DACAN
Joint discrete and continuous action spaces in Deep Reinforcement Learning

Nichlas Ørts Lisby*, Thomas Højriis Knudsen*, Chenjuan Guo
{nlisby16, thkn16}@student.aau.dk, cguo@cs.aau.dk
Aalborg University
Department of Computer Science

Abstract

In this work we tackle the problem of domains with hybrid action spaces, i.e. both discrete and continuous. These environments have proven challenging for traditional Deep Reinforcement Learning (DRL) methods, and may be sub-optimally handled by using discretized continuous actions. The addition of continuous actions are especially ideal for modern games, with input from devices like a mouse or an analog stick. Other relevant domains for continuous actions include robotics and other tasks where you cannot achieve sufficient precision with a limited number of predefined actions. While discrete Deep Reinforcement learning agents can be modified to work in such environments, they typically struggle when high precision is required. We introduce two different methods for combining discrete and continuous action spaces, the first being a naive combination of continuous and discrete networks and the other being an Actor-Critic based approach, with a central critic that can critique the various actors. We show that the naive combination of networks result in sub-optimal and unstable learning, and thereby confirming the need for a method in which continuous and discrete actions can be combined in a sensible and coherent way. Our central critic approach outperforms our Double DQN (DDQN) baselines in the DOOM environment on the VizDOOM scenarios *Deadly Corridor* and *Defend The Center*. It quickly reaches a score which is better than the DDQN baselines and then further improves the score. We also show that our approach significantly outperforms DDQN when using large actions spaces, for example to introduce precision in discretized actions, in which the DDQN will not scale properly.

1 Introduction

Deep Reinforcement Learning (DRL) experienced a big leap forward with the publication of the original Deep Q-Network (DQN)(Mnih et al. 2015). In the paper they teach an agent to play old-school Atari 2600 games at a level comparable to or even exceeding human-level performance in some Atari games, using their novel DQN which combines traditional Q-learning with Deep Neural Networks. The DQN receives raw pixels as the only input from the Atari games, and feeds those to the neural network, which results in DQN being able to reason about high-dimensional inputs. However, the method introduced in the DQN paper only handles low-dimensional and discrete action spaces. While this is plenty for the Atari games, which only have a few available actions, all of which are discrete, modern games have much larger actions spaces, especially with the introduction of continuous actions. Multiple papers have also applied the traditional DQN to more complex domains, as well as further improved upon the traditional DQN to either perform better in the traditional Atari games or to be able to handle more complex domains.

However, most of the improvements to DQN as well as other DRL methods still focus solely on discrete action spaces. This may be sub-optimal, as the usage of discrete actions alone can become troublesome in modern games, which predominantly use 3D environments. While using discrete actions for interaction with movement keys is fine, a problem arises once the agent has to "look around" (hereafter referred to as player orientation) in the environment. This action is often accomplished using either the mouse or an analog stick, both of which will yield a continuous action as opposed to a discrete action.

A continuous output can be any real value within a continuous sequence of numbers, i.e. all numbers within a range is defined. A continuous action can be anything from updating the position of the mouse on the x or y axis, to the degree of bending in a robot joint or the fuel to apply to a car to get it to go up a hill.

Continuous outputs do not integrate well with traditional DRL approaches. The (Tessler et al. 2016) paper did it by discretizing the mouse input into two discrete actions, namely *Look 30 degrees left* and *Look 30 degrees right* in the 3D domain of Minecraft.

Being restricted to only a couple of discrete actions for player orientation comes with some fairly obvious limitations. First of all there is a loss in precision, which could be vital in, for instance, First Person Shooter (FPS) or Multiplayer online battle arena (MOBA) games where the precision of the mouse is essential, as you can only move in predefined increments. Furthermore the very rough, step based control is very different to the smooth control from a human player. Another downside to discretizing continuous actions into many discrete actions is the increased size of the action-space, with which a traditional DQN struggles to learn, due

* These authors contributed equally

to the high amount of exploration needed.

