# Reinforcement Learning Exercise

Joshua Glaspey

EEL6938: AI for Autonomous Systems

Due Date: March 21, 2025

# Table of Contents

## Contents

Joshua Glaspey
Spring 2025
Reinforcement Learning Exercise

# Introduction

In this assignment, the goal was to train an RL agent to follow a reference speed profile. Given a pre-existing codebase, I modified the model, experimented with different hyperparameters, and adjusted the episode length to study its impact on learning. Additionally, I explored different reward structures and evaluated how they influenced performance. To assess the effectiveness of these changes, I implemented various quantitative metrics and visualized the results through plots comparing predicted and reference speeds. This report walks through the modifications, experimental findings, and key results gained from the process.

# Code Modifications

To improve the reinforcement model's performance, I added several changes. These focused on addressing each point provided in the assignment's description, namely model types, hyperparameters, episode length, reward structures, and performance metrics. This section will cover all these changes.

## Models

The provided code utilizes a Soft Actor Critic (SAC) model. To evaluate the impact of different RL algorithms, three additional models were implemented alongside SAC: Proximal Policy Optimization (PPO), Twin Delayed DDPG (TD3), and Deep Deterministic Policy Gradient (DDPG). Each model has its own respective configuration of hyperparameters, but for consistency, no additional parameters outside of those provided in the SAC model were added. The code is shown below for each additional model.

```
model = PPO(policy="MlpPolicy", env=train_env, verbose=1, policy_kwargs=policy_kwargs, learning_rate=args.learning_rate, batch_size=args.batch_size, gamma=args.gamma, ent_coef=ent_coef_param, device=device)

model = TD3(policy="MlpPolicy", env=train_env, verbose=1, policy_kwargs=policy_kwargs, learning_rate=args.learning_rate, batch_size=args.batch_size, buffer_size=args.buffer_size, tau=args.tau, gamma=args.gamma, device=device)

model = DDPG(policy="MlpPolicy", env=train_env, verbose=1, policy_kwargs=policy_kwargs, learning_rate=args.learning_rate, batch_size=args.batch_size, buffer_size=args.buffer_size, tau=args.tau, gamma=args.gamma, device=device)
```

## Hyperparameters

Each of the models has a set of configurable hyperparameters. Below is the list of modifiable hyperparameters for each model.

1. SAC: Learning rate, Batch size, Buffer size, Tau, Gamma, Entropy Coefficient, Net Arch.
2. PPO: Learning rate, Batch size, Gamma, Entropy Coefficient, Net Arch.
3. TD3: Learning rate, Batch size, Buffer size, Tau, Gamma, Net Arch.
4. DDPG: Learning rate, Batch size, Buffer size, Tau, Gamma, Net Arch

Each hyperparameter was tested using a comprehensive range of values. For each training iteration, all hyperparameters were set to their default value except the one being tested to isolate functionality. These ranges are provided below (**bold** indicates the default value). For example, if currently testing the effects of "batch_size," all other hyperparameters are set to their default, and batch

size iterates across its five values. Additionally, this procedure was repeated for each model (where hyperparameters are applicable).

- Learning Rates: 1e-5, 3e-5, 1e-4**, 3e-4**, 1e-3, 3e-3, 1e-2, 3e-2, 1e-1
- Batch Sizes: 32, 64, 128, **256**, 512
- Buffer Sizes: 50000, 100000, **200000**, 500000, 1000000
- Tau: 0.0001, 0.001, **0.005**, 0.01, 0.02
- Gamma: 0.90, 0.95, **0.99**, 0.999, 1.0
- Entropy Coefficients: **'auto'**, 0.01, 0.05, 0.1, 0.2
- Net Archs: [64x64], [128x128], **[256x256]**, [512x512]

Each hyperparameter range was carefully selected to explore different trade-offs in model performance:

1. **Learning Rate (1e-5 to 1e-1)** – The values span several orders of magnitude to compare slow convergence with rapid learning. This broad range helps identify the optimal balance between learning speed and stability.
2. **Batch Size (32 to 512)** – The batch sizes are selected in powers of 2 to examine the effect of variance in updates. Smaller batches introduce more variability, which can enhance exploration, while larger batches provide more stable but computationally expensive updates.
3. **Buffer Size (50,000 to 1,000,000)** – This range tests the impact of memory capacity on training. Smaller buffers prioritize recent experiences, while larger buffers retain extensive history at the cost of increased resource consumption.
4. **Tau (0.0001 to 0.02)** – This parameter controls the rate of network updates. Lower values ensure smoother, incremental updates, while higher values enable faster adaptation at the risk of increased variability.
5. **Gamma (0.9 to 1.0)** – The discount factor determines the weighting of future rewards. Lower values emphasize short-term gains, whereas higher values favor long-term rewards, influencing overall policy behavior.
6. **Entropy Coefficient ('auto' / 0.0, 0.01 to 0.2)** – This regulates the exploration-exploitation balance. Higher values encourage more exploration, while the 'auto' setting dynamically adjusts based on policy confidence. For the SAC model, the default value is 'auto' and for the PPO model, the default value is 0.0.
7. **Network Architecture ([64×64] to [512×512])** – The varying sizes of the neural network test the trade-off between computational efficiency and representational power. Smaller architectures train faster, while larger networks capture more complex patterns.

The hyperparameters were added to the skeleton code using the argument parser. Following the same structure as the "—chunk_size" parameter, each of the provided hyperparameters were added such that they can be manually set per run via the terminal. The code to scan each hyperparameter is provided below – this is in the beginning of the main function. Note that, for the entropy coefficient, "-1.0" is a temporary value used to be masked with a value of "auto" for the variable.

```
parser.add_argument("--model", type=int, default=0)
parser.add_argument("--learning_rate", type=float, default=3e-4)
parser.add_argument("--batch_size", type=int, default=256)
parser.add_argument("--buffer_size", type=int, default=200000)
```

```
parser.add_argument("--tau", type=float, default=0.005)
parser.add_argument("--gamma", type=float, default=0.99)
parser.add_argument("--ent_coef", type=float, default=-1.0)
parser.add_argument("--net_arch", type=str, default="256,256")
```

Moreover, the processing of these hyperparameters is provided below. Many of these values can be inserted into the models in their raw state, but a small amount of preprocessing is required. Note that there is a unique log directory created for this configuration of hyperparameters. Also, ignore the "args.reward" variable, as this is another inputted variable representative of the reward function and will be explained in a later section.

```
# Define the entropy coefficient and network architecture variables
args = parser.parse_args()
ent_coef_param = 'auto' if args.ent_coef == -1.0 else args.ent_coef
net_arch_param = list(map(int, args.net_arch.split(",")))

# Create a unique log directory for this configuration
log_dir = os.path.join(
    args.output_dir,
    f"model-{args.model}_chunk-{args.chunk_size}_lr-{args.learning_rate}_batch-{args.batch_size}_buffer-{args.buffer_size}_tau-{args.tau}_gamma-{args.gamma}_ent-{ent_coef_param}_arch-{'-'.join(map(str, net_arch_param))}_reward-{args.reward}"
)
os.makedirs(log_dir, exist_ok=True)

# Assign the network architecture variable in policy_kwargs
policy_kwargs = dict(net_arch=net_arch_param, activation_fn=nn.ReLU)
```

## Episode Lengths

Originally, the data was split into episodes of 100 steps. This is determined by the chunk size variable. Therefore, a range of episode lengths can be tested by simply passing in a new chunk size through the terminal ("—chunk_size <number here>"). For this experiment, the effect of the episode length was tested using the SAC model and default hyperparameters. In theory, shorter episodes should provide more frequent updates but overall higher variance, where longer episodes should improve stability but require more training time. The range of chunk sizes is provided below.

- Chunk sizes: 1, 10, 20, 50, **100**, 200, 600

## Reward Structures

The reward function operates by favoring larger values. For this assignment, the default reward was calculated as the negative of the absolute error between the current and reference speeds. Therefore, the smaller the difference, the larger the negative equivalent of the value. Specifically, it follows the equation below, given that $s_r$ is the reference speed $s_c$ is the current speed, and $R$ is the reward.

$$R = -|s_c - s_r|$$

To determine the effect of this reward structure, I introduced four new reward functions. These reward functions are tested against the SAC model and default parameters (including chunk size of 100). This section provides a definition of each function and the justification of its selection.

First, a squared error function can be used to penalize large deviations in speed differences. The default error uses a linear difference, so this reward function analyzes the effect of increasing the magnitude of larger differences. Theoretically, this should force the model to minimize significant speed differences. The reward function is provided below.

$$R = -(s_c - s_r)^2$$

Second, an exponential error equation is tested. This harshly penalizes errors of any kind to force the model to be as precise as possible. However, it may make the learning process unstable. The reward function is provided below.

$$R = -\exp(s_c - s_r)$$

Third, a thresholded absolute error is used. This provides a small tolerance for minor errors to encourage the model to continue finding near-perfect speeds while discouraging larger deviations. The goal is to reduce the exploration of the policy after finding a close enough speed to the reference value. The equation is provided below. In this case, the threshold is selected to be an absolute difference of 0.5.

$$R = \begin{cases} 1.0 & if\ |s_c - s_r| < 0.5 \\ -|s_c - s_r| & if\ |s_c - s_r| \geq 0.5 \end{cases}$$

Fourth, a cubed error equation is tested. This should strongly discourage large errors while magnifying the difference between smaller errors, which can potentially lead to smoother corrections. This is a glorified version of the squared error equation. The reward function is provided below.

$$R = -|s_c - s_r|^3$$

The code implementation for each reward function is provided below. Note that the equation is selected depending on an input flag "ERROR_SELECTION."

```
# 1: Absolute error
if ERROR_SELECTION == 0:
    error = abs(self.current_speed - self.ref_speed)
    reward = -error

# 2: Squared error
elif ERROR_SELECTION == 1:
    error = self.current_speed - self.ref_speed
    reward = -error**2

# 3: Exponential error
elif ERROR_SELECTION == 2:
    error = abs(self.current_speed - self.ref_speed)
    reward = -np.exp(error)

# 4. Thresholded absolute error
elif ERROR_SELECTION == 3:
    error = abs(self.current_speed - self.ref_speed)
    reward = 1.0 if error < 0.5 else -error

# 5. Cubed error
elif ERROR_SELECTION == 4:
```

```
    error = abs(self.current_speed - self.ref_speed)
    reward = -error**3
```

## Training Procedure

The RL model for speed following was trained using a batch script ($runall.bat$) that automates the execution of many different training runs with different hyperparameter configurations. The training process involves defining a reference speed profile, training an RL agent to follow it, and evaluating its performance using one of the provided reward functions and model architectures. For consistency, once one reference speed profile is created, it is stored locally and reused for all subsequent model trainings. This is done through the code below.

```
# Force-generate a 1200-step sinusoidal + noise speed profile
if not os.path.exists(CSV_FILE):
    speeds = 10 + 5 * np.sin(0.02 * np.arange(DATA_LEN)) + 2 * np.random.randn(DATA_LEN)
    df_fake = pd.DataFrame({"speed": speeds})
    df_fake.to_csv(CSV_FILE, index=False)
    print(f"Created {CSV_FILE} with {DATA_LEN} steps.")
else:
    print(f"{CSV_FILE} already exists. Skipping creation.")
```

Each line in $runall.bat$ executes $RL\_assignment.py$ with a unique combination of parameters that affect model learning. The required terminal flags are listed below.

- **--model [int]:** Maps the integer range [0,3] to the respective models $\{SAC, PPO, TD3, DDPG\}$.
- **--chunk_size [int]:** Defines the episode length for training.
- **--learning_rate [float]:** Defines the scale of parameter updating for the model.
- **--batch_size [int]:** Controls the number of training samples used per update.
- **--buffer_size [int]:** Defines the memory capacity for experience replay.
- **--tau [float]:** Influences how smoothly the target network updates.
- **--gamma [float]:** Sets the discount factor for future rewards.
- **--ent_coef [float]:** Manages the balance between exploration and exploitation.
    - A value of -1.0 maps to ‘auto’ – a dynamic variable controlled by model confidence.
- **--net_arch [str]:** Specifies the neural network size for function approximation.
    - Provided as a string of 2 comma-separated numbers (ex. “64,64”).
- **--reward [int]:** Maps the integer range [0,4] to the respective reward functions {Absolute error, Squared error, Exponential error, Thresholded absolute error, Cubed error}.

The batch script $runall.bat$ systematically varies these hyperparameters to analyze their effects on model performance. These results will be explored in the **Results** section

## Running the Training Script

To execute the training script, perform the following steps.

1. Open a command prompt and navigate to the project directory (containing both $RL\_assignment.py$ and $runall.bat$).
2. Type the following command: $./\boldsymbol{runall.bat}$

# Performance Metrics

This section outlines the key performance metrics used to evaluate the speed-following RL model. These metrics quantify the model's ability to track reference speeds accurately and efficiently. By analyzing these measures, we can assess the model performance, compare different approaches, and optimize a final training strategy.

## Key Performance Metrics

### 1. Mean Absolute Error (MAE)

MAE measures the average magnitude of the errors between predicted speed and reference speed, regardless of direction. It is defined by the equation below, where $s_c$ is the current speed, $s_r$ is the reference speed, and $n$ is the number of steps taken.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|(s_r)_i - (s_c)_i|$$

MAE provides an intuitive measure of error, making it useful for evaluating how well the model follows the reference speed. A lower MAE value indicates better performance. It is particularly useful when small deviations are equally significant.

### 2. Mean Squared Error (MSE)

MSE calculates the average squared differences between predicted and reference speeds. This penalizes larger errors more significantly. The formula is shown below.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}((s_r)_i - (s_c)_i)^2$$

MSE is particularly useful when larger errors should be more heavily penalized, as applying the square amplifies the impact of significant deviations. Lower MSE values indicate better performance, but due to squaring, it may exaggerate the effect of outliers.

### 3. Root Mean Squared Error (RMSE)

RMSE is the square root of the MSE, providing an error measure in the same unit as the predicted and reference speeds. The formula is shown below.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}((s_r)_i - (s_c)_i)^2}$$

RMSE is useful for interpreting errors on the same scale as the original data, making it more intuitive for understanding deviations. Lower RMSE values indicate better performance, similar to MSE, but without the exaggerated effect of squaring large errors.

### 4. $R^2$ Score

The $R^2$ score indicates how well the predicted speeds fit the reference speeds in terms of distribution. The formula is shown below, where $\widehat{s_r}$ is the mean of the reference speeds.

$$R^2 = 1 - \frac{\sum\{(s_r)_i - (s_c)_i\}^2}{\sum\{(s_r)_i - (\widehat{s_r})_i\}^2}$$

$R^2$ measures the proportion of variance explained by the model. A higher value suggests a better fit. Specifically, the $R^2$ score falls into these ranges:

- $R^2 = 1$: Perfect prediction.
- $R^2 = 0$: Model performs as poorly as a constant mean predictor.
- $R^2 < 0$: Model performs worse than a constant mean predictor.

**5. Average Reward**

The average reward measures the RL agent's performance by averaging the cumulative reward over all time steps. The formula is shown below, where $R_i$ is the reward at time step $i$.

$$Average\ Reward = \frac{1}{n}\sum_{i=1}^{n} R_i$$

The higher an average reward value is, the better the model performance is at achieving the task efficiently. Specifically, a higher reward value implies that the model successfully follows the reference speed while optimizing its policy.

## Visuals and Tabulations

To better compare the performance of different model configurations, the results will present graphical and tabular representations of these metrics. These files will be in the log directory after final execution of each model configuration. There are three structures that will be used for comparison:

**1. Performance Summary Table**

A performance summary table is provided for every result. It provides the final values for each performance metric defined above and can allow for direct comparison of the final policies of each model. An example table is provided below. These values are located at the bottom of $test\_results.txt$.

---

[TEST] Average reward over 1200-step test: -19.514
[TEST] Mean Absolute Error (MAE): 19.514
[TEST] Mean Squared Error (MSE): 404.408
[TEST] Root Mean Squared Error (RMSE): 20.110
[TEST] R2 Score (R2): -23.595

---

**2. Predicted Speed vs. Reference Speed Visualization**

A visualization of the reference speed versus predicted speed for all time steps is provided. It helps visualize when and where models deviate the most in terms of speed from the reference profile. The X-axis represents the time steps, and the Y-axis represents speed. The blue dotted line will be the reference speed, and the orange solid line will be the predicted speed. After a successful program run, this will be saved as $test\_plot\_chunk\_\#.png$.

**3. Training Convergence Visualization**

A visualization of the training convergence over time is provided here. It helps visualize areas during the model training where it struggles to determine the reference speed, highlighting problematic

areas and training behavior. The X-axis represents the time steps, and the Y-axis represents the average reward. Note that the reward is determined by the selection of the reward function, which may explain larger deviations in rewards. After a successful program run, this will be saved as $training\_convergence.png$.

Joshua Glaspey
Spring 2025
Reinforcement Learning Exercise

# Results

This section presents the results of training and evaluating the RL models on the speed-following task. Performance is assessed using the key metrics, including MAE, MSE, RMSE, $R^2$ Score, and Average Reward. Additionally, the best configuration for each hyperparameter's training progress and speed-tracking accuracy will be visualized to provide an insight into model behavior.

