of a lambda with five different seed values (1, 2, 3, 4, 5) to account for the randomness in the algorithm. Table 2 shows the values of the percentage of admissibility violations and the average number of expansions, averaged across the seed values. The admissibility violations percentage is calculated over all the nodes which are queried during rollout. As expected, the percentage of admissibility violations decreases as we increase the value of λ in the equation. We also observe the average number of expansions is almost constant till the value of $\lambda = 10^3$, but increases significantly for $\lambda = 10^4$. Thus, we do not experiment with higher values of lambda. We see that the best result in terms of admissibility violations while not compromising on average expansions is achieved when using $\lambda = 10^3$.

Although we do not observe an improvement in average expansions for the regularized loss, we see that the training proceeds in a much more stable way compared to the baseline. This can be seen in Figure 2 where we have compared the average expansions for validation across training iterations for the regularized loss with $\lambda = 10^3$ with the baseline (We have similar figures for other values of λ in the appendix A.2). We have denoted the mean across random seeds as dark lines and the standard deviation as light-shaded regions. For the regularized loss case we observe that the spreads are significantly lesser than the baseline and it decreases further for the later iterations.

5.3 Image Patch

The Image Patch architecture was tested on the Forest environment, with patch sizes of 5 and 9, and were compared against the baseline performance. All three architectures were run using the

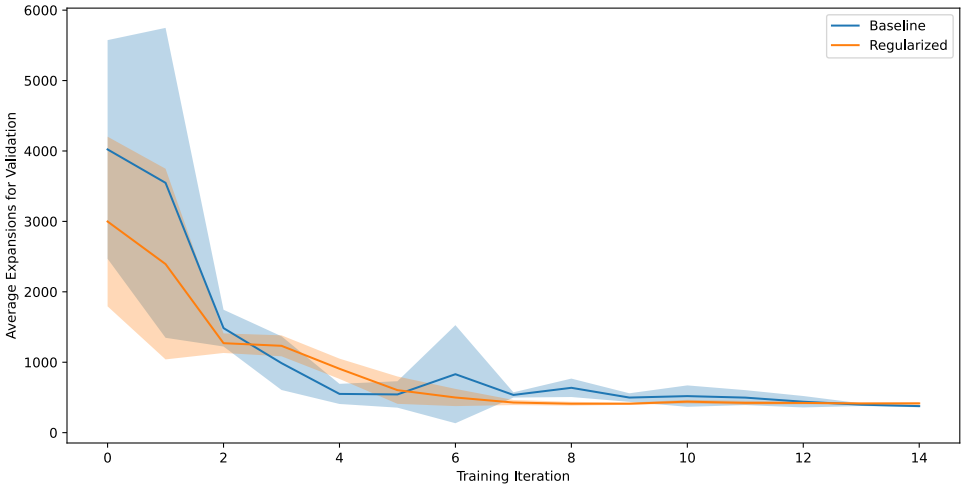

Figure 2: Baseline vs $\lambda = 10^3$ for random seeds 1 to 5

Lambda	Percentage of Admissibility Violations	Average Expansions
Baseline (no patch)	66.37	357.02
Patch Size 5	29.16	413.7
Patch Size 9	**7.34**	**338.94**

Table 3: Comparison across different image patch configurations, for Seed 1

Pytorch version of the network code. Due to time constraints, analysis over multiple seeds to study variability was not performed. The results are displayed in Table 3. It is observed that using an image patch does not significantly alter the number of expansions. Surprisingly, it is seen to reduce the occurrences of admissibility violations, despite not having been explicitly designed for doing the same, unlike in Section 5.2. This may be due to the additional information provided by including knowledge of the local surrounding area during heuristic computation.

6 Limitations and Future Work

Although we have implemented a loss function regularization to encourage admissibility, we have not implemented such a regularization to encourage monotonicity. An important direction of future work would be to incorporate a regularization term or network architecture which explicitly encourages or enforces monotonicity.

Also, in the reproduced results of the SaIL paper, we used seed value 1. We would like to study the variance in the results by running with more seed values.

The proposed modification to the loss function was tested only in the Forest environment. In future, we have to experiment with other environments and see how well the loss function generalises well and reduces the admissibility violations.

