A Closer Look at Invalid Action Masking in Policy Gradient Algorithms

Shengyi Huang
College of Computing & Informatics
Drexel University
Philadelphia, PA 19104
sh3397@drexel.edu

Santiago Ontañón *
College of Computing & Informatics
Drexel University
Philadelphia, PA 19104
so367@drexel.edu

Abstract

In recent years, Deep Reinforcement Learning (DRL) algorithms have achieved state-of-the-art performance in many challenging strategy games. Because these games have complicated rules, an action sampled from the full discrete action space will typically be invalid. The usual approach to deal with this problem in policy gradient algorithms is to "mask out" invalid actions and just sample from the set of valid actions. The implications of this process, however, remain under-investigated. In this paper, we show that the standard working mechanism of invalid action masking corresponds to valid policy gradient updates. More interestingly, it works by applying a *state-dependent differentiable function* during the calculation of action probability distribution. Additionally, we show its critical importance to the performance of policy gradient algorithms. Specifically, our experiments show that invalid action masking scales well when the space of invalid actions is large, while the common approach of giving negative rewards for invalid actions will fail. Finally, we provide further insights by evaluating different action masking regimes, such as removing masking after an agent has been trained using masking.

1 Introduction

Deep Reinforcement Learning (DRL) algorithms have yielded state-of-the-art game playing agents in challenging domains such as Real-time Strategy (RTS) games [21, 20] and Multiplayer Online Battle Arena (MOBA) games [1, 23]. Because these games have complicated rules, the discrete action spaces of different states usually have different sizes. That is, one state might have 5 valid actions and another state might have 7 valid actions. To formulate these games as a standard reinforcement learning problem with a singular action set, previous work combine these discrete action spaces to a *full discrete action space* that contains available actions of all states [21, 1, 23]. Although such full discrete action space makes it easier to apply DRL algorithms, one issue is that an action sampled from this full discrete action space could be invalid for some game states, and this action will have to be discarded. To make matters worse, some games have extremely large full discrete action spaces and an action sampled will typically be invalid. As an example, the full discrete action space of Dota 2 has 1,837,080 dimensions [1], and an action sampled might be to buy an item when there is not enough gold. To avoid repeatedly sampling invalid actions in full discrete action spaces, recent work applies policy gradient algorithm in conjunction with an technique known as invalid action masking, which "masks out" invalid actions and then just sample from those actions that are valid [21, 1, 23]. To the best of our knowledge, however, the theoretical foundations of invalid action masking have not been studied and its empirical effect is under-investigated.

*Currently at Google

Preprint. Under review.

In this paper, we take a closer look at invalid action masking, pointing out the gradient produced by invalid action masking corresponds to a valid policy gradient. More interestingly, we show that in fact, invalid action masking can be seen as applying a *state-dependent differentiable function* during the calculation of action probability distribution, to produce a behavior policy. Next, we design experiments to compare the performance of *invalid action masking* versus *invalid action penalty*, which is a common approach that gives negative rewards for invalid actions so that the agent learns to maximize reward by not executing any invalid actions. We empirically show that, when the space of invalid actions grows, invalid action masking scales well and the agent solves our desired task while invalid action penalty struggles to explore even the very first reward. Then, we design experiments to answer two questions: (1) What happens if we remove the invalid action mask once the agent was trained with the mask? (2) What is the agent's performance when we implement the invalid action masking naively by sampling the action from the masked action probability distribution but updating the policy gradient using the unmasked action probability distribution? Finally, we made our source code available at GitHub for the purpose of reproducibility[2]

2 Background

We consider the Reinforcement Learning problem in a Markov Decision Process (MDP) denoted as $(S, A, P, \rho_0, r, \gamma, T)$, where S is the state space, A is the discrete action space, $P : S \times A \times S \to [0, 1]$ is the state transition probability, $\rho_0 : S \to [0, 1]$ is the the initial state distribution, $r : S \times A \to \mathbb{R}$ is the reward function, γ is the discount factor, and T is the maximum episode length. A stochastic policy $\pi_\theta : S \times A \to [0, 1]$, parameterized by a parameter vector θ, assigns a probability value to an action given a state. The goal is to maximize the expected discounted return:

$$J = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \gamma^t r_t \right], \text{ where } \tau \text{ is the trajectory } (s_0, a_0, r_0, s_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1})$$
$$\text{and } s_0 \sim \rho_0, s_t \sim P(\cdot|s_{t-1}, a_{t-1}), a_t \sim \pi_\theta(\cdot|s_t), r_t = r(s_t, a_t)$$

The notation $P(\cdot|s_{t-1}, a_{t-1})$ represents the states transition distribution given the previous state s_{t-1} and action a_{t-1}, and the notation $s_t \sim P(\cdot|s_{t-1}, a_{t-1})$ represents that the state s_t visited at time t is sampled from $P(\cdot|s_{t-1}, a_{t-1})$. Similarly, $\pi_\theta(\cdot|s_t)$ represents the action distribution given state s_t, and $a_t \sim \pi_\theta(\cdot|s_t)$ means the action a_t at time t is sampled from $\pi_\theta(\cdot|s_t)$.

Policy Gradient Algorithms. The core idea behind policy gradient algorithms is to obtain the *policy gradient* $\nabla_\theta J$ of the expected discounted return with respect to the policy parameter θ. Doing gradient ascent $\theta = \theta + \nabla_\theta J$ therefore maximizes the expected discounted reward. Earlier work proposed the following policy gradient estimate to the objective J [19, 18]:

$$g_{\text{policy}} = \mathbb{E}_\tau \left[\nabla_\theta \log \pi_\theta(a_\tau|s_\tau) G_\tau \right] = \mathbb{E}_\tau \left[\nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) G_t \right], \tag{1}$$

where $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ denotes the discounted return following time t.

