Reinforcement Learning with Domain Randomization and SimOpt: a comparison between optimizers and discrepancy functions

Emanuele Francesco Elias
s344489

Dalia Lemmi
s344440

Abstract

In this paper, a reinforcement learning framework is extended to enable comparison between various algorithms, with the goal of analyzing their performance in the same continuous control environment. Then, Uniform Domain Randomization and Adaptive Domain Randomization techniques are implemented to improve the generalization capabilities of policies trained in simulation. Specifically, the latter is applied using the SimOpt framework to evaluate the impact of different optimization strategies and discrepancy metrics on policy performance. This analysis helps identify the most effective combinations for sim-to-real transfer. The full code is available at `https://github.com/Emanuele-Elias/rl_mldl_25.git`.

1 Introduction

Reinforcement Learning (RL) is a computational approach to learning through interaction, where an agent improves its behavior by trial and error, guided by reward signals [1]. Unlike supervised or unsupervised learning, RL focuses on learning to act optimally in dynamic environments based on reward feedback.
This paradigm is particularly useful for problems where explicit supervision is impractical or infeasible, such as robotics, game playing, or autonomous control systems.

2 Related Work

2.1 What is reinforcement learning?

Reinforcement Learning is defined as the problem of learning what to do, in practice how to map situations to actions, so as to maximize a numerical reward signal [1]. The learning agent interacts with its environment through a cycle of sensing the current state, choosing and executing an action, and receiving feedback in the form of a reward and a new state. The agent must not only learn to act based on immediate rewards but also consider how current actions influence future outcomes. This leads to the exploration-exploitation trade-off, where the agent must balance the need to explore new actions to discover better rewards with exploiting known actions that yield high returns.
The main components of an RL system are:

- **Policy:** a strategy used by the agent to determine the next action based on the current state.
- **Reward signal:** feedback from the environment indicating the success of an action.
- **Value function:** a prediction of future rewards used to evaluate the desirability of states.
- **Model (optional):** a representation of the environment used for planning.

2.2 What are MDP, Action spaces, state spaces, continuous/discrete environments?

RL problems are often formalized as a *Markov Decision Process* (MDP), a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ is the set of possible states;
- $\mathcal{A}$ is the set of possible actions;
- $P(s'|s, a)$ is the transition probability of moving to state s' after taking action a in state s;
- $R(s, a)$ is the reward function;
- $\gamma \in [0, 1]$ is the discount factor.

MDPs capture the dynamics of the environment and the sequential nature of decision making. Environments may be *discrete* (finite sets of states and actions) or *continuous* (states and/or actions lie in continuous domains). Continuous environments pose significant challenges due to the infinite nature of the space and the need for function approximation techniques.

2.3 What is Sim2Real Transfer?

Simulations enable safe and low cost training, but often fail to perform effectively when deployed in the real world. This challenge is known as the **Sim-to-Real transfer** problem. It arises due to the *reality gap*, that is the inevitable mismatch between the simulated and real environments, due to factors such as noisy dynamics, unmodeled interactions, and sensor uncertainties [3].
In their summary of the R:SS 2020 Workshop, Hofer et al. [3] emphasized that while simulation is crucial for safe and scalable learning, it is insufficient on its own. The consensus is that simulations are best viewed as a tool to support, but not replace, empirical testing.
Strategies to improve Sim2Real transfer include:

- **Domain Randomization**, which trains policies on a wide distribution of simulated environments to improve robustness [3].
- **System Identification**, where real-world observations are used to iteratively tune the simulator [2].
- **Task-Specific Simulation Tuning**, balancing between simulation precision and abstraction to prioritize policy generalization over raw accuracy.

There is also an ongoing debate on whether accurate simulation is a necessity. Some researchers argue that overly precise simulation may hinder generalization, and that deliberate abstraction or even biased simulation may yield more robust policies. Furthermore, the field increasingly acknowledges the importance of *policy representations* that are adaptable, scalable, and resilient to domain shifts [2].
To move forward, the community is exploring hybrid approaches combining real-world feedback with simulated environments (for example Real2Sim-Sim2Real loops), as well as meta-learning, differentiable simulators, and more expressive policy architectures. These efforts support the notion that Sim2Real transfer is not just a byproduct of other techniques, but a research field in its own right [3].