The following sections will explore every configurable variable's contribution to model learning and present a comparison of all results.

## Learning Rate Analysis

The results of using various learning rates are shown below. Table 1 compares the key metrics for each model. Note that, for SAC, using learning rates of 1e-1 or 3e-2 threw exceptions, signifying its sensitivity to high learning rates.

**Table 1:** Testing various learning rates against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| Model | Metric | Learning Rate | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **1e-5** | **3e-5** | **1e-4** | **3e-4** | **1e-3** | **3e-3** | **1e-2** | **3e-2** | **1e-1** |
| **SAC** | **Avg. Reward** | -2.721 | -1.972 | -2.186 | -1.958 | -2.260 | -2.118 | -1.904 | N/A | N/A |
| | **MAE** | 2.721 | 1.972 | 2.186 | 1.958 | 2.260 | 2.118 | 1.904 | N/A | N/A |
| | **MSE** | 11.274 | 6.127 | 7.625 | 6.010 | 7.927 | 7.062 | 5.661 | N/A | N/A |
| | **RMSE** | 3.358 | 2.475 | 2.761 | 2.452 | 2.816 | 2.658 | 2.379 | N/A | N/A |
| | **$R^2$** | 0.314 | 0.62 | 0.536 | 0.634 | 0.518 | 0.570 | 0.656 | N/A | N/A |
| **PPO** | **Avg. Reward** | -1.906 | -1.766 | -1.756 | -1.754 | -1.718 | -1.783 | **-1.712** | -1787.1 | -1791.42 |
| | **MAE** | 1.906 | 1.766 | 1.756 | 1.754 | 1.718 | 1.783 | **1.712** | 1787.1 | 1791.42 |
| | **MSE** | 5.698 | 4.831 | 4.743 | 4.743 | 4.549 | 4.877 | **4.523** | 4274623 | 4290175 |
| | **RMSE** | 2.387 | 2.198 | 2.178 | 2.178 | 2.133 | 2.208 | **2.127** | 2067.52 | 2071.27 |
| | **$R^2$** | 0.653 | 0.706 | 0.712 | 0.712 | 0.723 | 0.703 | **0.725** | -259973 | -260919 |
| **TD3** | **Avg. Reward** | -2.790 | -2.571 | -2.985 | -2.244 | -3.450 | -1791.42 | -1791.42 | -1791.42 | -1791.42 |
| | **MAE** | 2.790 | 2.571 | 2.985 | 2.244 | 3.450 | 1791.42 | 1791.42 | 1791.42 | 1791.42 |
| | **MSE** | 11.954 | 10.177 | 13.265 | 7.747 | 17.465 | 4290175 | 4290175 | 4290175 | 4290175 |
| | **RMSE** | 3.458 | 3.190 | 3.642 | 2.783 | 4.179 | 2071.27 | 2071.27 | 2071.27 | 2071.27 |
| | **$R^2$** | 0.273 | 0.381 | 0.193 | 0.529 | -0.062 | -260919 | -260919 | -260919 | -260919 |
| **DDPG** | **Avg. Reward** | -9.011 | -2.758 | -2.274 | -2.370 | -2.426 | -2.040 | -1791.42 | -10.097 | -10.097 |
| | **MAE** | 9.011 | 2.758 | 2.274 | 2.370 | 2.426 | 2.040 | 1791.42 | 10.097 | 10.097 |
| | **MSE** | 104.38 | 11.104 | 8.060 | 8.780 | 9.127 | 6.565 | 4290175 | 118.400 | 118.400 |
| | **RMSE** | 10.216 | 3.332 | 2.839 | 2.963 | 3.021 | 2.562 | 2071.27 | 10.881 | 10.881 |
| | **$R^2$** | -5.348 | 0.325 | 0.510 | 0.466 | 0.445 | 0.601 | -260919 | -6.201 | -6.201 |

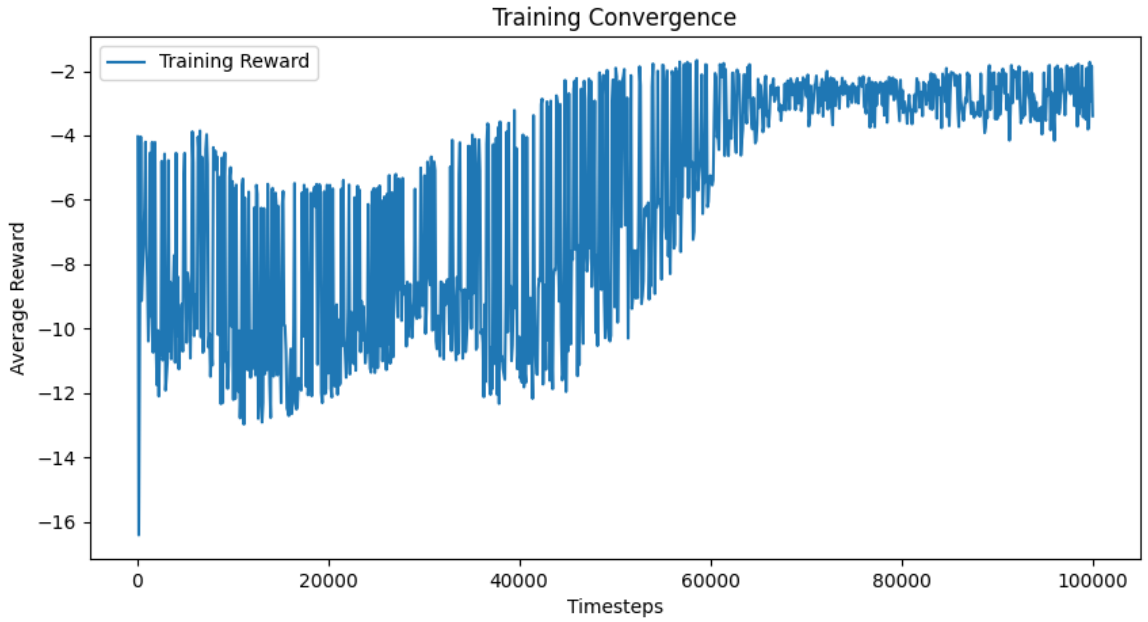
### Analysis

The learning rate significantly impacts the stability and convergence of reinforcement learning models. A lower learning rate, such as 1e-5 or 3e-5, resulted in slow but stable learning. These rates often led to suboptimal performance as the model failed to update weights effectively, resulting in higher MAE

and MSE values. The slow weight updates prevented the agent from adapting quickly, leading to prolonged training times and suboptimal final policies.

A demonstration of the slow improvement of low learning rates is shown below. Figure 1 shows the convergence of the average reward for a SAC model with a learning rate of 1e-5 across all the timesteps. As visualized, it takes around 60,000 time steps to converge to a smaller reward density, with high fluctuations while the timesteps were fewer than 60,000.

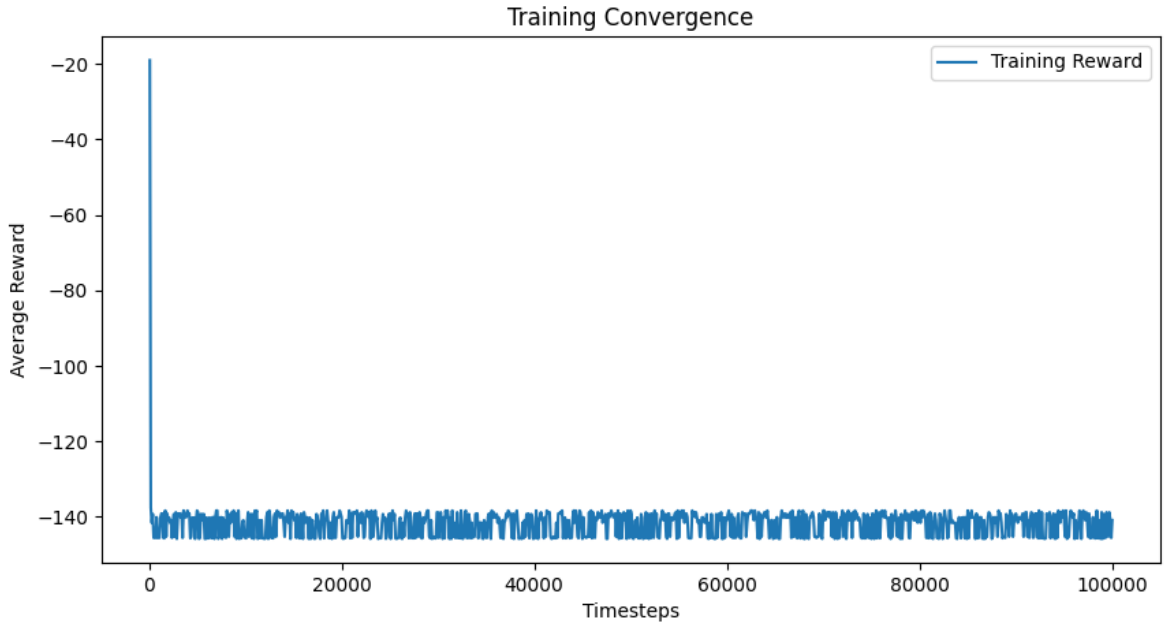
**Figure 1**: The training convergence of a SAC model using a learning rate of 1e-5.




Conversely, extremely high learning rates, such as 1e-1 or 3e-2, led to instability, as shown by erratic training curves and high variance in rewards. Large updates to policy parameters often caused the model to diverge instead of converging, leading to poor generalization. The high variance in error metrics across training runs further supports this instability.

Moreover, a demonstration of the divergent behavior of high learning rates is shown below. Figure 2 shows the divergence of the average reward for a TD3 model with a learning rate of 1e-1. It rapidly grows, on average, to a value of 140, and continues to remain high for the remainder of the training despite the large learning rate.

**Figure 2**: The training convergence of a TD3 model using a learning rate of 0.1.




Specifically, for each model, the exact learning rate can be shown for focusing too largely on exploration and not converging to an optimal policy: for SAC, it is 1e-2; for PPO, it is 3e-2; for TD3, it is 3e-4; for DDPG, it is 3e-3.

First, SAC experienced high sensitivity to high learning rates. SAC encountered exceptions at learning rates greater than or equal to 3e-2, indicating instability at these high values. The best performance was at 1e-2, yielding the highest $R^2$ (0.656) and lowest errors ($MAE = 1.904, MSE = 5.661$). At very low learning rates, SAC performed poorly, suggesting that a small step size slows down convergence significantly.

Second, PPO remained stable across a wide range of learning rates but catastrophically failed at 3e-2 and larger. The best PPO performance was observed at 1e-2, with the highest $R^2$ score (0.725) suggesting the policy explains nearly 3/4ths of the behavior of the input. Similarly, this learning rate saw the lowest errors ($MAE = 1.712, MSE = 4.523$), showcasing the minimized discrepancies between the reference and predicted speeds. PPO's performance degraded slightly at 3e-3, but remained within a reasonable range, confirming its robustness.

Third, TD3 performed poorly at learning rates above 1e-3, with even more extreme values at 3e-3 and beyond. The best TD3 performance was at 3e-4, with the lowest MSE (7.747), lowest MAE (2.244), and highest $R^2$ (0.529). However, this performance is weak in consideration to the other models, suggesting this is not an ideal architecture to explore beyond an ideal learning rate. Additionally, the poor performance at a learning rate of 1e-3 suggests that TD3 is very sensitive to excessive step sizes.

Fourth, DDPG also exhibited poor performance across most learning rates, particularly at high values (1e-2 and beyond). Its best performance was at 3e-3, where it achieved the lowest MAE (2.040) and highest $R^2$ (0.601). However, these values are only marginally better than TD3's metrics, and do not compare to the performance of PPO. Moreover, this policy was optimized using a high exploration factor,

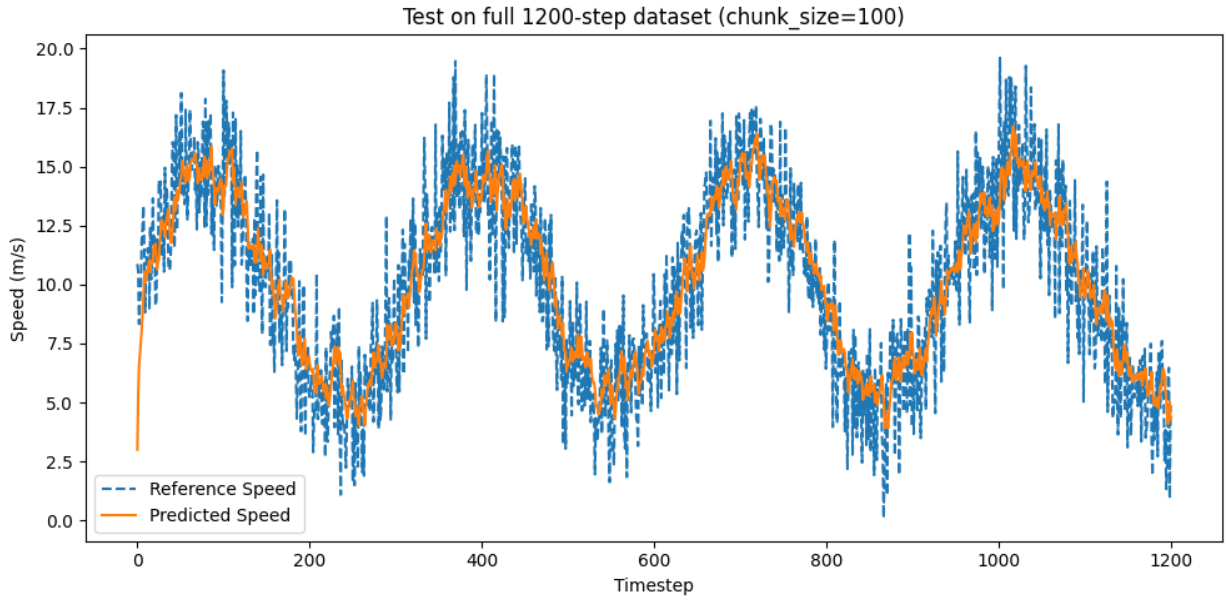
as raising the learning rate any higher (1e-2) causes for a divergence in reward. Moreover, using a low learning rate (1e-5) yielded awful performance as well (-9.011 average reward).

PPO at 1e-2 yielded the best results of the entire table, but using a learning rate of 1e-3 yielded the best performance in general across all four models. Not to mention, the performance scores of PPO at a learning rate of 1e-3 ($R^2 = 0.723$) are insignificantly worse than 1e-2. Figure 3 compares the reward divergence for PPO using a learning rate of 1e-3 versus 1e-2. Looking at the visualization, using a learning rate of 1e-2 fluctuates much more rapidly than 1e-2. In early time steps, the reward skyrockets to worse values, and in later time steps, it oscillates at higher rates than 1e-3. Therefore, for generalizability, using a learning rate of 1e-3 is ideal for this RL assignment.

Moreover, Figure 4 shows a visualization for the predicted speed of PPO using a learning rate of 1e-3 against the reference speed for all 1200 timesteps during testing. Generally, the predicted speed follows the overall trend of the reference speed reasonably well. This indicates that the model has successfully learned the primary structure of the speed variations, which is supported by its high $R^2$ score of 0.723. However, the predicted data appears to be smoother than the reference speed, suggesting that the PPO model does not capture all high frequency fluctuations in the reference data. In other words, the PPO model is over-regularizing to capture to pattern of the sinusoidal wave over all noisy data points in the reference data. On the positive side, the predicted speed does not exhibit extreme deviations, which suggests the PPO learning was stable.

**Figure 3:** Reward convergence over time for using a PPO model with learning rates of 1e-3 and 1e-2.

**Figure 4:** Speed comparison for testing PPO with a learning rate of 1e-3 against the reference speeds.




## Batch Size Analysis

The results of using various batch sizes are shown below. Table 2 compares the key metrics for each model.

**Table 2:** Testing various batch sizes against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| Model | Metric | Batch Size | | | | |
|---|---|---|---|---|---|---|
| | | **32** | **64** | **128** | **256** | **512** |
| **SAC** | **Avg. Reward** | -2.149 | -1.882 | -2.149 | -1.958 | -2.301 |
| | **MAE** | 2.149 | 1.882 | 2.149 | 1.958 | 2.301 |
| | **MSE** | 7.348 | 5.547 | 7.272 | 6.010 | 8.433 |
| | **RMSE** | 2.711 | 2.355 | 2.697 | 2.452 | 2.904 |
| | $R^2$ | 0.553 | 0.663 | 0.558 | 0.634 | 0.487 |
| **PPO** | **Avg. Reward** | -1.754 | -1.828 | **-1.751** | -1.754 | -1.921 |
| | **MAE** | 1.754 | 1.828 | **1.751** | 1.754 | 1.921 |
| | **MSE** | 4.785 | 5.205 | 4.778 | **4.743** | 5.673 |
| | **RMSE** | 2.187 | 2.281 | 2.186 | **2.178** | 2.382 |
| | $R^2$ | 0.709 | 0.683 | 0.709 | **0.712** | 0.655 |
| **TD3** | **Avg. Reward** | -3.071 | -2.232 | -1.971 | -2.244 | -2.135 |
| | **MAE** | 3.071 | 2.232 | 1.971 | 2.244 | 2.135 |
| | **MSE** | 13.717 | 7.991 | 6.100 | 7.747 | 7.091 |
| | **RMSE** | 3.704 | 2.827 | 2.470 | 2.783 | 2.663 |
| | $R^2$ | 0.166 | 0.514 | 0.629 | 0.529 | 0.569 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **DDPG** | **Avg. Reward** | -2.092 | -2.045 | -5.674 | -2.370 | -2.830 |
| | **MAE** | 2.092 | 2.045 | 5.674 | 2.370 | 2.830 |
| | **MSE** | 6.779 | 6.546 | 46.216 | 8.780 | 12.368 |
| | **RMSE** | 2.604 | 2.558 | 6.798 | 2.963 | 3.517 |
| | $\boldsymbol{R^2}$ | 0.588 | 0.602 | -1.811 | 0.466 | 0.248 |

**Analysis**

Batch size is directly related to the stability of updates in RL training. A higher batch size generally enables more stable updates but may lead to slower adaptability. Conversely, smaller batch sizes allow for more frequent updates but may add higher variance due to a smaller information domain.