3 Invalid Action Masking

Invalid action masking is a common technique implemented to avoid repeatedly generating invalid actions in large discrete action spaces [21, 1, 23]. To the best of our knowledge, there is no literature providing detailed descriptions of the implementation of invalid action masking. Existing work [21, 1] seems to treat invalid action masking as an auxiliary detail, usually describing it using only a few sentences. Additionally, there is no literature providing theoretical justification to explain why it works with policy gradient algorithms. In this section, we examine how invalid action masking is implemented and prove it indeed corresponds to valid policy gradient updates [19]. More interestingly, we show it works by applying a *state-dependent differentiable function* during the calculation of action probability distribution.

First, let us see how a discrete action is typically generated through policy gradient algorithms. Most policy gradient algorithms employ a neural network to represent the policy, which usually

[2] https://github.com/vwxyzjn/invalid-action-masking

outputs unnormalized scores (logits) and then converts them into an action probability distribution using a softmax operation or equivalent, which is the framework we will assume in the rest of the paper. For illustration purposes, consider an MDP with the action set $A = \{a_0, a_1, a_2, a_3\}$ and $S = \{s_0, s_1\}$, where the MDP reaches the terminal state s_1 immediately after an action is taken in the initial state s_0 and the reward is always +1. Further, consider a policy π_θ parameterized by $\theta = [l_0, l_1, l_2, l_3] = [1.0, 1.0, 1.0, 1.0]$ that, for the sake of this example, directly produces θ as the output logits. Then in s_0 we have:

$$\begin{aligned}\pi_\theta(\cdot|s_0) &= [\pi_\theta(a_0|s_0), \pi_\theta(a_1|s_0), \pi_\theta(a_2|s_0), \pi_\theta(a_3|s_0)] = \text{softmax}([l_0, l_1, l_2, l_3]) \\ &= [0.25, 0.25, 0.25, 0.25], \quad \pi_\theta(a_i|s_0) = \frac{\exp(l_i)}{\sum_j \exp(l_j)}\end{aligned} \tag{2}$$

At this point, regular policy gradient algorithms will sample an action from $\pi_\theta(\cdot|s_0)$. Suppose a_0 is sampled from $\pi_\theta(\cdot|s_0)$, and the policy gradient can be calculated as follows:

$$\begin{aligned}g_{\text{policy}} &= \mathbb{E}_\tau \left[\nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) G_t \right] \\ &= \nabla_\theta \log \pi_\theta(a_0|s_0) G_0 \\ &= [0.75, -0.25, -0.25, -0.25]\end{aligned} \qquad (\nabla_\theta \log \text{softmax}(\theta)_j)_i = \begin{cases} (1 - \frac{\exp(l_j)}{\sum_j \exp(l_j)}) & \text{if } i = j \\ \frac{-\exp(l_j)}{\sum_j \exp(l_j)} & \text{otherwise} \end{cases}$$

Now suppose a_2 is invalid for state s_0, and the only valid actions are a_0, a_1, a_3. Invalid action masking helps to avoid sampling invalid actions by "masking out" the logits corresponding to the invalid actions. This is usually accomplished by replacing the logits of the actions to be masked by a large negative number M (e.g. $M = -1 \times 10^8$). Let us use inv_s to denote this masking process and we can calculate the re-normalized probability distribution $\pi'_\theta(\cdot|s_0)$ as the following:

$$\pi'_\theta(\cdot|s_0) = \text{softmax}(\text{inv}_s([l_0, l_1, l_2, l_3])) \tag{3}$$

$$\begin{aligned}&= \text{softmax}([l_0, l_1, M, l_3]) = [\pi'_\theta(a_0|s_0), \pi'_\theta(a_1|s_0), \epsilon, \pi'_\theta(a_3|s_0)] \\ &= [0.33, 0.33, 0.0000, 0.33]\end{aligned} \tag{4}$$

where ϵ is the resulting probability of the masked invalid action, which should be a small number. If M is chosen to be sufficiently negative, the probability of choosing the masked invalid action a_2 will be virtually zero. After finishing the episode, the policy is updated according to the following gradient, which we refer to as the *invalid action policy gradient*.

$$\begin{aligned}g_{\text{invalid action policy}} &= \mathbb{E}_\tau \left[\nabla_\theta \sum_{t=0}^{T-1} \log \pi'_\theta(a_t|s_t) G_t \right] \\ &= \nabla_\theta \log \pi'_\theta(a_0|s_0) G_0 = [0.67, -0.33, 0.0000, -0.33]\end{aligned} \tag{5}$$

This example highlights that invalid action masking appears to do more than just "renormalizing the probability distributiuon"; it in fact makes the gradient corresponding to the logits of the invalid action to zero.

3.1 Invalid Action Masking Produces a Valid Policy Gradient

The action selection process is affected by a process that seems external to π_θ that calculates the mask. It is therefore natural to wonder how does the policy gradient theorem [19] apply. As a matter of fact, our analysis shows that the process of invalid action masking can be considered as a state-dependent differentiable function applied for the calculation of π'_θ, and therefore $g_{\text{invalid action policy}}$ can be considered as a policy gradient update for π'_θ.