Tackling continuous actions in DRL is not a new problem. The paper on Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al. 2019) was a breakthrough in determining continuous actions through their DDPG method. However, the DDPG method has primarily been applied in the context of controlling parts and joints of a robot, along with multiple physics simulation environments like the *Cartpole*[1] environment. DDPG has never been applied within the context of controlling player orientation, or more specifically determining mouse inputs.

In this paper we present two methods for dealing with both discrete and continuous actions in game environments. This is needed in order to further advance DRL progress in the domain of 3D games and continue the quest for human level control in 3D environments.

- We present a naive implementation of a mixed, but disjointed continuous action and discrete action agent, where there is no collaborative learning, but each agent tries to learn on its own.
- In addition to this, we present a framework which is capable of taking the usual high-dimensional input known from DRL, whilst producing both discrete and continuous outputs, using a conjunction of Actor-Critic methods for both discrete and continuous actions. We will show how to combine and train these methods in a cohesive and non-destructive manner.
- We demonstrate with empirical data that our novel method is able to outperform existing methods working on solely discrete or continuous actions spaces.

2 Environment

A challenging FPS 3D first person shooter environment can be set up in ViZDoom[2], which is the AI research platform that has been chosen for agents implemented and tested in this paper. ViZDoom is based on the classic FPS game DOOM II[3], and has a rich API for controlling the player with both discrete and continuous actions. Please consult section 9 and section 9 in appendix for a full overview of available actions. Furthermore it is built with reinforcement learning in mind, which means that the API includes a step function for providing a player action and getting back a reward as well as a function call for receiving the current game state. ViZDoom allows for different game state representations, however, for this paper we will solely be focusing on the raw pixel representation. Furthermore, ViZdoom provides multiple different maps or scenarios in which you can test your agent. It even provides a custom map builder to create custom scenarios.

Defining the domain

As with most traditional DRL algorithms the goal is to learn a policy using nothing but the raw pixels as input to the agent. However, in order for the algorithm to be able to reason about ongoing actions, like movement or acceleration, which is not obvious from a single frame, we will use frame stacking on the last x frames. Furthermore each frame will be preprocessed by a preprocessing function σ. σ will be responsible for downscaling and greyscaling the image. σ will downscale each frame to $w \times h$ pixels and perform greyscaling leaving each pixel defined by a single numeric value between 0 and 1.

Output from the agent is categorized in discrete actions and continuous actions. Discrete actions A_d for things like moving and shooting, from which we select a single action a_d. Continuous actions A_c being things like for instance the delta degree change of the player orientation, or force to apply to some element. A_c will be of size n where n is the total amount of continuous actions needed. For instance, if we would like to represent the continuous output of a computer mouse you would need two values, one for representing horizontal and vertical offset values respectively. The continuous actions are represented by a floating point value[4].

Thus once the preprocessing of σ has been applied, the agents will use the mapping:

$$w \times h \times x \rightarrow (a_d \in A_d, A_c)$$

Output being a tuple with both a discrete action, a_d, and n continuous action, all of which will then be executed simultaneously.

3 Related work

Combining continuous and discrete actions are not a completely unexplored domain. Prior work has tried to tackle the same task of combining continuous and discrete actions.

One of the more relevant methods is Hybrid SAC (Soft Actor-Critic) presented in (Delalleau et al. 2019). In the paper the authors also strive to combine discrete and continuous actions for video games, with discrete actions controlling buttons and continuous actions controlling an analog stick. It is an extension of the Soft Actor-Critic method, which was originally designed for continuous action tasks. In order to handle both continuous and discrete actions they theorize both converting continuous output to a discrete action and using a completely discrete actor. They end up combining both continuous and discrete actions into the SAC algorithm, and thereby making it a hybrid capable of outputting both continuous and discrete actions.

In table 1 we list how prior algorithms differs from ours.

As is evident, the Hybrid SAC method achieves the same goal as us and in our work we build on top of some of the same ideas as from the (Delalleau et al. 2019) paper, but we go with a fundamentally different approach where we utilize DDPG and our own discrete Actor-Critic method as our foundation as opposed to using SAC.

