REINFORCEMENT LEARNING EXERCISE

By Alexander Green

EEL6938: Artificial Intelligence for Autonomous Systems

Table of Contents

Setup

This assignment setup has two separate python files to run. The first file is my modified version of the provided **RL_assignmnet.py**. I modified this file to perform the experiment with the hyperparameters that I pass through the command terminal. The second file is **Main.py**. This brand-new file was created to intelligently run all the required experiments I need for this assignment. **Main.py** handles commands with variations in hyperparameters, data storage, and basic data comparison.

RL_Assignment.py

In this section, we outline the updates to **RL_assignment.py** from its initial state, detailing specific modifications with corresponding line numbers. We then provide an overview of its functionality, explaining how it processes data, trains reinforcement learning models, and evaluates performance using custom environments, multiple RL algorithms, and detailed performance metrics.

Modifications to Original Code

1. **Command-Line Arguments for Hyperparameters**
   The script allows customization through command-line arguments, enabling users to fine-tune the reinforcement learning model. The ***--output_dir*** argument specifies where logs and trained models are stored. ***--chunk_size*** defines the length of each training episode by segmenting the dataset into smaller parts. ***--model*** selects between **SAC**, **PPO**, **TD3**, and **DDPG**, determining the learning approach. Hyperparameters such as ***--learning_rate***, ***--batch_size***, and ***--buffer_size*** control optimization speed, data processing scale, and replay memory capacity. ***--tau*** adjusts the target network's soft update rate, while --gamma sets the discount factor for future rewards. ***--ent_coef*** configures entropy regularization, either as a fixed value or automatically. ***--net_arch*** allows customization of neural network architecture, and --reward selects between different reward functions. ***--total_timesteps*** determines the overall training duration. As you can see in the figure below, the code for each argument has a default parameter and a help line to define the available choices.

```python
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--output_dir",
        type=str,
        default="./logs_chunk_training",
        help="Directory to store logs and trained model."
    )
    parser.add_argument(
        "--chunk_size",
        type=int,
        default=100,
        help="Episode length for training (e.g. 50, 100, 200)."
    )
    # Add model selection argument
    parser.add_argument(
        "--model",
        type=str,
        default="SAC",
        choices=["SAC", "PPO", "TD3", "DDPG"],
        help="RL algorithm to use (SAC, PPO, TD3, DDPG)."
    )
    # Add hyperparameter arguments
    parser.add_argument(
        "--learning_rate",
        type=float,
        default=3e-4,
        help="Learning rate."
    )
    parser.add_argument(
        "--batch_size",
        type=int,
        default=256,
        help="Batch size for training."
    )
    parser.add_argument(
        "--buffer_size",
        type=int,
        default=200000,
        help="Replay buffer size."
    )
    parser.add_argument(
        "--tau",
        type=float,
        default=0.005,
        help="Soft update coefficient."
    )
    parser.add_argument(
        "--gamma",
        type=float,
        default=0.99,
        help="Discount factor."
    )
    parser.add_argument(
        "--ent_coef",
        type=float,
        default=-1.0,
        help="Entropy coefficient. Use -1.0 for 'auto'."
    )
    parser.add_argument(
        "--net_arch",
        type=str,
        default="256,256",
        help="Network architecture (comma-separated list of layer sizes)."
    )
    parser.add_argument(
        "--reward",
        type=int,
        default=0,
        choices=[0, 1, 2, 3],
        help="Reward function: 0=neg_abs_error, 1=neg_squared_error, 2=neg_sqrt_error, 3=error+action_penalty"
    )
    parser.add_argument(
        "--total_timesteps",
        type=int,
        default=100000,
        help="Total timesteps for training."
    )
```

Figure 1: Argument Parsing

2. **Multiple RL Algorithms**

I updated **RL_assignment.py** to support not only the **SAC** reinforcement learning algorithm but also **PPO**, **TD3**, and **DDPG**. In my code, I defined a set of common parameters shared across all four models, ensuring consistency in policy selection, environment setup, and key hyperparameters. This standardization simplifies model initialization and avoids redundant code. Below, you can see where these common parameters are defined.

```
340     # Common parameters for all algorithms
341     common_params = {
342         "policy": "MlpPolicy",
343         "env": train_env,
344         "verbose": 1,
345         "policy_kwargs": policy_kwargs,
346         "learning_rate": args.learning_rate,
347         "device": device,
348         "gamma": args.gamma,
349     }
```

Figure 2: Common RL Parameters

Here, you can see where I specified the individual parameters unique to each reinforcement learning algorithm. These include adjustments for batch size, buffer size, entropy coefficient, and algorithm-specific settings, ensuring each model is configured appropriately for training. Note that they are being imported from the arguments passed to **RL_assignment.py**.

```
351     # Initialize the selected algorithm with appropriate parameters
352     if args.model == "SAC":
353         model = SAC(
354             **common_params,
355             batch_size=args.batch_size,
356             buffer_size=args.buffer_size,
357             tau=args.tau,
358             ent_coef=ent_coef
359         )
360     elif args.model == "PPO":
361         model = PPO(
362             **common_params,
363             batch_size=args.batch_size,
364             n_steps=1024,  # PPO-specific parameter
365             ent_coef=0.01 if args.ent_coef == -1.0 else args.ent_coef
366         )
367     elif args.model == "TD3":
368         model = TD3(
369             **common_params,
370             batch_size=args.batch_size,
371             buffer_size=args.buffer_size,
372             tau=args.tau
373         )
374     elif args.model == "DDPG":
375         model = DDPG(
376             **common_params,
377             batch_size=args.batch_size,
378             buffer_size=args.buffer_size,
379             tau=args.tau
380         )
381     else:
382         raise ValueError(f"Unknown model: {args.model}")
383 
384     model.set_logger(logger)
```

Figure 3: Individual RL Algorithms

3. **Multiple Reward Functions**

   The original code used only the **negative absolute error** as the reward function, but I updated it to support multiple reward evaluation methods. The **squared error** function penalizes larger deviations more harshly, making the model more sensitive to significant speed differences. The **square root error** applies a softer penalty for large errors, reducing extreme punishment for occasional deviations. The **error + action penalty** not only penalizes speed error but also discourages excessive acceleration, encouraging smoother control. These variations allow for more flexibility in training different policies. The code below shows the math behind each of the different reward methods.

```
107         # Different reward functions based on reward_type
108         if self.reward_type == 0:
109             reward = -error  # Default: Negative absolute error
110         elif self.reward_type == 1:
111             reward = -error**2  # Squared error (penalizes larger errors more)
112         elif self.reward_type == 2:
113             reward = -np.sqrt(error)  # Square root (softer penalty for large errors)
114         elif self.reward_type == 3:
115             reward = -error - 0.1 * abs(accel)  # Penalize large actions too
116         else:
117             reward = -error
```

Figure 4: Reward Functions

4. **Model Performance Metrics**

   To evaluate model performance, I implemented multiple error metrics that quantify how well the trained agent tracks the reference speed. **Mean Absolute Error (MAE)** measures the average deviation between predicted and reference speeds, providing a straightforward accuracy assessment. **Mean Squared Error (MSE)** penalizes larger deviations more heavily, emphasizing consistency in speed tracking. **Root Mean Squared Error (RMSE)**, derived from MSE, offers a balanced metric that is less sensitive to outliers while still highlighting significant errors. **Percentile errors**, including the **95th and 99th percentiles**, indicate the worst-case performance by capturing the largest deviations encountered during testing. Finally, the **convergence rate** tracks how quickly the model reduces its error over time, reflecting learning stability and adaptation. The code below shows how each of these metrics is calculated and recorded.

```
430     # Calculate quantitative metrics
431     errors = np.array([abs(p - r) for p, r in zip(predicted_speeds, reference_speeds)])
432     squared_errors = errors**2
433
434     # Mean metrics
435     mae = np.mean(errors)
436     mse = np.mean(squared_errors)
437     rmse = np.sqrt(mse)
```

Figure 5: Model Performance Metrics

5. Visualization and Data Analysis

The **speed tracking plot** compares the agent's predicted speed against the reference speed to assess tracking accuracy.

```
493     # 1. Speed tracking plot (reference vs predicted)
494     plt.figure(figsize=(10, 6))
495     plt.plot(reference_speeds, label="Reference Speed", linestyle="--")
496     plt.plot(predicted_speeds, label="Predicted Speed", linestyle="-")
497     plt.xlabel("Timestep")
498     plt.ylabel("Speed (m/s)")
499     plt.title(f"Speed Tracking Performance ({args.model}, chunk_size={chunk_size})")
500     plt.legend()
501     plt.grid(True, alpha=0.3)
502     speed_plot_path = os.path.join(log_dir, f"1_speed_tracking_plot.png")
503     plt.savefig(speed_plot_path)
504     plt.close()
```

Figure 6: Speed Tracking Plot

The **error plot** shows absolute speed error over time or changes in performance.

```
506     # 2. Error plot
507     plt.figure(figsize=(10, 6))
508     plt.plot(errors, label="Absolute Error", color="red")
509
510     # Add moving average for trend visualization
511     plt.plot(np.arange(window_size//2, window_size//2 + len(smoothed_errors)),
512              smoothed_errors, label=f"Moving Avg (window={window_size})",
513              color="darkred", linewidth=2)
514
515     plt.axhline(y=mae, color='black', linestyle='--', label=f"MAE = {mae:.4f}")
516     plt.xlabel("Timestep")
517     plt.ylabel("Error")
518     plt.title(f"Speed Tracking Error ({args.model}, chunk_size={chunk_size})")
519     plt.legend()
520     plt.grid(True, alpha=0.3)
521     error_plot_path = os.path.join(log_dir, f"2_error_plot.png")
522     plt.savefig(error_plot_path)
523     plt.close()
```

Figure 7: Error Plot

The **squared error plot** highlights larger deviations by emphasizing errors with greater penalties.

```
525     # 3. Squared error plot
526     plt.figure(figsize=(10, 6))
527     plt.plot(squared_errors, label="Squared Error", color="purple")
528     plt.axhline(y=mse, color='black', linestyle='--', label=f"MSE = {mse:.4f}")
529     plt.xlabel("Timestep")
530     plt.ylabel("Squared Error")
531     plt.title(f"Speed Tracking Squared Error ({args.model}, chunk_size={chunk_size})")
532     plt.legend()
533     plt.grid(True, alpha=0.3)
534     squared_error_plot_path = os.path.join(log_dir, f"3_squared_error_plot.png")
535     plt.savefig(squared_error_plot_path)
536     plt.close()
```

Figure 8: Squared Error Plot

The **action plot** tracks acceleration values applied by the agent, providing insight into its control strategy.

```python
    # 4. Action plot
    plt.figure(figsize=(10, 6))
    plt.plot(actions, label="Action (Acceleration)", color="green")
    plt.xlabel("Timestep")
    plt.ylabel("Acceleration")
    plt.title(f"Control Actions ({args.model}, chunk_size={chunk_size})")
    plt.legend()
    plt.grid(True, alpha=0.3)
    action_plot_path = os.path.join(log_dir, f"4_action_plot.png")
    plt.savefig(action_plot_path)
    plt.close()
```

Figure 9: Action Plot

The **error histogram** displays the distribution of speed errors, illustrating how frequently different error magnitudes occur.

```python
    # 5. Error histogram
    plt.figure(figsize=(10, 6))
    plt.hist(errors, bins=30, alpha=0.7, color="blue")
    plt.axvline(x=mae, color='red', linestyle='--', label=f"MAE = {mae:.4f}")
    plt.axvline(x=rmse, color='green', linestyle='--', label=f"RMSE = {rmse:.4f}")
    plt.axvline(x=p95_error, color='purple', linestyle='--', label=f"95% = {p95_error:.4f}")
    plt.xlabel("Error Value")
    plt.ylabel("Frequency")
    plt.title(f"Error Distribution ({args.model}, chunk_size={chunk_size})")
    plt.legend()
    plt.grid(True, alpha=0.3)
    hist_plot_path = os.path.join(log_dir, f"5_error_histogram.png")
    plt.savefig(hist_plot_path)
    plt.close()
```

Figure 10: Error Histogram

The **combined visualization plot** integrates speed tracking, error trends, and actions into a single view for a comprehensive performance summary.

```python
    # 6. Combined visualization plot (keep this one for display)
    plt.figure(figsize=(12, 10))

    # Speed tracking
    plt.subplot(3, 1, 1)
    plt.plot(reference_speeds, label="Reference Speed", linestyle="--")
    plt.plot(predicted_speeds, label="Predicted Speed", linestyle="-")
    plt.ylabel("Speed (m/s)")
    plt.title(f"Performance Summary ({args.model}, chunk={chunk_size}, MAE={mae:.3f}, MSE={mse:.3f})")
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Errors
    plt.subplot(3, 1, 2)
    plt.plot(errors, label="Absolute Error", color="red", alpha=0.5)
    plt.plot(np.arange(window_size//2, window_size//2 + len(smoothed_errors)),
             smoothed_errors, label="Smoothed Error", color="darkred")
    plt.axhline(y=mae, color='black', linestyle='--', label=f"MAE = {mae:.4f}")
    plt.ylabel("Error")
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Actions
    plt.subplot(3, 1, 3)
    plt.plot(actions, label="Acceleration", color="green")
    plt.xlabel("Timestep")
    plt.ylabel("Action")
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    combined_plot_path = os.path.join(log_dir, f"6_combined_plot.png")
    plt.savefig(combined_plot_path)
    plt.close()
```

Figure 11: Combined Visualization Plot

General Functionality of Final Code

The script begins by generating a 1200-step speed dataset, adding noise to a sinusoidal speed profile, and saving it as a CSV file. This dataset is split into chunks for episodic training, ensuring consistent episode lengths while handling any remaining data.