Proposition 1. *$g_{\text{invalid action policy}}$ is the policy gradient of policy π'_θ.*

Proof. Let $s \in S$ to be arbitrary and consider the process of invalid action masking as a differentiable function inv_s to be applied to the logits $l(s)$ outputted by policy π_θ given state s. Then we have:

$$\pi'_\theta(\cdot|s_t) = \text{softmax}(\text{inv}_s(l(s)))$$

$$\text{inv}_s(l(s))_i = \begin{cases} l_i & \text{if } a_i \text{ is valid in } s \\ M & \text{otherwise} \end{cases}$$

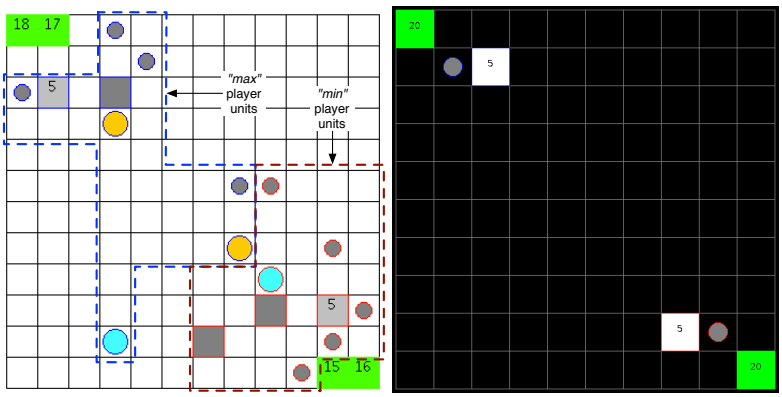


Figure 1: On the left is a screenshot of μRTS . Square units are "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units are "workers" (small, dark grey) and military units (large, yellow or light blue), and on the right is the 10×10 map we used to train agents to harvest resources. The agents could control units at the top left, and the units in the bottom left will remain stationary.

Clearly, inv_{s_t} applies either an identity function or a constant function for elements in the logits. Since these two kinds of functions are differentiable, inv_s is differentiable. Therefore, π'_θ is differentiable to its parameters θ. That is, $\frac{\partial \pi'_\theta(a|s)}{\partial \theta}$ exists for all $a \in A, s \in S$, which satisfies the assumption of policy gradient theorem [19]. Hence, $g_{\text{invalid action policy}}$ is the policy gradient of policy π'_θ. □

Although Proposition 1 suggests invalid action masking is theoretically supported by policy gradient theorem [19], note that inv_s is a state-dependent differentiable function. That is, given a vector x of length $|A|$ and two states s, s' with different number of invalid actions available in these states, $\text{inv}_s(x) \neq \text{inv}_{s'}(x)$.

4 Experimental Setup

In the remaining of this paper, we provide a series of empirical results showing the practical implications of invalid action masking.

4.1 Evaluation Environment

We use μRTS[3] as our testbed, which is a minimalistic RTS game maintaining the core features that make RTS games challenging from an AI point of view: simultaneous and durative actions, large branching factors and real-time decision making. A screenshot of the game can be found in Figure 1. It is the perfect testbed for our experiments because the action space in μRTS grows combinatorially and so does the number of invalid actions that could be generated by the DRL agent. We now present the technical details of the environment for our experiments.

- **Observation Space.** Given a map of size $h \times w$, the observation is a tensor of shape (h, w, n_f), where n_f is a number of feature planes that have binary values. The observation space used in this paper uses 27 feature planes as shown in Table 3 in the Appendices, similar to previous work in μRTS [17, 22, 8]. A feature plane can be thought of as a concatenation of multiple one-hot encoded features. As an example, if there is a worker with hit points equal to 1, not carrying any resources, owner being Player 1, and currently not executing any actions, then the one-hot encoding features will look like the following:

$$[0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0], [1, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0]$$

[3] https://github.com/santiontanon/microrts

Table 1: The action components and their descriptions.

Action Components	Range	Description
Source Unit	$[0, h \times w - 1]$	the location of unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 5]$	resource, base, barrack, worker, light, heavy, ranged
Attack Target Unit	$[0, h \times w - 1]$	the location of unit that will be attacked

The 27 values of each feature plane for the position in the map of such worker will thus be:

$$[0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]$$

- **Action Space.** Given a map of size $h \times w$, the action is an 8-dimensional vector of discrete values as specified in Table 1. The action space is designed similar to the action space formulation by Hausknecht, et al., [7]. The first component of the action vector represents the unit in the map to issue actions to, the second is the action type, and the rest of components represent the different parameters different action types can take. Depending on which action type is selected, the game engine will use the corresponding parameters to execute the action. Additional details on how to interface actions with μRTS internally can be found at Appendix B.
- **Rewards.** We are evaluating our agents on the simple task of harvesting resources as fast as they can for Player 1 who controls units at the top left of the map. A $+1$ reward is given when a worker harvests a resource, and another $+1$ is received once the worker returns the resource to a base.
- **Termination Condition.** We set the maximum game length to be of 200 time steps, but the game could be terminated earlier if the all of the resources in the map are harvested first.

Notice that the space of invalid actions becomes significantly larger in larger maps. This is because the range of the first and last discrete values in the action space, corresponding to *Source Unit* and *Attack Target Unit* selection, grows linearly with the size of the map. To illustrate, in our experiments, there are usually only two units that can be selected as the *Source Unit* (the base and the worker). Although it is possible to produce more units or buildings to be selected, the production behavior has no contribution to reward and therefore is generally not learned by the agent. Note the range of *Source Unit* is $4 \times 4 = 16$ and $24 \times 24 = 576$, in maps of size 4×4 and 24×24, respectively. Selecting a valid *Source Unit* at random has a probability of $2/16 = 0.125$ in the 4×4 map and $2/576 = 0.0034$ in the 24×24 map. With such action space, we can examine the scalability of invalid action masking.