Across all models, using moderate batch sizes (64, 128, 256) often produced the best results for all metrics. The extremes (32, 512) tended to degrade performance, either due to instability in updates or insufficient adaptation to dynamic changes. Using a batch size of 128 and a PPO model saw the best results from Table 2.

For SAC, using a batch size of 64 was ideal. The average reward improved to -1.882, with a MSE of 5.547 and $R^2$ of 0.663 being all the best performers. However, a batch size of 512 performed the worst. Likely, SAC benefits from moderate batch sizes because it stabilizes updates while maintaining adaptability. Using a large batch size likely dampens the stochasticity too much and renders it difficult for the model to adjust.

For TD3, the best batch size was 128. It had the lowest MSE (6.100) and RMSE (2.470), signifying that it showed minimal prediction error. Similarly, it had the highest $R^2$, signifying the best model fitting. Using too small of a batch size (32) introduced high variance and instability, where larger batch sizes (256, 512) also poorly performed due to the dampened stochasticity. Having a moderate tradeoff between these two sizes enabled the TD3 model to balance exploration with exploitation.

For DDPG, using a batch size of 64 was the ideal configuration. It had the lowest MSE/RMSE, as well as the highest $R^2$, which signifies minimal error and better fitness than the other configurations. Using a batch size of 128 showed a catastrophic failure, with a MSE of 46.216 and a $R^2$ of -1.811. This suggests a collapse of the model. The model performance drop at batch sizes of 256 and 512 second this point.

For PPO, using a batch size of 256 is ideal, but a batch size of 128 showed insignificant loss. A batch size of 256 performed the best in MSE (4.743), indicating the best model error reduction. It also performed the best with $R^2$ (0.712), suggesting an ideal model fit. However, using a batch size of 128 showed the best average reward, suggesting its policy was optimized to fit the reference speeds greater. Additionally, its difference in RMSE and $R^2$ is minimal ($\pm 0.08$), suggesting a near equivalent fit. Overall, PPO performs the best with moderately large batch sizes (128, 256) because it benefits from a more stable gradient estimation. Unlike SAC, PPO can leverage this higher batch size effectively due to it having a clipping mechanism which prevents excessive updates.

Overall, PPO was the best performing model for every hyperparameter scale in every metric. Therefore, PPO is the best performing model, with its ideal result being a batch size of 256. A comparison of the convergence rates in all four models using a batch size of 128 is shown in Figure 5, and a comparison of the convergence rates using a batch size of 256 is shown in Figure 6. Hese two batch sizes were selected due to their generalized better performance across all metrics for all models (besides

DDPG). Ultimately, using a batch size of 64 saw the smoothest convergence for all four models, and is the ideal batch size.

**Figure 5:** Reward convergence over time for all four models and a batch size of 128.

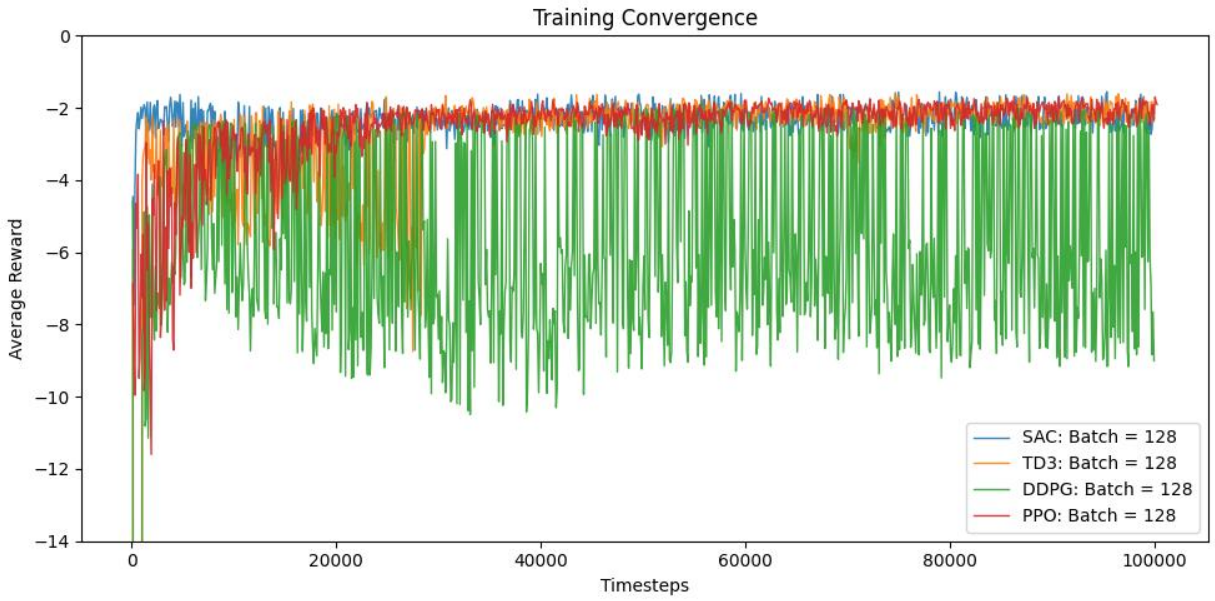



**Figure 6:** Reward convergence over time for all four models and a batch size of 256.

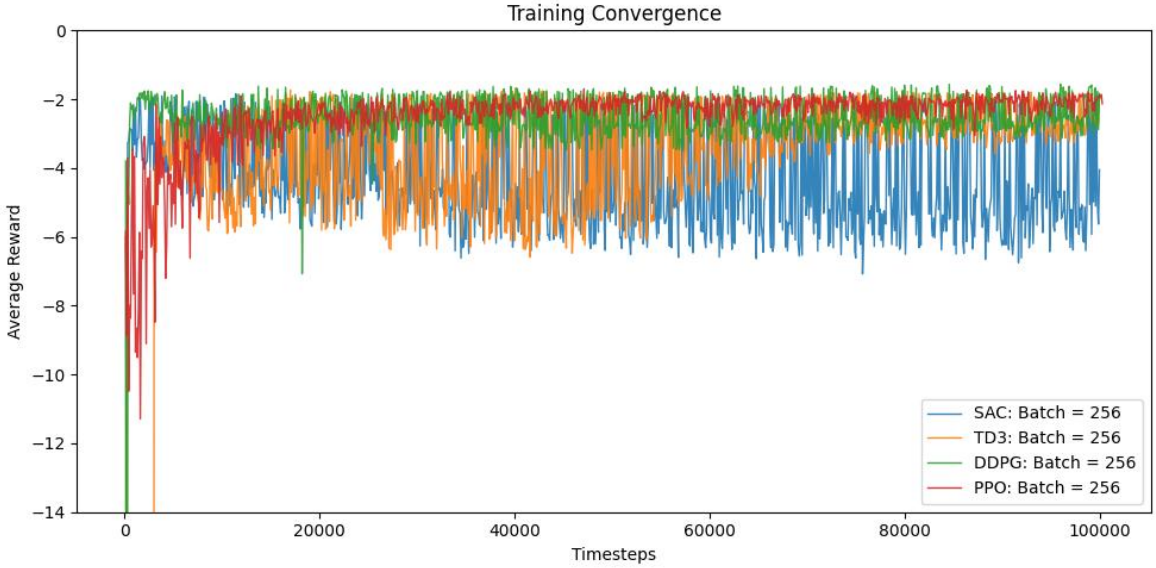
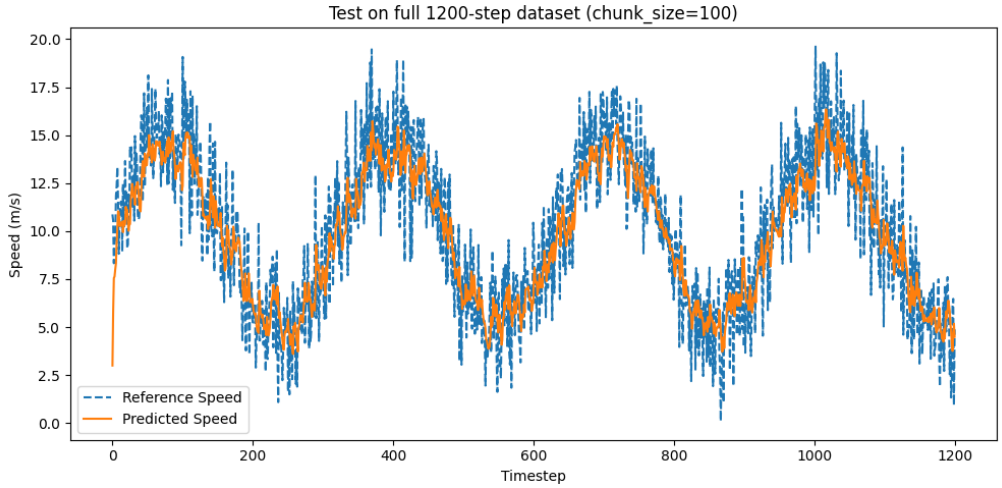



Figure 7 shows the predicted speeds of using a PPO model with a batch size of 256 against the reference speeds. The predicted speed closely follows the reference speed, capturing the general shape and periodic fluctuations. It captures the sinusoidal relationship largely. However, it struggles with perfect alignment with the high-frequency fluctuations. There is visible lag where the predicted speed underestimates or overestimates the peaks, suggesting the model isn't adapting fast enough to sudden speed variations. This could indicate that the batch size of 256 could be dampening the stochasticity of the

data, making the results too largely generalized to the sinusoidal shape of the reference data. Lowering the batch size could help with this problem but lowering it too far would introduce variability into the results. This was apparent in the results for a batch size of 128 showing insignificantly better results in the average reward, suggesting its policy may follow the reference speed a little better but suffer from higher errors at other instances in time.

**Figure 7:** Speed comparison for testing PPO with a batch size of 256 against the reference speeds.




## Buffer Size Analysis

The results of using various buffer sizes are shown below. Table 3 compares the key metrics for each model.

**Table 3:** Testing various buffer sizes against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| Model | Metric | Buffer Size | | | | |
|---|---|---|---|---|---|---|
| | | **50,000** | **100,000** | **200,000** | **500,000** | **1,000,000** |
| **SAC** | **Avg. Reward** | -2.049 | **-1.879** | -1.958 | -2.031 | -2.129 |
| | **MAE** | 2.049 | **1.879** | 1.958 | 2.031 | 2.129 |
| | **MSE** | 6.601 | **5.550** | 6.010 | 6.513 | 7.191 |
| | **RMSE** | 2.569 | **2.356** | 2.452 | 2.552 | 2.682 |
| | $R^2$ | 0.599 | **0.662** | 0.634 | 0.604 | 0.563 |
| **TD3** | **Avg. Reward** | -2.380 | -2.529 | -2.244 | -4.377 | -2.126 |
| | **MAE** | 2.380 | 2.529 | 2.244 | 4.377 | 2.126 |
| | **MSE** | 8.784 | 9.954 | 7.747 | 27.945 | 7.051 |
| | **RMSE** | 2.964 | 3.155 | 2.783 | 5.286 | 2.655 |
| | $R^2$ | 0.466 | 0.395 | 0.529 | -0.700 | 0.571 |
| **DDPG** | **Avg. Reward** | -1.941 | -4.468 | -2.370 | -2.581 | -2.121 |
| | **MAE** | 1.941 | 4.468 | 2.370 | 2.581 | 2.121 |
| | **MSE** | 5.872 | 29.354 | 8.780 | 10.407 | 7.061 |
| | **RMSE** | 2.423 | 5.418 | 2.963 | 3.226 | 2.657 |
| | $R^2$ | 0.643 | -0.785 | 0.466 | 0.367 | 0.571 |

**Analysis**

Buffer size is crucial in experience replay-based RL algorithms. The results in Table 3 demonstrate how modifying the buffer size can cause significant jumps in performance. Given the models, buffer size is not available as a hyperparameter in PPO.

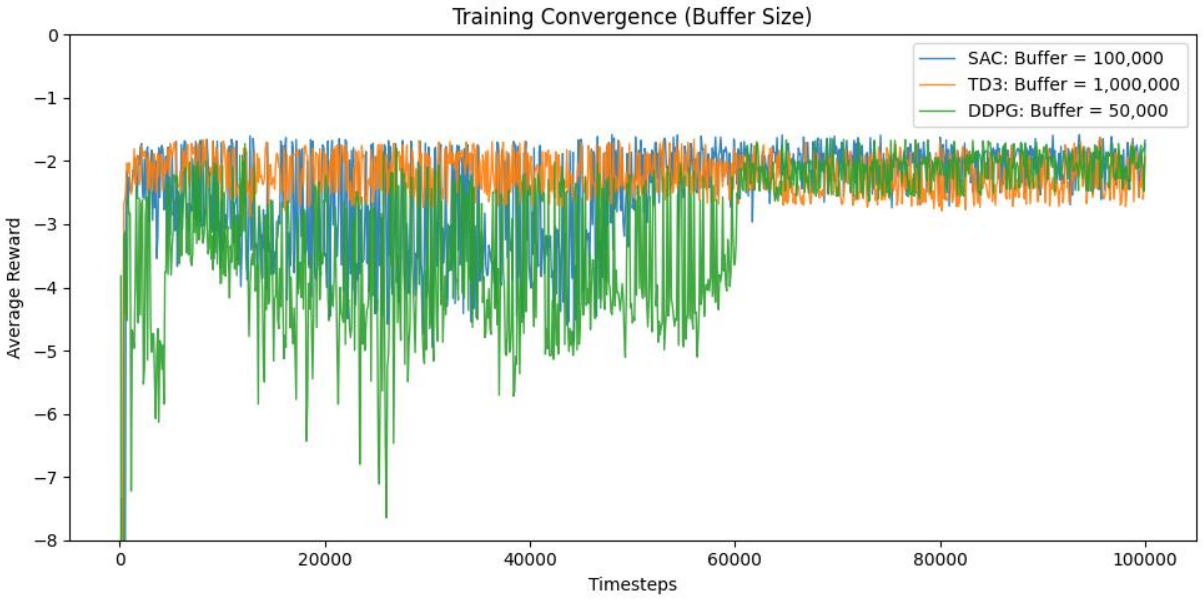
For SAC, the best average reward is observed at buffer size 100,000. Not to mention, these are the best results of the entire table. An intermediate buffer size is ideal for SAC, where increasing the buffer size beyond this point graduate worsens every metric. Also, the lowest error for MSE/RMSE occurs here as well, confirming this theory. Having a larger buffer can introduce outdated experiences into training, whereas having too small a buffer can forget important experiences too quickly.

For TD3, the best reward is observed at 1,000,000. Similarly, the best MSE/RMSE and $R^2$ scores are at a 1,000,000-buffer size, suggesting that for this model, having a very large buffer enables more effective learning. This is likely due to its delayed update mechanism, which requires a diverse sample set to stabilize training.

For DDPG, it is much more sensitive to high buffer sizes. The best average reward is 50,000, which is the lowest scale of buffer sizes tested. Similarly, the lowest errors (MSE/RMSE) and highest $R^2$ score occur at 50,000, confirming this fact. Interestingly, increasing the buffer size to 100,000 results in worse performance degradation than 200,000 or higher, suggesting that DDPG suffers from overfitting when too many outdated experiences are retained.