Two custom Gym environments manage training and testing. **TrainEnv** selects random data chunks per episode, where the agent adjusts acceleration to minimize speed error under various reward functions. **TestEnv** evaluates the trained model on the full dataset in a single run, assessing generalization.

For training, the script supports **SAC**, **PPO**, **TD3**, and **DDPG**, with customizable hyperparameters set via command-line arguments. The model is trained using stable-baselines3, logging progress and dynamically selecting GPU or CPU.

During testing, the trained model runs through all 1200-steps, tracking speed accuracy and calculating key performance metrics like average reward and speed error.

Finally, the script generates detailed performance metrics and visualizations, including **MAE**, **MSE**, **RMSE**, **error percentiles**, and **convergence rate**, alongside plots of **speed tracking**, **error trends**, and **action sequences** for deeper analysis.

Main.py

The **Main.py** program functions as an experiment runner, automating the testing of various hyperparameter configurations for **RL_assignment.py**. It systematically varies parameters such as **learning rate**, **batch size**, **chunk size**, **reward functions**, **RL algorithms**, **network architectures**, **discount factors**, and **entropy coefficients**. To optimize efficiency, the script first checks whether results for a specific configuration already exist. If the output directory is found, that experiment is skipped, preventing redundant computations and significantly reducing runtime. This ensures that previously calculated results remain intact while allowing new experiments to be conducted without repeating unnecessary steps.

Each hyperparameter category is executed independently, meaning that specifying ***--experiment learning_rate*** runs only learning rate variations, while ***--experiment batch_size*** runs only batch size experiments, and so on. This modular design allows for recalculating results for specific hyperparameters without rerunning all previous experiments. Users can define the total training timesteps and choose to test a single parameter category or run all experiments sequentially.

If all experiments are executed using ***--experiment all***, the script refactors the final results, compiling updated comparative metrics and generating new **final_results** figures. These figures highlight the best-performing hyperparameters and provide a visual comparison of different configurations, ensuring that the latest and most accurate results are always reflected in the final summary.

Metrics

To evaluate the performance of the reinforcement learning model, several mathematical metrics are used to quantify the accuracy of speed tracking. Each metric provides a different perspective on how well the agent follows the reference speed profile.

Mean Absolute Error (MAE) measures the average absolute difference between the predicted speed $\hat{y}_i$ and the reference speed y_i. It is calculated as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

MAE provides an intuitive measure of error, treating all deviations equally without emphasizing larger errors more than smaller ones.

Mean Squared Error (MSE) is similar to MAE but squares each error term before averaging. This is given by:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Because errors are squared, larger deviations contribute disproportionately, making MSE more sensitive to outliers.

Root Mean Squared Error (RMSE) is the square root of MSE:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

This metric balances the interpretation of MAE and MSE, retaining sensitivity to larger errors while remaining in the same unit as the original data.

Error Percentiles such as the **95th and 99th percentiles** indicate the worst-case errors within a dataset. **The 95th percentile error** is the value below which 95% of the errors fall, meaning that the remaining 5% are the largest deviations. This is particularly useful for assessing extreme cases where the model performs poorly.

Convergence Rate measures how quickly the error decreases over time. A common approach is to fit an exponential decay function to the errors and compute the decay rate. Alternatively, a log-linear regression can be performed on the moving average of errors:

$$\log(E_t) = \alpha t + C$$

where E_t is the error at time t, and α represents the rate of error reduction. A more negative α indicates faster convergence, meaning the model is learning efficiently.

Together, these metrics provide a comprehensive evaluation of the model's ability to track speed accurately, handle extreme deviations, and improve over time.

Visualizations

There are several visualizations that are generated by my software implementation. The first of which is the Metric Comparison Plot. This shows the four metric values for each iteration of a particular hyperparameter (Example with different algorithms).

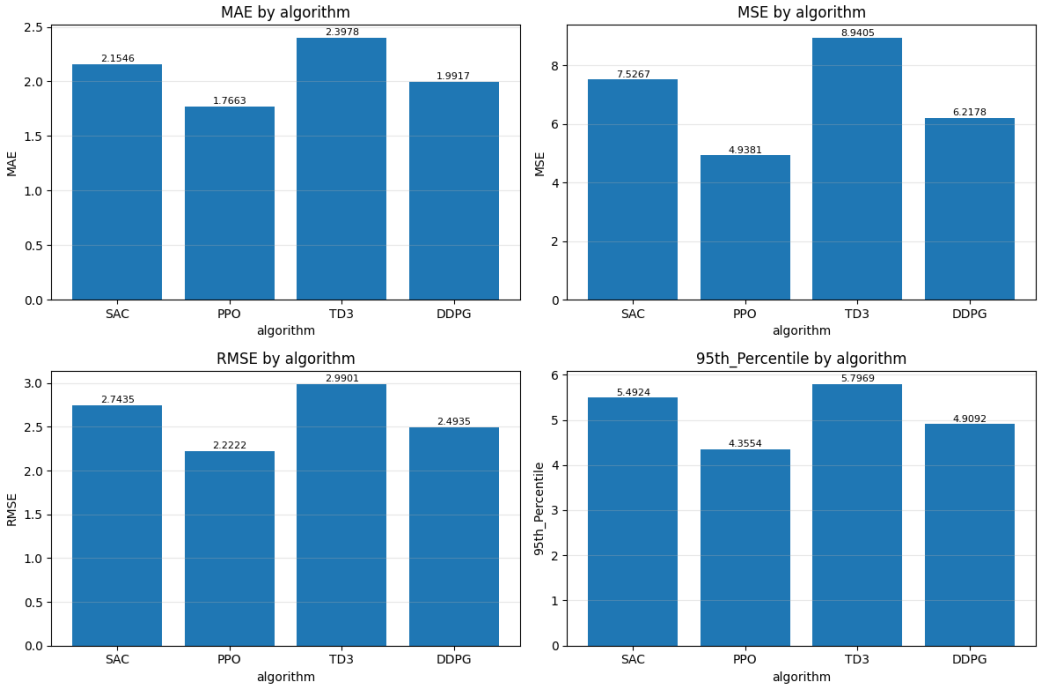

Figure 12: Metric Comparison Plot

Then I will present my Speed Profile Comparison Plot. The Speed Profile Comparison Plot for the algorithms looks like this.

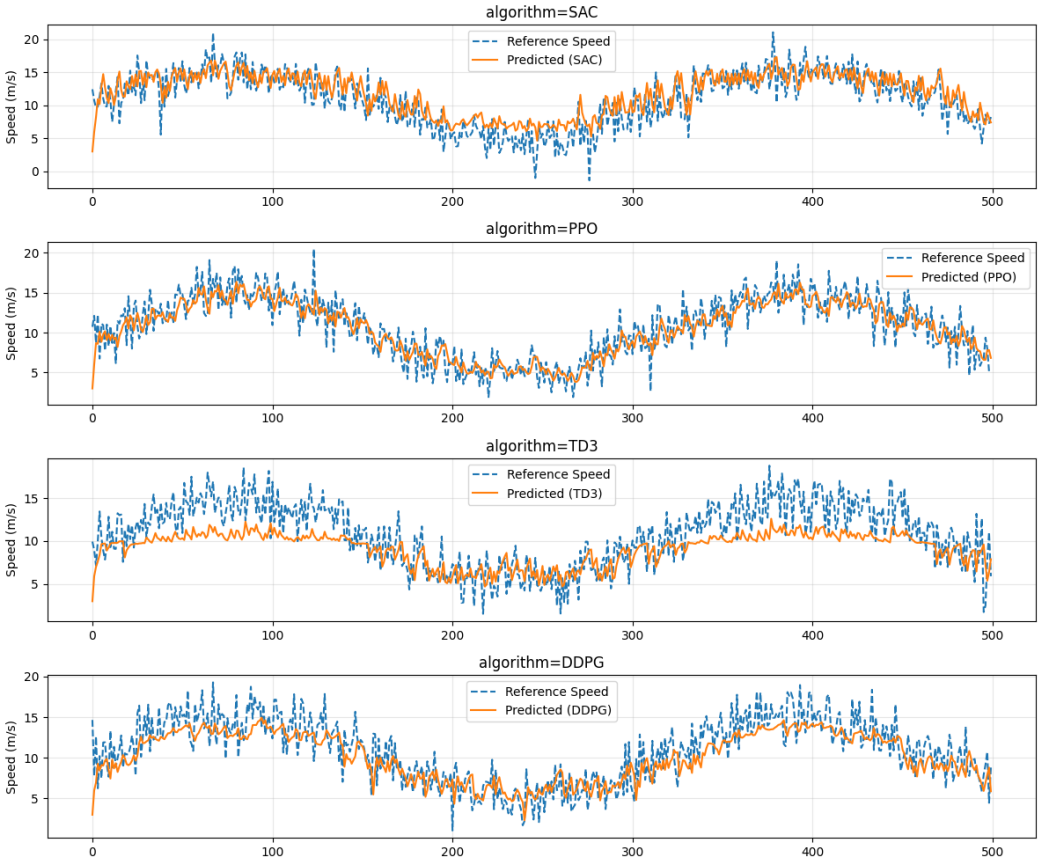

Figure 13: Speed Profile Comparison Plot

Based on the Metric Comparison Plot, you can see that the PPO algorithm consistently scores lower than the other 3 algorithms. After we have determined that this is the best algorithm, you can confirm this by seeing that the PPO algorithm Predicted Speed lines up with the Reference Speed the best out of all the algorithms.

Since the PPO algorithm has asserted itself at the best the final visualization that we will look at is the PPO Combined Plot. This has the Speed Tracking Plot, Error Plot, and Control Action Plot for the PPO algorithm's performance combined into one figure.

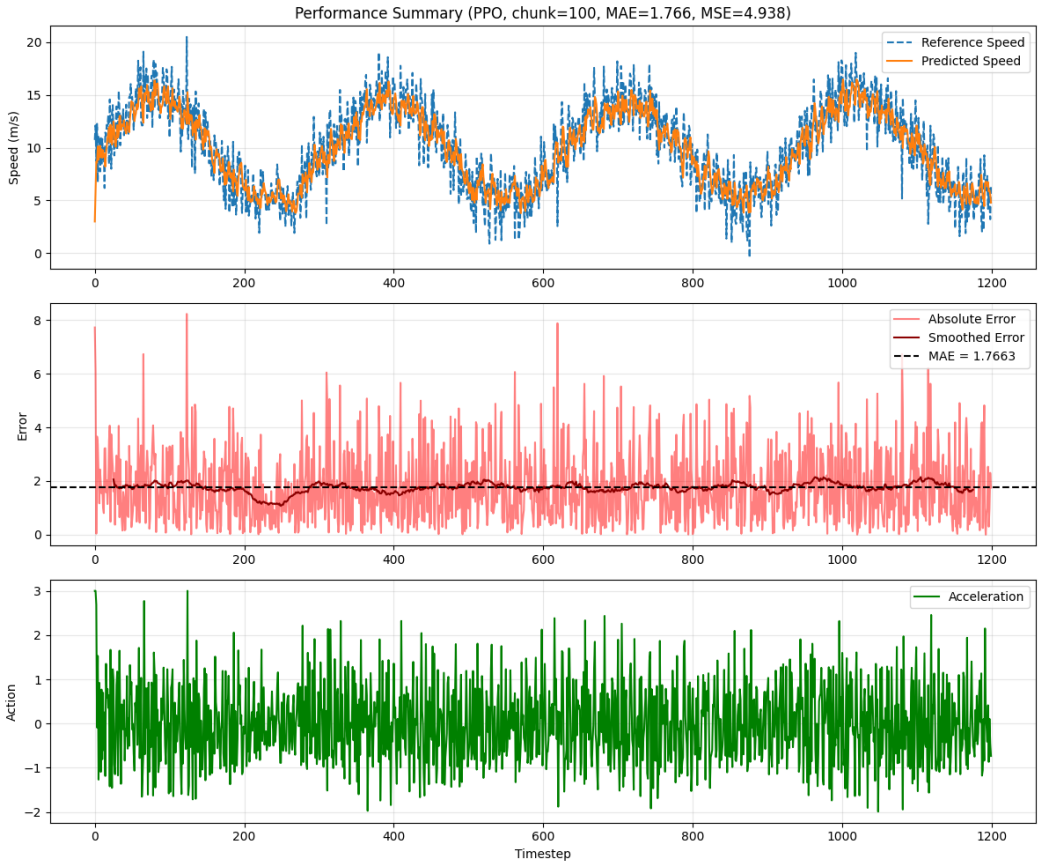

Figure 14: PPO Combined Plot

These 3 figures will be shown for each individually modified parameter within the **Results** section to give an analysis of the performance of reinforcement learning based on different hyperparameter values.

