The Use of Imitation Learning to Provide Eigenspectrum Optimization to Mitigate Threat of Resonance From Adversarial Attacks in a Distributed Manner

Nicolas Kozachuk
Department of Electrical & Computer Engineering
Lehigh University
ngk324@lehigh.edu

Date: May 11th, 2024

Introduction/Motivation

A method to provide resonance reduction against attacks from external agents in dynamic networks via eigenspectrum optimization was found via previous research and proved to be successful(paper is attached if extra background knowledge is wanted, but it is not needed). An objective function was derived to provide eigenspectrum optimization, while maintaining a graph network's properties to ensure fair comparison, and by performing gradient descent, one is able to manipulate a graph's edges to provide resonance reduction via eigenspectrum optimization. This can be seen as a flattening of a plot of the eigenvalue spectrum, as a greater multiplicity of mode eigenvalues corresponds to resonance. This is extremely useful when one has a centralized system, but some difficulty arises when one wants to implement this eigenspectrum optimization to mitigate the threat of attacks from external agents in a decentralized system. This is because in a decentralized system, each node of a network operates independently, making it impossible to perform a gradient descent as the full network information is unknown to an individual node. This report goes into detail about an imitation learning approach to provide eigenspectrum optimization for decentralized networks by using preference comparison to perform imitation learning in order to provide resonance reduction for decentralized networks.

Basic imitation learning involves an agent(s) first observing the actions of an expert for various states during the training phase. The agent then uses this training set to learn a policy that tries to mimic the actions demonstrated or queried by the expert, in order to achieve the best performance. For example, an imitation learning agent would observe a human expert driver and will register her actions at the various states. Based on that, it will create a policy of what actions to take at any given state based on what the expert did. At run time it will try its best to approximate the optimal action based on the learned policy, as the states won't be exactly similar, and a probabilistic element will creep in. Due to the complexity of providing this eigenspectrum optimization in a distributed manner, imitation learning is a great approach to solve the problem as an agent can operate on one node at a time.

Problem Description

The task that is set to be solved is to use imitation learning to learn a policy of how to manipulate the edge weights connected to a node, for each node in a graph, in order to reduce an objective value that is a function of the eigenvalues of the graph. The objective function used is a function derived to perform eigenspectrum optimization from previous research, and can be seen as a flatten of the eigenvalue spectrum curve. There are many different approaches for algorithms to perform imitation learning depending on one's problem and the task at hand, as well as many different types of algorithms for each of the different approaches. The two main classical approaches for imitation learning are behavioral cloning and preference comparison[1]. The complication for the current task arises due to the large size of the problem domain, as there is an infinite sized continuous action space since the edge weights can be manipulated in a seemingly infinite number of ways, as well as near infinite sized state space due to the many number of different configurations of graphs. There are also additional problems with this task since it builds off of novel research, resulting in a lack of resources. Due to this, it is important to choose the best imitation learning approach, as well as the best algorithm for the approach.

The behavioral cloning approach is a broad approach, with many different algorithms, but it has its limitations. As mentioned in the paper "Exploring Limitation of Behavioral Cloning for Autonomous Driving"[2], behavioral cloning has limitations due to data bias and overfitting, which causes it to generalize poorly and recover poorly from errors. Although the problem Behavioral cloning is used for in the paper is much different than my application, the limitations in the paper result from similar problems that occur in my task in that there is a lack of causal models and training instability as a result of a very large state and action space. Alternatives to behavioral cloning have been cited to have more promising results, such as Data Aggregation(DAgger), which is similar but gathers on-policy demonstrations. The paper "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning" proposes the DAgger algorithm, which is an improvement upon the behavioral cloning algorithm. DAgger works by training a stationary deterministic policy that can be seen as a no regret algorithm in an online

learning setting[3]. The paper shows that any such no regret algorithm, combined with additional reduction assumptions, finds a policy with good performance under the distribution of observations it induces. Although this may be an improvement to behavioral cloning, this algorithm will not be valid for my application, as it uses on-policy learning, while I will be using off-policy learning via demonstrations using results from objective functions with desirable values.

