Gaming the Video Against Yourself

Joshua Boisvert
Stanford University
450 Jane Stanford Way, Stanford, CA 94305
joshmboi@stanford.edu

Abstract

This paper explores the integration of computer vision, deep reinforcement learning, and self-play in training an autonomous agent to play a custom two-player game inspired by League of Legends and Dota 2. A toy game environment was developed using pygame, featuring discrete abilities, resource management, and spatial dynamics. To interpret the visual input and account for temporal dependencies, a convolutional neural network (CNN) was combined with a long short-term memory (LSTM) network. The agent uses a Soft Actor-Critic (SAC) framework, augmented with continuous action embeddings and entropy regularization, to learn a policy through experience replay and self-play. Experimental results show that while the agent can exploit the environment to achieve maximal rewards, the actor and critic models do not accurately predict the environment dynamics. This work highlights the challenges of learning in partially observable, multi-agent environments and suggests directions for improving agent behavior through curriculum learning and enhanced architectural design.

1. Introduction

The real world is a dynamic and complex environment, governed by systems that are often only partially understood through theorems and postulates built on simplifying assumptions. Despite these abstractions, perceiving the world and extracting meaningful information from sensory input remains inherently difficult. For a robot or artificial agent to interact effectively with its surroundings, it must first interpret the environment through sensing, then translate those observations into purposeful actions.

In such complex settings, it is difficult to map sensed data into a meaningful feature or "state" representation. Furthermore, for an agent to apply itself to a variety of tasks, it must be able to generalize over a larger feature space rather than train for specific features. This link between the input and the actions taken relies on the agent focusing on features that it deems important rather than hand-crafted information.

Selecting actions that maximize the likelihood of success is challenging not only because feedback can be sparse or ambiguous, but also because "success" itself is often poorly defined. Consider now the action of walking. Though it may be easy to create a heuristic that captures the amount of time in the air, it provides little feedback for the actual motion of walking itself (e.g. putting one foot in front of the other). Furthermore, the method of feedback comes through raw input rather than pre-processed information. While heuristics may provide rough guidance, they rarely capture the full complexity of what makes an action effective. This challenge is amplified for autonomous agents, which rely on well-specified reward signals to distinguish between desirable and undesirable behavior. The task of mapping high-dimensional sensory inputs to coherent, goal-directed actions remains a fundamental obstacle in real-world decision-making.

Mirroring the complexity of real life in a much simpler setting, video games tend to display raw pixel input and other raw input (sound) and receive actions to alter the game's state. In video games, a player must convert raw visual input into an understanding of the game state (player health, enemy positions, objectives) and use this inferred state to decide on actions that maximize long-term reward. Adversarial settings only further complicate this process, as players must take into account the potential actions of the adversarial agents, as well as adjust to a more complex and evolving environment.

This paper aims to replicate the complex process of using visual input to take reward-maximizing actions through the use of a simple two-player game. I use sequences of 12 frames of input 200 pixels by 150 pixels in RGB as input to a convolutional neural network (CNN) and long short term memory (LSTM) model. The output of two different CNN-LSTM blocks are used to determine an anticipated reward value within the critic and a specified action to take in the actor. The critic serves as a method of understanding the rewards of taking specific actions and the actor serves to maximize the rewards through taking specific actions using

the critic values as a heuristic. The critic is a fully connected neural network (FCN) with one value output and the actor is an FCN with two heads, one for the selection of an action and one for the position values within the game to take the action.

2. Related Work

Previous work in deep reinforcement learning has demonstrated the capacity of autonomous agents to process raw pixel input and learn effective, reward-maximizing behaviors. Notably, [7] and [6] showcase this capability in different but complementary ways. In [7], the agent receives sequences of four consecutive frames as input, enabling it to extract temporal information directly from raw visual data. This method allows the agent to infer velocity and momentum without explitictly modeling time.However, due to the partially observability of my two-player game, where the agent lacks access to the internal states or hidden actions of the oppenent, I could not utilize frame-stacking, leading me to pursue a different time-dependent neural network.