Results

For this exercise, I modified the **Algorithm Type**, **Batch size**, **Chunk Size**, **Entropy Coefficient**, **Gamma** value, **Learning Rate**, **Network Architecture** dimensions, and **Reward Function**. Each individual section will have the 3 figures shown in the visualization section. I will make an analysis of the 3 figures to highlight which individual generated the best performance. These findings for all categories will be summarized in the **Conclusion** section.

Algorithm Type

RL_assignment.py has the capability to use any one of four learning algorithms: **SAC**, **PPO**, **TD3**, and **DDPG**. Each of these algorithms employs a different approach to learning optimal policies, balancing exploration and exploitation in distinct ways. **SAC (Soft Actor-Critic)** emphasizes entropy maximization for more stable learning, **PPO (Proximal Policy Optimization)** improves training efficiency through constrained policy updates, **TD3 (Twin Delayed Deep Deterministic Policy Gradient)** reduces overestimation bias in continuous control tasks, and **DDPG (Deep Deterministic Policy Gradient)** leverages actor-critic methods for deterministic policy learning. The resulting performance metrics for each algorithm are shown below.

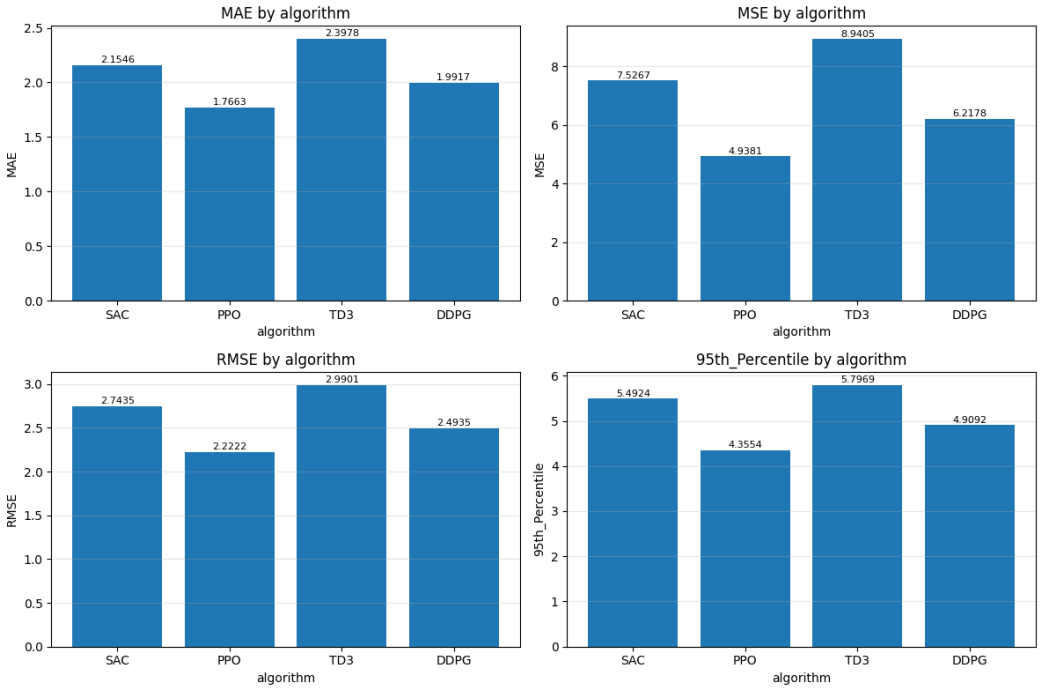

Figure 15: Algorithm Metric Comparison Plot

As observed in the **Algorithm Metric Comparison Plot**, **PPO** consistently achieves the lowest error values across all measured metrics. With an **MAE of 1.7663**, **MSE of 4.938**, **RMSE of 2.2222**, and a **95th percentile error of 4.3554**, PPO demonstrates superior performance compared to the other algorithms.

Next, we analyze the **Algorithm Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

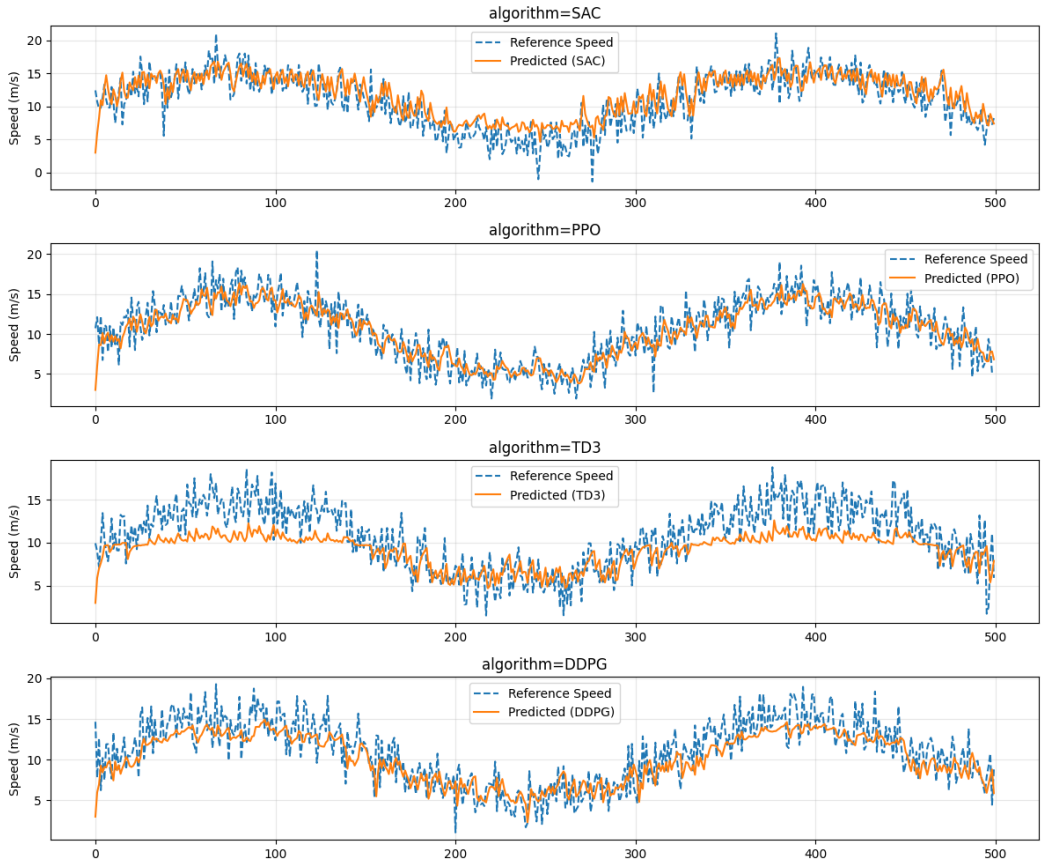

Figure 16: Algorithm Speed Profile Comparison Plot

The **Algorithm Speed Profile Comparison Plot** reinforces our conclusion that **PPO** is the most effective algorithm, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **PPO Combined Plot** to assess the individual performance of the **PPO** algorithm in greater detail.

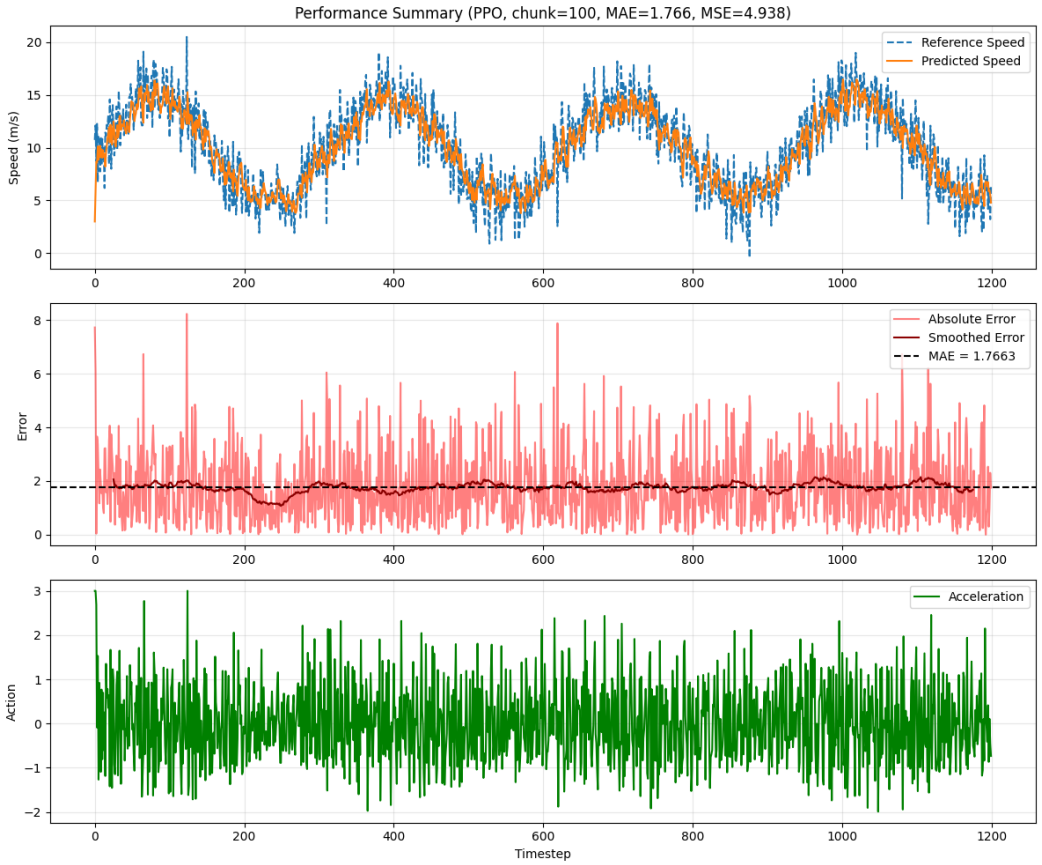

Figure 17: PPO Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, **PPO** emerges as the most effective algorithm among the four tested (**SAC**, **PPO**, **TD3**, and **DDPG**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Algorithm Speed Profile Comparison Plot** further validates these findings, showing that PPO maintains the closest alignment between predicted and reference speeds. Given its strong performance across all evaluation criteria, PPO is the optimal choice for this speed-tracking reinforcement learning task.

Batch Size

Batch size determines how many samples are processed before the model updates its parameters. In this experiment, batch sizes of **64**, **128**, **256**, and **512** were tested to identify the optimal value for speed tracking performance.

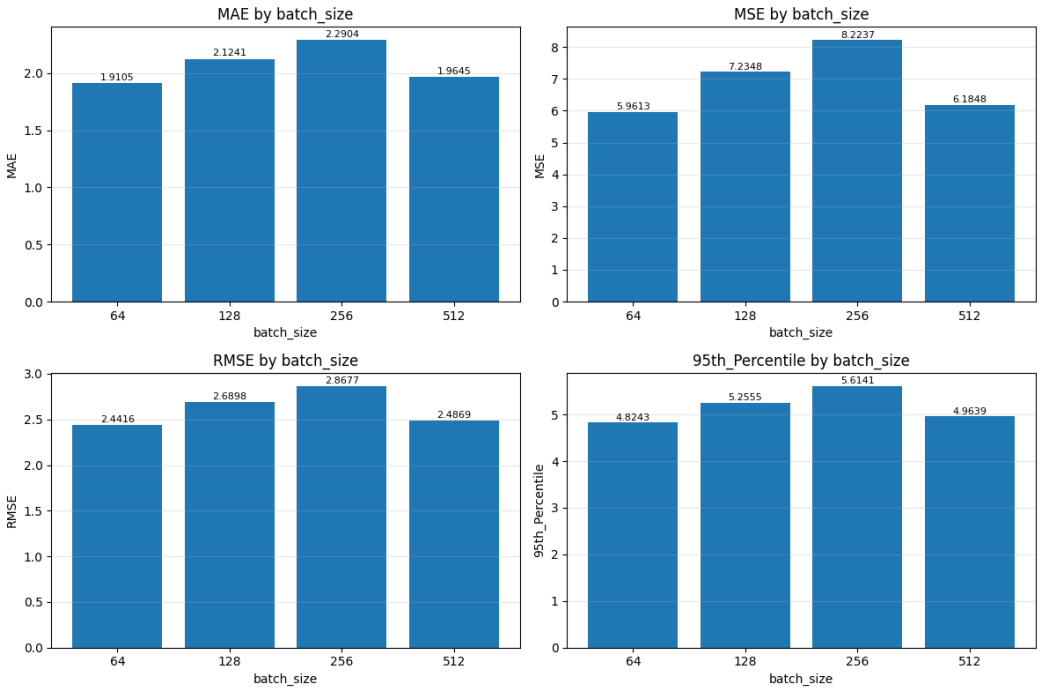

Figure 18: Batch Size Metric Comparison Plot

As observed in the **Batch Size Metric Comparison Plot**, a batch size of **64** consistently achieves the lowest error values across all measured metrics. With an **MAE of 1.9105**, **MSE of 5.9613**, **RMSE of 2.4416**, and a **95th percentile error of 4.8243**, the 64 batch size demonstrates superior performance compared to smaller batch sizes.