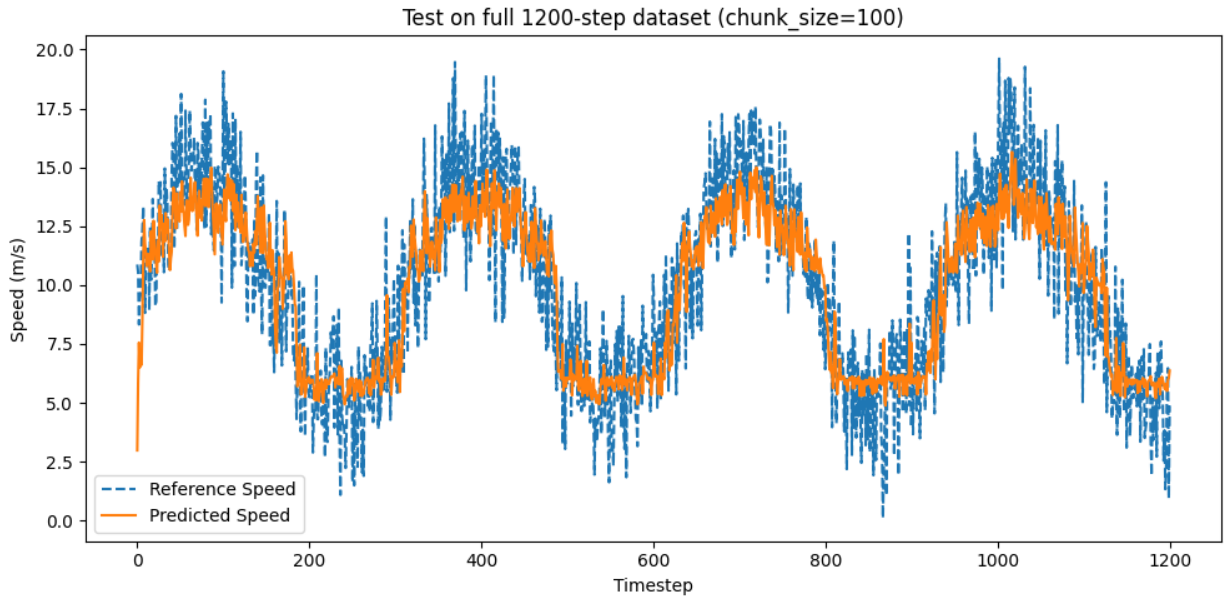
These results suggest that the buffer size should be adjusted by model instead of selecting based on the most generalizable result due to the vastly different performances from each one. Figure 8 compares the reward convergence for each model's best configuration (SAC: 100,000; TD3: 1,000,000; DDPG: 50,000). Going purely off the best metrics, SAC achieves the best model learning in every metric. However, in terms of training stability, TD3 with a buffer size of 1,000,000 maintains the best average reward without fluctuating.

**Figure 8:** Reward convergence over time. SAC uses a buffer size of 100,000. TD3 uses a buffer size of 1,000,000. DPG uses a buffer size of 50,000.



To analyze the best performing configuration, Figure 9 visualizes the predicted versus reference speed when using SAC with a buffer size of 100,000. The predicted speed closely follows the direction of the reference speed, signifying it was effective in capturing the overall trend well. However, the predicted values are smoother and fail to capture the high-frequency fluctuations in the reference speed. This suggests that the SAC model may be applying excessive smoothing, possibly due to a suboptimal configuration of other hyperparameters. This is supported by the $R^2$ value of 0.662, which indicates that the model explains a significant portion of the original function but still has room for improvements. Potentially, it could benefit from increasing the buffer size to learn more information in each replay, but that may introduce irrelevant history when grown too large in scale, as signified by the next highest buffer size (200,000).

**Figure 9:** Speed comparison for testing SAC with a buffer size of 100,000 against the reference speeds.




## Tau Analysis

The results of using various tau values are shown below. Table 4 compares the key metrics for each model.

**Table 4:** Testing various taus against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| **Model** | **Metric** | **Tau** | | | | |
|---|---|---|---|---|---|---|
| | | **0.0001** | **0.001** | **0.005** | **0.01** | **0.02** |
| **SAC** | **Avg. Reward** | -1.817 | -2.079 | -1.958 | -1.844 | -1.970 |
| | **MAE** | 1.817 | 2.079 | 1.958 | 1.844 | 1.970 |
| | **MSE** | 5.131 | 6.976 | 6.010 | 5.269 | 6.051 |
| | **RMSE** | 2.265 | 2.641 | 2.452 | 2.295 | 2.460 |
| | **$R^2$** | 0.688 | 0.576 | 0.634 | 0.680 | 0.632 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **TD3** | **Avg. Reward** | -1.920 | -2.613 | -2.244 | -2.075 | -2.111 |
| | **MAE** | 1.920 | 2.613 | 2.244 | 2.075 | 2.111 |
| | **MSE** | 5.782 | 10.582 | 7.747 | 6.741 | 6.976 |
| | **RMSE** | 2.405 | 3.253 | 2.783 | 2.596 | 2.641 |
| | $R^2$ | 0.648 | 0.356 | 0.529 | 0.590 | 0.576 |
| **DDPG** | **Avg. Reward** | **-1.793** | -2.108 | -2.370 | -2.231 | -2.408 |
| | **MAE** | **1.793** | 2.108 | 2.370 | 2.231 | 2.408 |
| | **MSE** | **5.092** | 6.975 | 8.780 | 7.742 | 9.086 |
| | **RMSE** | **2.256** | 2.641 | 2.963 | 2.782 | 3.014 |
| | $R^2$ | **0.690** | 0.576 | 0.466 | 0.529 | 0.447 |

**Analysis**

Table 4 presents the results for different tau values across each model. It governs the update rate of the target networks, balancing the stability of training with its responsiveness. Overall, this table presents a dominant performance from using a tau of 0.0001.

For SAC, the best average reward (-1.817) is obtained at a tau of 0.0001. The error rates follow a similar trend ($MSE = 5.131, R^2 = 0.688$), implying that smaller tau values improve the model's predictive power. As tau increases to 0.02, the performance slightly deteriorates ($R^2 = 0.632$), implying both that tau does not contribute as largely to performance as the previous hyperparameters, but indeed improves as it approaches lower values.

The exact same trends are apparent for both TD3 and DDPG architectures. They both experience their best average rewards (-1.920 and -1.793, respectively), as well as best error rates. Therefore, for all models, lower tau values facilitate more stable and accurate learning. Increasing tau generally results in worse performance.

A visualization of this point is shown in Figure 10, where using a SAC architecture with tau values of 0.0001, 0.001, and 0.01 are provided. Using a tau of 0.001 shows a very large initial fluctuation in error, with consistent larger oscillations than a tau of 0.0001. Similarly, using a tau of 0.01 shows very large oscillations in average reward for the first 20,000-time steps, but follows this oscillating pattern. Using a tau of 0.0001 showcases the most stable training behavior with the smallest oscillations in later stages. It is important to note that all tau configurations find a good final average reward, despite the high oscillations.

To visualize the best configuration, Figure 11 provides the predicted speeds versus the reference speeds for the DDPG architecture with a tau of 0.0001. This was the best performing configuration according to Table 4. The predicted speed follows the overall trend of the reference speed, signifying that the model has learned the underlying pattern of the data. This is supported by a high $R^2$ score of 0.690, suggesting it has learned 69% of the original data. The periodic oscillations are captured reasonably well without an extreme overfitting to the spikes. However, the high-frequency variations are smoothed over by the predictions, likely due to the impact of the very small tau. Using a smaller tau results in slower updated, which contributes to the high training stability shown in Figure 10, but provides lagging predictions. Specifically, the predicted speed starts lower than the reference speed and appears to take a couple time steps to catch up to that value. Overall, increasing tau could help the model adapt more quickly to changes in speed, but at a sacrifice in stability.

**Figure 10:** Reward convergence over time using SAC with various tau values (0.0001, 0.001, 0.01).

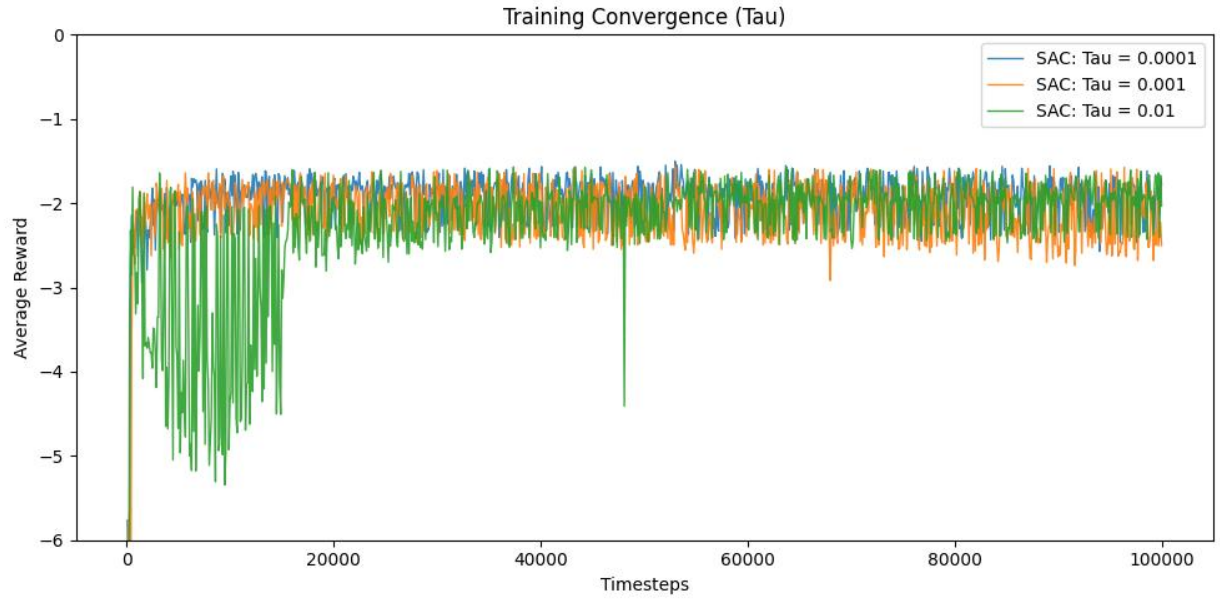
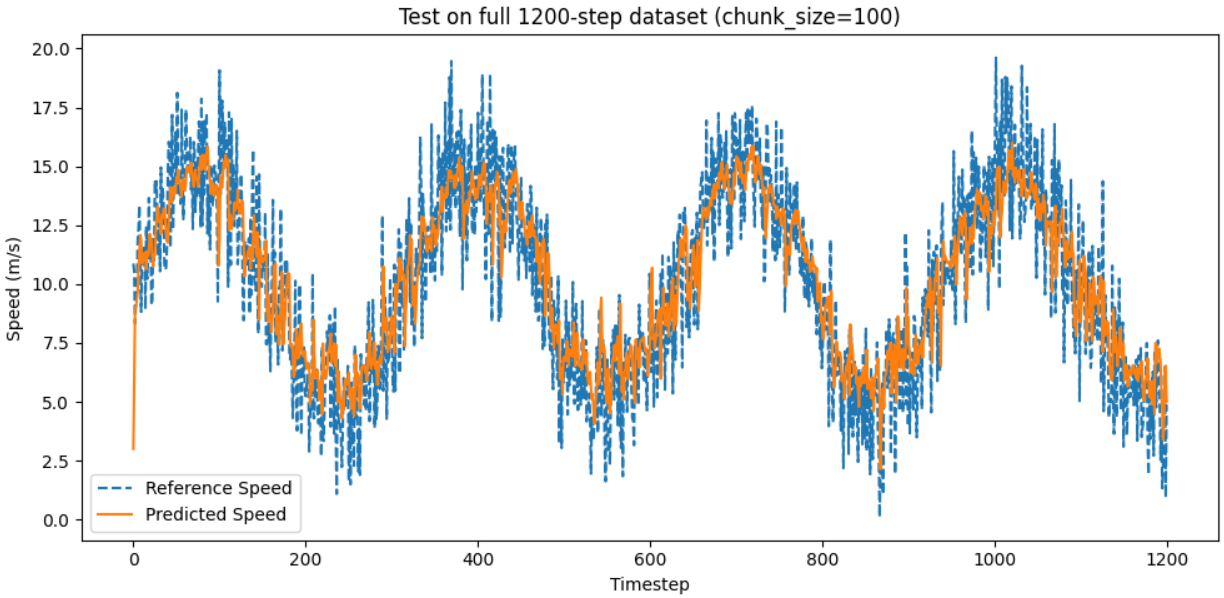



**Figure 11:** Speed comparison for testing DDPG with a tau value of 0.0001 against reference speeds.




## Gamma Analysis

The results of using various gamma values are shown below. Table 5 compares the key metrics for each model.

**Table 5:** Testing various gammas against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| Model | Metric | Gamma | | | | |
|---|---|---|---|---|---|---|
| | | **0.90** | **0.95** | **0.99** | **0.999** | **1.0** |
| **SAC** | **Avg. Reward** | -1.744 | -1.927 | -1.958 | -2.146 | -2.464 |
| | **MAE** | 1.744 | 1.927 | 1.958 | 2.146 | 2.464 |
| | **MSE** | 4.770 | 5.808 | 6.010 | 7.332 | 9.542 |
| | **RMSE** | 2.184 | 2.410 | 2.452 | 2.708 | 3.089 |
| | $R^2$ | 0.710 | 0.647 | 0.634 | 0.554 | 0.420 |
| **PPO** | **Avg. Reward** | **-1.735** | -1.744 | -1.754 | -1.781 | -1.773 |
| | **MAE** | **1.735** | 1.744 | 1.754 | 1.781 | 1.773 |
| | **MSE** | **4.656** | 4.734 | 4.743 | 4.909 | 4.830 |
| | **RMSE** | **2.158** | 2.176 | 2.178 | 2.216 | 2.198 |
| | $R^2$ | **0.717** | 0.712 | 0.712 | 0.701 | 0.706 |
| **TD3** | **Avg. Reward** | -2.260 | -3.102 | -2.244 | -2.480 | -2.374 |
| | **MAE** | 2.260 | 3.102 | 2.244 | 2.480 | 2.374 |
| | **MSE** | 7.908 | 14.395 | 7.747 | 9.359 | 9.002 |
| | **RMSE** | 2.812 | 3.794 | 2.783 | 3.059 | 3.000 |
| | $R^2$ | 0.519 | 0.124 | 0.529 | 0.431 | 0.453 |
| **DDPG** | **Avg. Reward** | -1.871 | -1.829 | -2.370 | -4.492 | -2.900 |
| | **MAE** | 1.871 | 1.829 | 2.370 | 4.492 | 2.900 |
| | **MSE** | 5.484 | 5.201 | 8.780 | 28.572 | 12.826 |
| | **RMSE** | 2.342 | 2.281 | 2.963 | 5.345 | 3.581 |
| | $R^2$ | 0.666 | 0.684 | 0.466 | -0.738 | 0.220 |

**Analysis**

Gamma determines the weight of future rewards in RL relative to immediate rewards. A lower gamma (i.e. 0.90) places more emphasis on short-term rewards, whereas higher gamma (1.0) prioritizes long-term rewards.

For SAC, as gamma generally increases, performance decreases. The best results occur at a gamma of 0.90, with the best average reward of -1.744. Similarly, the best error scores are at this gamma ($MSE = 4.770, R^2 = 0.710$), indicating that SAC benefits from shorter-term reward prioritization. At a gamma of 1.0, the model's performance is significantly worse ($R^2 = 0.420$), suggesting that it struggles to generalize effectively with long-term investment.

For PPO, the performance is stable across all different gamma configurations. The best performance is seen at 0.90, with a high average reward of -1.735 and $R^2$ of 0.717. However, increasing gamma never significantly decreases the results. For example, at a gamma of 1.0, the $R^2$ score is 0.706, which is higher than the best SAC configuration. Therefore, PPO is more robust to change in this hyperparameter than the others.

For TD3, the best result is at a gamma level of 0.99, which is a balance between high and low scales. The average reward is -2.244, with relatively strong performance scores ($MSE = 7.747, R^2 = 0.529$). However, these results are significantly worse than SAC and PPO, suggesting this is not the ideal

architecture to explore gamma configurations. Additionally, TD3 is highly sensitive to changes, as decreasing the gamma to 0.95 results in a degrade of $R^2$ to 0.124.

For DDPG, the best performance is found at a gamma value of 0.95. This is still a significantly lower value than 0.99, but is better performant than the absolute lowest tested of 0.90. Increasing the gamma beyond 0.95 leads to drastic performance reduction, where at a gamma of 0.999, the $R^2$ performance becomes negative (-0.738), indicating poor generalization. This suggests that DDPG is also sensitive to gamma adjustments. However, it's performance is only slightly worse than SAC, with an average error of -1.829 and errors of MSE at 5.201 and $R^2$ at 0.684.

Overall, lower gamma values consistently outperform higher values. High gamma values tend to introduce stability with PPO being the only exception with its generalizability across all gamma. Figure 12 visualizes the convergence rate of all models using a gamma of 0.9, while Figure 13 visualizes the convergence rate of all models using a gamma of 0.95. As shown, all models outside SAC struggle with early rewards. However, the training stabilization of a gamma with 0.90 is more than 0.95. Most notably, TD3 and DDPG struggle with higher oscillations throughout the entirety of the timesteps using a gamma of 0.95. PPO's and SAC's behavior is generally the same for both figures, but the oscillations of both models towards later stages of training is less with a gamma of 0.9 than 0.95.