4 Background

An important concept in reinforcement learning is the Markov Decision Process (MDP). The idea is to model your

[1] https://gym.openai.com/envs/CartPole-v0/

[2] http://vizdoom.cs.put.edu.pl

[3] https://store.steampowered.com/app/2300/DOOM_II

[4] In our implementation specifically represented by a 64 bit floating point variable.

Action space / Work	Discrete actions	Continuous actions	Combined actions	Hybrid foundation
DQN based methods	✓	✗	✗	-
Advantage Actor Critc	✓	✗	✗	-
DDPG	✗	✓	✗	-
Soft Actor Critic	✓[5]	✓	✗	-
Hybrid SAC	✓	✓	✓	Soft Actor Critic
This work	✓	✓	✓	DDPG

Table 1: **Related work**

domain and environment using the 5-tuple

$$(S, A, R, \mathbb{P}, \gamma)$$

where

S is the set of all possible states in the environment.

A is the set of all possible actions in the environment.

R is a reward function that maps $(s, a) \to r$.

$\mathbb{P}$ is a probability function $P(s'|s_t, a_t)$ specifying how likely we are to transition to s' given that we at time step t take action a_t in state s_t.

γ is a discount factor used to cope with uncertainty in future rewards, which is why we will often use the wording "expected *discounted* reward".

With this we can now follow the usual reinforcement learning flow of observing a state s_t at every time step t and taking an action based on our policy $a_t = \mu(s_t)$ which will result in a reward r_t and a transition to the next state s_{t+1}

With regards to the states, there is another useful concept called the Markov Property, which is something our states are assumed to fulfill. The Markov Property states that the next state should only depend on the current state. This means that the probability of transitioning to some state given the current state should be the same as the probability of transitioning to the state given all prior states. Mathematically we say that

$$P[s_{t+1}|s_1, ..., s_t] = P[s_{t+1}|s_t]$$

This is a convenient assumption, since it means we do not have to store the complete history (Although the current state could be the sequence of all states before it).

DQN

The DQN algorithm was first introduced in the paper by Mnih et al. in 2015 (Mnih et al. 2015). The paper proved to be a break through in modern DRL since it managed to combine traditional Q-learning with high dimensional input by using a Convolutional Neural Network (CNN), a type of deep neural networks ideal for recognizing patterns in images. The neural network was utilized to approximate the Q-value, that is the expected discounted reward.

The DQN algorithm uses the Bellman equation to calculate the Q-value, which states that the total expected reward is given by the immediate reward plus the reward expected in the future.

Following this the Q function can be defined in a nice recursive manner:

$$Q(s, a) = \mathbb{E}\left[R(s, a) + \gamma \max_{a'} \left(Q\left(S\left(s, a\right), a'\right)\right)\right]$$

where S is now the transition dynamics defined for taking action a in some state s. It is used to calculate the Q-value for a given action, allowing us to choose the best action based on the Q-value. The goal is to calculate the optimal reward for all actions, and the optimal Q-function is often denoted as Q^*.

DQN uses Experience Replay, meaning it samples from the set δ of previous experiences. The replay buffer is standard for most DRL algorithms, and leads to more stable behavior. The replay buffer is simply a large store of past transitions in the form of tuples given by (s_t, a_t, r_t, s_{t+1}).

The second key contribution of DQN was the introduction of a target network Q' to help improve the learning stability. The target network is a copy of the actual network that is not updated for a period of time allowing the network to have something stable to converge towards. The target network is then updated with the weights of the actual network at some given interval.

During each learning step, the algorithm uses a mean-squared error function to minimize the error during training and maximize the Q-value. At a given time step t we define the target value for our network as:

$$y_t = \begin{cases} r_i & \text{if } i+1 \text{ is terminal} \\ r_i + \gamma \max_{a'} Q'(s_{t+1}, a') & \text{else} \end{cases}$$

Which leads us to the following definition of loss:

$$\left(y_t - Q\left(s_t, a_t\right)\right)^2$$

often refereed to as TD-loss,is temporal difference loss, since we calculate the difference between what our Q-network predicts at time step t and the immediate reward gained at t added to the value predicted by our target network on the next state s_{t+1}.