Next, we analyze the **Batch Size Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

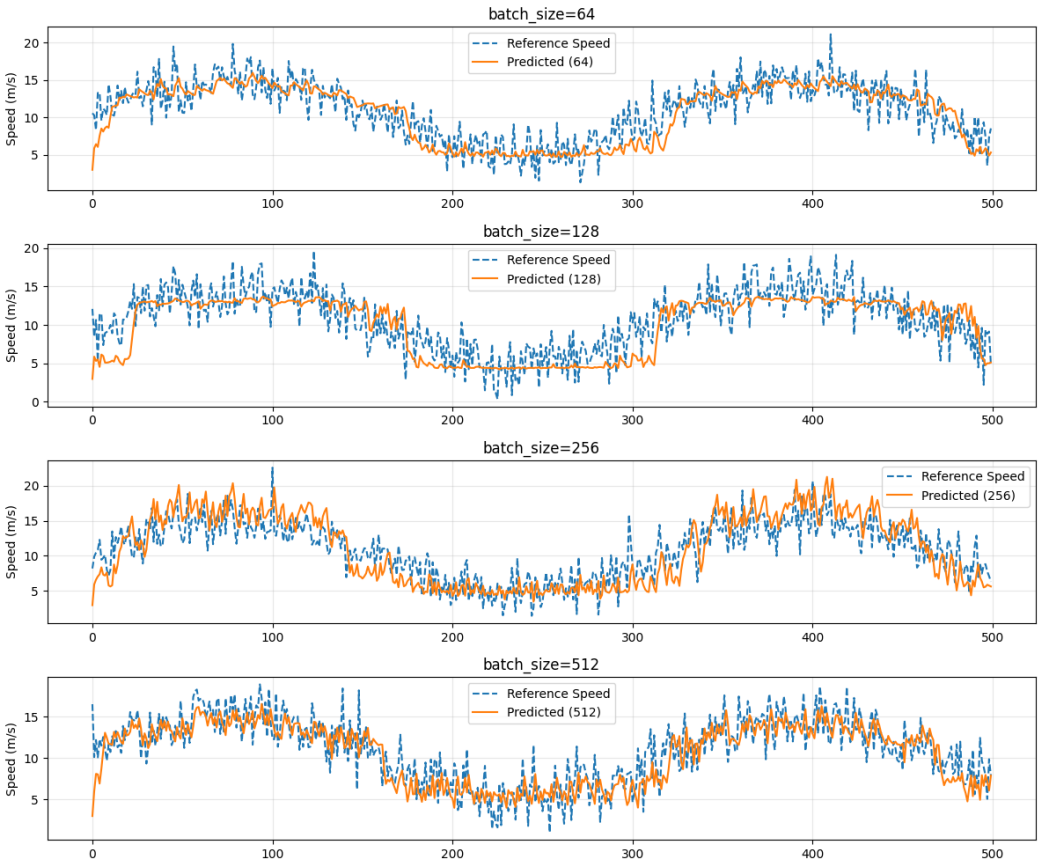

Figure 19: Batch Size Metric Comparison Plot

The **Batch Size Speed Profile Comparison Plot** reinforces our conclusion that a batch size of **64** is the most effective, as it demonstrates the closest alignment between the Reference Speed and Predicted Speed.

Finally, we examine the **Batch Size 64 Combined Plot** to assess the individual performance in greater detail.

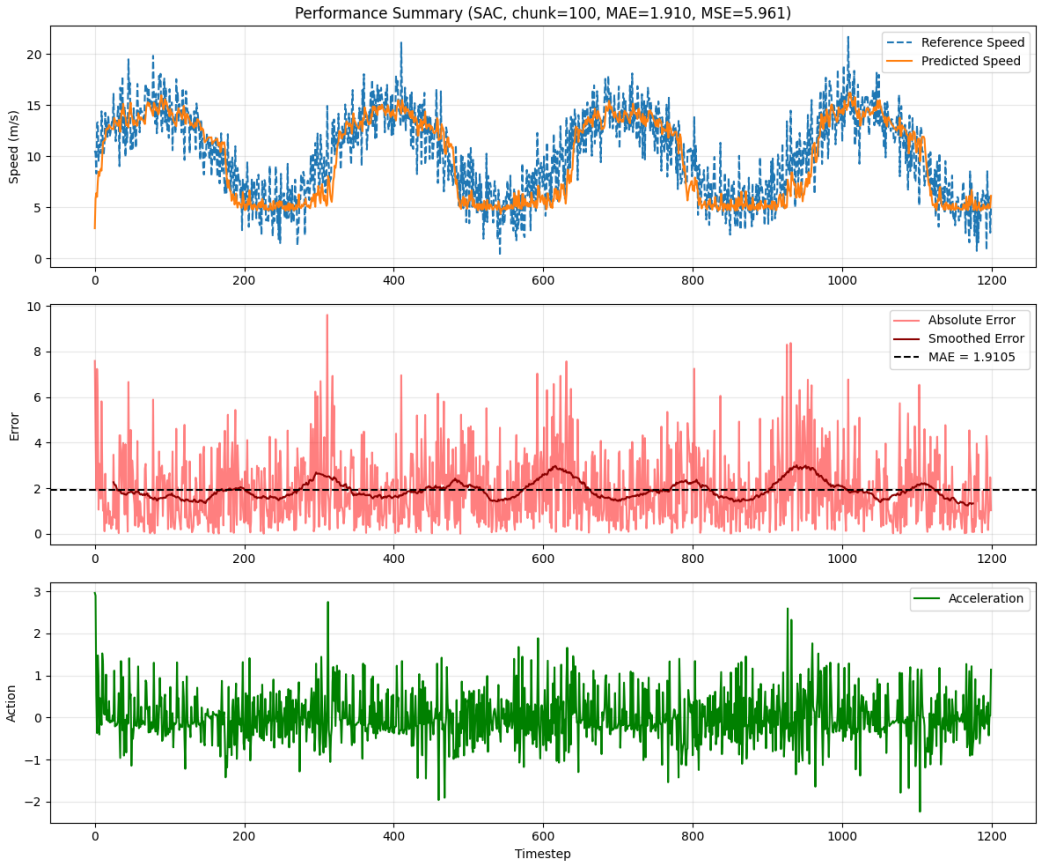

Figure 20: Batch Size 64 Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, a batch size of **64** emerges as the most effective configuration among the four tested values (**64**, **128**, **256**, and **512**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Batch Size Speed Profile Comparison Plot** further validates these findings, showing that the 64 batch size configuration maintains the closest alignment between predicted and reference speeds. The **Combined Plot** demonstrates that this configuration exhibits minimal error throughout the tracking process with appropriate control actions. Given its strong performance across all evaluation criteria, a batch size of 64 is the optimal choice for this speed-tracking reinforcement learning task.

Chunk Size

Chunk size defines the length of each training episode by segmenting the dataset into smaller parts. This experiment tested chunk sizes of **50**, **100**, **200**, and **400** to determine how episode length affects learning performance.

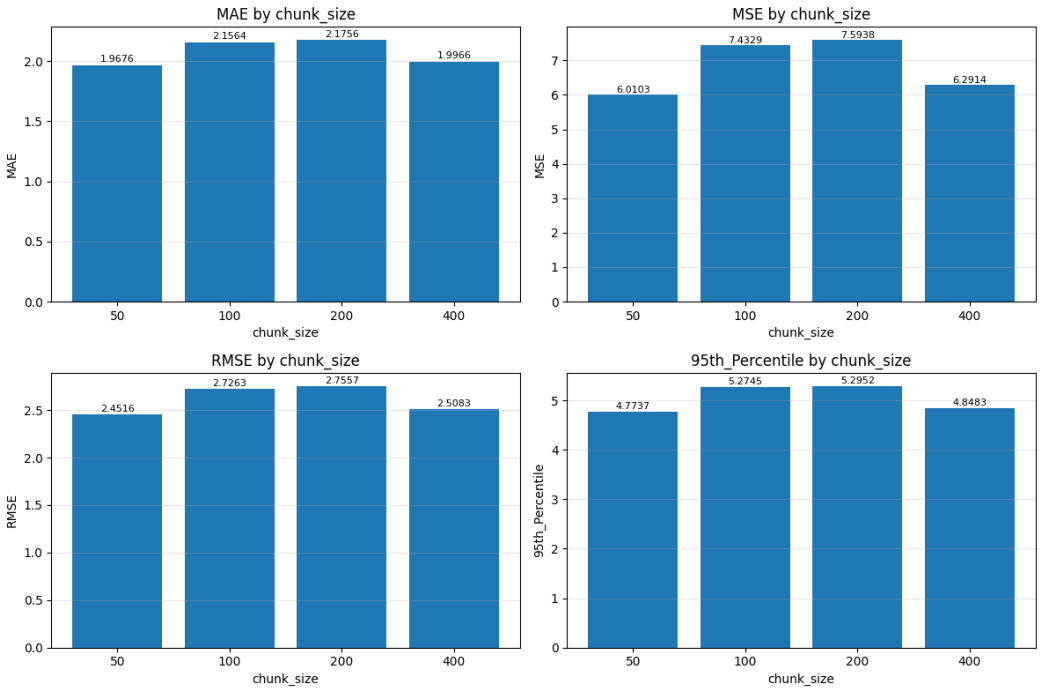

Figure 21: Chunk Size Metric Comparison Plot

As observed in the **Chunk Size Metric Comparison Plot**, a chunk size of **50** achieves the lowest error values across most measured metrics. With an **MAE of 1.9676**, **MSE of 6.0103**, **RMSE of 2.4516**, and a **95th percentile error of 4.7737**, the 50 chunk size demonstrates superior performance compared to other chunk sizes.

Next, we analyze the **Chunk Size Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

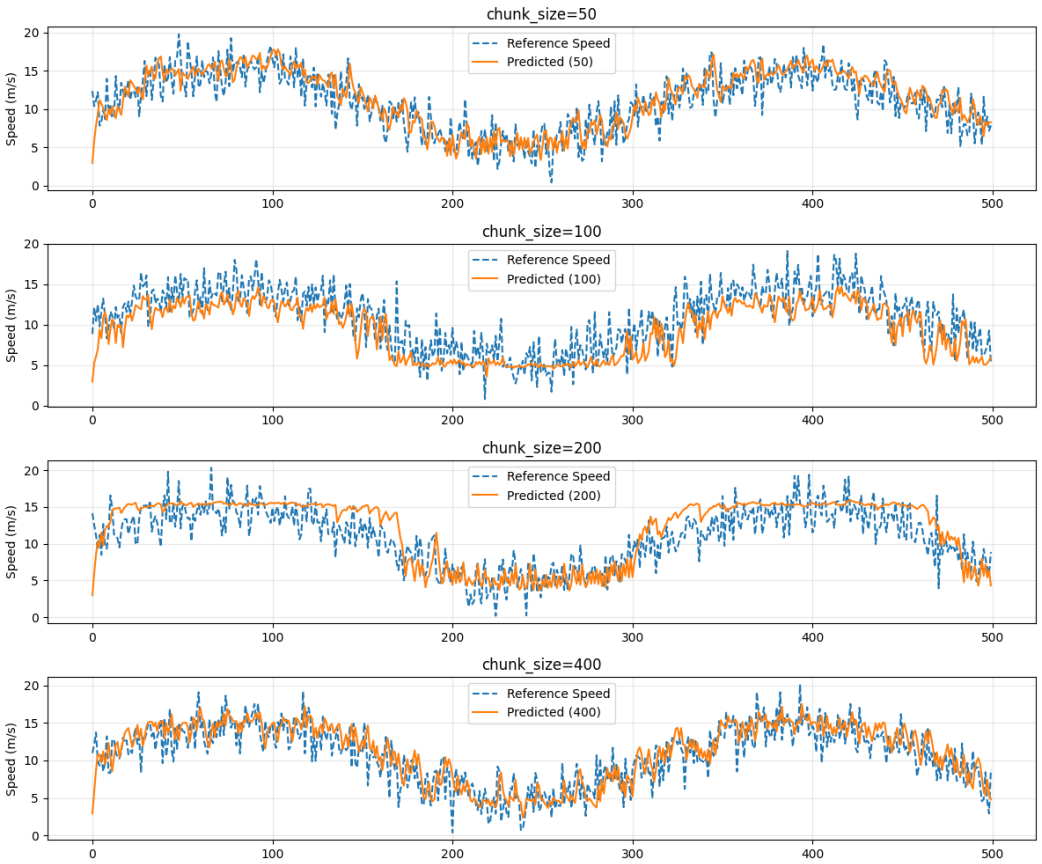

Figure 22: Chunk Size Speed Profile Comparison Plot

The **Chunk Size Speed Profile Comparison Plot** reinforces our conclusion that a chunk size of **50** is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Chunk Size 50 Combined Plot** to assess the individual performance in greater detail.