In contrast, [6] focuses on transforming raw pixel input into a condensed latent feature space using a convolutional encoder. This latent space is then used for policy learning, improving sample efficiency and robustness, while also reducing the agent model's complexity. Inspired by this approach, I adopt a similar strategy, compressing high-dimensional visual input into a lower-dimensional, abstract feature space using a convolutional encoder.

Another relevant example of visual reinforcement learning is presented in [5], which employs a CNN-LSTM architecture to train an agent to play the game DOOM. The convolutional layers extract spatial features from raw input frames, which are then passed through an LSTM to maintain a temporal context. While this work effectively handles pixel-based inputs and demonstrates temporal reasoning, its singleplayer setup lacks the adaptive complexity introduced by opponent modeling. The absence of self-play or adversarial training limits its generalization to multi-agent environments, where the agent must continually adapt to evolving opponent strategies.

To address this limitation in other areas, several works have employed self-play as a means of generating a natural curriculum and increasing agent robustness. [2], [1], and [8] demonstrate that training against a past versions of the agent leads to more generalized and adaptable policies. The use of self-play introduces stochasticity into the environment, encouraging agents to continuously improve. In particular, [8] developed a league-based training framework where agents are pitted against previous policies to prevent overfitting and collapse into narrow strategies, resulting in a more stable and scalable learning process.

Regarding the learning algorithm itself, I draw from the Soft Actor-Critic (SAC) framework introduced in [3]. SAC is an off-policy algorithm that combines entropy maximization with actor-critic learning, promoting exploration and stabilizing policy updates. Its off-policy nature allows the use of a replay buffer, enabling the agent to learn from past experiences that may differ from the current policy, which is a useful trait in symmetric, self-play environments where roles between player and opponent can switch. Building on this, [4] extends SAC to visual domains by incorporating latent feature representations for image-based state encoding. Like [6] and [5], this work uses convolutional encoders to extract meaningful state features but instead discards value based methods like deep Q-learning in favor of actor-critic techniques, which generally offer better stability and convergence in continuous or partially observable settings.

Collectively, these prior works inform the architectural and methodological choices in this paper. By integrating latent representation learning, recurrent modeling, and self-play within a SAC-based reinforcement learning framework, this work aims to demonstrate that an agent is capable of learning robust and adaptable strategies in a visually complex, adversarial two-player game.

3. Methods

To facilitate my agent's learning, I created a custom two-player video game, a deep learning architecture consisting of a CNN-LSTM blocks for state representation, and actor-critic modlues for decision-making and value estimation.

3.1. Game Environment

I designed a custom two-player game using the pygame library and drawing inspiration from MOBA-style games like League of Legends and Dota 2. The game includes fundamental mechanics such as movement, ability casting, and resource management. Each player can perform one of three abilities: a directional projectile attack ("Q"), an area-of-effect damage zone ("W"), and a temporary shield ("E"). These abilities differ in terms of damage output and cooldown durations, encouraging diverse strategic behaviors.

Player actions such as movement and targeting are derived from the mouse position, allowing for spatially dependent interaction. To accommodate visual processing constraints, game sprites are deliberately larger to reduce the visual resolution passed to the learning agent. Health and stamina bars are visible for both players, reflecting current health and current stamina available, respectively. Figure 1 shows an example in-game screenshot with random policy agents.

3.1.1 CNN-LSTM Architecture

I process the visual input using a CNN and an LSTM. A convolutional neural network is a series of layers that per-

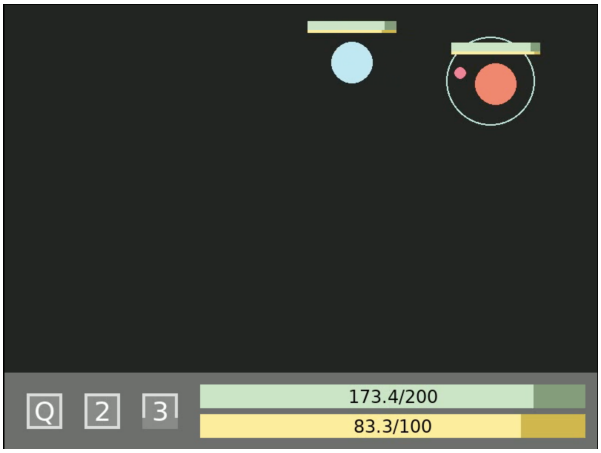

Figure 1. Example image of the toy game environment.