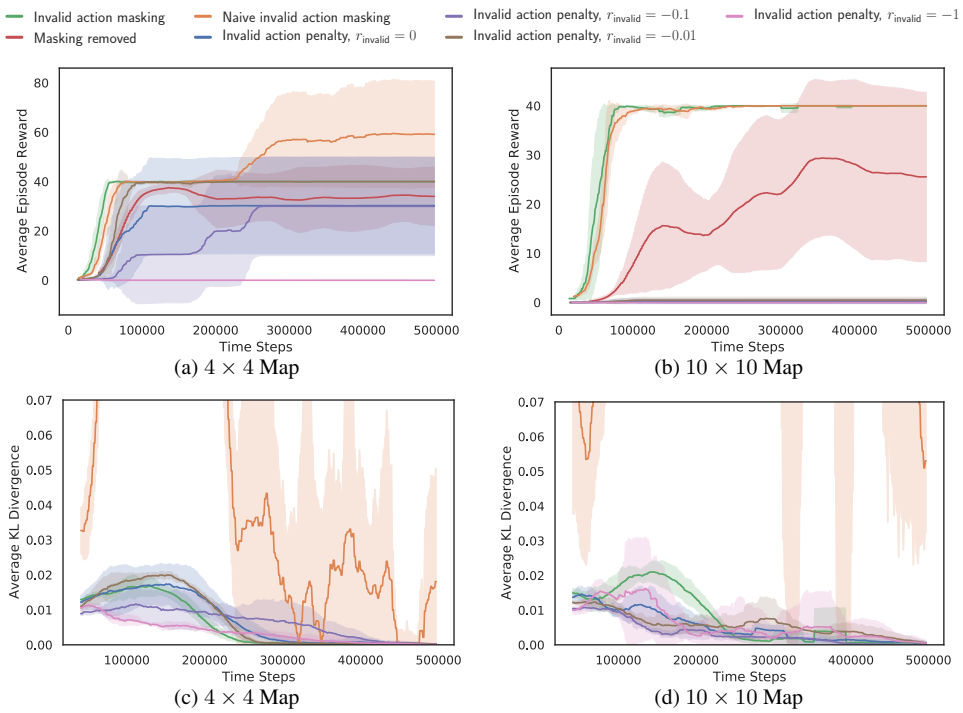
4.2 Training Algorithm

We use Proximal Policy Optimization [16] as the DRL algorithm to train our agents. The details of the implementation and neural network architecture, hyperparameters can be found in Appendix A.

4.3 Strategies to Handle Invalid Actions

To examine the empirical importance of invalid action masking, we compare the following four strategies to handle invalid actions.

1. **Invalid action penalty.** Every time the agent issues an invalid action, the game environment adds a non-positive reward $r_{\text{invalid}} \leq 0$ to the reward produced by the current time step. This technique is standard in previous work [4]. We experiment with $r_{\text{invalid}} \in \{0, -0.01, -0.1, -1\}$, respectively, to study the effect of the different scales on the negative reward.
2. **Invalid action masking.** At each time step t, the agent receives a mask on the *Source Unit* and *Attack Target Unit* features such that only valid units can be selected and targeted. Note that in our experiments, invalid actions still could be sampled because the agent could still



(a) 4×4 Map (b) 10×10 Map

(c) 4×4 Map (d) 10×10 Map

Figure 2: (a) and (b) show the learning curves of agents with different strategies to handle invalid actions. The x-axis shows the number of game steps and y-axis shows the average episode reward gathered. (c) and (d) have the x-axis showing the number of game steps and y-axis showing the average Kullback–Leibler (KL) divergence between the target and current policy of PPO. Shaded area represents one standard deviation of the data over 4 random seeds. Curves are smoothed for readability. For results in other maps, see Figure 3 in the Appendices.

select incorrect parameters for the current action type. We didn't provide a feature-complete invalid action mask for simplicity, as the mask on *Source Unit* and *Attack Target Unit* already significantly reduce the action space.

3. **Naive invalid action masking.** At each time step t, the agent receives the same mask on the *Source Unit* and *Attack Target Unit* as described for invalid action masking. The action shall still be sampled according to the re-normalized probability calculated in Equation 4, which ensures no invalid actions could be sampled, but the gradient is updated according to probability calculated in Equation 2. We call this implementation *naive invalid action masking* because its gradient does not replace the gradient of the logits corresponds to invalid actions with zero.
4. **Masking removed.** At each time step t, the agent receives the same mask on the *Source Unit* and *Attack Target Unit* as described for invalid action masking, and trains in the same way as the agent trained under invalid action masking. However, we then evaluate the agent without providing the mask. In other words, in this scenario, we evaluate what happens if we train with a mask, but then perform without it.

We evaluate the agent's performance in maps of sizes 4×4, 10×10, 16×16, and 24×24. All maps have one base and one worker for each player, and each worker is located near the resources.