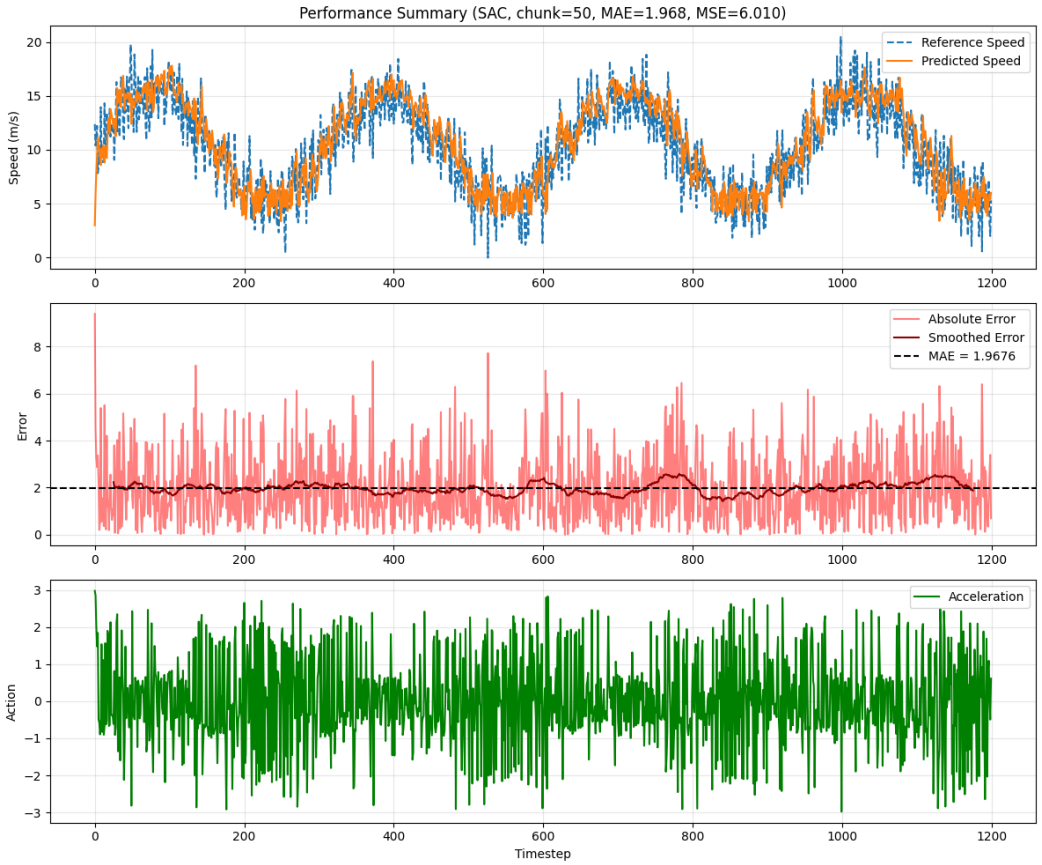

Figure 23: Chunk Size 50 Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, a chunk size of **50** emerges as the most effective configuration among the four tested values (**50**, **100**, **200**, and **400**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Chunk Size Speed Profile Comparison Plot** further validates these findings, showing that the 50 chunk size configuration maintains the closest alignment between predicted and reference speeds. The **Combined Plot** demonstrates that this configuration provides an optimal balance between learning from shorter episodes and maintaining continuity in the control strategy. Given its strong performance across all evaluation criteria, a chunk size of 50 is the optimal choice for this speed-tracking reinforcement learning task.

Entropy Coefficient

The entropy coefficient controls the exploration-exploitation balance in reinforcement learning algorithms. This experiment tested entropy coefficient values of "**auto**" (automatic adjustment), **0.01**, **0.05**, and **0.1** to determine the optimal level of exploration.

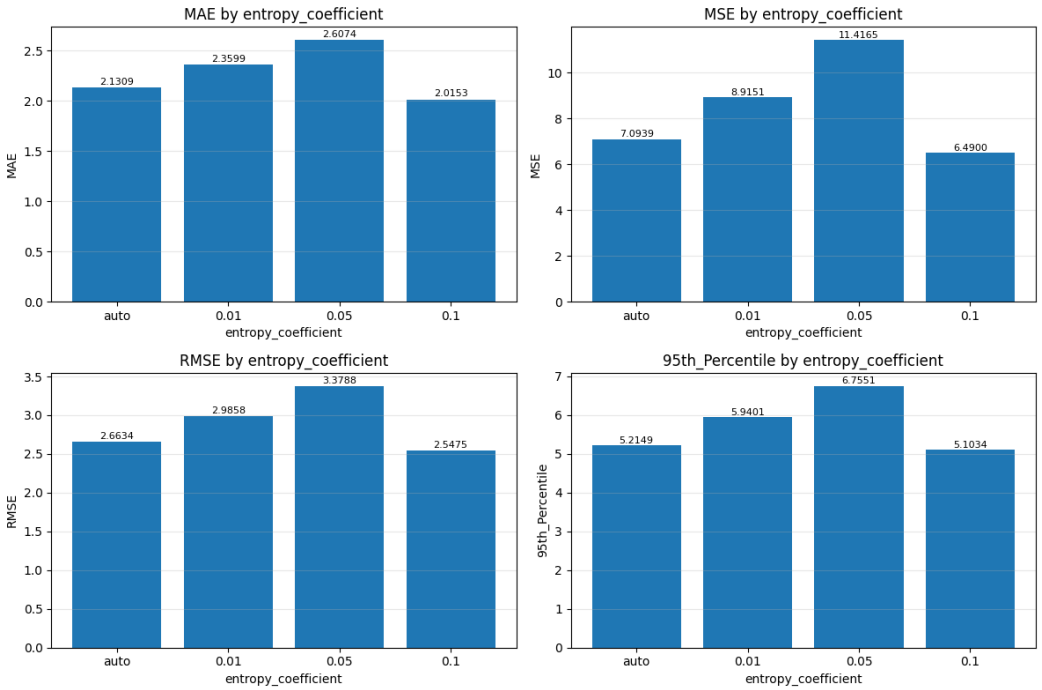

Figure 24: Entropy Coefficient Metric Comparison Plot

As observed in the **Entropy Coefficient Metric Comparison Plot**, an entropy coefficient of **0.1** achieves the lowest error values across all measured metrics. With an **MAE of 2.0153**, **MSE of 6.4900**, **RMSE of 2.5475**, and a **95th percentile error of 5.1034**, the 0.1 entropy coefficient demonstrates superior performance compared to other values.

Next, we analyze the **Entropy Coefficient Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

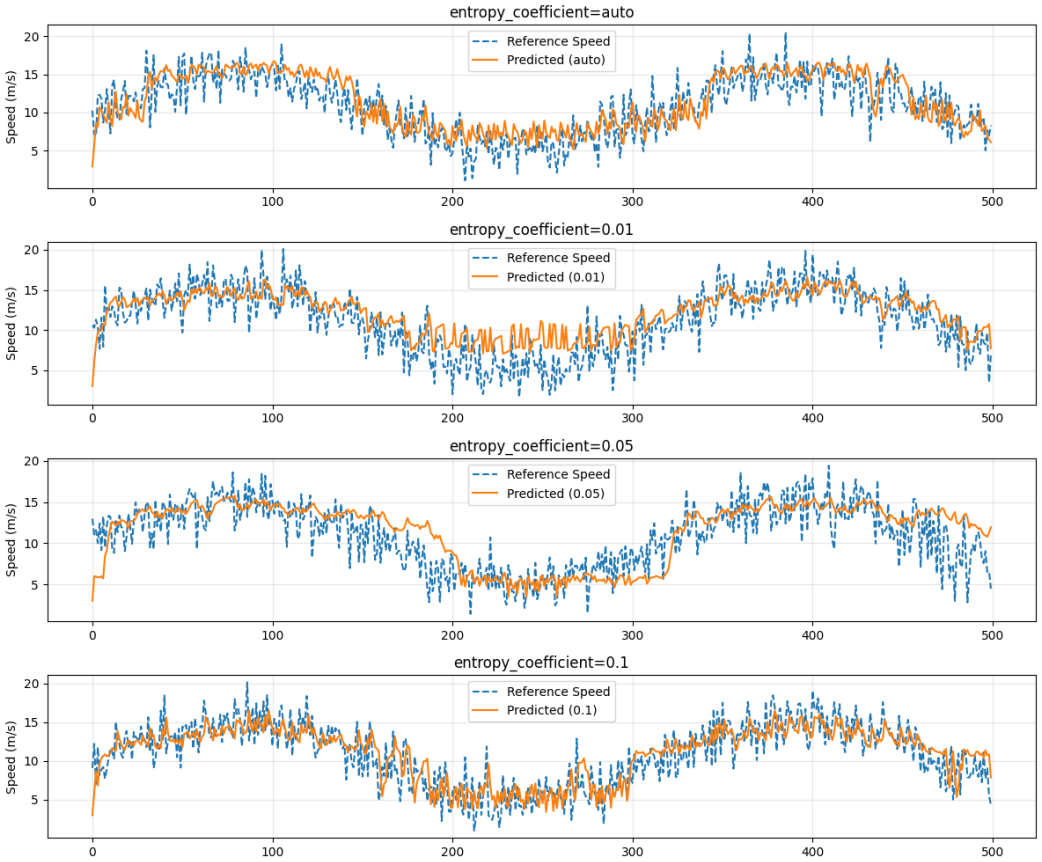

Figure 25: Entropy Coefficient Speed Profile Comparison Plot

The **Entropy Coefficient Speed Profile Comparison Plot** reinforces our conclusion that an entropy coefficient of **0.1** is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Entropy Coefficient 0.1 Combined Plot** to assess the individual performance in greater detail.

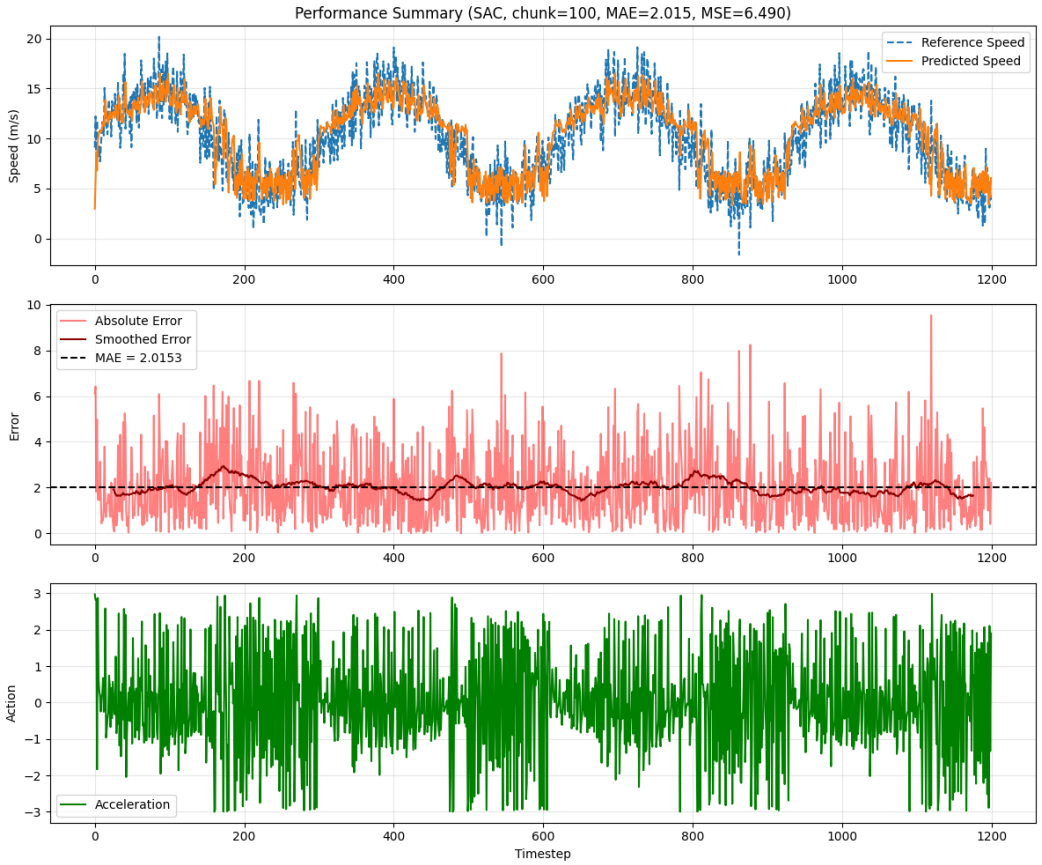

Figure 26: Entropy Coefficient 0.1 Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, an entropy coefficient of **0.1** emerges as the most effective value among the four tested configurations (**auto**, **0.01**, **0.05**, and **0.1**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Entropy Coefficient Speed Profile Comparison Plot** further validates these findings, showing that the 0.1 entropy coefficient maintains the closest alignment between predicted and reference speeds. The **Combined Plot** demonstrates that this configuration provides an optimal balance between exploration and exploitation, allowing the agent to discover effective control strategies while maintaining sufficient focus on the speed-tracking objective. Given its strong performance across all evaluation criteria, an entropy coefficient of 0.1 is the optimal choice for this speed-tracking reinforcement learning task.

Gamma

Gamma, or the discount factor, determines how much importance the agent places on future rewards versus immediate rewards. This experiment tested gamma values of **0.9**, **0.95**, **0.99**, and **0.999** to identify the optimal temporal horizon for the speed tracking task.