3 Methodology

3.1 Basic algorithms

3.1.1 REINFORCE

REINFORCE [1] is a Monte Carlo policy-gradient algorithm that directly optimizes the policy parameters by following the gradient of the expected return. The central idea of REINFORCE is to adjust the policy parameter θ in the direction of greater expected return $J(\theta)$:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi(A_t|S_t, \theta), \quad (1)$$

where G_t is the return obtained after taking action A_t in state S_t, and α is the learning rate.
The gradient $\nabla_\theta \log \pi(A_t|S_t, \theta)$ encourages actions that yield higher returns, while penalizing less favorable ones. REINFORCE is known for its simplicity and unbiased gradient estimates, but it suffers from

high variance.
To address this issue, a common extension is REINFORCE with baseline, where the return is centered by subtracting a baseline $b(S_t)$, often an estimate of the state value function $\hat{v}(S_t, w)$. The update rule becomes:

$$\theta \leftarrow \theta + \alpha(G_t - b(S_t))\nabla_\theta \ln \pi(A_t|S_t, \theta). \tag{2}$$

This does not change the expected value of the gradient but can significantly reduce its variance, thereby improving the learning process.

3.1.2 Actor-Critic

Actor-Critic methods combine the strengths of policy gradient (actor) and value function approximation (critic). The actor updates the policy parameters θ using gradient ascent, while the critic learns the value function parameters w to approximate the expected return.
Differently than REINFORCE, which waits until the end of an episode, Actor-Critic methods use bootstrapping. A typical update uses the one-step return:

$$\delta_t = R_{t+1} + \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w), \tag{3}$$

$$\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \ln \pi(A_t|S_t, \theta), \tag{4}$$

$$w \leftarrow w + \alpha_w \delta_t \nabla_w \hat{v}(S_t, w). \tag{5}$$

This approach enables online, incremental updates and typically exhibits faster learning due to reduced variance. The use of eligibility traces, n-step returns, or the λ-return can further enhance learning stability and efficiency [1].

3.1.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [4] is a widely used on-policy reinforcement learning algorithm that balances performance and simplicity. It belongs to the class of policy gradient methods, but it improves training stability by limiting the deviation between the old and updated policies through a clipped surrogate objective. This way it encourages small policy updates while still allowing multiple epochs of minibatch optimization per batch of collected data.

3.2 Domain Randomization

3.2.1 Uniform Domain Randomization

Uniform Domain Randomization (UDR) is a simple yet effective sim-to-real transfer strategy. UDR operates by training the agent on a distribution of environments rather than a single fixed one: specific physical parameters are randomly perturbed at the start of each episode, forcing the policy to generalize beyond any single instantiation of the task. However, UDR requires manual tuning of the parameter distributions, does not leverage real-world feedback, and may fail to capture domain shifts that lie outside the randomized range.
In our case, we applied UDR to the Hopper agent by varying the masses of selected body parts (thigh, leg, and foot) within predefined uniform intervals. This exposed the policy to a range of dynamic conditions during training, without ever altering the torso mass. The idea is that, by learning over such a randomized domain, the agent can become more invariant to dynamics changes and potentially generalize better to unseen target environments. Later we provide quantitative results comparing PPO with and without UDR.

3.2.2 Adaptive Domain Randomization

Unlike UDR, Adaptive Domain Randomization (ADR) modifies the distribution over domain parameters based on observed discrepancies between simulated and real-world rollouts. The goal is to iteratively refine the simulation environment so that policies trained within it generalize better to the real world. ADR uses feedback from the real environment to guide this refinement, typically by optimizing the parameter distribution to minimize a divergence between simulated and real trajectories.
One implementation of ADR is SimOpt [5], which frames the problem of domain refinement as an optimization task over simulation parameters.
The objective is to minimize a divergence metric $\mathcal{D}$ between real-world rollouts τ^{real} and simulated rollouts $\tau^{\text{sim}}(\psi)$:

$$\psi^* = \arg\min_{\psi} \mathcal{D}(\tau^{\text{real}}, \tau^{\text{sim}}(\psi)), \tag{6}$$

where ψ denotes the distribution parameters over randomized simulation domains.
Common choices for $\mathcal{D}$ include metrics that compare aggregated state observations across trajectories [6]. SimOpt typically operates in a loop:

1. Train a policy π_θ in simulation under the current parameter distribution $p(\psi)$.

2. Execute the policy in the real environment and collect real rollouts τ^{real}.

3. Evaluate the discrepancy $\mathcal{D}(\tau^{\text{real}}, \tau^{\text{sim}}(\psi))$.

4. Use an optimizer (we initially focused on CMA-ES) to update $p(\psi)$.

5. Repeat until convergence or satisfactory performance.

In our implementation, we apply SimOpt to the Hopper environment, focusing on the mass distribution across selected links.
In the early stages, we used a custom discrepancy function combining L1 and L2 distances between real and simulated trajectories, smoothed with a Gaussian filter:

$$\text{discrepancy} = w_1 \sum_t \text{smooth}_\sigma \left(\|x_t^{\text{sim}} - x_t^{\text{real}}\|_1 \right) \quad (7)$$

$$+ w_2 \sum_t \text{smooth}_\sigma \left(\|x_t^{\text{sim}} - x_t^{\text{real}}\|_2^2 \right), \quad (8)$$

where $w_1 = 1.0$ and $w_2 = 0.1$ balance the contributions of the L1 and squared L2 norms, and $\sigma = 1.0$ controls the temporal smoothing. This function was chosen for its simplicity and ability to highlight persistent deviations across entire trajectories [5].
Later, we explored more statistically grounded discrepancy metrics such as Maximum Mean Discrepancy (MMD) and the Wasserstein distance [6]. These distances offer improved theoretical properties over norm-based methods, including robustness to noise and better sensitivity in high-dimensional observation spaces.
The MMD between two distributions P and Q with samples $\{x_i\}_{i=1}^n \sim P$ and $\{y_j\}_{j=1}^m \sim Q$ is defined as:

$$\text{MMD}^2(P, Q) = \mathbb{E}[k(x, x')] + \mathbb{E}[k(y, y')] - 2\mathbb{E}[k(x, y)], \quad (9)$$

where $k(\cdot, \cdot)$ is a kernel function. In our implementation, we used a Gaussian kernel with bandwidth selected via the median heuristic.
For the Wasserstein distance, we used a multivariate empirical version computed on matched trajectory observations. Given empirical distributions $P = \{x_i\}_{i=1}^n$ and $Q = \{y_j\}_{j=1}^n$, the 1-Wasserstein distance is approximated as:

$$W(P, Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\| \right], \quad (10)$$

where $\Pi(P, Q)$ is the set of joint distributions (couplings) with marginals P and Q. We computed this using standard linear programming solvers and regularized approximations for computational efficiency.
The discrepancy score directly influenced the optimizer, which we used to adjust the simulation mass parameters. The optimization targeted minimizing the gap between simulated and real rollouts.
Compared to UDR, SimOpt bridges the simulation-reality gap not just via randomization but through targeted domain adaptation grounded in empirical evidence.

4 Experiments and Results

All experiments were carried out using a fixed random seed of 42 and, unless otherwise specified, evaluated on the source environment. A more comprehensive evaluation across multiple seeds would be required to assess the robustness of the results.

4.1 Baseline Policy Gradient Algorithms

We first evaluated several classical policy gradient algorithms. For each one we show a graph of training returns with uncertainty bounds (standard deviation across the 100 samples used for the moving average).

REINFORCE (Vanilla) The simplest policy gradient method estimates the gradient using the return G_t as a baseline-free signal. The update rule is:

$$\nabla J(\theta) = \mathbb{E}_t \left[\gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right], \tag{11}$$

where $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ is the Monte Carlo return.