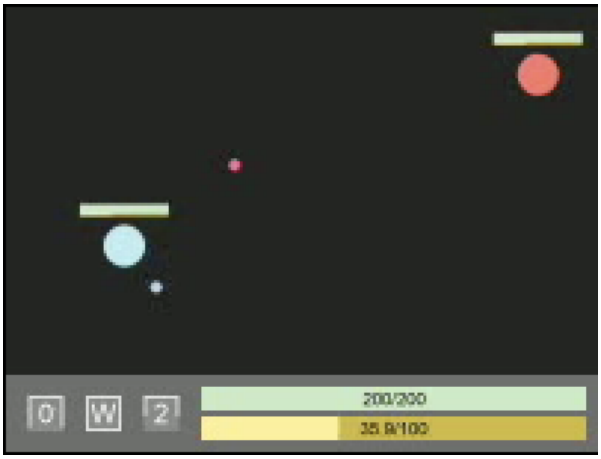

Figure 2. Example input to the CNN.

form convolution on a multi-dimensional input. Oftentimes convolution layers will contain multiple filters to gather information in varying methods. The CNN captures increasingly complex features from an input as the input makes its way through the model.

The architecture of my CNN takes in a sequence of 12 $200 \times 150 \times 3$ images and consists of three convolution layers with ReLU activation layers and max pooling layers in between. The three layers of my arcitecture are as follows:

- Conv1: 16 filters, kernel size 3, stride 2, padding 1
- Conv2: 32 filters, kernel size 3, stride 2, padding 1
- Conv3: 32 filters, kernel size 3, stride 2, padding 1

Each convolution layer is followed by a ReLU activation layer and a max pooling layer of kernel size 2, resulting in a final flattened vector size of 192. The ReLU layers are activation layers that use the ReLU function to introduce nonlinearities in the neural network, and the max pooling layers take the value of the maximum pixel within the kernel area.

The output of the CNN is then flattened and compressed into a feature representation of size 128. This feature vector serves as the latent state space vector representation of the visual input. In order to fit within the memory of my machine I needed to reduce the original resolution of 800×600 to 200×150 neccessitating an increase in the size of the sprites of the gae. Figure 2 shows a reduced-resolution image input example.

This vector is then fed into an long short-term memory network, which updates a hidden state and a cell state while collecting inputs. The LSTM module has a hidden state size of 256. The recurrent layer allows the model to maintain temporal context, which is essential in this partially observable setting where opponents' internal states (e.g. cooldowns) are not directly visible.

3.1.2 Actor and Critic Networks

The actor and critic components are each equipped with independent CNN-LSTM blocks to interpret the game state and mitigate cross-network interference during training. This separation ensures that actor and critic representations evolve independently, maintaining network stability. Additionally, if I were to link the two network into a shared feature representation, it would be hard to argue for whether the actor or critic should update the shared module and at what frequency. Additionally, since the actor and critic optimize for separate objectives (value approximation and policy exploitation), it made more sense to separate the architectures.

Both the actor and critic utilize the hidden state of their corresponding CNN-LSTM to determine an action to take and a value estimation respectively.

The actor outputs two sets of values:

- Action selection: A 4-dimensional unnormalized embedding for each of the 5 possible actions (idle, move, projectile, zone, shield). These are compared against an embedding matrix using cosine similarity to produce action logits.
- Positional execution: Mean and standard deviation parameters for a Gaussian distribution over (x, y) coordinates.

Actions are sampled using softmax over logits and positions are sampled from the corresponding Gaussian. This stochasticity promotes exploration and policy robustness. Fully connected layers are used for both the action embedding and the positional parameters.

The critic network produces a scalar estimate of the state-action value. After processing input through its CNN-LSTM stack, the critic applies three fully connected layers

with dimensions 256, 128, and 1, using ReLU activations between layers.