One last thing to tackle is when to select which action. To strike a balance between exploration and exploiting previous learned experience, DQN uses the Epsilon-greedy method. It is simply a slowly decreasing value, ϵ, which determines the probability of selecting a random action or using the action with the maximum Q-value as decided by the Q-function. Initially ϵ will often be 100% meaning all actions are random, and exploration is favored, but as time goes on the ϵ value will decrease and often end at 5% where exploitation is heavily favored, but a random action is still chosen ever so often.

[5] Can easily be modified for discrete action spaces

DDQN Since the introduction of DQN it has been improved multiple times. In this paper we will utilize the extension called Double DQN (DDQN) which is an almost regular DQN improved to reduce the overly optimistic estimates that a vanilla DQN can yield when determining the value of the current state. This is done by updating the way the target value is calculated:

$$y_t = \begin{cases} r_i & \text{if } i+1 \text{ is terminal} \\ r_i + \gamma Q'(s_{t+1}, \max_{a'} Q(s_{t+1}, a')) & \text{else} \end{cases}$$

Now utilizing the the current network for action selection.

Actor-Critic

A somewhat different approach for DRL is the Actor-Critic method. It differs from DQN in multiple ways by being both a mix of value based and policy based algorithm and also by being an on-policy algorithm. The Actor-Critic method utilizes two networks, one being a traditional Q-Function (the critic) and the other being a policy function (the actor). The responsibility of the actor is to determine which actions to pick, while the critic should be used to evaluate the actors behavior. The output of the actor should in the case of discrete actions be activated using the Softmax activation function, which will result in a probability distribution over actions to pick. This differs from the off-policy method of using for instance ϵ-greedy for selecting the action to pick.

Throughout the paper we will use the fundamental ideas from Actor-Critic, but our different usages will also differ significantly from traditional Actor-Critic methods, and we will therefore not go more in to depth for the time being.

DDPG

The ideas for determining continuous actions detailed in this paper is inspired by the paper Continuous control with deep reinforcement learning(Lillicrap et al. 2019), which introduces a novel method for continuous control in DRL with the DDPG algorithm. The general method for DDPG is largely inspired by DQN and Actor Critic networks.

The agents behavior is based on its policy π, which is a learned probability distribution. This is also where the primary difference between DQN and DDPG exists, as DQN utilizes a Q value to estimate a policy, while DDPG uses an architecture similar to the one found in Actor Critic networks, which uses both a Q network and a policy network. As with DQN the DDPG also utilizes target networks, which leaves us with a total of 4 networks: A Q network Q (the critic), a deterministic policy network μ (the actor), a target Q network Q' and a target policy network μ'.

The DDPG network receives an observation and outputs a single continuous value. This value will most often correspond to some position or an amount to add or subtract from a value. For example, consider a joint on a robot, where the current position of the joint is observed in every state. The network value can then be used to determine the update to the joints position or angle.

Contrary to Reinforcement Learning (RL) in discrete action spaces, where exploration is done by probabilistically selecting a random action (for example using epsilon-greedy), DDPG, and continuous action RL in general, simulates randomness by adding noise to the action itself, to ensure different actions are tried during training to find useful learning signals. This leaves us with an exploration policy $\mu^t(s_t)$, which takes a state and modifies the action using some noise $\mathcal{N}$:

$$\mu^t(s_t) = \mu(s_t) + \mathcal{N}$$

So we simply output an action directly as a real number, that can then be applied.

To prevent canceling out training the noise, $\mathcal{N}$, is generated so that it is somewhat correlated with previous noise, using the Ornstein-Uhlenbeck (OU) process(Uhlenbeck and Ornstein 1930), as detailed in the DDPG paper.