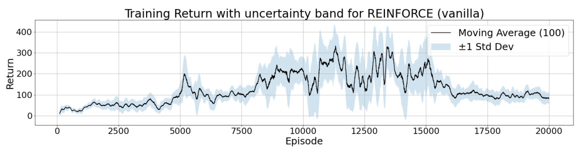

Figure 1: Training return curve REINFORCE_vanilla

REINFORCE + Average Baseline To reduce variance, we subtracted the mean return across the episode from G_t:

$$\nabla J(\theta) \propto \gamma^t (G_t - \bar{G}) \nabla_\theta \log \pi_\theta(a_t|s_t), \tag{12}$$

where $\bar{G}$ is the episode-average return.

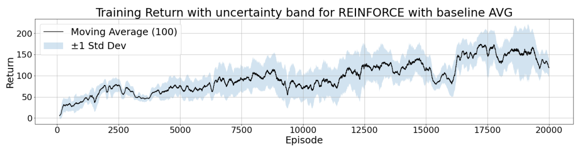

Figure 2: Training return curve REINFORCE_baselineAVG

REINFORCE + Value Network Baseline A learned baseline was introduced by fitting a separate value network $\hat{v}(s_t; w)$ to predict G_t. The advantage function becomes $A_t = G_t - \hat{v}(s_t)$, and the policy and value networks are trained separately. This approach reduces variance more effectively and improves convergence speed.

Actor–Critic This method uses a shared policy and value function to estimate the advantage $A_t = G_t - \hat{v}(s_t)$, training both networks jointly. Since not extensively tuned, it could be further improved.

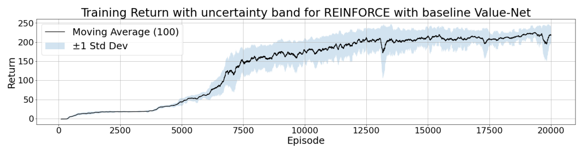

Figure 3: Training return curve REINFORCE_baselineVLN

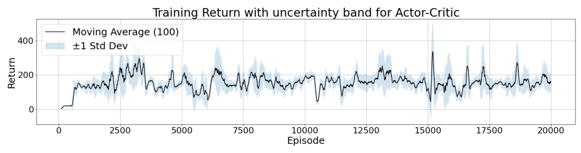

Figure 4: Training return curve Actor-Critic

Table 1 reports mean and standard deviation of the returns across 100 test episodes. From these results we can note how introducing a simple average baseline increased both the return and reduced variability with respect to the simple Reinforce Vanilla. Further improvements were observed with the value-network baseline (226.27 mean return), which offered a more accurate estimate of the expected return, greatly stabilizing training. The Actor–Critic method performed similarly well, with a slightly lower mean return (198.15) but the lowest variance (7.96), indicating very stable learning.

Proximal Policy Optimization (Stable-Baselines3) We employed the default PPO implementation from Stable-Baselines3 without any tuning.

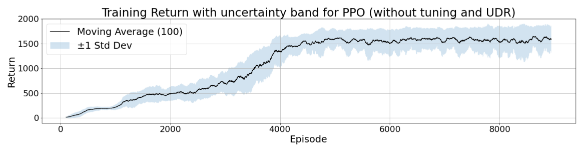

Figure 5: Training return curve default PPO

Empirical Comparison All algorithms were trained over fixed length episodes and tested on 100 evaluation episodes on the source domain. We can see the final performance comparison in Figure 6.

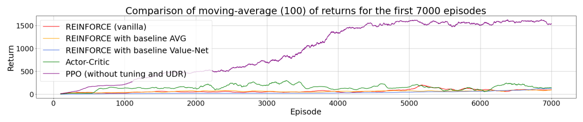

Figure 6: Comparison of smoothed returns across policy gradient methods.

Even without tuning, PPO outperforms the other baselines significantly. This observation motivated us to continue with PPO as the base algorithm for the remainder of our study.

4.1.1 PPO Tuning and Robustness

A grid search over several key hyperparameters was conducted to improve the performance of the Stable-Baselines3's PPO implementation on the source domain. The best configuration was selected based on the highest average return over 5 evaluation episodes per configuration.
Table 2 summarizes the tested and selected hyperparameters. In particular, the learning rate 3×10^{-4} and $\gamma = 0.99$ yielded the best performance.
We evaluated different variants and the final comparison between PPO tuned (source domain), PPO tuned (target domain) and PPO tuned with UDR, is presented in Figure 7. Tuning significantly improved PPO's performance.