3.2. Training Methods

3.2.1 Frame Skipping and Temporal Sequences

Inspired by [7], [5], and [8], I implement a frame skip strategy to reduce computational overhead. I use a skip interval of 6 frames rather than the skip interval of 4 frames that is conventional. The agent selects an action every 6 frames, which corresponds to every fifth of a second in a 30 frames per second game. Each training sample is composed of a 12-frame sequence, allowing the model to learn long-term dependencies and assign credit appropriately over time. This overlap in sequences is key for maintaining continuity as well.

3.2.2 Soft Actor-Critic Optimization

The agent is trained using the Soft Actor-Critic (SAC) algorithm [3], which facilitates off-policy learning and entropy-regularized policy optimization. SAC is well-suited for environments involving self-play due to its ability to learn from diverse and past experiences stored in a replay buffer. Specifically for this toy game, since the player agent will eventually switch places with the opponent, learning a policy over a larger state space becomes increasingly important to reduce the stochasticity of the opponent when training.

To support gradient-based learning over discrete actions, an action embedding mechanism is used. By mapping actions into a continuous embedding space, we allow gradients to flow through action selection, enabling end-to-end differentiability. This embedding works similarly to embedding tokens used in natural language processing. The embeddings only serve to allow for the use of gradient-based methods.

Actor updates are performed by minimizing the following loss:

$$\mathcal{L}_{actor} = \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{a\sim\pi}\left[-Q_\theta(s,a) + \alpha\cdot\mathcal{H}(\pi(\cdot|s))\right]\right] \quad (1)$$

$Q_\theta(s,a)$ is the expected return for taking an action a within a state s. These values are determined using the current critic. The states are sampled from the replay buffer $\mathcal{D}$ and the actions are sampled from the current policy π. $\mathcal{H}$ represents the entropy of the policy at the specified state. One thing to note is the use of approximate gradients rather than analytical gradients. The gradients are determined using replay buffer sampling. The actor loss is used to update both the actor and the action embedding layer.

The temperature parameter α is adjusted automatically to ensure the entropy remains close to a target value:

$$\mathcal{L}_\alpha = \mathbb{E}_{s\sim\mathcal{D}}\left[-\alpha\cdot\mathcal{H}(\pi(s)) - \mathcal{H}_{target}\right] \quad (2)$$

For this game, since there is a discrete dimensionality of 5 and a continuous dimensionality, the target entropy that I used was -5, directly equaling the number of distinct discrete actions.

Critic loss is computed using the mean squared error between predicted Q-values and the expected return from the Bellman Eauation and target Q-values of the next state-action pair:

$$\mathcal{L}_{critic} = \frac{1}{N}\sum(Q_\theta(s,a) - y)^2 \quad (3)$$

$$y = r + \gamma(1-d)Q_{target}(s',a') \quad (4)$$

r represents the rewards at the next state, d is a flag determining whether the state is a terminal state, and γ is the discount factor. Once again, the critic loss is determined through sampling from the replay buffer. The critic loss is backpropagated through the critic network.

Target networks are updated using a Polyak averaging strategy:

$$\overline{\theta} \leftarrow \tau\cdot\theta + (1-\tau)\cdot\overline{\theta} \quad (5)$$

$\overline{\theta}$ is a parameter in the target critic network and θ is a parameter in the critic network. τ determines the rate at which the target critic network is updated, usually set to 0.005.

All learned parameters are optimized using Adam with a learning rate of 1×10^{-3}.

3.2.3 Replay Buffer

I used a replay buffer with 20,000 transitions (observation, action, reward, next observation sets). Due to memory limits this is the maximum that I could store without risk of memory overflow. However, having 20,000 transitions is satisfactory for conducting my policy optimization, as it encompasses approximates the past 20 rollouts (full trajectory run throughs of my game).

3.2.4 Self-Play Curriculum

Self-play introduces a natural curriculum by incrementally challenging the agent. To avoid destabilizing learning, only the player is updated while the opponent is held fixed. Every 20,000 training steps, the opponent's policy is replaced with the latest version of the player's policy. This semi-static opponent scheme ensures that the training environment evolves gradually, enabling stable and consistent policy improvement over time.