4.4 Evaluation Metrics

We used the following metrics to measure the performance of the agents in our experiments: r_{episode} is the average episode reward over last 10 episodes. a_{null} is the average number of actions that select a *Source Unit* that is not valid over last 10 episodes. $a_{\text{busy}}$is the average number of actions that select

Table 2: Results averaged over 4 random seeds. The symbol "-" means "not applicable". Higher is better for r_{episode} and lower is better for a_{null}, a_{busy}, a_{owner}, t_{solve}, and t_{first}.

Strategies	Map size	r_{invalid}	r_{episode}	a_{null}	a_{busy}	a_{owner}	t_{solve}	t_{first}
Invalid action penalty	4×4	-1.00	0.00	0.00	0.00	0.00	-	0.53%
		-0.10	30.00	0.02	0.00	0.00	50.94%	0.52%
		-0.01	**40.00**	0.02	0.00	0.00	14.32%	0.51%
		0.00	30.25	2.17	0.22	2.70	36.00%	0.60%
	10×10	-1.00	0.00	0.00	0.00	0.00	-	3.43%
		-0.10	0.00	0.00	0.00	0.00	-	2.18%
		-0.01	0.50	0.00	0.00	0.00	-	1.57%
		0.00	0.25	90.10	0.00	102.95	-	3.41%
	16×16	-1.00	0.25	0.00	0.00	0.00	-	0.44%
		-0.10	0.75	0.00	0.00	0.00	-	0.44%
		-0.01	1.00	0.02	0.00	0.00	-	0.44%
		0.00	1.00	184.68	0.00	2.53	-	0.40%
	24×24	-1.00	0.00	49.72	0.00	0.02	-	1.40%
		-0.10	0.25	0.00	0.00	0.00	-	1.40%
		-0.01	0.50	0.00	0.00	0.00	-	1.92%
		0.00	0.50	197.68	0.00	0.90	-	1.83%
Invalid action masking	04x04	-	**40.00**	-	-	-	8.67%	**0.07%**
	10x10	-	**40.00**	-	-	-	11.13%	**0.05%**
	16x16	-	**40.00**	-	-	-	11.47%	**0.08%**
	24x24	-	**40.00**	-	-	-	18.38%	**0.07%**
Masking removed	04x04	-	33.53	63.57	0.00	17.57	76.42%	-
	10x10	-	25.93	128.76	0.00	7.75	94.15%	-
	16x16	-	17.32	165.12	0.00	0.52	-	-
	24x24	-	17.37	150.06	0.00	0.94	-	-
Naive invalid action masking	4×4	-	**59.61**	-	-	-	11.74%	**0.07%**
	10×10	-	**40.00**	-	-	-	13.97%	**0.05%**
	16×16	-	**40.00**	-	-	-	30.59%	**0.10%**
	24×24	-	**38.50**	-	-	-	49.14%	**0.07%**

a *Source Unit* that is already busy executing other actions over last 10 episodes. a_{owner} is the average number of actions that select a *Source Unit* that does not belong to Player 1 over last 10 episodes. t_{solve} is the percentage of total training time steps that it takes for the agents' moving average episode reward of the last 10 episodes to exceed 40. t_{first} is the percentage of total training time step that it takes for the agent to receive the first positive reward.

4.5 Evaluation Results

We report the results in Figure 2 and in Table 2. Here we present a list of important observations:

Invalid action masking scales well. Invalid action masking is shown to scale well as the number of invalid actions increases; t_{solve} is roughly 12% and very similar across different map sizes. In addition, the t_{first} for invalid action masking is not only the lowest across all experiments (only taking about $0.05 - 0.08\%$ of the total time steps), but also consistent against different map sizes. This would mean the agent was able to find the first reward very quickly regardless of the map sizes.

Invalid action penalty does not scale. Invalid action penalty is able to achieve good results in 4×4 maps, but it does not scale to larger maps. As the space of invalid action gets larger, sometimes it struggles to even find the very first reward. E.g. in the 10×10 map, agents trained with invalid action penalty with $r_{\text{invalid}} = -0.01$ spent a 3.43% of the entire training time just discovering the first reward, while agents trained with invalid action masking take roughly 0.06% of the time in all maps. In addition, the hyper-parameter r_{invalid} can be difficult to tune. Although having a negative r_{invalid} did encourage the agents not to execute any invalid actions (e.g. a_{null}, a_{busy}, a_{owner} are usually very close to zero for these agents), setting $r_{\text{invalid}} = -1$ seems to have an adverse effect of discouraging exploration by the agent, therefore achieving consistently the worst performance across maps.

KL Explosion of naive invalid action masking. According to Table 2, the r_{episode} of naive invalid action masking is the best across almost all maps. In the 4×4 map, the agent trained with naive invalid action masking even learns to travel to the other side of the map to harvest additional resources. However, naive invalid action masking has two main issues: 1) As shown in Figure 2c, 2d, the average Kullback–Leibler (KL) divergence [12] between the target and current policy of PPO for naive invalid action masking is significantly higher than that of any other experiments. Since the policy changes so drastically between policy updates, the performance of naive invalid action masking might suffer when dealing with more challenging tasks. 2) As shown in Table 2, the t_{solve} of naive invalid action masking is more volatile and sensitive to the map sizes. In the 24×24 map, for example, the agents trained with naive invalid action masking take 49.14% of the entire training time to converge. In comparison, agents trained with invalid action masking exhibit a consistent $t_{\text{solve}} \approx 12\%$ in all maps.