**Figure 12:** Reward convergence over time using all four models and a gamma of 0.90.

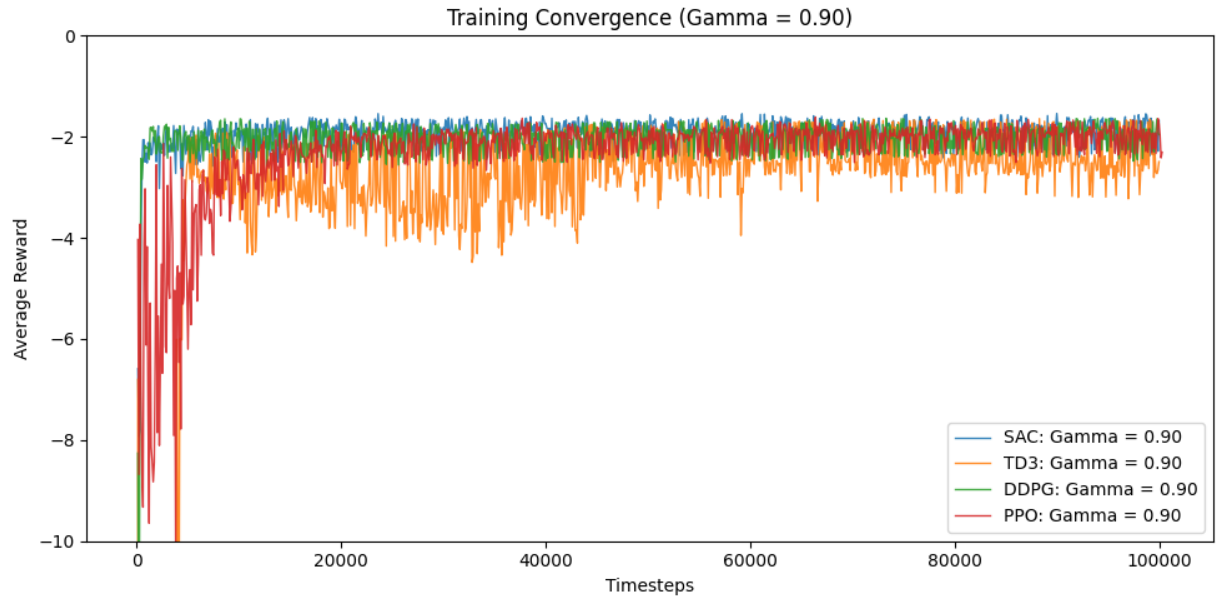


**Figure 13:** Reward convergence over time using all four models and a gamma of 0.95.

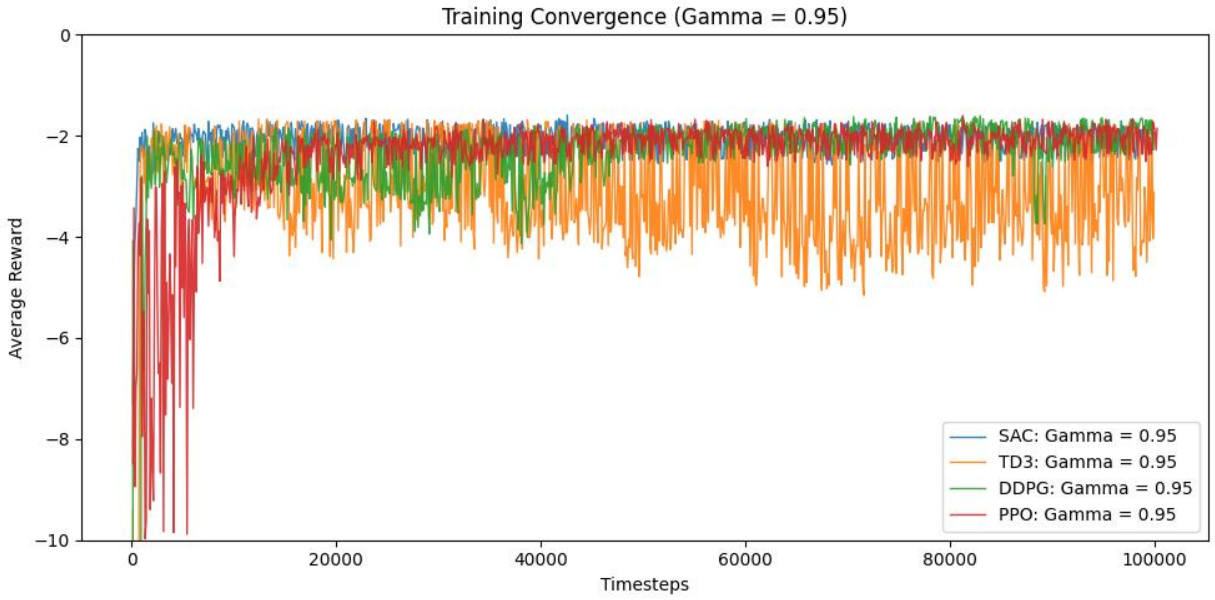
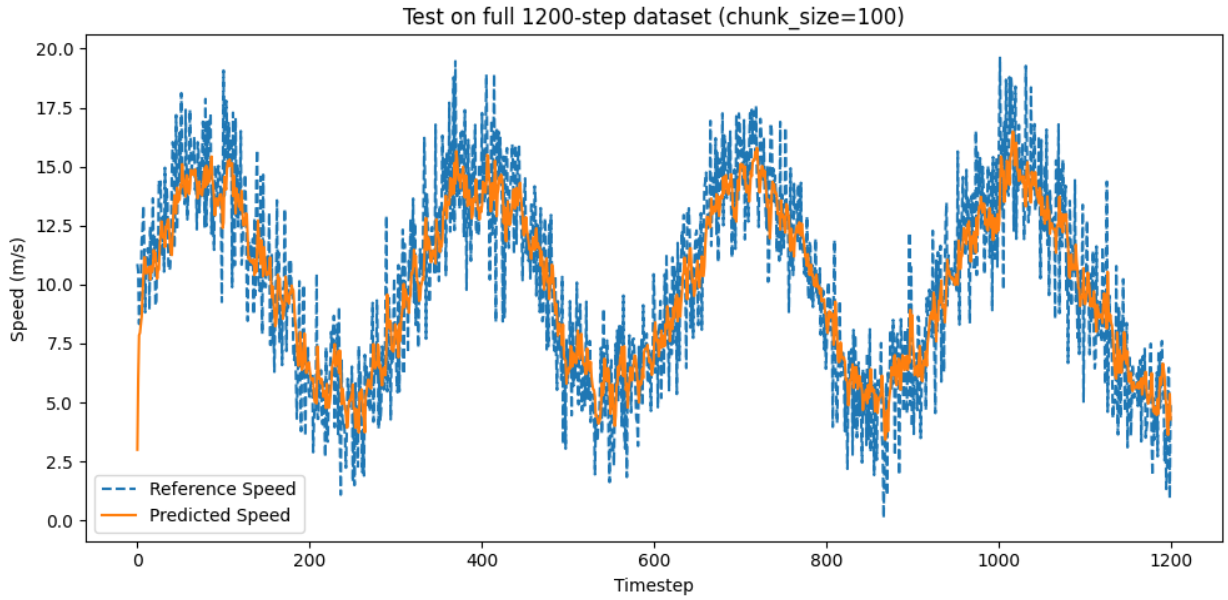



Figure 14 illustrates the performance of PPO with a gamma of 0.90 when predicting speed compared to the reference speed test dataset. The predicted speed closely follows the reference speed's overall trend, suggesting that the model captures the behavior of the reference data. This is supported by a very high $R^2$ of 0.717. Similar to the other figures, the predicted speeds are smoother than the reference data, which showcases this model's ability to generalize the form of the sinusoidal wave but not to capture high oscillations. However, the waveform does showcase fairly consistent oscillations, which is supported by a low MSE of 4.656. Additionally, the average reward of -1.735 suggests that the error from the learned policy is very small when compared to the reference speeds in this dataset. Using the lower gamma value prioritizes short-term rewards, which can lead to quicker adaptations to current data but reduced long-term consistency. The model appears to emphasize immediate speed changes, which explains the rapidly oscillating behavior of the predicted values. Increasing the gamma value might reduce deviations from the reference speed, but raising it too high would diminish the policy's optimization as demonstrated with a gamma of 0.95.

**Figure 14:** Speed comparison for testing PPO with a gamma value of 0.90 against the reference speeds.




## Entropy Coefficient Analysis

The results of using various entropy coefficients are shown below. Table 6 compares the key metrics for each model. Note that SAC uses a default entropy coefficient of 'auto' and PPO uses a default entropy coefficient of 0.0.

**Table 6:** Testing various entropy coefficients against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| **Model** | **Metric** | **Entropy Coefficient** | | | | |
|---|---|---|---|---|---|---|
| | | **auto / 0.0** | **0.01** | **0.05** | **0.1** | **0.2** |
| **SAC** | **Avg. Reward** | -1.958 | -3.616 | -2.227 | -2.111 | -2.495 |
| | **MAE** | 1.958 | 3.616 | 2.227 | 2.111 | 2.495 |
| | **MSE** | 6.010 | 19.749 | 7.823 | 6.901 | 9.982 |
| | **RMSE** | 2.452 | 4.444 | 2.797 | 2.627 | 3.159 |
| | $R^2$ | 0.634 | -0.201 | 0.524 | 0.580 | 0.393 |
| **PPO** | **Avg. Reward** | **-1.754** | -1.812 | -1.784 | -1.957 | -1.950 |
| | **MAE** | **1.754** | 1.812 | 1.784 | 1.957 | 1.950 |
| | **MSE** | **4.743** | 5.050 | 4.930 | 5.936 | 5.922 |
| | **RMSE** | **2.178** | 2.247 | 2.220 | 2.436 | 2.433 |
| | $R^2$ | **0.712** | 0.693 | 0.700 | 0.639 | 0.640 |

### Analysis

The entropy coefficient encourages exploration by preventing premature convergence to suboptimal policies. By increasing the scale of this value, the exploration is increased. By decreasing it, the exploitation to one policy is increased. For SAC, the default entropy coefficient is set to 'auto,' which

dynamically adjusts the entropy during training. For PPO, the default entropy coefficient is 0.0, meaning that there is no entropy-based exploration incentive.

First, for SAC, the default auto setting provides the best performance in terms of average reward (-1.958) and MSE (6.010) compared to the other entropy coefficients. The second-best configuration is 0.1, which achieves lower MSE (6.901) than the other fixed values. However, as a general trend, using higher entropy values (0.2) leads to a significant performance drop, as seen in the average reward of -2.495 and MSE of 9.982, which indicates excessive exploration. Overall, MSE/RMSE increase with larger entropy coefficients, and $R^2$ scores decline as entropy increases.

Second, for PPO, using the default setting of 0.0 achieves the best average reward (-1.754) and lowest MSE (4.743). Also, the $R^2$ score is very high (0.712). The 0.01 entropy coefficient slightly worsens performance, but overall, there is not an insignificant drop in performance ($R^2 = 0.693$). Increasing steadily worsens performance, but it is not a horrendous dropoff. For example, at 0.2, the $R^2$ score is 0.640. Overall, MSE/RMSE increase with entropy, which confirms that higher entropy values make it harder for PPO to converge. Similarly, $R^2$ scores decrease with entropy.

Figure 15 visualizes the convergence rates between the two default configurations for each model (SAC: auto; PPO: 0.0). Obviously, the SAC convergence is much less stable, with high oscillations around -6 during later stages of training. PPO initially experiences high oscillations, but after 10,000 timesteps, the convergence stabilizes and converges towards -2. Therefore, PPO is the optimal model configuration by large for this hyperparameter configuration.

**Figure 15:** Reward convergence over time using SAC and PPO with entropy coefficients of 'auto' and 0.0, respectively.

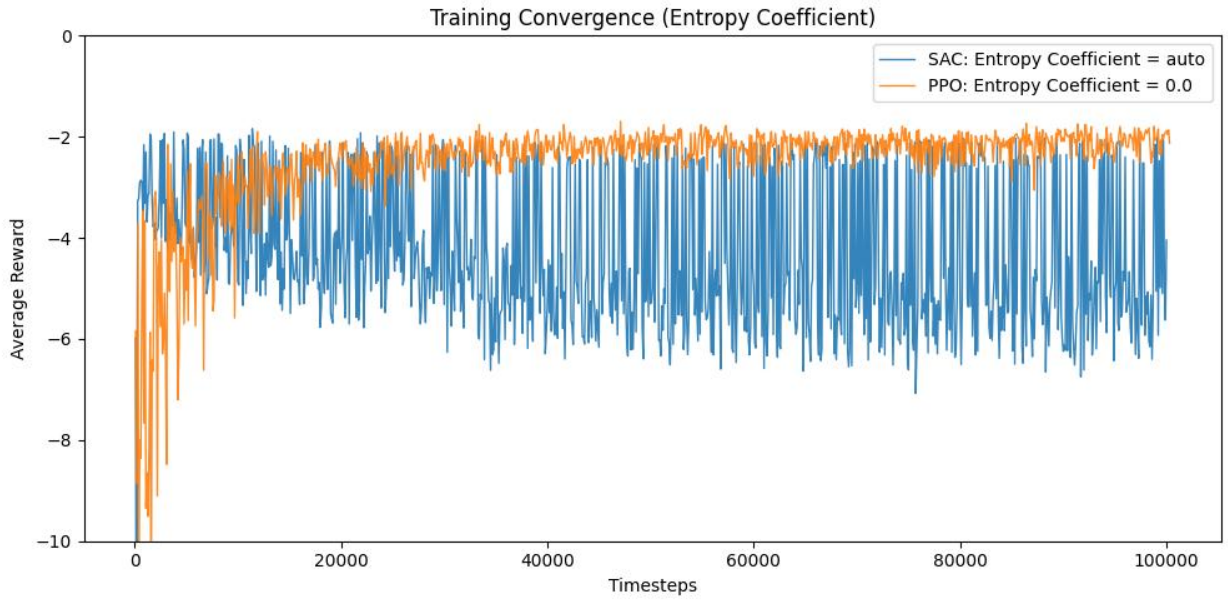
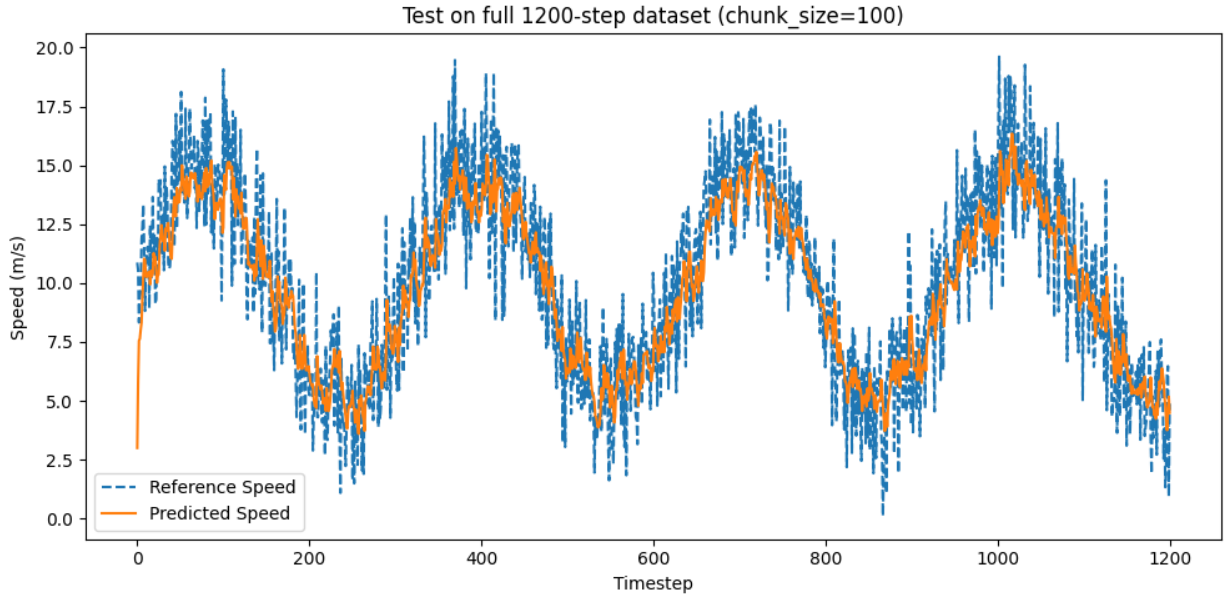



Figure 16 visualizes the predicted speeds of PPO using the default entropy coefficient of 0.0 against the reference speeds. Generally, the predicted speed closely follows the reference speed across the entire sine wave, suggesting the policy follows the reference speeds well. This is supported by the $R^2$ value of 0.712, which is very high. Again, the predicted speeds are smoother than the reference speed, which is the result of the policy generalizing the sinusoidal function and not overfitting itself to the noise. Therefore, without entropy regularization, PPO prioritizes stability in its predictions and may struggle

with capturing high-frequency variations. Also, the model likely converged to a more deterministic policy while avoiding unnecessary exploration. While this stabilized the learning (Figure 15), it can limit the adaptability of this policy to rapid oscillations, which may miss important information. Raising the entropy coefficient may encourage exploration, but as discovered by Table 6, raising it too high will increase the error of the model ($MSE = 5.050$ for an entropy coefficient of 0.01).