Preference comparison is the other type of imitation learning approach that could be successful for my task. As explained in the paper “Contextual Bandits and Imitation Learning via Preference-Based Active Queries” in the preference comparison algorithm, the learner actively queries an expert at each round to compare two actions and receive noisy preference feedback[4]. The learner's objective is two fold, to minimize the regret associated with the executed actions, while also simultaneously, minimizing the number of comparison queries made to the expert. The paper explains how a learner has access to a function class that can represent the expert's preference model, which for my task would be the objective function I defined. This algorithm is a possible good fit for my task, as for a state action pair, the objective function can be queried in order to determine which action is more optimal.

After looking through the different types of behavioral cloning and preference comparison algorithms, I will likely be more successful performing imitation learning using a preference comparison imitation learning algorithm. A general preference comparison algorithm can be created using the preference comparison API from the Stable-Baseline3 library. The preference comparison algorithm will allow one to be able to query directly from the objective function in order to determine the best action from a set of actions and thus determine the optimal policy after many iterations. This will be more successful than using a behavioral cloning approach, as it is easier to query preference than to show desirable demonstrations due to the infinitely large state and action space. One may think it may be difficult to query many different sets of actions and learn an optimal policy, but preference comparisons algorithms such as the Proximal Policy Optimization Algorithm have proved to be successful. In the paper “Proximal

Policy Optimization Algorithm"[5] , it explains how PPO is a new family of policy gradient methods for reinforcement learning, which alternates between sampling data through interaction with the environment, and optimizing a surrogate objective function using stochastic gradient ascent/descent. Where standard policy gradient methods perform one gradient update per data sample, PPO uses a novel objective function that enables multiple epochs of minibatch updates. This is the algorithm that best suits the task at hand, and thus will be the one used to build the model.

Formulation of Markov Decision Process

Generally, for this task the state space is defined as all of the individual nodes and their corresponding connected edge weights. In this report, an undirected uniform complete graph with 30 nodes is used, as is similarly used in the research for the eigenspectrum optimization. Due to the goal of maintaining the graph properties, the sum of the edge weights must remain constant, which is 870 for a 30 node uniform complete graph with edge weights equal to 1. It is with noting that the sum of the edge weights is equivalent to the sum of the eigenvalues. Another constraint is that the minimum allowed edge weight is 0.1, as the researchers who derived the objective function and the eigenspectrum optimization approach insist that the minimum edge weights be 0.1, since any lower edge weight could be a negligible value and result in the essential removal of the edge if the weight approaches 0. This means for the graph used in this report, the state space is defined as the edge weights of the nodes of a unit 30 node uniform complete graph, where the minimum edge weight is 0.1, the sum of the edge weights are 870, and thus the maximum possible edge weight is 783.2, which would occur when all other edge weights are 0.1.

The general action space for this task is all possible manipulations of edge weights for nodes. Since this report uses a unit 30 node uniform complete graph, and must adhere to the defined constraints, the action space can be slightly more defined as any positive or negative manipulation of edge weights, where the edge weight must remain greater than or equal to 0.1 and the sum of the edge must remain constant at 870, thus the sum of the actions performed on the edges must sum to 0. This means that for a given edge

weight, the minimum action value that can be performed on it is 0.1 minus the edge weight and thus the maximum action value is 783.2 minus the edge weight. These are the state and action spaces for the graph used for this given problem presented, but complications arise when attempting to implement these state and action spaces in code.

The reward function is simply defined as the objective function used in the gradient descent. This is used as for each set of actions generated and performed, the reward function is queried to determine the optimal action. Since the model is attempting to learn how to perform the gradient descent via preference comparison, it makes sense that the reward function is the objective function used to perform the gradient descent for the eigenspectrum optimization.