Another important direction of future work would be to reduce the overhead for our parallel rollout optimization and verify its equivalence with respect to the standard rollout method.

Our changes (CNN operating on image patch input features) may offer a computation-performance tradeoff in comparison to the original SaIL framework, and thus a comparison of computational intensity against the SaIL algorithm may provide further insights as to the utility of our algorithm.

This may require significant usage of dedicated computing resources, as we observed up to 10 hours of run time for training in a single environment.

7 Conclusions

We have shown that our work improves upon the existing framework in an additive and extension-friendly manner. The regularization term was seen to significantly reduce the number of admissibility violations while preserving the same level of performance. In a similar vein, the image patch inclusion also independently was seen to reduce the overall occurrence of admissibility violations. A consistent heuristic may be better performing in more challenging environments and so our results indicate that our proposed algorithms show promise in addressing complex tasks. Hence, they may be of great practical utility in solving robotic motion planning problems.

References

[1] M. Bhardwaj, S. Choudhury, and S. Scherer. Learning heuristic search via imitation, 2017. URL https://arxiv.org/abs/1707.03034.

[2] A. Kumar, A. Zhou, G. Tucker, and S. Levine. Conservative q-learning for offline reinforcement learning, 2020. URL https://arxiv.org/abs/2006.04779.

A Appendix

A.1 Team Contribution

Authors are listed in alphabetical order. All members contributed equally throughout the duration of the project. All decisions related to the different parts of the project (like problem formulation, literature review, experimental design, and inferences) were taken collectively after discussion. Some specific contributions are as follows: Adeem took the initiative to reproduce the results reported in the SaIL paper for various environments and was responsible for code optimization. Mahesh took charge of implementing the PyTorch code for the Linear and CNN networks, and experimenting with different patch sizes. Naveen's efforts were aimed towards modifying the loss function and was responsible for experimenting with different values of the regularization parameter λ.