Masking removed still behaves to some extent. As shown in Figures 2a, 2b, masking removed is still able to perform well to a certain degree. As the map size gets larger, its performance degrades and starts to execute more invalid actions by, most prominently, selecting an invalid *Source Unit*. Nevertheless, its performance is significantly better than that of the agents trained with invalid action penalty even though they are evaluated without the use of invalid action masking. This shows that the agents trained with invalid action masking can, to some extent, still produce useful behavior when the invalid action masking can no longer be provided.

5 Related Work

There have been other approaches of dealing with invalid actions. Dulac-Arnold, Evans, et al. [5] suggest to embed discrete action spaces into a continuous action space, use nearest-neighbor methods to locate the nearest valid actions. In the field of games with natural language, others propose to trains an Action Elimination Network (AEN) [24] to reduce the action set.

The purpose of avoiding executing invalid actions arguably is to boost the exploration efficiency. Some less related work achieves this purpose by reducing the full discrete action space to a simpler action space. Kanervisto, et al. [10] describes this kind of work as "action space shaping", which typical involves 1) action removals (e.g. Minecraft RL environment removes non useful actions such as "sneak" [9]), and 2) discretization of continuous action space (e.g. the famous CartPole-v0 environment discretize the continuous forces to be applied to the cart [2]). Although a well-shaped action space could help the agent efficiently explore and learn a useful policy, action space shaping is shown to be potentially difficult to tune and some times detrimental in helping the agent solve the desired tasks [5].

Lastly, Kanervisto, et al. [10] and Ye, et al. [23] provide ablation studies to show invalid action masking could be important to the performance of agents, but they do not study the empirical effect of invalid action masking as the space of invalid action grows, which is addressed in this paper.

6 Conclusions

In this paper, we examined the technique of invalid action masking, which is a technique commonly implemented in policy gradient algorithms to avoid executing invalid actions especially in domains where the action space is large. Our work shows that: 1) the gradient produced by invalid action masking is a valid policy gradient, 2) it works by applying a *state-dependent differentiable function* during the calculation of action probability distribution, 3) invalid action masking empirically scales well as the space of invalid action gets larger; in comparison, the common technique of giving a negative reward when an invalid action is issued fails to scale, sometimes struggling to find even the first reward in our environment, 4) the agent trained with invalid action masking was still able to produce useful behaviors with masking removed.

For future work, we hope to provide a better theoretical framework to explain the working mechanism of invalid action masking. In particular we plan to study whether the use of action masking has any consequences on the convergence guarantees of the learning algorithms, and design methods that could exploit masks during training but do not rely on them, in order to apply them to application domains where masks might be available during trainig, but not when deployed in the real world.

References

[1] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pond'e de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[3] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[4] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[5] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[6] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2019.

[7] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015.

[8] Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in μrts. 2019.

[9] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 4246–4247. AAAI Press, 2016.

[10] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. *arXiv preprint arXiv:2004.00980*, 2020.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[13] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[14] Marc'Aurelio Ranzato, Fu Jie Huang, Y-Lan Boureau, and Yann LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *2007 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2007.

[15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[17] Marius Stanescu, Nicolas A. Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. 09 2016.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[20] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[21] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[22] Zuozhi Yang and Santiago Ontañón. Learning map-independent evaluation functions for real-time strategy games. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7, 2018.

[23] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, et al. Mastering complex control in moba games with deep reinforcement learning. *arXiv preprint arXiv:1912.09729*, 2019.

[24] Tom Zahavy, Matan Haroush, Nadav Merlis, Daniel J Mankowitz, and Shie Mannor. Learn what not to learn: Action elimination with deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3562–3573, 2018.

Appendices

A Details on the Training Algorithm Proximal Policy Optimization

The DRL algorithm that we use to train the agent is Proximal Policy Optimization (PPO) [16], one of the state of the art algorithms available. There are two important details regarding our PPO implementation that warrants explanation, as those details are not elaborated in the original paper. The first detail concerns how to generate an action in the `MultiDiscrete` action space as defined in the OpenAI Gym environment [2] of gym-microrts [8], while the second detail is about the various code-level optimizations utilized to augment performance. As pointed out by Engstrom, Ilyas, et al. [6], such code-level optimizations could be critical to the performance of PPO.

A.1 Multi Discrete Action Generation

To perform an action a_t in μRTS, according to Table 1, we have to select a Source Unit, Action Type, and its corresponding action parameters. So in total, there are $hw \times 6 \times 4 \times 4 \times 4 \times 4 \times 6 \times hw = 9216(hw)^2$ number of possible discrete actions (including invalid ones), which grows exponentially as we increase the map size. If we apply the PPO directly to this discrete action space, it would be computationally expensive to generate the distribution for $9216(hw)^2$ possible actions. To simplify this combinatorial action space, `openai/baselines` [3] library proposes an idea to consider this discrete action to be composed from some smaller *independent* discrete actions. Namely, a_t is composed of smaller actions

$$a_t^{\text{Source Unit}}, a_t^{\text{Action Type}}, a_t^{\text{Move Parameter}}, a_t^{\text{Harvest Parameter}},$$
$$a_t^{\text{Return Parameter}}, a_t^{\text{Produce Direction Parameter}}, a_t^{\text{Produce Type Parameter}}, a_t^{\text{Attack Target Unit}}$$

And the policy gradient is updated in the following way (without considering the PPO's clipping for simplicity)

$$\begin{aligned}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t &= \sum_{t=0}^{T-1} \nabla_\theta \left(\sum_{d \in D} \log \pi_\theta(a_t^d|s_t) \right) G_t \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \left(\prod_{d \in D} \pi_\theta(a_t^d|s_t) \right) G_t\end{aligned}$$

$$\begin{aligned}D = \{&\text{Source Unit}, \text{Action Type}, \text{Move Parameter}, \text{Harvest Parameter}, \text{Return Parameter}, \\ &\text{Produce Direction Parameter}, \text{Produce Type Parameter}, \text{Attack Target Unit}, \}\end{aligned}$$

Implementation wise, for each Action Component of range $[0, x-1]$, the logits of the corresponding shape x is generated, which we call Action Component logits, and each a_t^d is sampled from this Action Component logits. Because of this idea, the algorithm now only has to generate $hw + 6 + 4 + 4 + 4 + 4 + 6 + hw = 2hw + 36$ number of logits, which is significantly less than $9216(hw)^2$. To the best of our knowledge, this approach of handling large multi discrete action space is only mentioned by Kanervisto et, al [10].