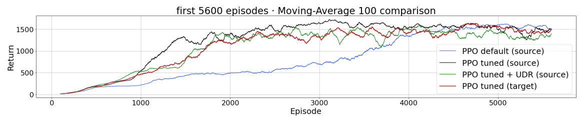

Figure 7: Training returns for PPO variants: tuned PPO on source, on target, and with UDR.

In Table 3 we observe that the `UDR True` configuration performs worse than `UDR False` in the source → target setting (mean return 791.54 vs 942.04), which is counterintuitive given that domain randomization is typically expected to improve generalization. One possible explanation is that the right combination of lower and upper bound has not been found, or that policies trained with UDR require more training time to converge, and the standard 2 million steps may not have been sufficient.

4.2 SimOpt

To further improve policy robustness under domain shifts, we implemented SimOpt. The method operates in a closed optimization loop: at each iteration, we sample a set of candidate simulation parameters (the limb masses of the Hopper agent, keeping the torso mass fixed), train a policy using PPO on the simulated environment, and then evaluate the discrepancy between real and simulated rollouts using the selected metric.
This discrepancy acts as the objective to minimize: we experimented with different statistical discrepancy scores (previously introduced), and supported multiple optimizers selectable via command-line arguments, including CMA-ES, Particle Swarm Optimization (PSO), and Differential Evolution (DE).
After the optimization converges, so when the variance of the parameter distribution becomes sufficiently small, we sample new simulation parameters from the optimized distribution and perform a final training phase using PPO. The resulting policy is then evaluated on both source and target environments. Results in Table 3 show that the performance is dependent on the chosen discrepancy function and optimizer. The success of `score2` (MMD) and `score3` (Wasserstein) in SimOpt suggests that statistical discrepancy measures are more effective than simple norm-based metrics (`score1`). They capture global distributional differences between simulated and real-world trajectories, while norm-based scores may overfit to local mismatches and fail to reflect long-term or structural differences.
Regarding the optimizers, PSO underperforms when combined with `score1` (701.31 on target), but achieves the best results when paired with `score2` or `score3` (1381.82 on target): performance depends on the landscape induced by the discrepancy function and works better when paired with high-dimensional, smooth discrepancy functions, which benefit from its global search capability and fast convergence. CMA-ES, though stable, may be less reactive to small dis-

crepancy reductions, particularly in noisy or non-convex settings. Lastly DE, being mutation-based and highly stochastic, is more variable: effective in some setups (`score2_de`), unreliable in others.
We show the training curves for the three optimizers with the Maximum Mean Discrepancy to further illustrate these differences.