3.3. Training Setup

When training my policy, repeatedly ran my game and updated my critic in real time at every step that I toook an action. I used batches of size 256 to update my actor and critic, making sure to use the same batch for both the actor and critic. Both the actor and critic trained once each

action step. When conducting rollouts, I made sure to collect 10 times the batch size of transitions to start training to decrease the correlation between samples within the same batch. I also had a pretraining period of 30,000 timesteps for the critic so that the critic and actor would have less instability when training.

4. Experiments and Results

Because of the complexity of my model and environment, I encountered a significant amount of difficulty training. One aspect of my results that completely confused me was the movement of the agents to the corner of the map to fight it out. Since there were only strong penalties for leaving the center, I have no idea why the agents chose to stay in the corners. Figure 3 shows an example of the agents traveling to the corner. Initially I thought that this could be due to vanishing gradients and the squashing of the position output into the pixel range of the game, but the agent appears to select the same position for both movement and placing the damage zones, as well as selects varying corners in different games. Interestingly enough, the agent appears to stop using the projectile ability, as shooting any projectile would not hit the opposing player since the projectile spawns outside of the players. Because of this difficulty in training

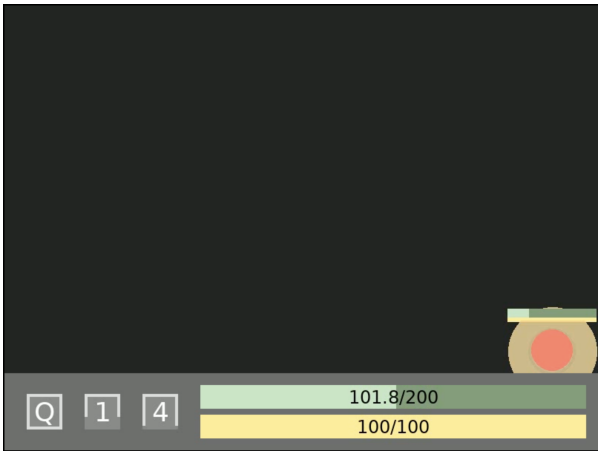

Figure 3. Both agents fighting in the bottom corner.

with the full action space, I repeatedly reduced the action space of my player and opponent, ultimately resulting in the player only being able to shoot projectiles and the opponent only being able to do nothing. Additionally, I gave the agents full stamina regeneration to prevent stamina issues from affecting the ability of the actor and critic to learn. Unfortunately, this reduces my problem into a really small action and state space, however, it still achieves the paper goal of translating visual input into a state space using an actor critic method and a self-play curriculum.

Comparing the results of my policy to a randomly acting agent, the trained policy was able to kill the opponent in seconds and the random policy was unable to kill the agent within a minute. In terms of returns received, the trained policy had average rewards of 295 over the course of 10 games, whereas the random policy agent received average rewards of 44 over 10 games. These results are shown in Table 4.

Policy	Average Returns	Time to Kill (s)
Random	44	12
Trained	295	N/A

Table 1. Average returns and time to kill.

Figures 4 and 5 show the actor and critic losses respectively. The actor loss stedily decreases, but this makes sense with a critic loss that continually increases. The critic loss increase means that the critic is having a hard time accurately predicting the rewards for different transitions. Due to this variance, the actor has much uncertainty when calculating updates. Due to the nature of the critic being used in the actor update and the actor being used in the critic update, the effects of these instabilities are compounding. Even with these poor loss values, the returns over episodes are incredible.

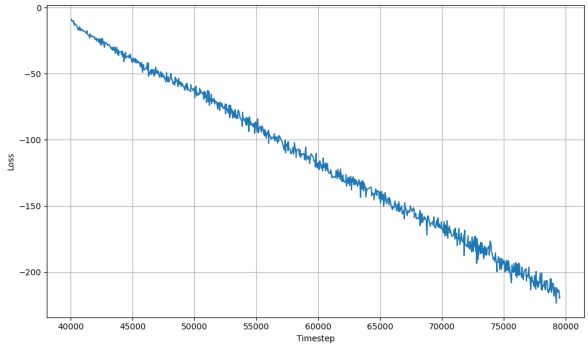

Figure 4. Actor loss across timesteps.