The Stable Baseline3 library allows discrete, box, and dictionary space types for defining observation spaces, and only discrete and box space types for action spaces. The discrete space type is used to define a discrete space that maps to a set of defined actions, such as a die roll, card draw etc. The box space type is used to define continuous spaces, as state space size can be defined, as well as the minimum and maximum values that each state space can be. The dictionary space type can be used to define a combination of discrete and box types. The observation(state) space can be defined using the box space type, with the shape space being the number of edges, the minimum value being 0.1 and maximum values being 783.2, as can be seen in Appendix A.1**.** The space shape may not be what is expected for a distributed approach, but that is due to the limitations of the code library. Ideally, an action would consist of the manipulation of edges for a specific node, but since the action space can only be defined as discrete or box space, this is not possible. Additionally, a multi agent approach is also not feasible due to the limitations of the code library. To account for this, it was attempted to have the space shape be a symmetric 2D matrix, where the elements in the i^{th} row corresponds to the edge weights of the i^{th} node, similar to an adjacency matrix, and actions can be performed on each row, allowing the algorithm to learn policies specific to a node. Problems also arose with this approach though as the state and action space

must be the same size for preference comparison with box spaces for state and observation space, so the action had to be defined as a 2D matrix. Due to limited control over the trajectories generated for querying when using preference comparison, actions were generated for 2D matrix observation space, but they were not symmetric matrices and unable to be altered during trajectory generation to become symmetric. This is problematic because the matrix must be symmetric as edge weight between node i and node j corresponds to the (i,j) and (j,i) matrix element. These elements being different is not possible for an undirected graph, so a different approach was looked into and used. As briefly mentioned previously, the observation space shape was defined as being a vector containing each of the edge weights, without repetition to account for symmetry of edge weights in an undirected graph.

Ideally the action space would be defined as a dictionary of discrete space value corresponding to each node, and a box space corresponding to the nodes edges, but as previously mentioned this is not possible for the Stable Baseline3 library. Thus, the action space is defined using a box space of shape equal to the number of edges, with the minimum value of each node corresponding the negative value that, when added, would reduce the edge to the minimum weight of 0.1, and the maximum action value being the positive value that when added would result in the maximum edge weight of 783.2, as can be seen in Appendix A.2. Problems arise when defining the action space as the sum of the elements must remain constant to maintain graph properties, so the sum of the action must be zero, but this is something that is not able to be defined using the Stable Baseline3 library. This problem is dealt with in the step function of the environment by normalizing the vector of edge weights after each action is performed.

Algorithm Design

To begin, a custom environment had to first be created to fit the problem at hand. The main components of this are an initialization function, step function, reset function, and render function. For initialization,

the environment creation function takes an adjacency matrix input and generates a vector of edge weights. During the environment initialization, the action and observation space are both defined in the manner explained in the previous section. The step function works by adding the action values to the current edge weight vector. The edge weight vector then undergoes normalization to ensure that the sum of the edges remains constant. The get_reward() function is called to get the reward, as well as a get_info() and get_observation() function in order for the step function to return the reward for the step, as well as the edge weights from the get_info() and get_observation() function. The get_reward() function is, as explained in the previous section, defined as the objective function used in the gradient descent. Another function is also defined that constructs a laplacian matrix for the current graph, which is called by the get_reward() function, since the reward, which is the objective function value for the graph, is a function of the graph's eigenvalues. A reset() function is defined to reset the environment, so for this problem the reset function sets all the edge weight back to the initial value of 1. Like the step() function, the reset() function also returns the graph's edge weights. Lastly, a render() function is defined, which displays the graph, with its edge's weight value corresponding to the grayscale color of the edge..