**Figure 16:** Speed comparison for testing PPO with an entropy coefficient of 0.0 against the reference speeds.




## Network Architecture Analysis

The results of using various network architectures are shown below. Table 7 compares the key metrics for each model

**Table 7:** Testing various network architectures against the default hyperparameters for each model. The best metric result for each hyperparameter value across all models is highlighted in red, while the best overall result for each model across all hyperparameter values is underlined. The best overall model-hyperparameter combination for each metric is **bold.**

| Model | Metric | Network Architecture | | | |
|---|---|---|---|---|---|
| | | **64 x 64** | **128 x 128** | **256 x 256** | **512 x 512** |
| **SAC** | **Avg. Reward** | -2.086 | -1.984 | -1.958 | -1.898 |
| | **MAE** | 2.086 | 1.984 | 1.958 | 1.898 |
| | **MSE** | 6.809 | 6.145 | 6.010 | 5.615 |
| | **RMSE** | 2.609 | 2.479 | 2.452 | 2.370 |
| | **$R^2$** | 0.586 | 0.626 | 0.634 | 0.659 |
| **PPO** | **Avg. Reward** | **-1.749** | -1.795 | -1.754 | -1.800 |
| | **MAE** | **1.749** | 1.795 | 1.754 | 1.800 |
| | **MSE** | **4.731** | 4.980 | 4.743 | 5.023 |
| | **RMSE** | **2.175** | 2.232 | 2.178 | 2.241 |
| | **$R^2$** | **0.712** | 0.697 | **0.712** | 0.695 |

| | | | | | |
|---|---|---|---|---|---|
| **TD3** | **Avg. Reward** | -2.507 | -1.904 | -2.244 | -2.315 |
| | **MAE** | 2.507 | 1.904 | 2.244 | 2.315 |
| | **MSE** | 9.816 | 5.622 | 7.747 | 8.206 |
| | **RMSE** | 3.133 | 2.371 | 2.783 | 2.865 |
| | $\boldsymbol{R^2}$ | 0.403 | 0.658 | 0.529 | 0.501 |
| **DDPG** | **Avg. Reward** | -1.847 | -2.761 | -2.370 | -2.001 |
| | **MAE** | 1.847 | 2.761 | 2.370 | 2.001 |
| | **MSE** | 5.360 | 12.169 | 8.780 | 6.284 |
| | **RMSE** | 2.315 | 3.488 | 2.963 | 2.507 |
| | $\boldsymbol{R^2}$ | 0.674 | 0.260 | 0.466 | 0.618 |

**Analysis**

Table 7 provides a comparison between the performances of each model using different scales of network architectures.

First, SAC experienced its best performance at a network architecture size of 512x512. It achieved its highest average reward (-1.898), and best error metrics ($MSE = 5.615, R^2 = 0.659$). Decreasing the network architecture gradually worsened the performance metrics, such as an architecture size of 256x256 having a $R^2$ score of 0.634 and MSE of 6.010. Therefore, increasing the network size leads to lower error and higher $R^2$, meaning SAC benefits from larger architectures.

Second, PPO experienced its best performance at an architecture size of 64x64 – the opposite of SAC. It experienced the highest average reward of -1.749, and best error scores ($MSE = 4.731, R^2 = 0.712$). Larger networks worsen performance gradually, but not significantly. In fact, using a network architecture of 512x512 is better performing for the PPO network over the SAC architecture ($R^2 = 0.695$). This model showcases that a larger network architecture can lead to overfitting to recent experiences. Also, the drop in $R^2$ supports this claim. Moreover, the increase in MSE and MAE with increased architecture suggests that PPO struggles with excessive network complexity.

Third, TD3 performed best at a size of 128x128, with a highest average reward of -1.904. It also has its lowest MSE of 5.622 and $R^2$ of 0.658. Performance declines at larger architectures with increasing errors and lower rewards. This score is nearly identical to the performance of SAC using 512x512, but has a significantly worse drop off in performance when modifying the architecture size. Also, the larger architecture suggests that TD3 is susceptible to overfitting.

Fourth, DDPG has its best performance at 64x64, with a highest average reward of -1.847, MSE of 5.360, and $R^2$ of 0.674. Increasing the architecture size at all leads to a significant performance degradation, where the $R^2$ score at a network architecture size of 128x128 is 0.260. This suggests that DDPG benefits from a small network architecture and is the most sensitive to change.

To achieve the best performance per model, it is important to understand the operation and benefits from each. For example, SAC benefits from a large architecture while PPO benefits from small architecture. But, to visualize the differences in training convergences per architecture size, Figure 17 shows the reward convergence during training for PPO with all network architecture sizes. All four network architectures generally followed the same trend, with 512x512 showing the most instability initially with a large leap. However, all four architectures converged to a reward around -2, showing the minimal difference between each architecture. They all maintain stable training and minimal oscillations.

Therefore, given by the metrics provided in Table 7, using a network architecture of 64x64 is the best selection for PPO.

Figure 18 shows the predicted speeds using PPO with a network architecture of 64x64 compared to the reference speeds. The predicted speed follows the general shape of the reference speed, which indicates that the model learned the overall temporal patterns in the data. This is supported by a $R^2$ score of 0.712, which is very high. However, the reference speed fluctuates rapidly, which is not fully captured by the predicted data. The predicted speed is smoother, which may suggest the PPO regularizes this function and ignores fine-grained variations. Increasing the network architecture size would overfit this prediction line to better capture these oscillations, but it would take more training iterations to converge properly. Additionally, the predicted speed does not capture the heights of the valleys and peaks of the reference dataset, which can be reflected by the MSE and MAE errors.

**Figure 17:** Reward convergence over time using PPO with all four different network architectures.

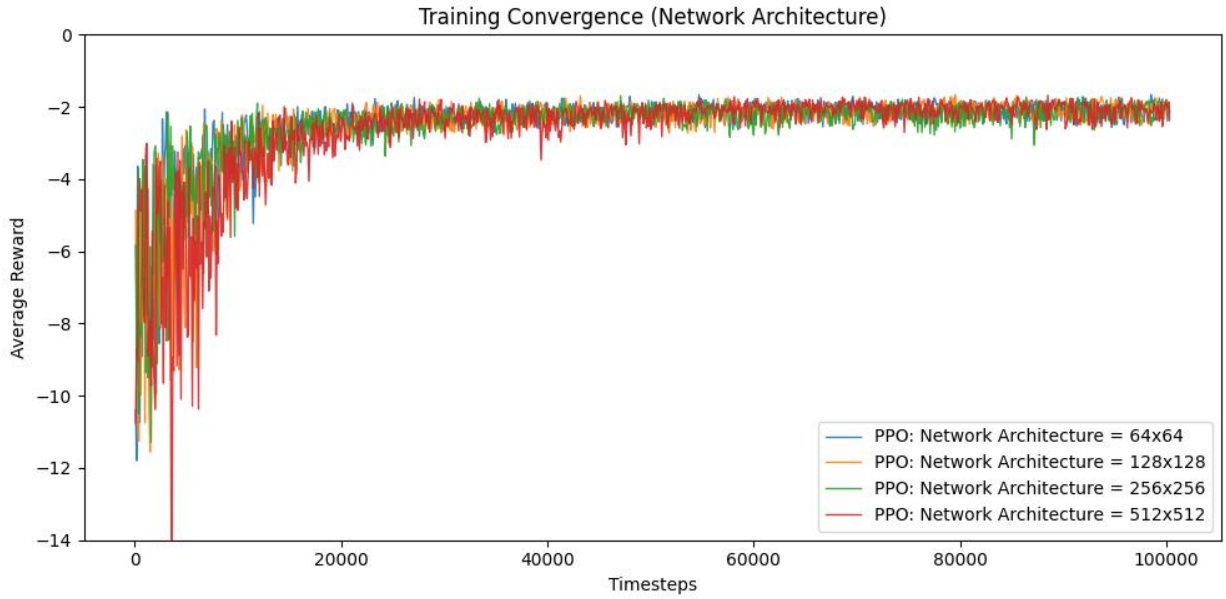
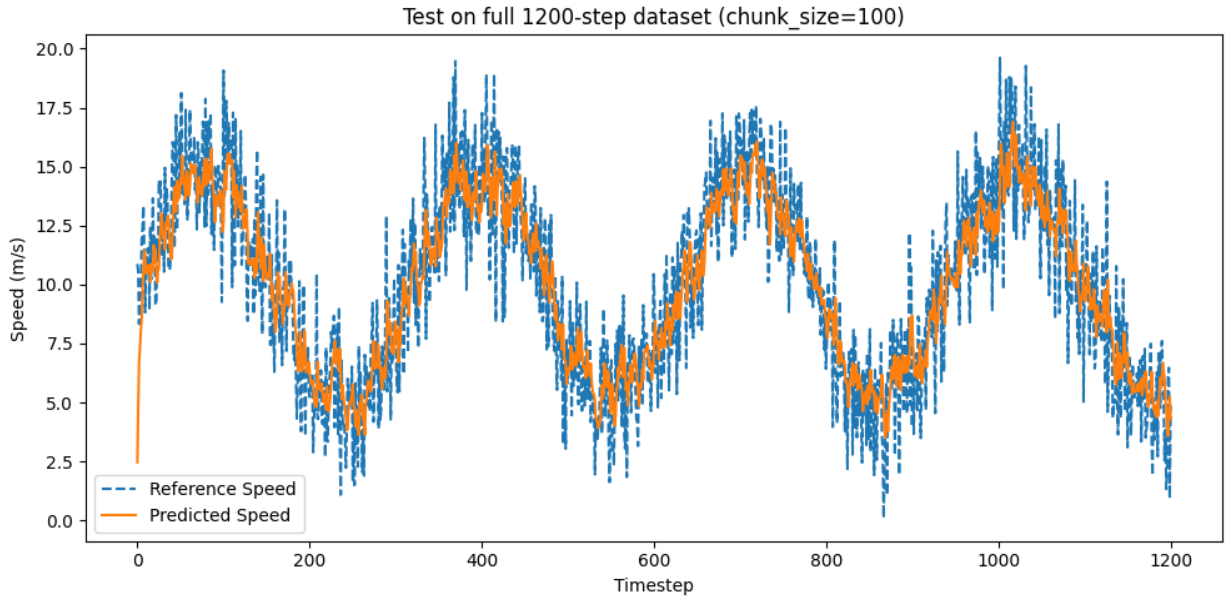


**Figure 18:** Speed comparison for testing PPO with a network architecture of 64x64 against the reference speeds.




## Episode Length Analysis

The results of using various episode lengths are shown below. Table 8 compares the key metrics for each model.

**Table 8:** Testing various episode lengths against the default hyperparameters for each model. The best result using all hyperparameter values for the SAC model is **underlined and bold.**

| Model | Metric | Episode Length | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **1** | **10** | **20** | **50** | **100** | **200** | **600** |
| **SAC** | **Avg. Reward** | -19.514 | -3.703 | -2.325 | -2.429 | -1.958 | **-1.832** | -2.018 |
| | **MAE** | 19.514 | 3.703 | 2.325 | 2.429 | 1.958 | **1.832** | 2.018 |
| | **MSE** | 404.408 | 18.623 | 8.302 | 9.111 | 6.010 | **5.316** | 6.632 |
| | **RMSE** | 20.110 | 4.315 | 2.881 | 3.018 | 2.452 | **2.316** | 2.575 |
| | $R^2$ | -23.595 | -0.133 | 0.495 | 0.446 | 0.634 | **0.674** | 0.597 |

### Analysis

The episode length determines how long the agent interacts with the environment before updating its policy, which directly impacts learning stability, exploration-exploitation balance, and performance generalization. In this project, the episode length corresponds to the number of timesteps per training segment. The SAC model was used for testing the difference of different episode lengths versus performance.

Generally speaking, shorter policy updates should correspond to frequent updates with less trajectory data, leading to higher variance. Longer episodes allow for more stable learning, but may also introduce sluggish adaptation. Short episodes encourage exploratory behavior, while long ones benefit exploitation. Therefore, shorter episodes favor underfitting while larger episodes favor overfitting. Finding an ideal balance between too-short and too-large episode lengths will be the best performing value.

First, there is extremely poor performance using an episode length of 1. The average reward is -19.514, which is significantly worse than any other episode length, suggesting that learning is unstable. Similarly, the MSE is 404.408, which is a higher error than any other hyperparameter configuration in all other experiments (outside of learning rate). The negative $R^2$ value implies the model is performing worse than a naïve baseline that predicts the mean.

Second, using an episode length of 10 returns the performance of the model to reasonable amounts. However, this is still too small, as the $R^2$ value is -0.133, which is still worse than a naïve baseline. The MSE is 18.623, which is a fairly high error.

Performances using episode lengths of 20, 50, and 100 are somewhat better, but still too small. There is significant improvement over an episode length of 10, seeing average rewards of -2.325, -2.429, and -1.958, respectively. Also, $R^2$ scores of 0.495, 0.446, and 0.634 suggest that the model is learning a good policy. Specifically, using an episode length of 100 shows a generalizable policy with smaller error ($MSE = 6.010$).

On the opposite side, using an episode length of 600 is too large, with a similar performance as using an episode length of 100. The $R^2$ score is 0.597, which is a good score but not as performant as an episode length of 200. Additionally, the decrease in average reward to -2.018 indicates that the model is losing adaptability in exchange with overfitting to the training data. The MSE of 6.632 suggests the model may be overfitting to long-term trends as well.

Therefore, an episode length of 200 yields the best average reward (-1.832), best MSE (5.316), and best $R^2$ score (0.674). This suggests that this episode length is the best balance between exploration and exploitation tested, and has diminishing returns to increase and decrease this value.

Figure 19 visualizes episode lengths of 50, 100, 200, and 600 using the SAC model. 1 and 10 were omitted due to their poor performance in Table 8. Using episode lengths of 50 and 100 showed high oscillations throughout the entire duration of training, Interestingly, using an episode length of 600 showed nearly no oscillations with training. This can be explained by the higher amount of time steps needed for each weight update, causing for less extreme updates. Overall, using an episode length of 200 converged to the lowest average reward, and displayed the most stable training process. Therefore, using an episode length of 200 is the significantly best value for balancing exploration and exploitation.

Figure 20 visualizes SAC with an episode length of 200 predicting the speeds during testing when compared to the reference speeds. The predicted speed aligns well with the reference speed, which indicates the policy learned the sinusoidal pattern of it. The $R^2$ score of 0.674 supports this fact, as this is a fairly confident value. Similarly, the predicted speeds follow a smoother path than the reference speed, suggesting that it is generalizable to the pattern of the input rather than overfit to the noisy data. However, it could also suggest that the model could still be underfitting to high-frequency fluctuations. In fact, the predicted speeds appear to be lagging behind the reference speeds. This reflects the MSE of 5.316, which is a good score but suggests that there is still some error with the policy. The RMSE is 2.316, which represents an average error of 2.316 with the predicted speed versus the reference speed. Lastly, there is some large initial error with the predicted speeds, which could be the result of needing more episodes to catch up to the reference data.

**Figure 19:** Reward convergence over time using SAC with episode lengths of 50, 100, 200, and 600.

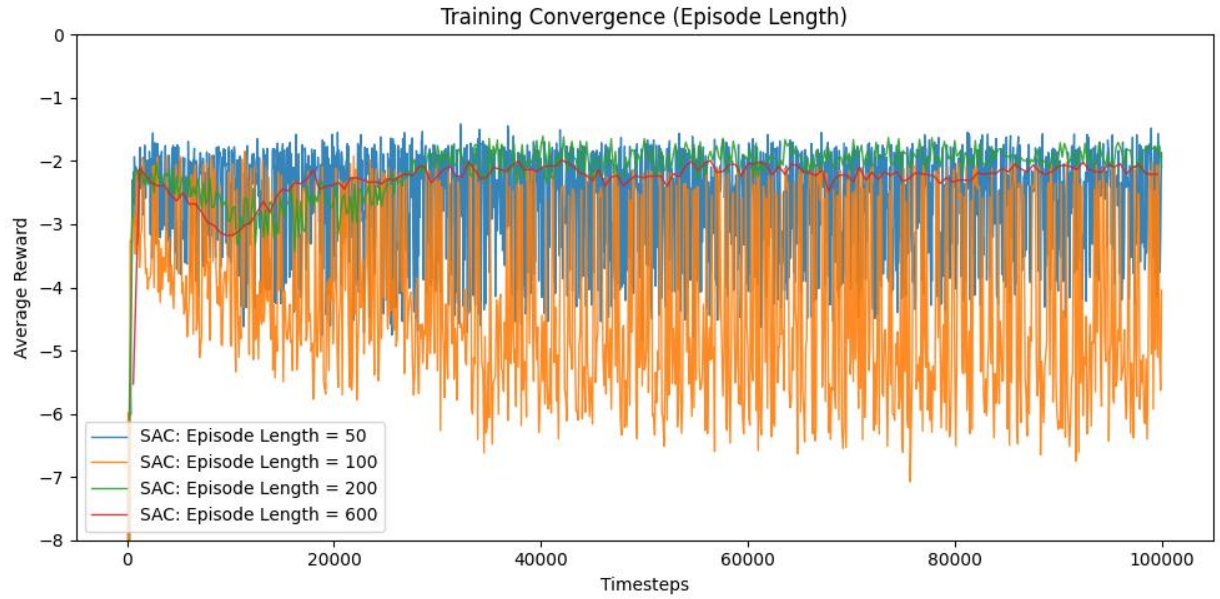
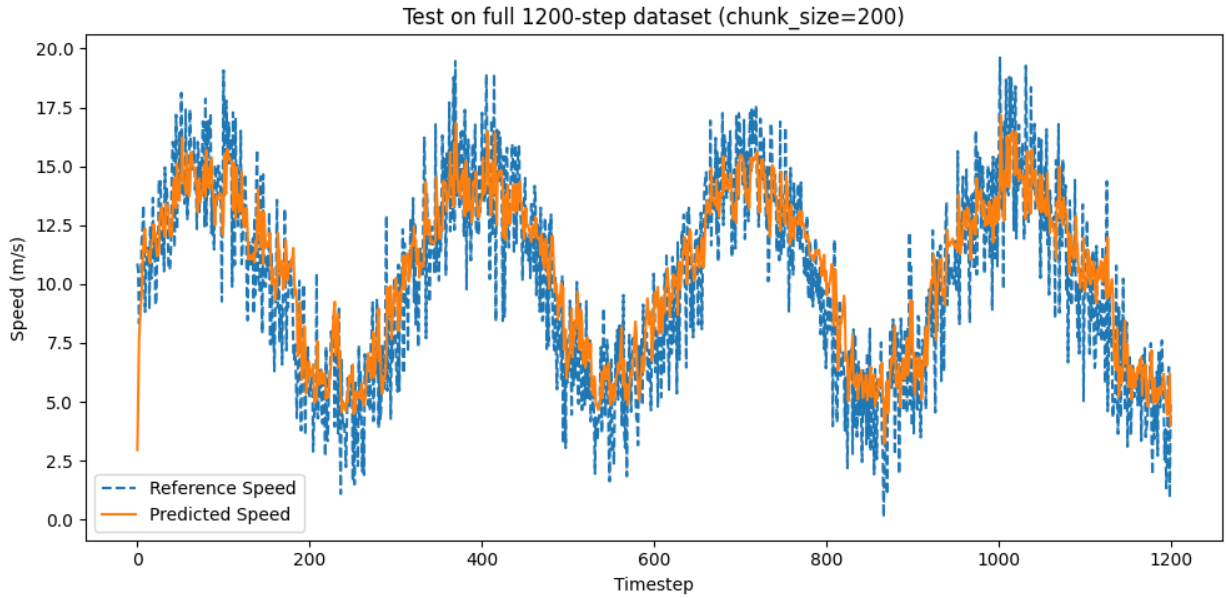



**Figure 20:** Speed comparison for testing SAC with a episode length of 200 against the reference speeds.




## Reward Function Analysis

The results of using various reward functions are shown below. Table 9 compares the key metrics for each model.