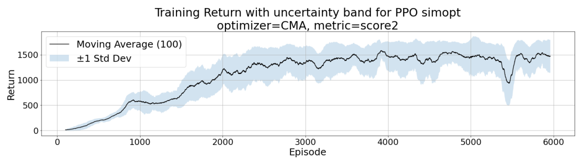

Figure 8: Training curve for SimOpt with Score 2 and CMA

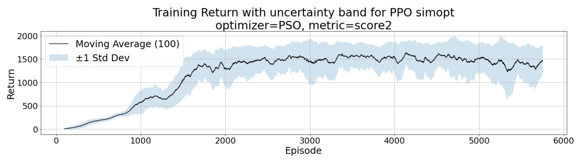

Figure 9: Training curve for SimOpt with Score 2 and PSO

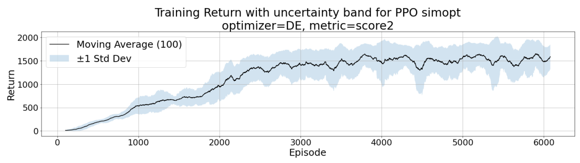

Figure 10: Training curve for SimOpt with Score 2 and DE

5 Conclusion

In this work, we explored how different reinforcement learning algorithms perform in a continuous control setting and how their transferability can be improved through domain randomization techniques.
Among classical policy gradient algorithms, PPO emerged as the best performing.
We then implemented SimOpt to perform adaptive domain randomization and discovered how its effectiveness is closely tied to the choice of discrepancy metric and optimizer. For example, while norm-based scores performed reasonably well with CMA, the combination of MMD or Wasserstein distance with PSO gave the highest returns on the target environment.
These findings suggest that adaptive simulation refinement can outperform fixed-domain randomization, especially under large or structured domain shifts. Future work may explore hybrid optimization strategies, integrate uncertainty estimation in the discrepancy evaluation, or apply SimOpt to more complex agents and tasks.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Second Edition)*. MIT Press, 2018. 1, 2, 3

[2] P. Kormushev, S. Calinon, and D. G. Caldwell, ”Reinforcement learning in robotics: Applications and real-world challenges,” *Robotics*, 2013. 2

[3] S. Höfer et al., ”Perspectives on sim2real transfer for robotics: A summary of the R:SS 2020 workshop,” 2020. 2

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, ”Proximal Policy Optimization Algorithms,” arXiv preprint arXiv:1707.06347, 2017. 3

[5] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, “Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience,” arXiv preprint arXiv:1810.05687, 2019. 3, 4

[6] A. J. Miller, F. Yu, M. Brauckmann, and F. Farshidian, “High-Performance Reinforcement Learning on Spot: Optimizing Simulation Parameters with Distributional Measures,” arXiv preprint arXiv:2504.17857, 2024. 4

Table 1: Performance of Various Algorithms over 100 Episodes as test

Algorithm	**Mean Return**	**Std. Dev.**
REINFORCE (vanilla)	85.24	28.13
REINFORCE + AVG Baseline	123.13	20.56
REINFORCE + Value-Net Baseline	226.27	9.42
Actor–Critic	198.15	7.96

Table 2: PPO and UDR Hyperparameter Tuning Results

Method	**Parameter**	**Tested Values**	**Selected**
PPO Tuning	γ	0.95, 0.98, 0.99	0.99
	Learning Rate	linear_schedule, 10^{-3}, 10^{-4}, 3×10^{-4}, 6×10^{-4}, 9×10^{-4}, 10^{-6}, 10^{-8}	3×10^{-4}
UDR Tuning	Thigh Mass Range	[0.90, 1.10], [0.70, 1.30], [0.50, 1.50]	[0.90, 1.10]
	Leg Mass Range	[0.90, 1.10], [0.70, 1.30], [0.50, 1.50]	[0.90, 1.10]
	Foot Mass Range	[0.90, 1.10], [0.70, 1.30], [0.50, 1.50]	[0.50, 1.50]

Table 3: PPO transfer performance for the basic UDR variants and all SimOpt (`score` $\times$ `optimizer`) combinations.

Transfer	**Variant**	**Mean Return**	**Std. Dev.**
source $\rightarrow$ source	UDR False	1716.83	205.87
	UDR True	1642.50	226.51
	SimOpt score1 with CMA	1762.65	13.00
	SimOpt score1 with DE	1654.24	164.53
	SimOpt score1 with PSO	1372.18	138.93
	SimOpt score2 with CMA	1644.85	4.04
	SimOpt score2 with DE	1808.35	44.00
	SimOpt score2 with PSO	1732.90	60.70
	SimOpt score3 with CMA	1759.80	59.16
	SimOpt score3 with DE	1753.20	4.26
	SimOpt score3 with PSO	1732.90	60.70
source $\rightarrow$ target	UDR False	942.04	53.00
	UDR True	791.54	14.68
	SimOpt score1 with CMA	1067.71	23.32
	SimOpt score1 with DE	1066.51	129.14
	SimOpt score1 with PSO	701.31	5.01
	SimOpt score2 with CMA	653.04	9.51
	SimOpt score2 with DE	535.67	28.56
	SimOpt score2 with PSO	1381.82	100.11
	SimOpt score3 with CMA	807.85	15.90
	SimOpt score3 with DE	637.74	4.63
	SimOpt score3 with PSO	1381.82	100.11
target $\rightarrow$ target	UDR False	1776.58	56.66