The algorithm chosen to use for preference comparison was PPO due to its robustness, simplicity to implement, and most importantly its alignment to the task at hand. Its reward net for determining preferences is defined to be a multi-layer perceptron that takes the state, action, next state, and done flag as inputs and flattens and then concatenates them together. A cross entropy loss is defined to be used as the loss function, and the FeedForward32Policy is the policy defined to be used for the PPO training. The FeedForward32Policy is a feed forward policy network with two hidden layers of 32 units. The sample procedure for generating trajectories is defined to be random, and the computation of synthetic preferences using ground-truth environment rewards is also defined. Using all of these important parameters, as well as a hyperbolic query scheduler, a PPO algorithm can be trained. After training, the learned policy can be evaluated on the environment in order to determine the final reward.

Implementation and Results

The PPO algorithm was implemented with various parameters, as briefly explained previously. First, a reward net is defined for the given observation and action space that flattens the inputs. The inputs to the reward net are normalized to be between 0 and 1 using a running average. Next, the sampling of fragments for trajectories of actions are defined to be sampled randomly with replacement. Then, synthetic preferences are defined to be computed using the ground-truth rewards. Additionally, a class is defined to convert the rewards of two generated fragments into a probability in order to be chosen based on preference. All of these code implementations can be seen in Appendix A.3. Next, the PPO model itself is defined and is set to use the FeedForward32Policy, the specified environment, as well as set other parameters such as the number of epochs, the learning rate, number of steps, etc., as can be seen in Appendix A.4. Each of the parameters were tuned in order to find the optimal parameters for this problem. Then, the agent trainer is defined in order to generate trajectories using the defined PPO model, reward net, and environment. Lastly, each of the defined components come together to build the main interface for reward learning using preference comparison, which can be seen in Appendix A.5. Finally, the defined main interface for preference comparison can be trained, and is done so with the parameters of the number of time steps being 25,000 and there being 200 comparisons. During training, a wrapper function is used in order to save the training data for output to be analyzed.

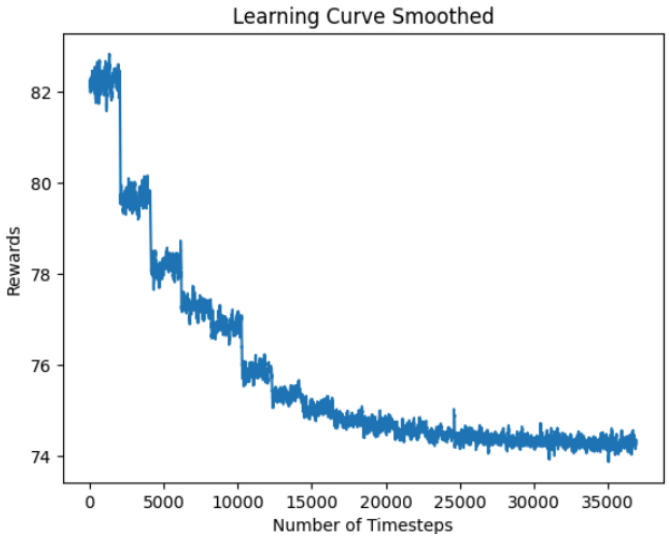

Figure 1: Training Plot

One can see the results from training above in Figure 1. It can be seen that there is an exponential decay in the reward value with respect to the number of time steps. In the beginning of training it can be seen that the reward briefly oscillates around a constant value before finding a semi-optimal action via comparison, which significantly reduces the reward. This occurs a few times in the training, where for some time steps there is no improvement, until a semi-optimal action is performed and selected from a comparison, which significantly decreases the reward. This can be most clearly seen at the timestep of around 3,000 and 5,000, where there is brief oscillation before there is a large almost instantaneous decrease in the reward value. As the algorithm learns and improves, this trend becomes less apparent due to a limit being approached, but even in the timesteps between 10,000 and 15,000 it can be seen that there is some oscillation before a slight decrease in the reward value. After training, the policy learned by the model can be used for evaluation, which when applied to the defined environment results in a value of 76.01.