**Table 9:** Testing various reward functions against the default hyperparameters for each model. The best result using all hyperparameter values for the SAC model is **underlined and bold.** However, the average reward scores are not considered due to calculation differences.

| Model | Metric | Reward Function | | | | |
|---|---|---|---|---|---|---|
| | | Absolute Error | Squared Error | Exponential Error | Thresholded Error | Cubed Error |

| | | | | | | |
|---|---|---|---|---|---|---|
| **SAC** | **Avg. Reward** | -1.958 | -2.032 | -2.813 | -2.072 | -1.958 |
| | **MAE** | 1.958 | 2.032 | 2.813 | 2.072 | 1.958 |
| | **MSE** | **6.010** | 6.719 | 7.497 | 6.641 | **6.010** |
| | **RMSE** | **2.452** | 2.592 | 2.738 | 2.577 | **2.452** |
| | $R^2$ | **0.634** | 0.591 | 0.544 | 0.596 | **0.634** |

**Analysis**

The choice of a reward function directly influences how the agent learns and optimizes its behavior. This analysis focuses on the benefits presented to the SAC architecture, as the reward function should operate independent of model type. Additionally, given the rewards are evaluated at different scales, this analysis will focus on MSE, RMSE, and $R^2$.

The best performing function in terms of these metrics is tied between absolute error and cubed error. Both functions saw a MSE of 0.610, RMSE of 2.452, and $R^2$ of 0.634. Absolute error directly penalizes the absolute deviation from the true value, making it a simple yet effective approach. It also does not disproportionately punish larger errors, which contributes to stable learning. However, cubed error amplifies larger deviations compared to absolute error. Given this result tied the absolute error, most errors in the dataset were small – hence, the cubing effect is negligible. With datasets with significant prediction outliers, cubed error could introduce instability.

Conversely, squared error saw worse performance than absolute error. Squaring the deviation between predictions and reference speeds can cause for larger deviations, but given the results with cubed error, this may be the network's response to the particular method of punishing larger deviations. Exponential error was the worst performing function with the highest MAE, MSE, and RMSE, alongside the lowest $R^2$ score. This suggests that exponentially scaling the penalty leads to erratic learning behavior, likely to the excessive sensitivity to minor errors. The model overcompensates for small deviation, which causes instability. Lastly, thresholded error indicates moderate performance with its MAE, MSE, RMSE, and $R^2$ scores. This function only penalizes errors beyond a certain threshold, reducing sensitivity to minor fluctuations. While useful for stabilizing training, it may not generalize well if the threshold is improperly tuned.

Figure 21 visualizes absolute error against cubed error in terms of convergence rate. Apparently, the cubed error function converges quicker than absolute error, and has a much more stable training performance. This can be due to the fact that lower errors are positively amplified given the cubing (< 1.0). Therefore, the most stable reward function, as well as the best performing, is the cubed error function.

Figure 22 visualizes the predicted speeds calculated during testing against the reference speeds using SAC with a cubed error reward function. Generally, the predicted line follows the trend of the reference speed well, indicating that this configuration captures the general patterns of the speed variations. This is supported by a high $R^2$ score of 0.634. The peaks and valleys are also mostly mirrored by the predicted speeds, which demonstrates reasonable model learning. Also, the predicted speeds are smoother, exhibiting its generalization to the reference speeds. However, there is a consistent lag from the predicted speed, particularly around sharp edges, which shows a conservative prediction approach. This can explain the low MSE and MAE errors. This also coincides with a cubic loss behavior, where extreme precision is favored in smaller area locations due to the large penalty of error. This is demonstrated by the thickness of the line during flat points in the sinusoidal wave. However, the model is also limited to react

to sharp changes due to very small penalties applied to small errors, leading to this lagging behavior at valleys and peaks. Overall, using a cubic function was effective in reducing small fluctuations, but cause for more extreme function lagging and struggling with edges.

**Figure 21:** Reward convergence over time using SAC with reward functions of absolute error and cubed error.

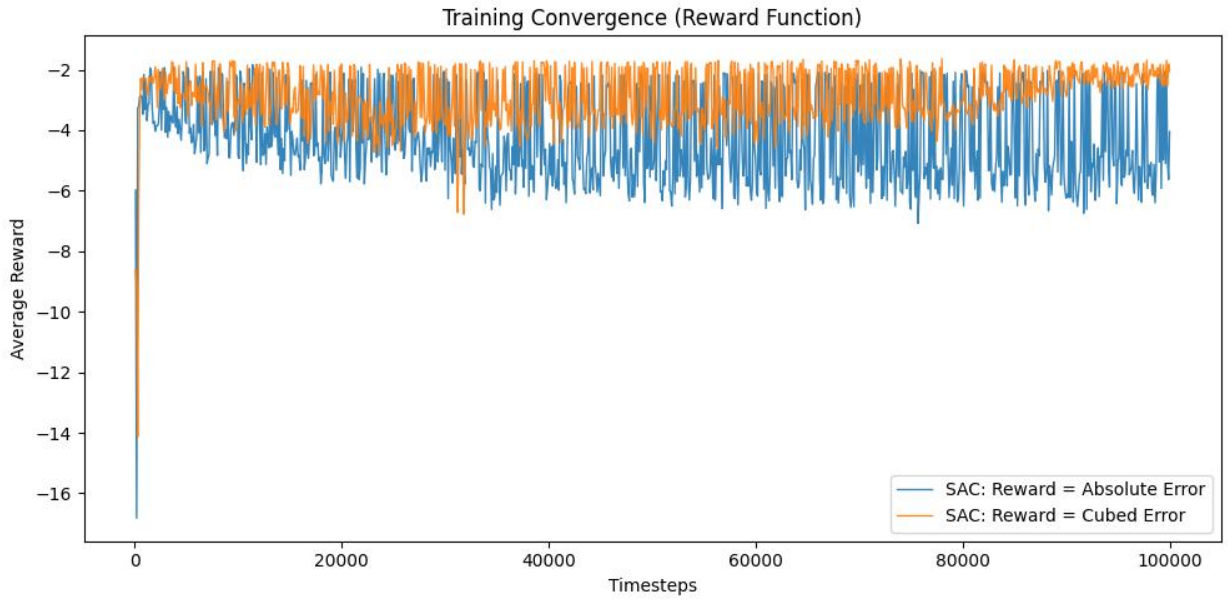
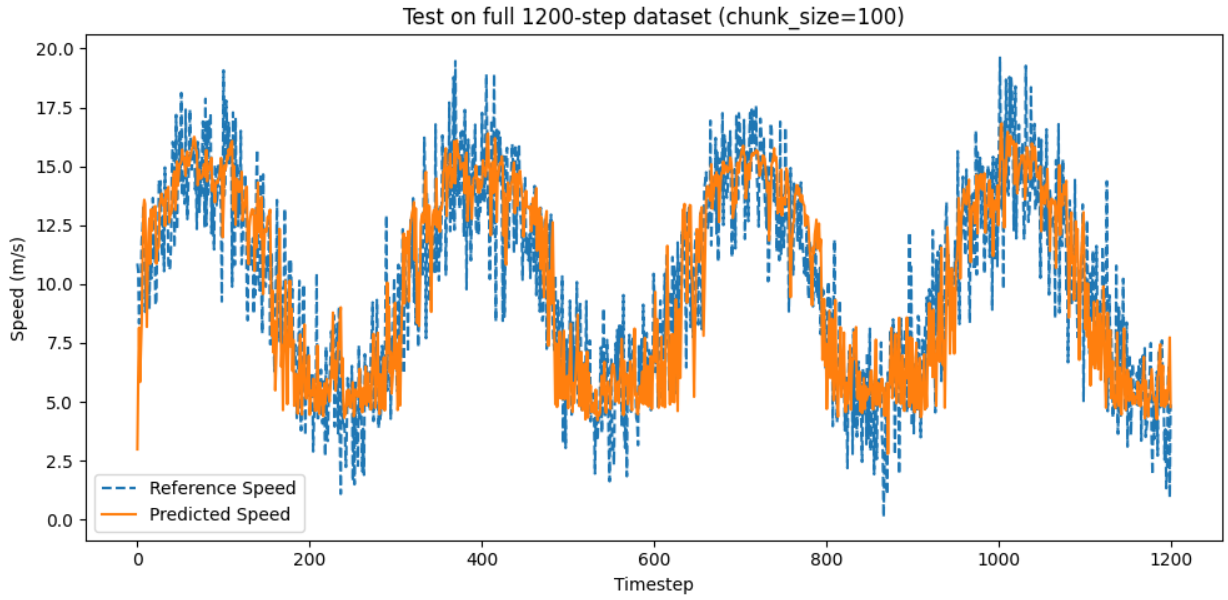



**Figure 22:** Speed comparison for testing SAC with a reward function of cubed error against the reference speeds.



## Best Configurable Results

This section will explore using the ideal combination of all hyperparameters for each model and comparing the performances. The best model will be selected based on the same performance metrics. The ideal configuration for each model is provided below.

1. **SAC**
    a. Learning Rate = **3e-4**
    b. Batch Size = **64**
    c. Buffer Size = **100,000**
    d. Tau = **0.0001**
    e. Gamma = **0.90**
    f. Entropy Coefficient = **auto**
    g. Network Architecture = **512x512**
    h. Episode Length = **200**
    i. Reward Function = **Cubic Error**
2. **PPO**
    a. Learning Rate = **1e-3**
    b. Batch Size = **256**
    c. Gamma = **0.90**
    d. Entropy Coefficient = **0.0**
    e. Network Architecture = **256x256**
    f. Episode Length = **200**
    g. Reward Function = **Cubic Error**
3. **TD3**
    a. Learning Rate = **3e-4**
    b. Batch Size = **128**
    c. Buffer Size = **1,000,000**
    d. Tau = **0.0001**
    e. Gamma = **0.99**
    f. Network Architecture = **128x128**
    g. Episode Length = **200**
    h. Reward Function = **Cubic Error**
4. **DDPG**
    a. Learning Rate = **1e-3**
    b. Batch Size = **64**
    c. Buffer Size = **50,000**
    d. Tau = **0.0001**
    e. Gamma = **0.95**
    f. Network Architecture = **64x64**
    g. Episode Length = **200**
    h. Reward Function = **Cubic Error**

The results from these configurations are shown below in Table 10.

**Table 10:** Testing the best hyperparameter configurations for every model. The best results between all models are **underlined and bold.**

| Model | Metric | Results |
|---|---|---|
| **SAC** | **Avg. Reward** | -1.751 |
| | **MAE** | 1.751 |
| | **MSE** | 4.803 |
| | **RMSE** | 2.192 |
| | $\boldsymbol{R^2}$ | 0.708 |
| **PPO** | **Avg. Reward** | -1.733 |
| | **MAE** | 1.733 |
| | **MSE** | 4.629 |
| | **RMSE** | 2.151 |
| | $\boldsymbol{R^2}$ | 0.718 |
| **TD3** | **Avg. Reward** | -1.835 |
| | **MAE** | 1.835 |
| | **MSE** | 5.286 |
| | **RMSE** | 2.299 |
| | $\boldsymbol{R^2}$ | 0.679 |
| **DDPG** | **Avg. Reward** | **-1.721** |
| | **MAE** | **1.721** |
| | **MSE** | **4.573** |
| | **RMSE** | **2.138** |
| | $\boldsymbol{R^2}$ | **0.722** |

**Analysis**

For this analysis, the hyperparameters (namely learning rate and batch size) were selected to optimize each model's performance while preventing divergence or instability. Among the models tested, PPO had previously demonstrated consistently superior performance in every metric. This section examines whether that trend holds when using the best possible configuration for each model.

With SAC, a relatively smaller learning rate (3e-4) was used to prevent the instability that using the original best selection (1e-2). Also, given the cubic reward function, large deviations will be exemplified and potentially overflow the program, favoring lower learning rates. A large replay buffer of 100,000 helps maintain the sample diversity, and using a large architecture size of 512x512 suggests that SAC benefits from a more complex policy representation.

With PPO, a higher learning rate (1e-3) was used because it was proven to be the most stable step size without a large compromise to performance. A larger batch size of 256 likely contributes to better generalization, and no entropy regularization was used, indicating that exploration is not an issue in this scenario. Also, a larger network architecture of 256x256 was selected to be more generalized to the larger batch size selection. Moreover, a sample run of PPO using a network architecture of 64x64 was tested and yielded insignificantly worse results than this configuration ($MAE = 1.735, MSE = 4.649, R^2 = 0.717$).

With TD3, a moderate gamma (0.99) suggests that the model places more weight on long-term rewards. Additionally, a very large buffer size of 1,000,000 ensures a diverse training set, complementing this long-term reward structure. However, sampling inefficiency may be present in this configuration.

With DDPG, a high learning rate of 1e-3 was also used, which is slightly shy of its best performing learning rate of 3e-3 to avoid divergence in the reward. Also, it used the smallest network architecture of 64x64 given that DDPG benefits from a more lightweight representation, likely to reduce overfitting given its deterministic nature.

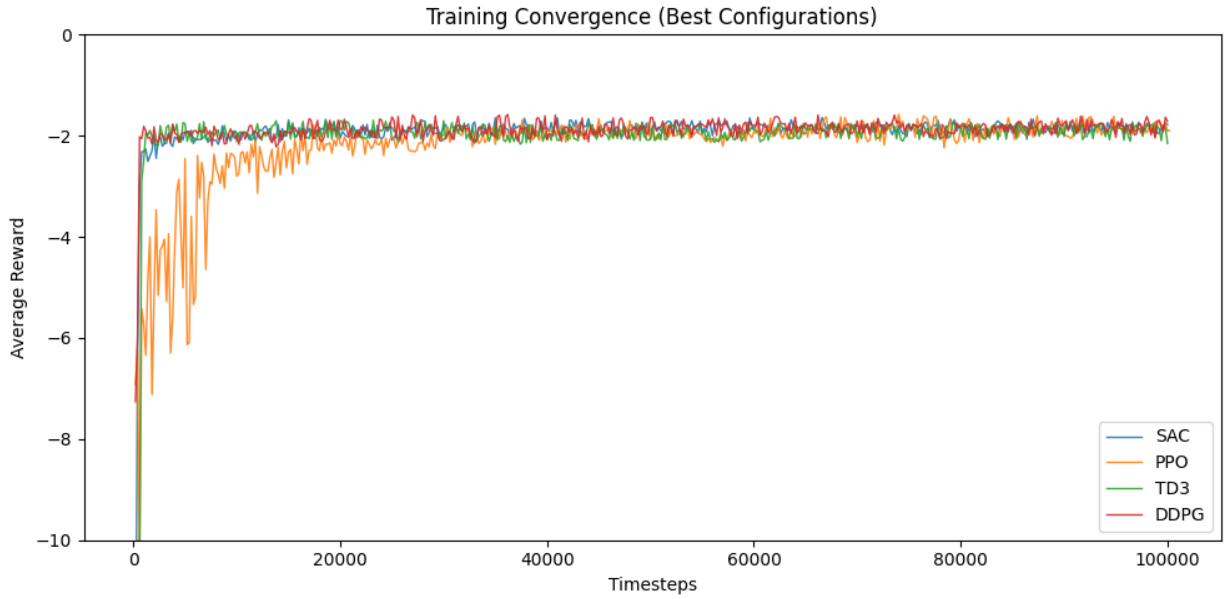
Overall, DDPG achieved the best performance in every metric. It achieved the highest $R^2$ score (0.722) and lowest error metrics ($MAE = 1.721, MSE = 4.573$). PPO was highly competitive to these scores and yielded the second-best metrics ($R^2 = 0.718$). Both models achieved strong error minimization in their predictions. SAC was effective, but did not perform as well as PPO and DDPG, likely due to its higher sample complexity. TD3 had the weakest performance, showing the highest error values across all metrics, indicating that it struggled to generalize well despite its theoretically improved stability over DDPG.