However, more recent results have found that OU noise is overly complicated, and that uncorrelated mean-zero Gaussian noise is perfectly adequate[6](Matheron, Perrin, and Sigaud 2019). Gaussian noise is a random noise drawn from a Gaussian distribution:

$$f(x) = \frac{1}{d\sqrt{2\pi}} e^{-1/2(\frac{x-m}{d})^2}$$

where m is the mean and d is the standard deviation. With Gaussian noise, the probability of larger deviations quickly decreases.

Learning The DDPG algorithm is in many ways very similar to DQN and borrows many training characteristics from DQN. It uses both a replay buffer and target networks, same as DQN. However, as mentioned, DDPG learns a total of 4 networks. The critic is updated using a mean-squared Bellman error, which is computed from mini-batch samples representing a transition, $s_i \rightarrow s_{i+1}$.

The loss functions for the critic is defined as:

$$J_Q = \frac{1}{N} \sum_i \left(r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1})) - Q(s_i, a_i)\right)^2$$

For the actor the objective is simply to maximize discounted reward, which we can do by using gradient ascent with the following 'loss':

$$J_\mu = \frac{1}{N} \sum_i Q(s_i, \mu(s_i))$$

The target networks for DDPG are updated differently from DQN, where the target network is frozen for a while, whereas DDPGs target networks are simply constrained with soft updates. Soft updates will essentially blend the two networks together, using a factor, tau, to determine the ratio between how much of the target network to keep and how much to take from the regular network. It is defined as:

$$\tau \ll 1 : \theta' \rightarrow \tau\theta + (1-\tau)\theta'$$

Where θ is the weights of the network and θ' is the weights of the target network.

Similar to DQN, as described in section 4, DDPG also uses a replay buffer. Since DDPG is an off-policy algorithm,

[6]https://spinningup.openai.com/en/latest/algorithms/ddpg.html

it can as opposed to other Actor-Critic methods utilize this large replay buffer to learn from more uncorrelated transitions.

MADDPG

Multi-agent DDPG (MADDPG) is a multi-agent method presented by (Lowe et al. 2020) which allows multiple DDPG agents to collaborate in order to achieve a common goal.

The multi-agent method of joining a continuous action agent with a discrete action agent is based on the MADDPG method of running two or more DDPG agents together. MADDPG extends the traditional DDPG agent into a multi-agent approach, using centralized planning by means of decentralized actors all being critiqued by a central critic, which has access to information about all policies of other agents. It was specifically designed for environments where multiple different agents all affect the same environment[7]. Each individual agent has direct access to only its local observations, and can be viewed as the actors from an actor-critic algorithm. Essentially, MADDPG is simply two or more DDPG networks that have learned to cooperate, thanks to a central actor.

5 Methodology

We introduce two methods for combining discrete and continuous actions in a single agent. The first method naively combines discrete and continuous action networks. Afterwards, we introduce a novel method to handle the combination of discrete and continuous action networks.

NNC

We first introduce a naive combination of multiple independent continuous and discrete action networks, which we will refer to as Naive Network Combination (NNC). The NNC method will have a DDPG network for controlling player orientation and a completely independent DDQN for controlling movement and shooting. The purpose of NNC is to observe how well independent networks will train together, as this is expected to result in bad and unstable learning.

With a naive combination, we expect that a good action in one network may still lead to a bad reward, resulting in negative learning for the network, even though it chose the correct action.

Consider a scenario where the network responsible for aiming will correctly aim exactly at an opponent, but the network responsible for moving and shooting decides to walk backward and fall down a cliff instead of shooting. In that case, even though the network responsible for aiming did the correct thing, the other network does something completely wrong, which will result in a bad reward and may eventually result in the network learning not to do the 'right' thing. It may instead learn to look directly into the air, preventing the movement network from running off the cliff, but now never being able to kill the opponent, and never actually receiving the optimal reward. The other network may then try to shoot, but as the player looks directly into the air, shooting will not yield any reward.