A.2 Code-level Optimizations

Here is a list of code-level optimizations utilized in this experiments. For each of these optimizations, we include a footnote directing the readers to the files in the *openai/baselines* [3] that implements these optimization.

1. **Normalization of Advantages**[4]: After calculating the advantages based on GAE, the advantages vector is normalized by subtracting its mean and divided by its standard deviation.
2. **Normalization of Observation**[5]: The observation is pre-processed before feeding to the PPO agent. The raw observation was normalized by subtracting its running mean and divided by its variance; then the raw observation is clipped to a range, usually $[-10, 10]$.

[4] `https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L139`

[5] `https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/common/vec_env/vec_normalize.py#L4`

3. **Rewards Scaling**[6]: Similarly, the reward is pre-processed by dividing the running variance of the discounted the returns, following by clipping it to a range, usually $[-10, 10]$.
4. **Value Function Loss Clipping**[7]**:** The PPO implementation of *openai/baselines* clips the value function loss in a manner that is similar to the PPO's clipped surrogate objective:

$$V_{loss} = \max\left[\left(V_{\theta_t} - V_{targ}\right)^2, \left(V_{\theta_{t-1}} + \text{clip}\left(V_{\theta_t} - V_{\theta_{t-1}}, -\varepsilon, \varepsilon\right)\right)^2\right]$$

where V_{targ} is calculated by adding $V_{\theta_{t-1}}$ and the A calculated by General Advantage Estimation[15].
5. **Adam Learning Rate Annealing**[8]**:** The Adam [11] optimizer's learning rate is set to decay as the number of timesteps agent trained increase.
6. **Mini-batch updates**[9]**:** The PPO implementation of the *openai/baselines* also uses mini-batches to compute the gradient and update the policy instead of the whole batch data such as in *open/spinningup*. The mini-batch sampling scheme, however, still makes sure that every transition is sampled only once, and that the all the transitions sampled are actually for the network update.
7. **Global Gradient Clipping**[10]**:** For each update iteration in an epoch, the gradients of the policy and value network are clipped so that the "global ℓ_2 norm" (i.e. the norm of the concatenated gradients of all parameters) does not exceed 0.5.
8. **Orthogonal Initialization of weights**[11]**:** The weights and biases of fully connected layers use with orthogonal initialization scheme with different scaling. For our experiments, however, we always use the scaling of 1 for historical reasons.

B Additional Details on the μRTS Environment Setup

Each action in μRTS takes some internal game time, measured in ticks, for the action to be completed. *gym-microrts* [8] sets the time of performing harvest action, return action, and move action to be 10 game ticks. Once an action is issued to a particular unit, the unit would be considered as a "busy" unit and would therefore no longer be able to execute any actions until its current action is finished. To prevent the DRL algorithms from repeatedly issuing actions to "busy" units, *gym-microrts* allows performing frame skipping of 9 frames such that from the agent's perspective, once it executes the harvest action, return action, or move action given the current observation, those actions would be finished in the next observation. Such frame skipping is used for all of our experiments.

C Reproducibility

It is important to for the research work to be reproducible. We now present the list of hyperparameters used in Table 4 and the list of neural network architecture in Table 5. In addition, we provide the source code to reproduce our experiments at GitHub[12]

[6] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/common/vec_env/vec_normalize.py#L4