As a side note, using this ideal configuration of parameters for PPO did not outperform its best performance with default parameters and a learning rate of 1e-2. This learning rate was not selected due to the instability of training, but it yields a better result than the ideal configuration of DDPG.

The justification for why PPO and DDPG performed the best can be given as follows. PPO benefitted from using a high learning rate (1e-3) to help it learn faster without divergence. Also, its clipped objective function provides it stability, avoiding drastic policy updates. DDPG benefitted from a high learning rate (1e-3) as well, and used a smaller network to learn faster without overfitting. SAC relied too heavily on entropy maximization, which most likely introduced instability while training in later stages of the process. Additionally, the 512x512 network architecture made training slower and less sample efficient. TTD3's overly large buffer size (1,000,000) might have hurt its performance due to stale data influencing policy updates. The small network architecture size of 128x128 may have limited its function approximation capacity, and the high gamma of 0.99 may have prioritized long-term gains and reduced its response to immediate changes.

Figure 23 visualizes the reward convergences for all four of these configurations. All models eventually converge to a similar reward level (<-2.0) after 100,000 timesteps. PPO showed the most instability early on, with sharp fluctuations in average reward during the initial training phase. This suggests that PPO explores more aggressively before stabilizing and coincides with its higher learning rate of 1e-3. SAC, TD3, and DDPG have relatively smoother learning curves, with fewer fluctuations early in training. This aligns with their deterministic nature. However, all models seem to stabilize around the same point, meaning that the hyperparameter tuning did not dramatically alter the final performance.

**Figure 23:** Reward convergence of all models using ideal hyperparameter configurations over training.




Figures 24 – 27 visualize the predicted speed of each model using these ideal configurations against the reference speeds. Figure 24 shows that SAC produces smoother predictions that closely follow the reference speed, but still retain some noise. SAC's entropy-regularized objective helps with stable learning, leading to good performance in this task.

**Figure 24:** Speed comparison for testing SAC with the ideal hyperparameter configuration against the reference speeds.

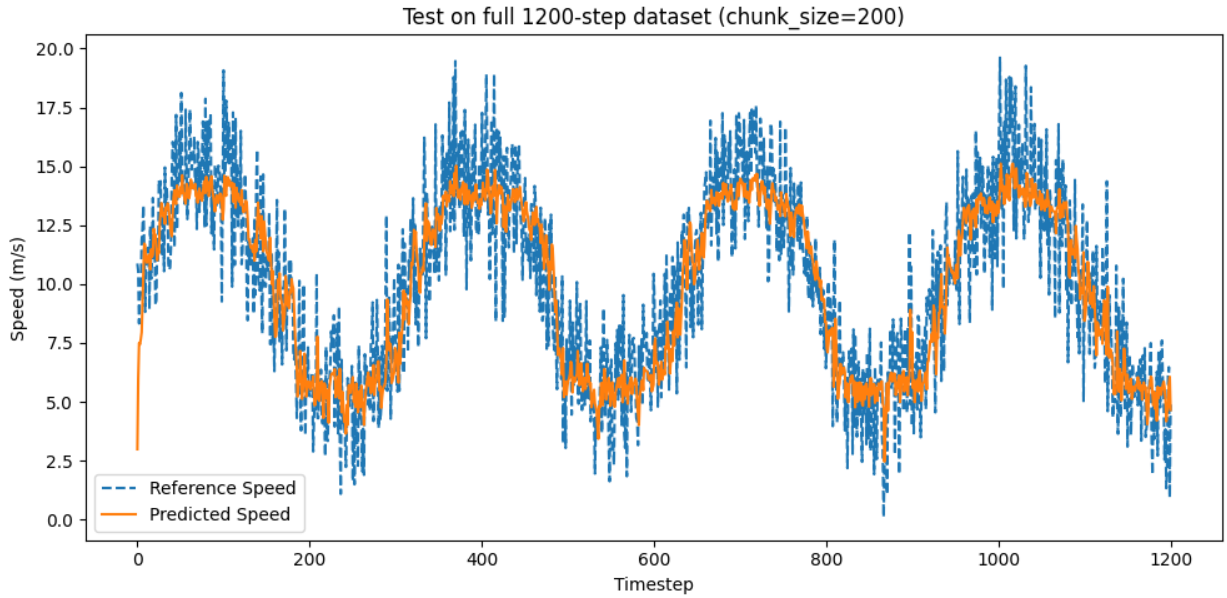



Figure 25 reveals that PPO's predictions are slightly noisier than SAC but still follow the reference speed well. PPO's policy updates via its clipped objective function likely contribute to this stable performance.

**Figure 25:** Speed comparison for testing PPO with the ideal hyperparameter configuration against the reference speeds.

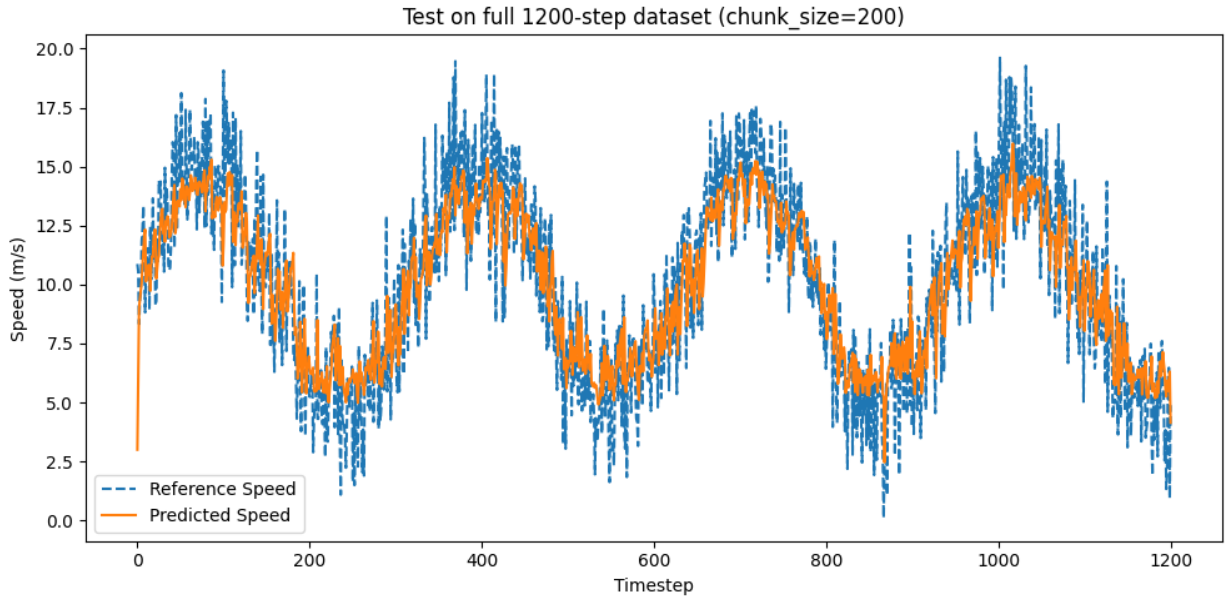



Figure 26 suggests that TD3 struggles more with matching the high-frequency fluctuations in the reference speed. However, it still captures the general trend, likely benefitting from its delayed actor updates and clipped target policy.

**Figure 26:** Speed comparison for testing TD3 with the ideal hyperparameter configuration against the reference speeds.

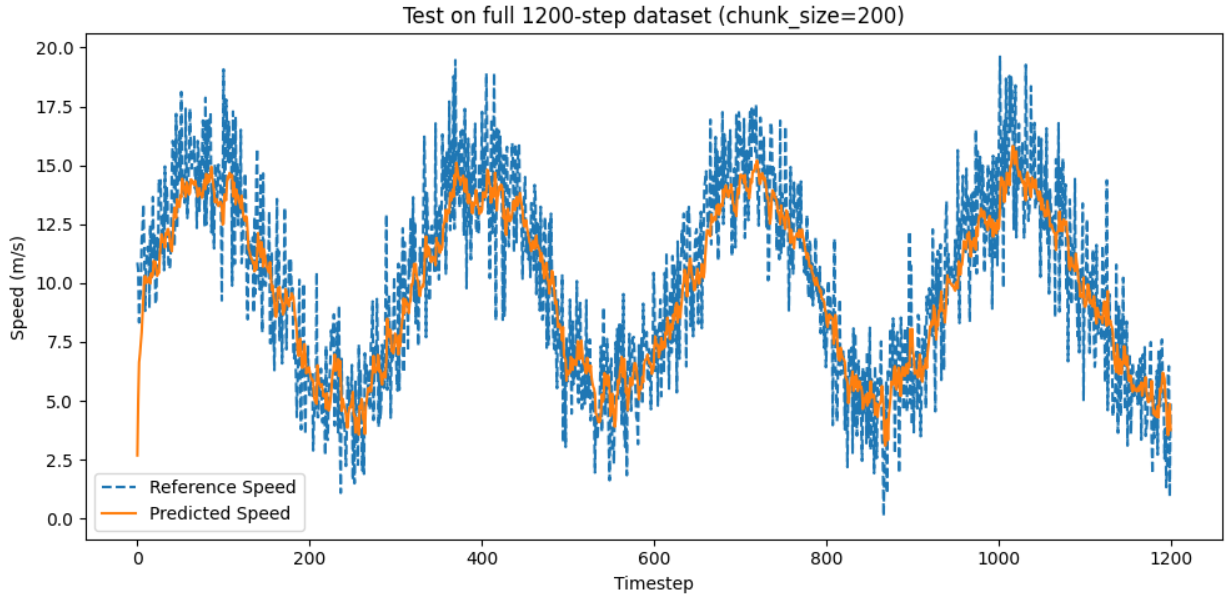
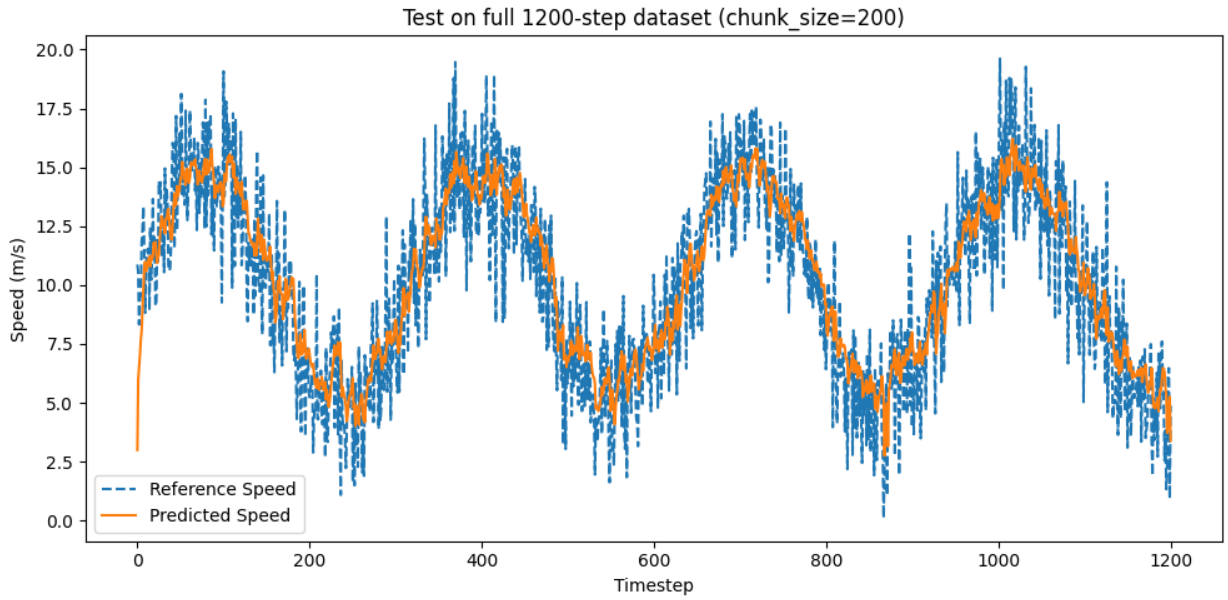



Figure 27 shows that DDPG has higher variance and appears to be prone to overfitting or slight instability, which aligns with its sensitivity to exploration noise and hyperparameters. As a result, this plot suggests that PPO may be the better performing model for this task, despite its slight improvement in error metrics and $R^2$ score.

**Figure 27:** Speed comparison for testing DDPG with the ideal hyperparameter configuration against the reference speeds.



# Discussion

The results of this RL experiment highlight the significant impact that hyperparameters and design choices have on the overall learning process. Through systematic analysis of various hyperparameters, it became clear that subtle modifications could drastically influence learning efficiency, convergence stability, and final policy performance.

The learning rate proved to be one of the most critical factors in determining the success of training. Extremely low learning rates (1e-5) resulted in slow convergence, while excessively high values (1e-1) led to unstable training dynamics and divergence. The optimal learning rate was found to be between 1e-4 and 1e-3, where convergence was stabilized and efficient. This suggests that a balance must be struck between making meaningful updates to the policy and maintaining stability.

The batch size influenced the stability of training and the variance of updates. Smaller batch sizes (32) introduced higher variance in gradient updates, promoting exploration but at the cost of increased noise. Larger batch sizes (512) resulted in more stable updates but required higher computational costs. A mid-range batch size of 128 to 256 appeared to offer the best trade-off.

The buffer size played a role in determining how much past experience the agent could use for training. A small buffer size (50,000) prioritized recent experiences but could lead to forgetting important past information. Conversely, a large buffer size (1,000,000) retained a more comprehensive training history but required significant memory overhead. The experiments suggest that a buffer size of around 200,000 provided a good balance for this task.

Tau, which controls the rate at which the target network updates its weights, significantly influenced the smoothness of learning. A very low tau value (0.0001) led to negligibly slow updates, which helps generalize the model. Higher tau values allowed for responsive updates, but introduced a bit of instability. Ultimately, lower tau values were favored for all models, given the testing range's lower bound was 0.0001.

Gamma determined the trade-off between short-term and long-term rewards. Lower gamma values (0.90) encouraged short-sighted policies, while higher values (0.999) promoted long-term planning. Most of the time, smaller gamma values were better performing (0.90, 0.95), which indicates that there is a slight preference for being able to react to shorter plans given the rapid oscillations of the reference speeds.

The entropy coefficient controlled exploration versus exploitation. Lower entropy values (0.01) led to more deterministic policies, while higher values (0.2) encouraged greater exploration at the cost of stability. The optimal setting seemed to favor the default configuration for both SAC and PPO. With SAC, having a dynamic entropy coefficient that updated depending on the current policy's confidence promoted the best results, and with PPO, excluding this coefficient boosted performance. For both models, having a smaller value for entropy (or nonexistent) was beneficial for convergence.

The network architecture significantly impacted the model's capacity to learn complex policies. Smaller networks (64x64) struggled with representing intricate relationships, despite potentially small error rates (MSE, MAE). Large networks were computationally expensive and were prone to overfitting, but allowed to model larger relationships. The best performance typically was a balance between small and large (128x128, 256x256).

Longer episode lengths provided more data per episode but required careful reward shaping to ensure meaningful learning. Shorter episodes made learning more sample-efficient but risked premature

termination before sufficient state exploration. The best approach involved tuning episode length in conjunction with an appropriate reward function.

The reward function itself played a crucial role. Reward structures that heavily penalized deviations from the reference speed promoted stable policies but limited flexibility. This includes exponential and cubic error functions. More adaptive reward designs, such as thresholding the absolute error, resulted in more flexibility, but also promoted a bit of instability during reward convergence. The cubic error function showed the best results in terms of reward convergence and stability.

The comparison of different models trained under varying hyperparameter settings reveals clear trends regarding optimal configurations. Models with excessively high learning rates tended to diverge, while those with low learning rates were slow to learn. Similarly, models trained with small batch sizes had erratic learning curves due to high gradient variance, where those trained with very large batch sizes demonstrated slower adaptation to dynamic environments. Generally speaking, PPO was the best model in terms of convergence stability and final policy. Using the default parameters, PPO outperformed every other model during the fine-tuning experiments, with SAC showing slightly worse but second-best results. During the final configuration, DDPG outperformed PPO, but only via metrics, as the predicted speeds were overfit, which explains the larger $R^2$ score. Overall, PPO was the best model in terms of robustness with creating an ideal policy for this RL task.

In conclusion, this study demonstrated that hyperparameter tuning and design choices are necessary for success with RL training.