[7] An example of this could be multiple robots or multiple joints on a robot

DACAN

The primary contribution of this paper is our off-policy and model-free algorithm and network architecture designed to combine a discrete action network with one or more continuous action networks, thus the network should be able to fully control a player with both continuous and discrete action inputs. As opposed to the NNC method, the networks should be combined in a non naive way, in which the network should be able to achieve stable and non destructive learning. We dub our contribution Discrete And Continuous Action Network (DACAN). The main idea for DACAN is to utilize the same strategy as in MADDPG, that is have a single 'global' critic that can critique, or evaluate, multiple actors. In this paper that would be one or more DDPG networks, and a single discrete actor network. However, the idea should be reusable with any continuous and discrete actor networks, and therefore not exclusive to DDPG and our actor network implementation.

Network architecture First we introduce a standard base network, as can be seen in fig. 1, consisting of 4 layers, the first 3 being convolutional layers and the last being a fully connected layer. All the layers uses the Rectified Linear Unit (ReLU) activation function. This base is very common in DRL networks working on pixel input. The idea is to have the convolutional layers pick up on patterns in the image, and then combine the patterns to a bigger meaning in the fully connected layer.

For further information of the exact values used in the network, please conform to the hyper parameters in section 10.

Connected to the last layer of the base network is our output layers.

The critic output is activated using the identity activation function (essentially no activation). The goal is for the critic to be able to predict the Q-value given a state and actions. Therefore the critic will, in addition to the stacked observations, take both a discrete and continuous actions as input.

Next up is one or more continuous output layers activated using tanh to force the output to be in range $[-1, 1]$. The goal of these layers are to output the continuous actions that will maximize the Q-value, i.e total discounted reward accumulated.

Finally, we have the discrete actor output using the softmax activation to get a probability distribution over discrete actions that can be taken. The goal is again to maximize the Q-value, so the action with the highest probability should reflect the action expected to give the highest total discounted reward.

The complete network architecture can be observed in fig. 1.

Learning As with both DQN and DDPG we will also introduce replay memory for storing experiences. During training we will sample from these experiences and use those to update the networks. Furthermore we will also be utilizing target networks, as both DQN and DDPG found

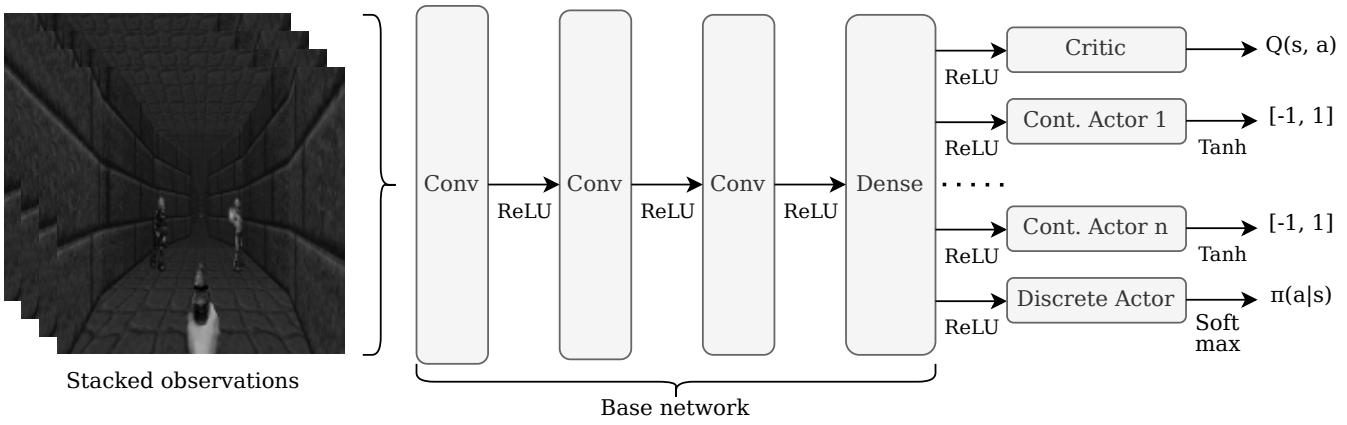

Figure 1: DACAN network architecture. Detailed information about layer sizes can be found in the appendix.

those to improve the learning stability. Therefore we will have the following networks:

Q	The critic network
μ_d	The discrete actor network
μ_c^1	The first continuous actor network
...	
μ_c^n	The n continuous actor network
Q'	The critic target network
μ_d'	The discrete actor target network
$\mu_c^{1\prime}$	The first continuous actor target network
...	
$\mu_c^{n\prime}$	The n continuous actor target network

Following this we define $\mu_c^{1,\dots,n}$ as all continuous actor networks, i.e. $\mu_c^i, \forall i \in \{1, \dots, n\}$.