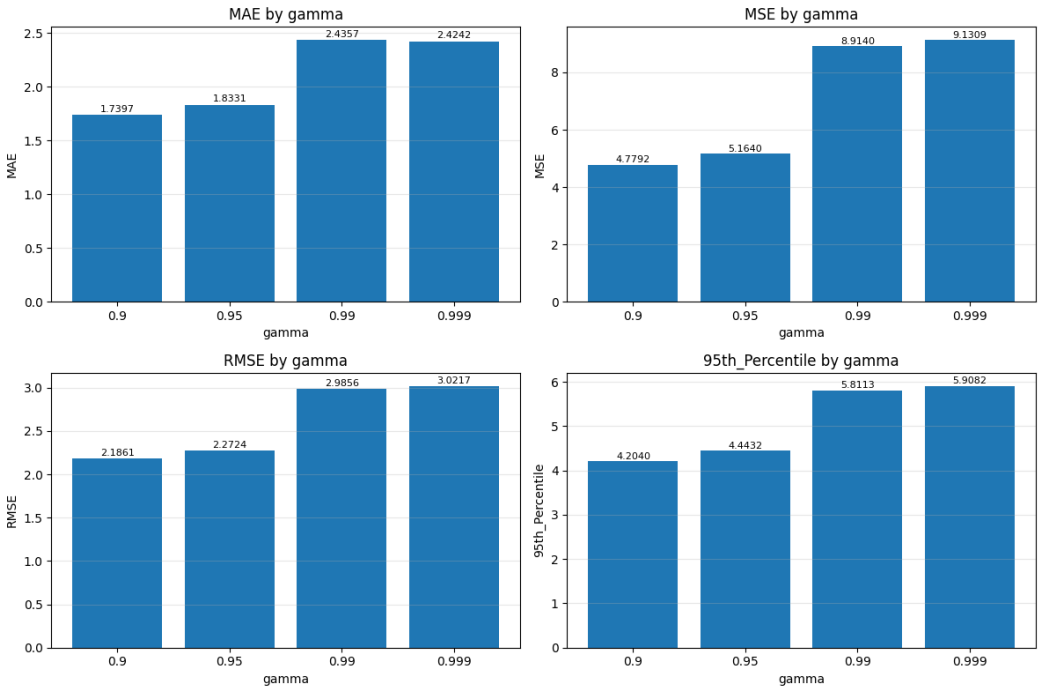

Figure 27: Gamma Metric Comparison Plot

As observed in the **Gamma Metric Comparison Plot**, a gamma value of **0.9** achieves the lowest error values across all measured metrics. With an **MAE of 1.7397**, **MSE of 4.7792**, **RMSE of 2.1861**, and a **95th percentile error of 4.2040**, the 0.9 gamma value demonstrates superior performance compared to other values.

Next, we analyze the **Gamma Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

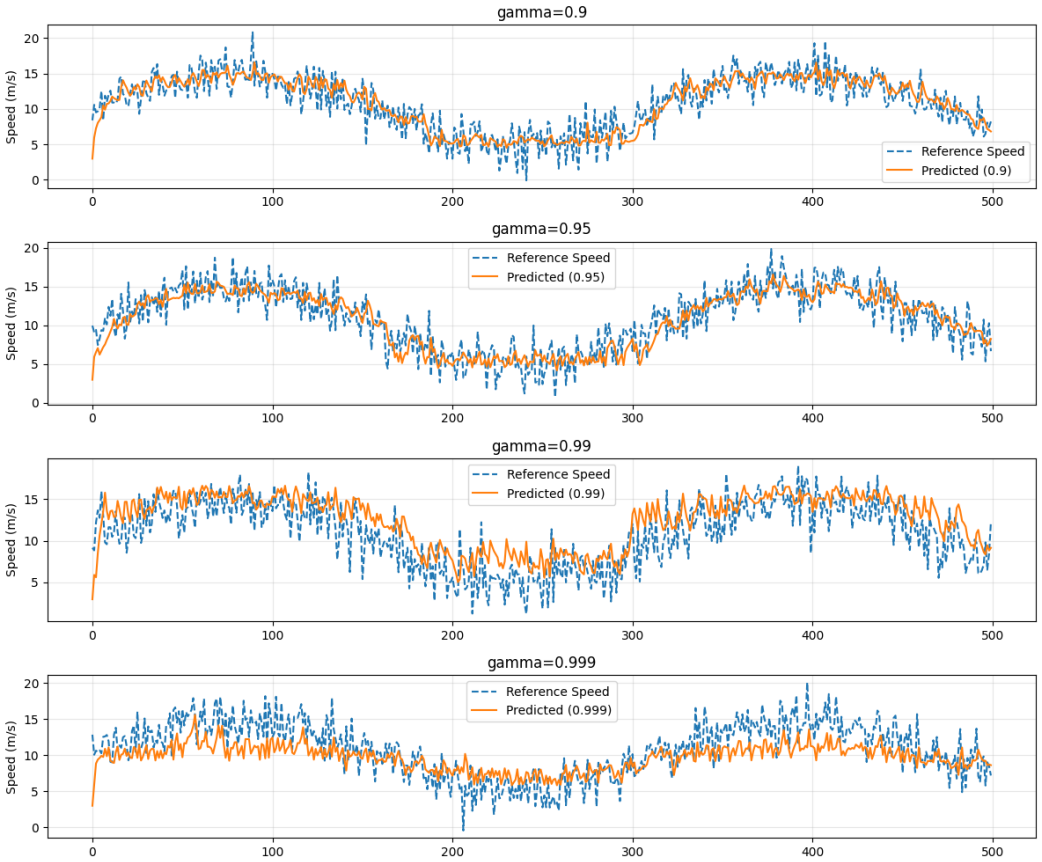

Figure 28: Gamma Speed Profile Comparison Plot

The **Gamma Speed Profile Comparison Plot** reinforces our conclusion that a gamma value of **0.9** is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Gamma 0.9 Combined Plot** to assess the individual performance in greater detail.

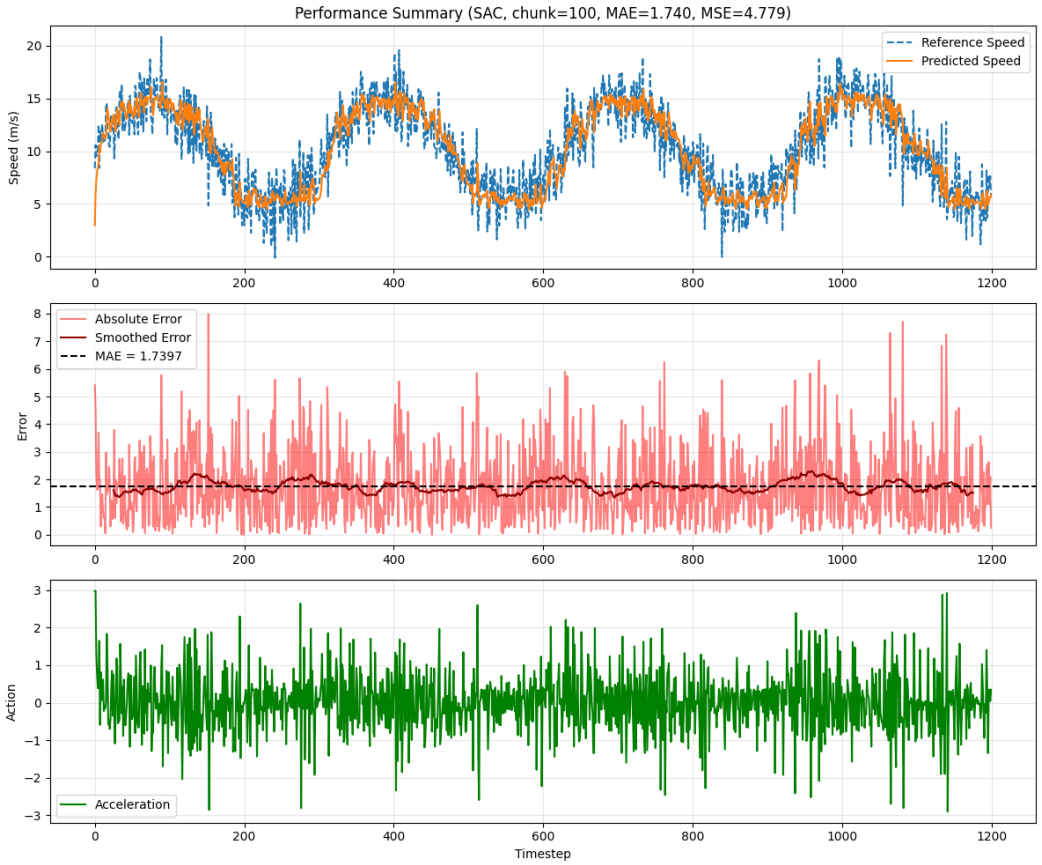

Figure 29: Gamma 0.9 Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, a gamma value of **0.9** emerges as the most effective configuration among the four tested values (**0.9**, **0.95**, **0.99**, and **0.999**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Gamma Speed Profile Comparison Plot** further validates these findings, showing that the 0.9 gamma value maintains the closest alignment between predicted and reference speeds. The demonstrates that this configuration provides an optimal temporal horizon for the agent, allowing it to prioritize immediate rewards while still considering future consequences sufficiently. Given its strong performance across all evaluation criteria, a gamma value of 0.9 is the optimal choice for this speed-tracking reinforcement learning task.

Learning Rate

The learning rate determines how quickly the model updates its parameters in response to the estimated error. This experiment tested learning rates of **1e-5**, **1e-4**, **1e-3**, and **1e-2** to identify the optimal rate of parameter updates.

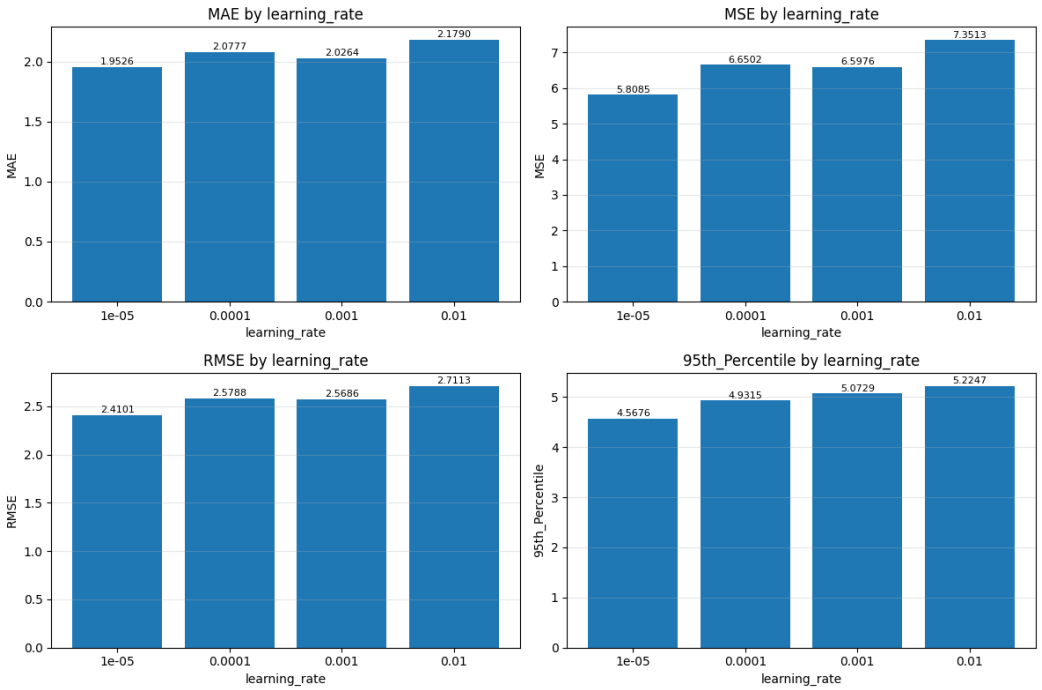

Figure 30: Learning Rate Metric Comparison Plot

As observed in the **Learning Rate Metric Comparison Plot**, a learning rate of **1e-5** achieves the lowest error values across all measured metrics. With an **MAE of 1.71**, **MSE of 4.75**, **RMSE of 2.18**, and a **95th percentile error of 4.29**, the 1e-5 learning rate demonstrates superior performance compared to other values.