Inferences from Results

The initial reward, which is the objective function value, is 213.23, so a decrease down to 76.01 is significant. This 213.23 value is not included in the training plot, since the model learns almost instantly to reduce the reward significantly, and because expanding the range of y-axis makes the learning done from timestep ~100 to 35,000 much less apparent. One may think this is an error, but it is not as if one queries the environment's reward before training, it is 213.23. Additionally, when implementing the code, many different models and parameters were tested, and this resulting one is the best one, so it makes sense the reward is significantly decreased. For example, initially there were some other models and parameters used over the course of implementation that had an initial plotted value of ~200 and only decreased to ~150, and I was able to gradually improve the algorithm to further reduce the reward, which is why I am confident in the results.

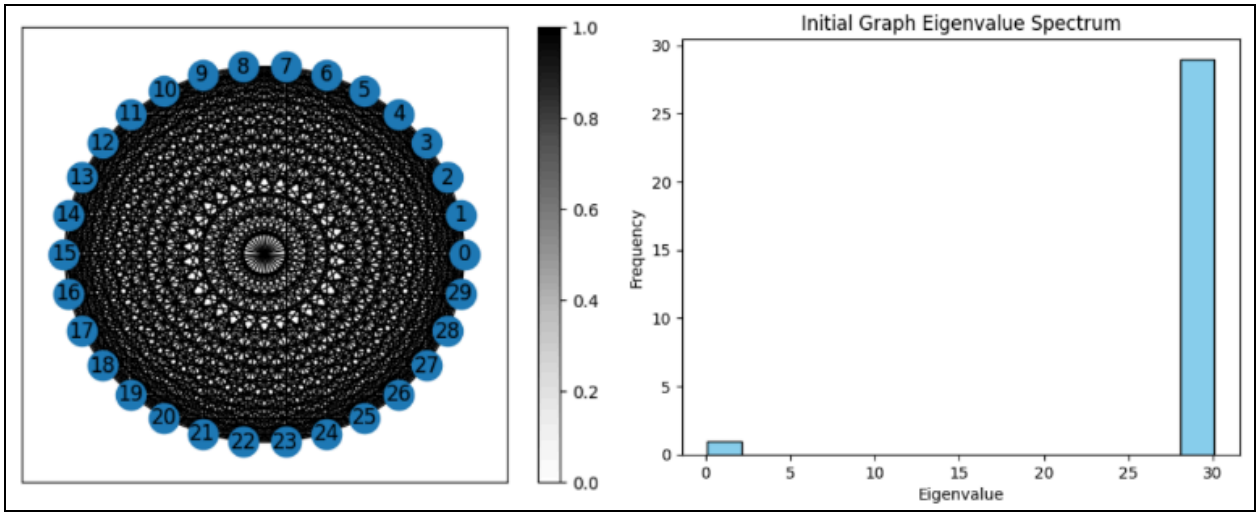

Figure 2: Initial Graph and Eigenvalue Spectrum

The initial graph and its eigenvalue spectrum can be seen above in Figure 2. The initial graph has all edge weights equivalent to 1, which is why the graph has all black edges and the eigenvalue spectrum is peaky. This initial graph is chosen due to its peaky eigenvalue spectrum, as the research showed that the peakier the eigenvalue spectrum is, the greater the risk of resonance. Since the peakiness of the eigenvalue spectrum correlates to greater resonance, the flatter the eigenvalue spectrum, the lower the resonance and its risk will be. For this initial graph, the objective value is 213.23.