In order to update the networks we will have to determine the loss functions for each.

Starting with the critic network, the premise is the same as in DDPG, the target value is the predicted Q-value using the Bellman equation, that is we take the immediate reward at a given step plus the discounted future reward predicted by the target network on the next state using the best actions. In order to determine the best continuous actions, we use the continuous target networks to determine the optimal action. For discrete actions, since the total number of actions are a lot more limited than for the continuous actions, we can simply maximize the Q-value over all the possible discrete actions. Formally the target would then be given by

$$y_i = r_i + \gamma \max_{a_d \in A_d} Q'\left(s_{i+1}, \mu_c^{1\prime}(s_{i+1}), \dots, \mu_c^{n\prime}(s_{i+1}), a_d\right)$$

Then we can determine the TD error using the predicted Q-value using the critic network. At last we will use Mean Squared Error (MSE) as our loss function, which will result in the following function for calculating loss across a mini batch with N elements:

$$J_Q = \frac{1}{N} \sum_i \left(y_i - Q\left(s_i, a_{c\,i}^1, \dots, a_{c\,i}^n, a_{di}\right)\right)^2$$

and i denoting the time step at each element in the mini batch.

The continuous actors are fairly straight forward, as the goal of the these are simply to gain as much discounted future reward as possible. Therefore the loss can simply be the negative sum of all predicted Q-values on a mini batch. For prediction we will utilize the discrete action resulting in the greatest predicted Q-value and for all continuous actions not being the one for which we are determine loss for, we will use target networks in order to differentiate the different continuous actor losses. If we use gradient ascent for these network updates, we can maximize with regards to the Q-value in which case the 'loss' for actor j will simply be:

$$J_{\mu_c^j} = \frac{1}{N} \sum_i \max_{a_d \in A_d} Q(s_i, \mu_c^{1\prime}(s_i), .., \mu_c^j(s_i), .., \mu_c^{n\prime}(s_i), a_d)$$

The discrete actor is somewhat different from actors in regular discrete Actor-Critic methods and it is more closely related to the actor from DDPG. In practice we will use these key differences: **1** Experience Replay instead of using the traditional N-step loss. **2** A Target network, as we sample from earlier experiences we should not be utilizing a rapidly changing network to calculate targets.

These changes are primarily due to DDPG being an off-policy method while most other Actor-Critic methods are on-policy. Although it should be noted using discrete off-policy Actor-Critic methods with experience replay is not a new thing. Previous work has shown to be very effective using these methods and is for instance demonstrated with Sample Efficient Actor-Critic with Experience Replay (Wang et al. 2017).

Apart from these two key changes, the main idea of the discrete actor remains the same as most previous discrete actors. The idea is for the actor to predict the 'advantage' gained by taking a particular action. The advantage is the delta value between the predicted discounted reward on an action and the value we expect a state to be 'worth'. Formally it is given by

$$A(s, a) = Q(s, a) - V(s)$$

Hence the goal for this network is to maximize the advantage gained given above. However, we want to do it in such a way, that the probabilities outputted from the actor network is updated in a sensible way, and not immediately favors one action a lot more than the others. Therefore we utilize the Negative Log-Likelihood (NLL) $-\log$ loss function, which is simply the negative $\log$ value of the probability of selecting an action(Miranda 2017)[8].