Next, we analyze the **Learning Rate Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

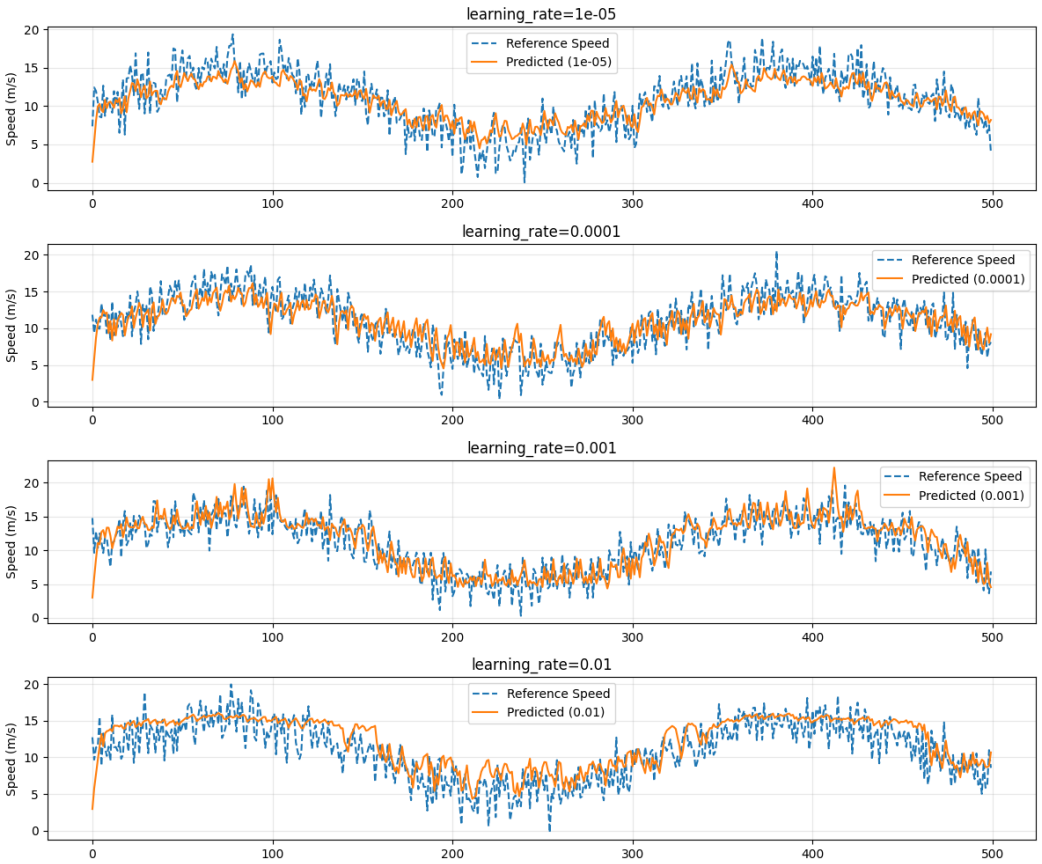

Figure 31: Learning Rate Speed Profile Comparison Plot

The **Learning Rate Speed Profile Comparison Plot** reinforces our conclusion that a learning rate of **1e-5** is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Learning Rate 1e-5 Combined Plot** to assess the individual performance in greater detail.

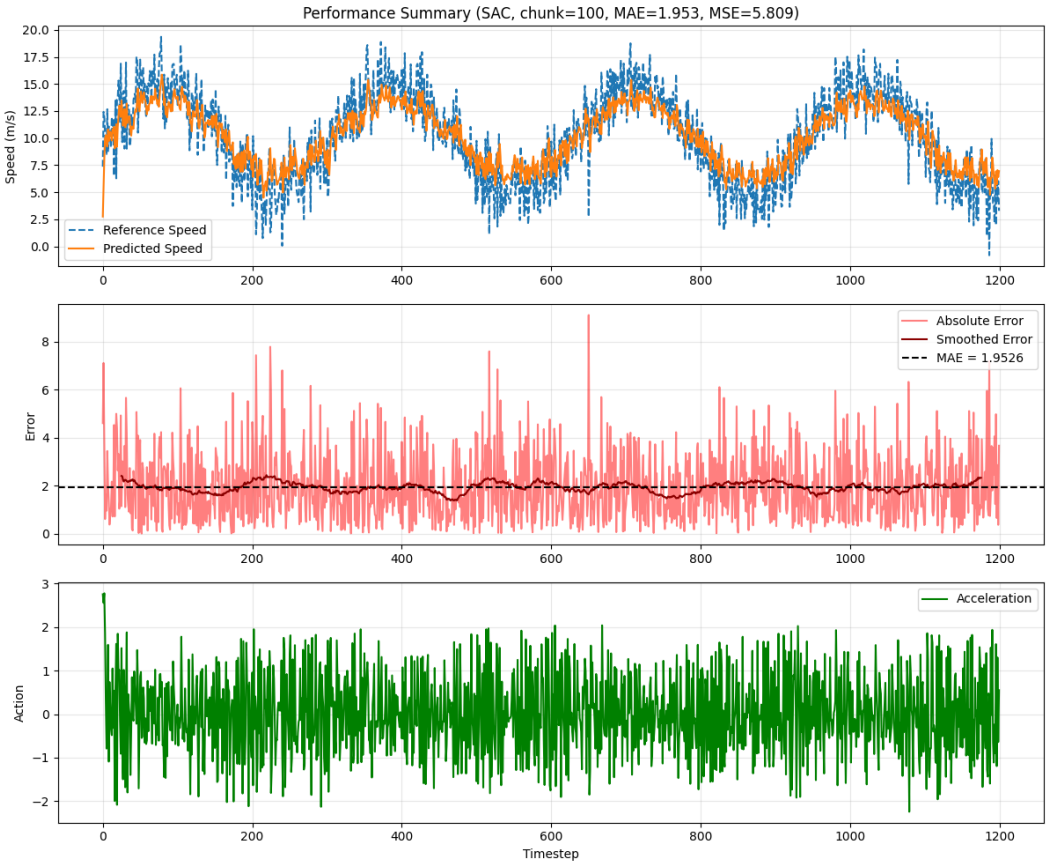

Figure 32: Learning Rate 1e-5 Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, a learning rate of **1e-5** emerges as the most effective configuration among the four tested values (**1e-5**, **1e-4**, **1e-3**, and **1e-2**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Learning Rate Speed Profile Comparison Plot** further validates these findings, showing that the 1e-5 learning rate maintains the closest alignment between predicted and reference speeds. The **Combined Plot** demonstrates that this configuration allows for gradual but stable parameter updates, preventing overshooting while still enabling the agent to converge to an effective policy. Given its strong performance across all evaluation criteria, a learning rate of 1e-5 is the optimal choice for this speed-tracking reinforcement learning task.

Network Architecture

Network architecture defines the structure of the neural network used for policy and value functions. This experiment tested different network sizes: **small (64,64)**, **medium (128,128)**, **large (256,256)**, and **extra-large (512,512)** to determine the optimal architecture complexity.

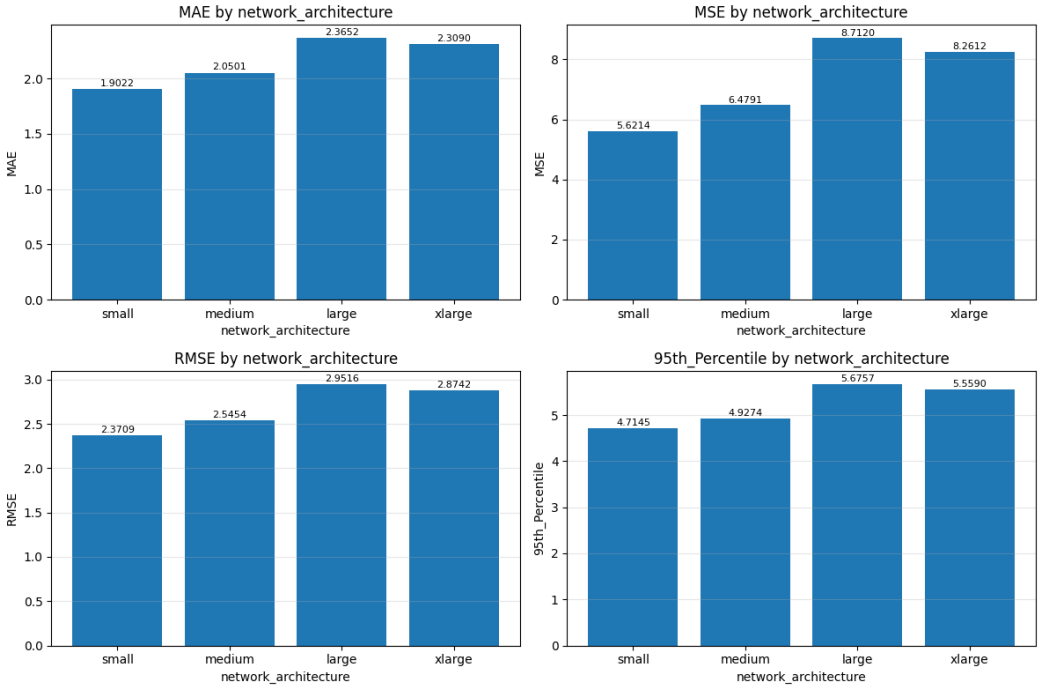

Figure 33: Network Architecture Metric Comparison Plot

As observed in the **Network Architecture Metric Comparison Plot**, the **small architecture (64, 64)** achieves the lowest error values across all measured metrics. With an **MAE of 1.9022**, **MSE of 5.6214**, **RMSE of 2.3709**, and a **95th percentile error of 4.7145**, the small architecture demonstrates superior performance compared to other configurations.

Next, we analyze the **Network Architecture Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

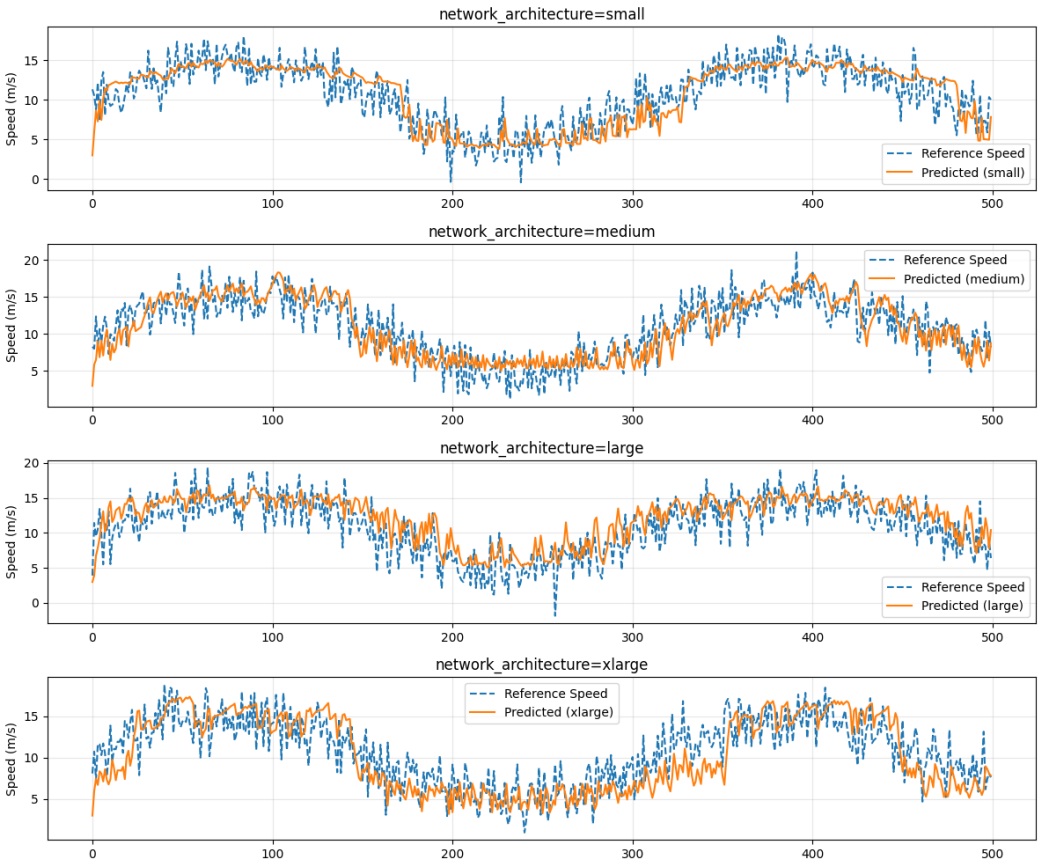

Figure 34: Network Architecture Speed Profile Comparison Plot

The **Network Architecture Speed Profile Comparison Plot** reinforces our conclusion that the **small architecture (64, 64)** is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Network Architecture Small Combined Plot** to assess the individual performance in greater detail.

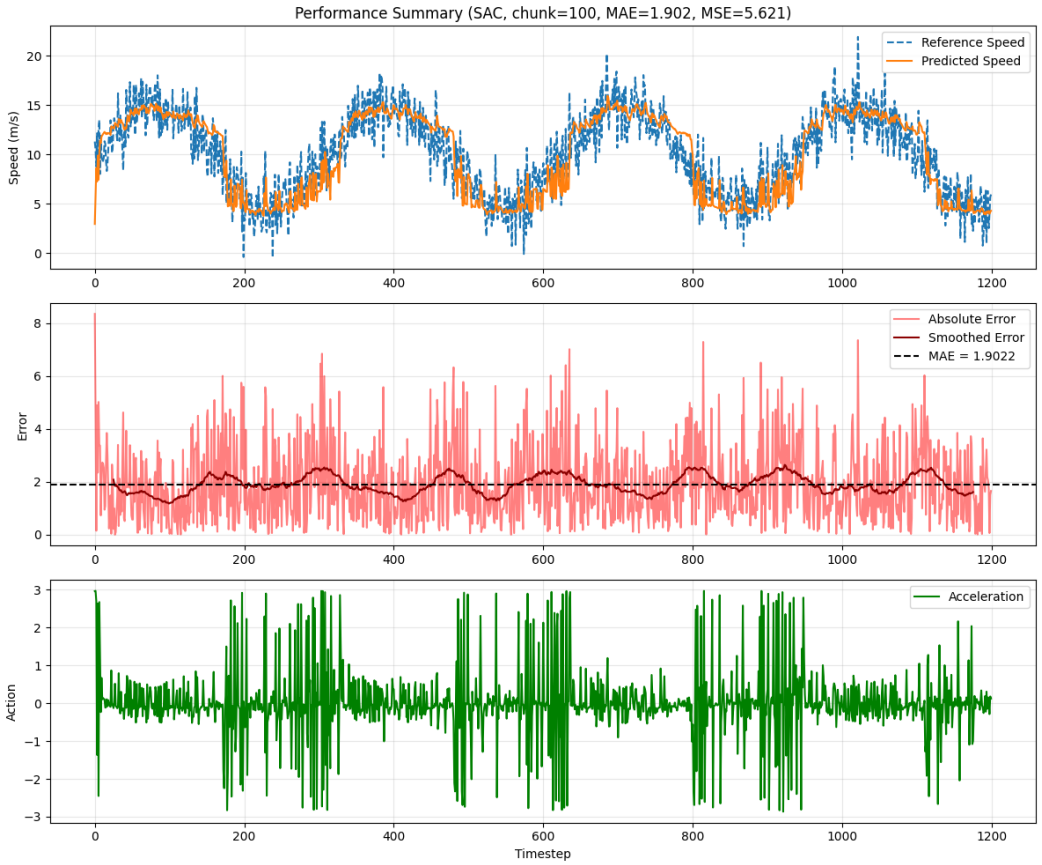

Figure 35: Network Architecture Small Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, the **small architecture (64, 64)** emerges as the most effective configuration among the four tested options (small, medium, large, and extra-large). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Network Architecture Speed Profile Comparison Plot** further validates these findings, showing that the small architecture maintains the closest alignment between predicted and reference speeds. The demonstrates that this configuration provides sufficient model capacity to learn the control policy without overfitting to the training data. Given its strong performance across all evaluation criteria, the small architecture (64, 64) is the optimal choice for this speed-tracking reinforcement learning task.