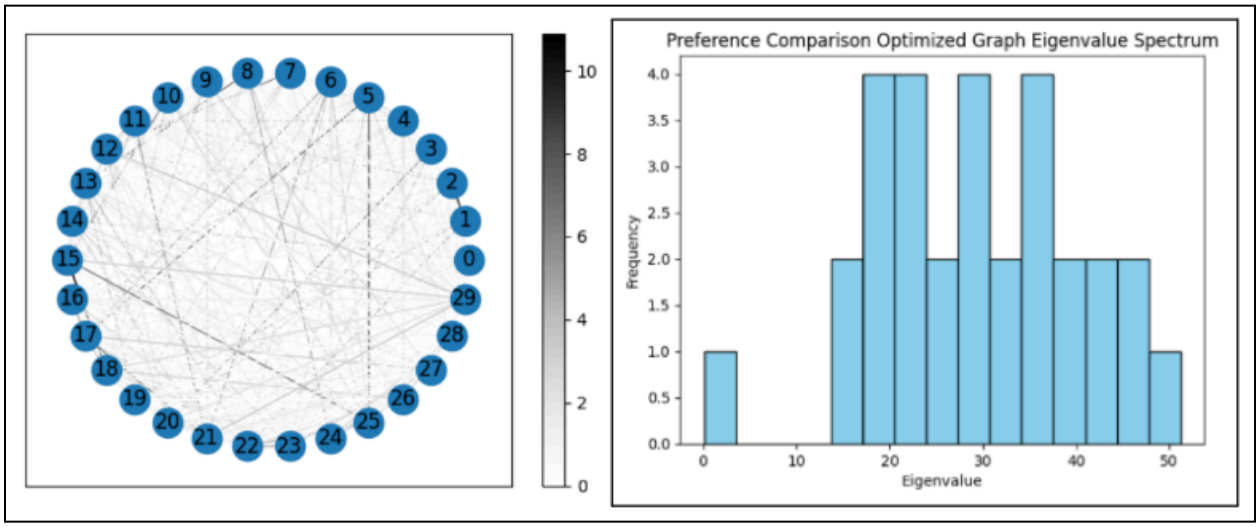

Figure 3: Preference Comparison Optimized Graph and Eigenvalue Spectrum

After training the model, the learned policy is used for evaluation on the graph. The resulting graph and its eigenvalue spectrum can be seen in Figure 3. It can be seen that the graph looks significantly different.

If one looks at the color bar, it can be seen that the max edge weight is about 10, which is why the graph is very white compared to the initial graph. This is because the model found that the optimal graph configuration is to have a few heavy edge weights and the rest be very low, since the sum of the edge weights must remain constant. The reward(objective value) for this optimized graph is 76.01, which is a significant decrease compared to 213.23. The code output also shows that the sum of the eigenvalues remains constant at 870, which is another success. Additionally, the gradient descent algorithm used in the research results in an optimized objective value of 71.04, which is very close to 76.01 considering the initial value is 213.23. This shows that the model is successful in achieving eigenspectrum optimization. It is also worth noting that the gradient descent algorithm takes about two hours to reach an optimized objective function value, while this model only takes 5 minutes. This highlights the power and the usefulness of this model.

Conclusion

In conclusion, this report presents a novel approach to address the challenge of providing resonance reduction in dynamic networks, particularly focusing on decentralized systems vulnerable to external attacks. By leveraging imitation learning, specifically preference comparison algorithms, the study aims to optimize a graph network's eigenspectrum in order to mitigate the potential threat of adversarial attacks. While acknowledging the complexities inherent in the problem domain, the report navigates through algorithm selection and formulation of the Markov Decision Process to propose a viable solution.

The implementation of the Proximal Policy Optimization (PPO) algorithm, tailored to the task at hand, demonstrates promising results. Through meticulous parameter tuning and rigorous training, the model showcases a significant reduction in objective function value, indicating successful eigenspectrum optimization. Moreover, the generated policy leads to tangible improvements in network resilience, as evidenced by the transformed graph and eigenvalue spectrum. The optimized network configuration

exhibits decreased resonance risk, achieving objectives comparable to traditional gradient descent methods but with significantly reduced computational overhead.