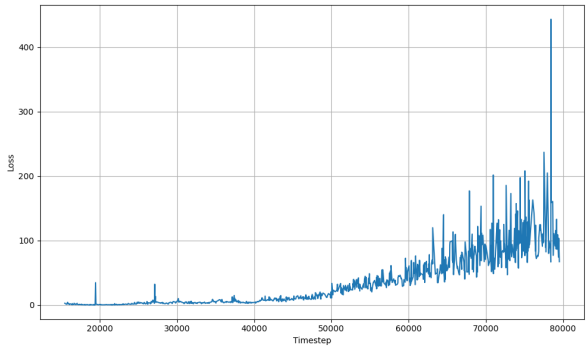

Figure 5. Critic loss across timesteps.

Figure 6 shows the average returns of an agent over timesteps. Since an agent has 200 health with a health

regeneration rate of about 1 health every second, within a timespan of 12 seconds, the agent can get a maximum damage score of 210. The reward function rewards the agent for every damage point inflicted, as well as 100 point if the agent wins. Therefore, the maximum score that an agent can get within a time period of less than 20 seconds is 310. This means that the agent performs the maximal amount of damage within the prescribed time and achieves maximum rewards. Even though the critic and actor have massive losses due to prediction difficulty, the agent was still able to quickly dispatch the opponent. I am truly unsure as

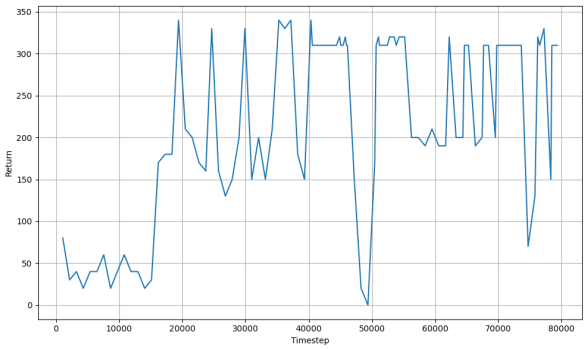

Figure 6. Returns over timesteps.

to how the agent can still achieve maximal results with a poorly defined actor and critic, and am wondering whether it is due to emergent properties or due to

5. Conclusion

This work demonstrates the feasibility of integrating multiple neural network architectures (convolutional neural network (CNN), long short-term memory network (LSTM), an actor-critic framework) to train an agent capable of interacting with and learning from a simulated game environment. While the resulting policy acts within a limited state and action space, the toy game provides a structured, visually rich setting that captures some of the dynamics found in real-world scenarios, offering the agent the possibility of exhibiting spatial and temporal processing.

The results also highlight the inherent challenges in combining these components. Training instabilities emerged as a central difficulty, likely due to the simultaneous optimization of interconnected networks with shifting objectives. Nonetheless, with an even simpler model, the agent was able to learn a policy that maximized returns within a given time window.

There are several avenues that I would like to pursue for improving both stability and agent behavior. A persistent issue observed was the agent's tendency to cluster near the corners of the map with a larger action space. I would like to investigate the cause of this strategy, whether it be credit assignment or reward-hacking. Additionally, with the discrepancy between actor and critic performance and agent performance, I would also like to look into if the simplicity of the game allowed the agent to receive maximal rewards with a suboptimal policy. Lastly, I would like to explore more principled curriculum learning techniques. For instance, progressively unmasking actions or gradually increasing environment complexity might allow the agent to fully experience and guide the agent toward more desirable behaviors like maintaining a central position or learning more nuanced tactics.

References

[1] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition, 2018.

[2] D. Dwibedi and A. Vemula. Playing games with deep reinforcement learning, 2020.

[3] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications, 2019.

[4] T. Kim, Y. Park, Y. Park, and I. H. Suh. Acceleration of actor-critic deep reinforcement learning for visual grasping in clutter by state representation learning based on disentanglement of a raw input image, 2020.

[5] G. Lample and D. S. Chaplot. Playing fps games with deep reinforcement learning, 2018.

[6] S. Lange, M. Riedmiller, and A. Voigtländer. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2012.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[8] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning, 2019.