A.2 Figures

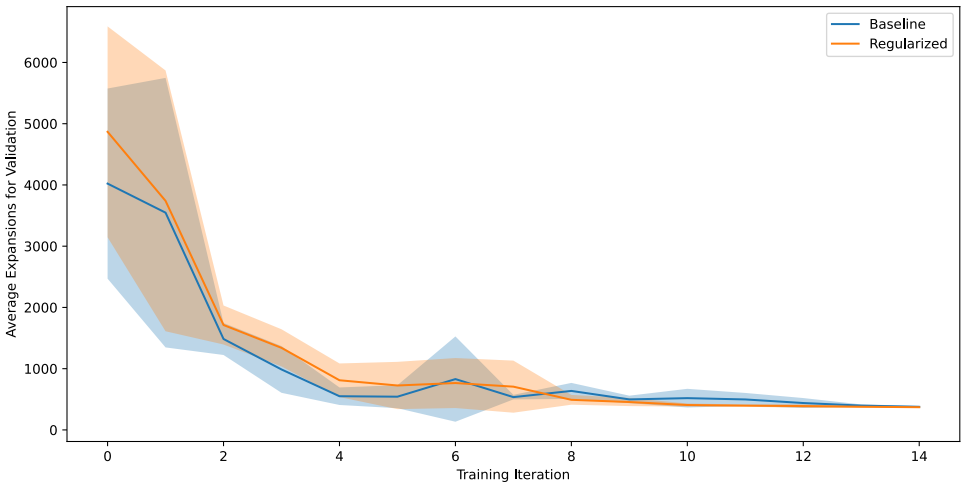

Figure 3: Baseline vs $\lambda = 1$

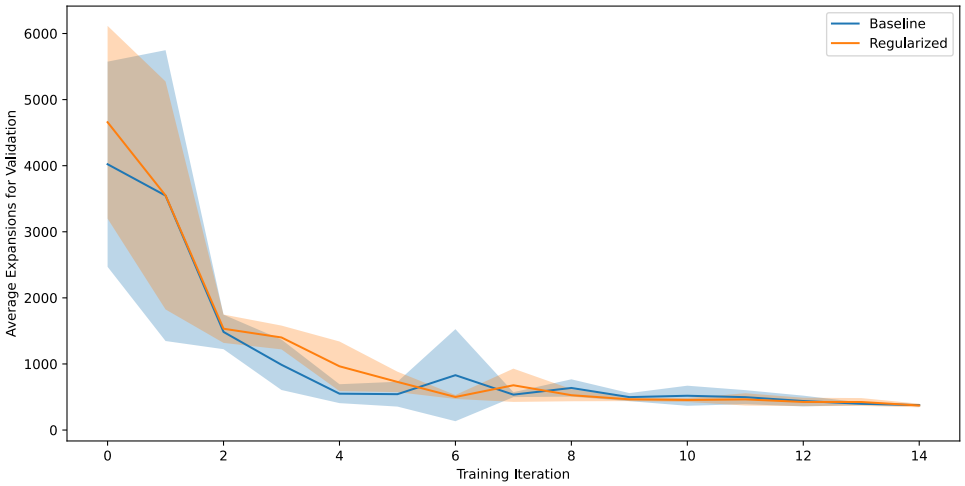

Figure 4: Baseline vs $\lambda = 10$

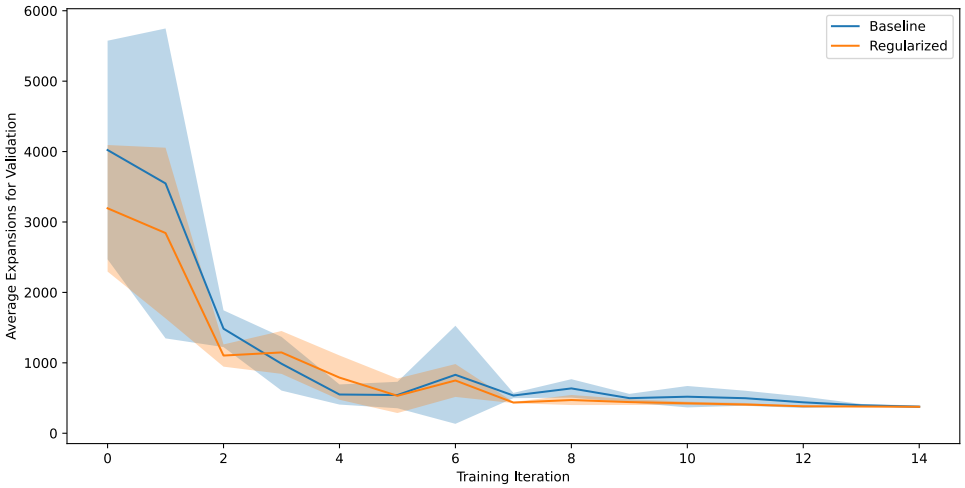

Figure 5: Baseline vs $\lambda = 10^2$

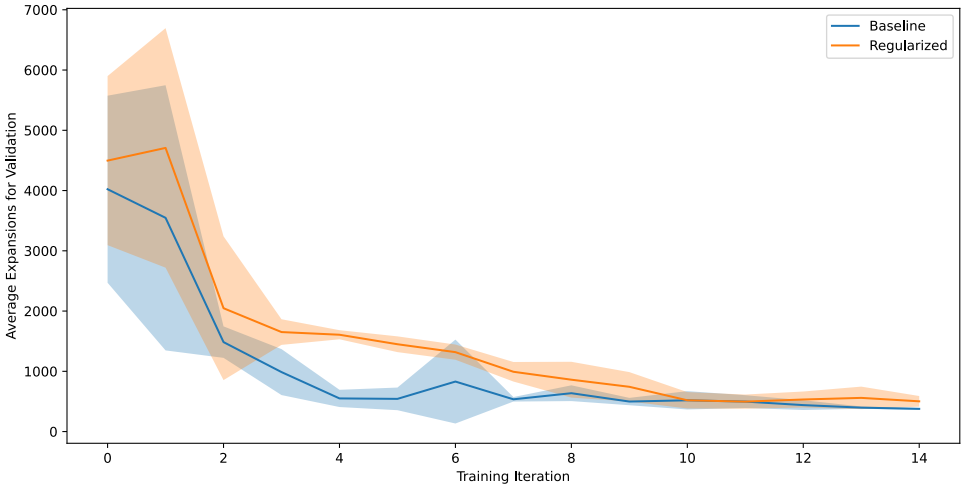

Figure 6: Baseline vs $\lambda = 10^4$