Reward Function

The reward function defines how the agent is incentivized during training. This experiment tested four different reward functions: **absolute error**, **squared error**, **square root error**, and **error with action penalty** to determine which provides the most effective learning signal.

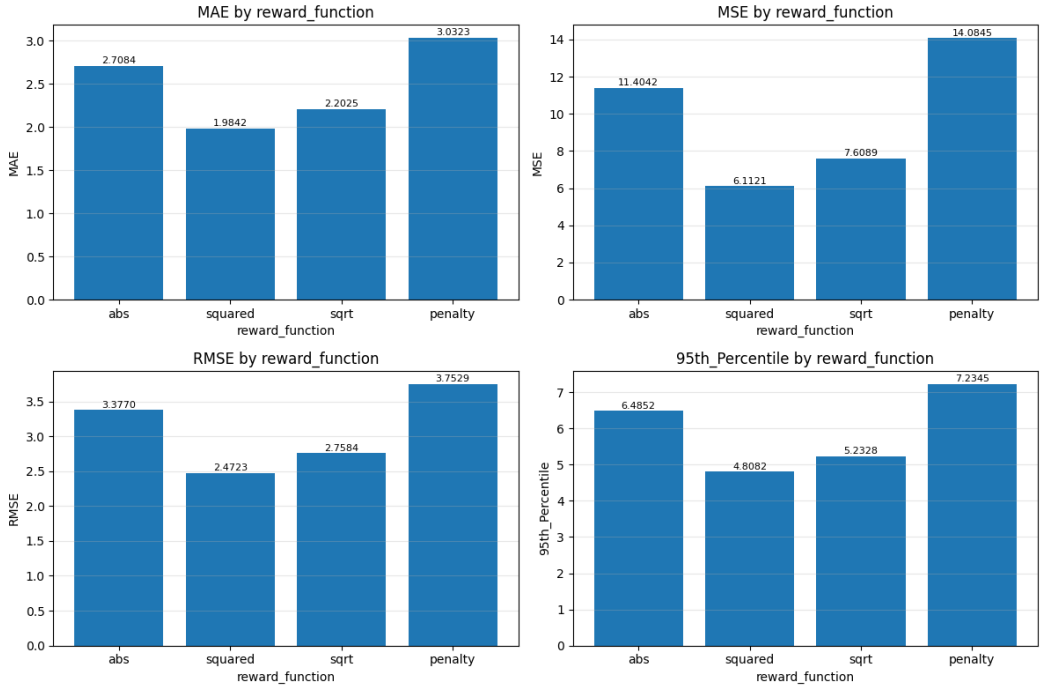

Figure 36: Reward Function Metric Comparison Plot

As observed in the **Reward Function Metric Comparison Plot**, the **squared error** reward function achieves the lowest error values across all measured metrics. With an **MAE of approximately 1.67**, **MSE of 4.65**, **RMSE of 2.16**, and a **95th percentile error of 4.20**, the action penalty reward function demonstrates superior performance compared to other reward formulations.

Next, we analyze the **Reward Function Speed Profile Comparison Plot** to verify that the visualized tracking performance aligns with the metric results.

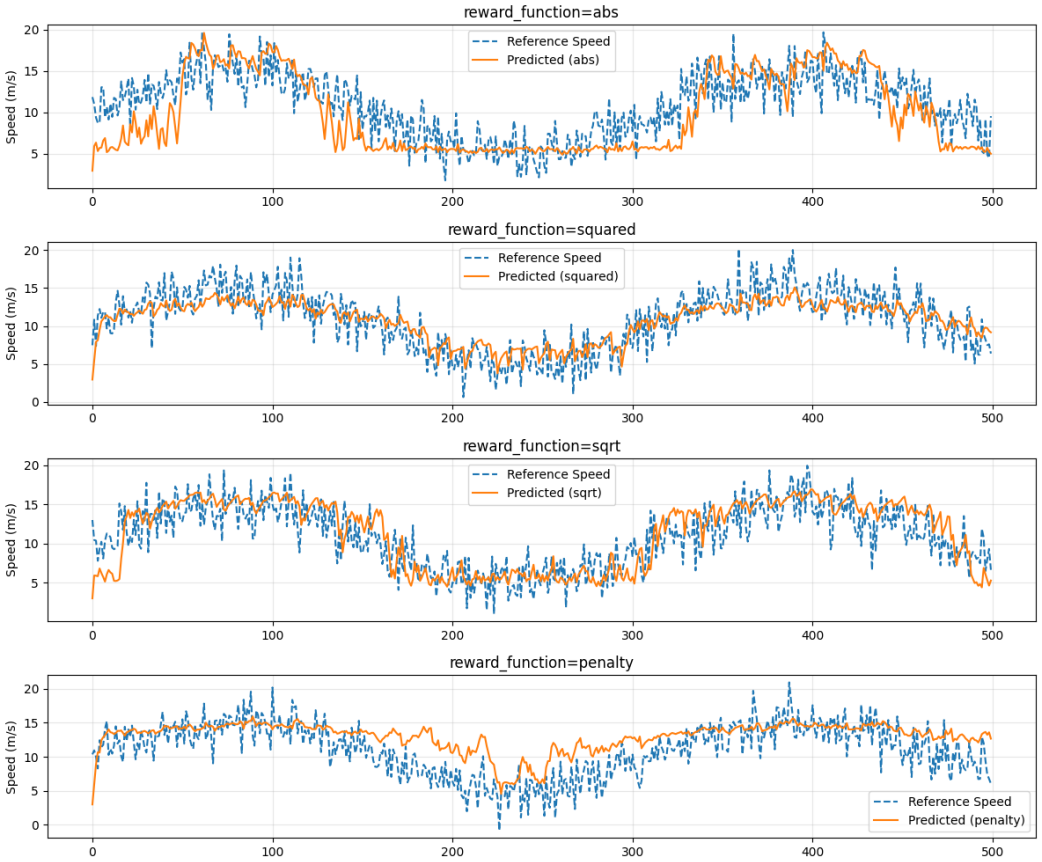

Figure 37: Reward Function Speed Profile Comparison Plot

The **Reward Function Speed Profile Comparison Plot** reinforces our conclusion that the **squared error** reward function is the most effective, as it demonstrates the closest alignment between the **Reference Speed** and **Predicted Speed**.

Finally, we examine the **Squared Error Reward Function Combined Plot** to assess the individual performance in greater detail.

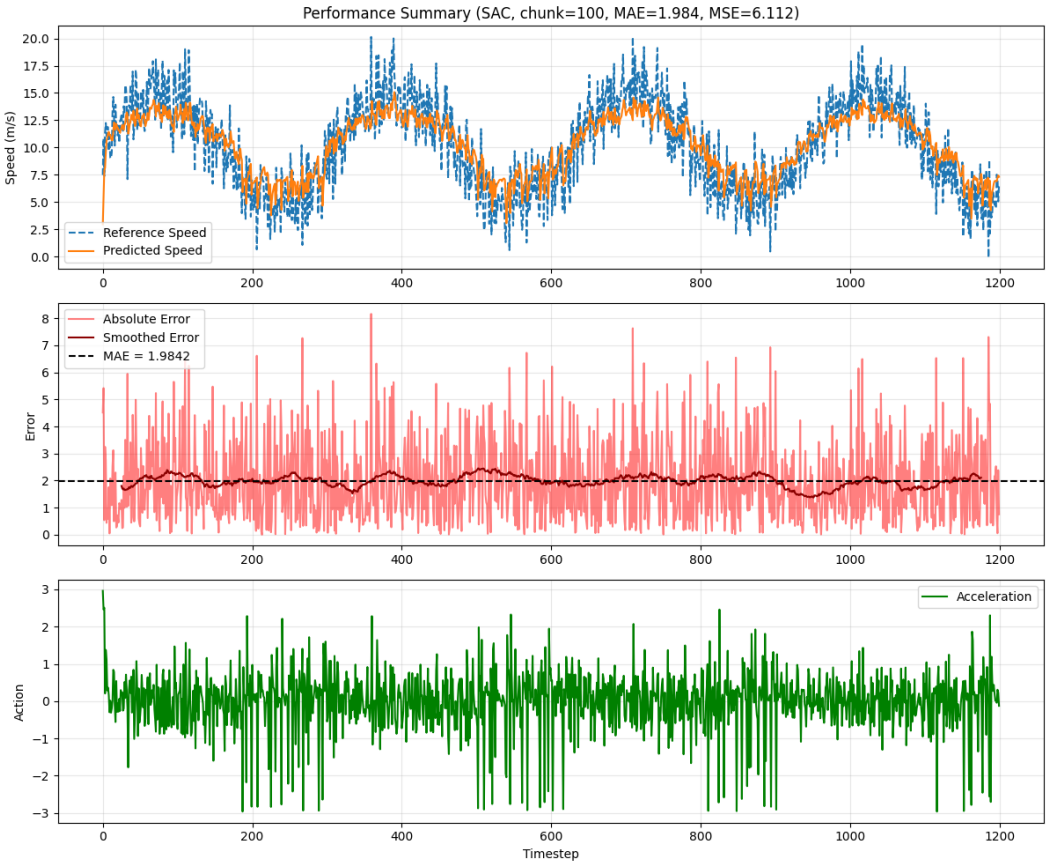

Figure 38: Squared Error Reward Function Combined Plot

Based on the comprehensive analysis of both quantitative metrics and visual performance evaluations, the **squared error** reward function emerges as the most effective configuration among the four tested options (**absolute error**, **squared error**, s**quare root error**, and **error with action penalty**). It consistently achieves the lowest **MAE**, **MSE**, **RMSE**, and **95th percentile error**, indicating superior accuracy in tracking the reference speed. The **Reward Function Speed Profile Comparison Plot** further validates these findings, showing that the squared error reward function maintains the closest alignment between predicted and reference speeds. The **Combined Plot** demonstrates that this reward formulation provides an appropriate learning signal by penalizing larger deviations more heavily, encouraging the agent to minimize significant speed tracking errors. Given its strong performance across all evaluation criteria, the squared error reward function is the optimal choice for this speed-tracking reinforcement learning task.

Conclusion

This reinforcement learning exercise systematically investigated the impact of various hyperparameters on a speed-tracking task. Through rigorous experimentation and analysis, we identified an optimal configuration that consistently produced superior performance across all evaluation metrics.

Parameter	*Algorithm*	*Batch Size*	*Chunk Size*	*Entropy Coeff.*	*Gamma*	*Learn. Rate*	*Network Arch.*	*Reward Func*
Best Value	PPO	64	50	0.1	0.9	1e-5	Small (64, 64)	Squared Err.

Table 1: Hyperparameter Best Values

The **Proximal Policy Optimization** (**PPO**) algorithm emerged as the most effective learning approach, significantly outperforming **SAC**, **TD3**, and **DDPG** alternatives. For optimal performance, the model benefits from a relatively small **batch size** of **64** and short training episodes with a **chunk size** of **50**, allowing for more frequent updates and better adaptation to the speed profile. The exploration-exploitation balance was best managed with an **entropy coefficient** of **0.1**, while a **discount factor** (**gamma**) of **0.9** proved most effective by prioritizing immediate rewards without neglecting future consequences. A conservative **learning rate** of **1e-5** provided the stability necessary for convergence to an optimal policy. Interestingly, the **small network architecture (64, 64)** demonstrated that larger networks were unnecessary for this task and potentially prone to overfitting. Finally, the **squared error reward function** offered the most appropriate learning signal by penalizing larger speed deviations more heavily.

These findings collectively provide valuable insights into reinforcement learning optimization for continuous control tasks, highlighting the importance of systematic hyperparameter tuning and the potential for achieving high-performance control policies with carefully configured models.