[7] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L68-L75

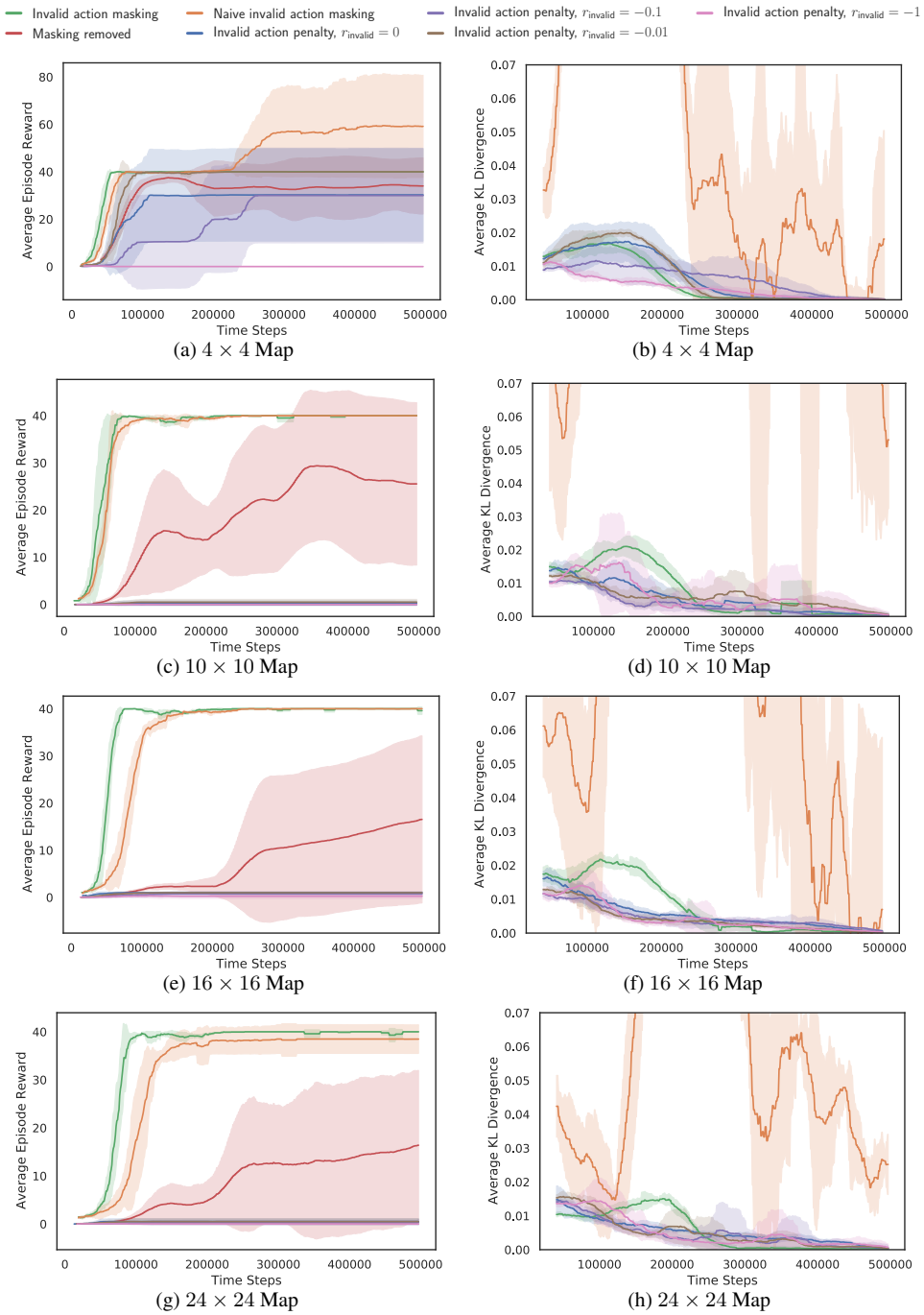
[8] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/ppo2.py#L135

[9] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/ppo2.py#L160-L162

[10] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L107

[11] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/a2c/utils.py#L58

[12] https://github.com/neurips2020submission/invalid-action-masking



(a) 4×4 Map (b) 4×4 Map

(c) 10×10 Map (d) 10×10 Map

(e) 16×16 Map (f) 16×16 Map

(g) 24×24 Map (h) 24×24 Map

Figure 3: The left column show the learning curves of agents with different strategies to handle invalid actions. The x-axis shows the number of game steps and y-axis shows the average episode reward gathered. The right column have the x-axis showing the number of game steps and y-axis showing the average Kullback–Leibler (KL) divergence between the target and current policy of PPO. Shaded area represents one standard deviation of the data over 4 random seeds. Curves are smoothed for readability.

Table 3: The list of feature maps and their descriptions.

Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack, worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack

Table 4: The list of experiment parameters and their values.

Parameter Names	Parameter Values
Total Time steps	500,000 time steps
γ (Discount Factor)	0.99
λ (for GAE)	0.97
ε (PPO's Clipping Coefficient)	0.2
η (Entropy Regularization Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5
K (Number of PPO Update Iteration Per Epoch)	10
α_π Policy's Learning Rate	0.0003
α_v Value Function's Learning Rate	0.0003

Table 5: Neural Network Architecture. To explain the notation, let us provide detailed description of the architecture used in 24×24 map as an example. The input to the neural network is a tensor of shape $(24, 24, 27)$. The first hidden layer convolves 16 3×3 filters with stride 1 with the input tensor followed by a 2×2 max pooling layer [14] and applies a rectifier nonlinearity [13]. The second hidden layer similarly convolves 32 2×2 filters with stride 1 followed by a 2×2 max pooling layer and applies a rectifier nonlinearity. The final hidden layer is a fully connected linear layer consisting of 128 rectifier units. The output layer is a fully connected linear layer with $2hw + 36 = 1188$ number of output.

4×4	10×10
Conv2d(27, 16, kernel_size=2,), MaxPool2d(1), ReLU() Flatten() Linear(144, 128), ReLU(), Linear(128, 68)	Conv2d(27, 16, kernel_size=3,), MaxPool2d(1), ReLU(), Conv2d(16, 32, kernel_size=3), MaxPool2d(1), ReLU() Flatten() Linear(1152, 128), ReLU(), Linear(128, 236)
16×16	24×24
Conv2d(27, 16, kernel_size=3), MaxPool2d(1), ReLU(), Conv2d(16, 32, kernel_size=3), MaxPool2d(1), ReLU() Flatten() Linear(4608, 128), ReLU(), Linear(128, 548)	Conv2d(27, 16, kernel_size=3, stride=1), MaxPool2d(2), ReLU(), Conv2d(16, 32, kernel_size=2, stride=1), MaxPool2d(2), ReLU() Flatten() Linear(800, 128), ReLU(), Linear(128, 1188)