Although the model is not a completely distributed approach due to limitations with libraries, it is an adequate start to meet this goal, as it is an improvement upon the gradient descent approach, which cannot be transformed to be distributed. This model or one similar will be continued to be worked on for research purposes in order to eventually achieve a distributed approach, if possible. In order to further advance the model, one will look into building custom wrappers in order to extend the model to learn and evaluate on many different types and sizes of graphs. Additionally, one will also look into other libraries that may allow for a similar model, but with multiple agents. Overall, this study not only contributes to advancing resilience strategies in dynamic networks but also underscores the efficacy of imitation learning, specifically preference comparison algorithms, in addressing complex optimization tasks. The findings offer valuable insights for future research and practical applications in network security and resilience engineering.

Appendix A: Code

```
highest_possible_edge_weight = 2*self.num_edges - 0.1*2*(self.num_edges-1)
self.observation_space = spaces.Box(low=0.1, high=highest_possible_edge_weight,
                                    shape=np.array([self.num_edges]))
```

Appendix A.1: Defining Observation Space in Custom Environment

```
low = 0.1 - self.edges
high = (2*self.num_edges - 0.1*2*(self.num_edges-1))*np.ones(self.num_edges) - self.edges
self.action_space = spaces.Box(low=low, high=high, shape=np.array([self.num_edges]),
                               dtype=np.float64)
```

Appendix A.2: Defining Action Space in Custom Environment

```
reward_net = BasicRewardNet(
    venv.observation_space, venv.action_space, normalize_input_layer=RunningNorm,)

fragmenter = preference_comparisons.RandomFragmenter(warning_threshold=0, rng=rng)

gatherer = preference_comparisons.SyntheticGatherer(rng=rng)

preference_model = preference_comparisons.PreferenceModel(reward_net)
```

Appendix A.3: Defining of Preference Comparison Parameters

```
model = PPO(
    policy=FeedForward32Policy,
    policy_kwargs=dict(
        features_extractor_class=NormalizeFeaturesExtractor,
        features_extractor_kwargs=dict(normalize_class=RunningNorm),
    ),
    env=venv,
    n_steps=2048 // venv.num_envs,
    clip_range=0.1,
    ent_coef=0.1,
    gae_lambda=0.95,
    n_epochs=5,
    gamma=0.97,
    learning_rate=2e-3,
)
```

Appendix A.4: Defining PPO Model

```
trajectory_generator = preference_comparisons.AgentTrainer(
    algorithm=model,
    reward_fn=reward_net,
    venv=venv,
    exploration_frac=0.05,
    rng=rng,
)

pref_comparisons = preference_comparisons.PreferenceComparisons(
    trajectory_generator,
    reward_net,
    num_iterations=5,
    fragmenter=fragmenter,
    preference_gatherer=gatherer,
    reward_trainer=reward_trainer,
    initial_epoch_multiplier=2,
    initial_comparison_frac=0.01,
    query_schedule="hyperbolic",
)
```

Appendix A.5: Initialize Agent Trainer and Main Preference Comparison Interface

References

[1] AI, SmartLab. "A Brief Overview of Imitation Learning." *Medium*, Medium, 19 Sept. 2019, smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c.

[2] Codevilla, Felipe, et al. "Exploring the Limitations of Behavior Cloning for Autonomous Driving." *arXiv.Org*, 18 Apr. 2019, arxiv.org/abs/1904.08980.

[3] Ross, Stephane, et al. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." *arXiv.Org*, 16 Mar. 2011, arxiv.org/abs/1011.0686.

[4] Sekhari, Ayush, et al. "Contextual Bandits and Imitation Learning via Preference-Based Active Queries." *arXiv.Org*, 24 July 2023, arxiv.org/abs/2307.12926.

[5] Schulman, John, et al. "Proximal Policy Optimization Algorithms." *arXiv.Org*, 28 Aug. 2017, arxiv.org/abs/1707.06347.