The loss function J_{μ_d} for the discrete actor therefore becomes:

$$J_{\mu_d} = \frac{1}{N} \sum_i -\log(\mu_d(s_i)) \cdot A(s_i, a_i)$$

Using the loss functions from above we calculate the partial derivative of the function with respect to all weights and biases in the network. This will result in gradients for each parameter in the network, and using these gradients we can perform gradient descent, i.e. subtract the gradient from the parameter in order to converge to the local minima. The gradient is multiplied by a small number α that is the learning rate, which is used to perform small granular updates and not overshoot the target.

In order to update the target networks we will utilize DDPGs method of doing soft updates to all the target networks. This is opposed to way it is done in DQN, where the target network is hard updated at some specific interval.

So after each iteration of training we will update target networks using

$$\begin{aligned}
\theta^{Q'} &\leftarrow \tau\theta^{Q'} + (1-\tau)\,\theta^{Q} \\
\theta^{\mu'_d} &\leftarrow \tau\theta^{\mu'_d} + (1-\tau)\,\theta^{\mu_d} \\
\theta^{\mu_a^{1\dots n'}} &\leftarrow \tau\theta^{\mu_a^{1\dots n'}} + (1-\tau)\,\theta^{Q\mu_a^{1\dots n}}
\end{aligned}$$

where θ denotes the weights of a given network and τ is some ratio determining how much the target network should be updated.

Pseudocode DACAN algorithm can be seen in algorithm 1.

Algorithm 1 DACAN algorithm

Init networks $Q, \mu_c^{1\dots n}, \mu_d$ with weights $\theta^Q, \theta^{\mu_c^{1\dots n}}, \theta^{\mu_d}$
Init target networks $Q', \mu_c^{1\dots n\prime}, \mu'_d$ with weights:

$$\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu_c^{1\dots n\prime}} \leftarrow \theta^{\mu_c^{1\dots n}}, \theta^{\mu'_d} \leftarrow \theta^{\mu_d}$$

Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select actions using current policy and noise:

$$\begin{aligned}
a_d &= \mu_d(s_t) \\
a_c^i &= \mu_c^i(s_t) + \mathcal{N}_{t+i}, \forall i \in \{1, \dots, n\}
\end{aligned}$$

    Take actions $a_t = \{a_d, a_c^1, \dots, a_c^n\}$ and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample mini-batch B of N transitions from R
    for $b \in B$ **do**
      Update $Q, \mu_c^{1\dots n}, \mu_d$ with values from b according to section 5
    end for
    Update $Q', \mu_c^{1\dots n\prime}, \mu'_d$ according to section 5
  end for
end for

6 Experiments

We test the 5 different methods discussed in the paper in order to determine and compare the results and benefits of DACAN over the existing methods. The purpose is to test how well DACAN performs when combining discrete and continuous actions, versus networks made for only one or the other. The methods will be tested against the *Deadly Corridor* scenario from VizDoom.

> *The purpose of this scenario is to teach the agent to navigate towards its fundamental goal (the vest) and make sure he survives at the same time.*
>
> *Map is a corridor with shooting monsters on both sides (6 monsters in total). A green vest is placed at the opposite end of the corridor. Reward is proportional (negative or positive) to change of the distance between the player and the vest. If player ignores monsters on the sides and runs straight for the vest he will be killed somewhere along the way.*[9]

Figure 2: Starting position in Deadly Corridor

For all experiments in this section, all agents will use the same base network as was explained in section 5. Furthermore all agents will utilize the same preprocessing algorithm σ which will scale the image to 80×80 greyscale and do 4 frames frame stacking. The discrete actions A_d will be $\{NoOp, Shoot, WalkAndShoot, Walk\}$ and the continuous actions will be limited to the players horizontal orienta-

[8] We can maximize by minimizing the negative log likelihood.

[9] https://github.com/mwydmuch/ViZDoom/tree/master/scenarios

tion, i.e. $A_c = \{\Delta_H\}$. Δ_H will be restricted to to a maximum update of 10 degrees pr. game tick. Additional hyper parameters used by the agents for the tests can be found in section 10.

DDQN

First baseline was created using a DDQN. Since the DDQN will only be able to output discrete actions, we need to discretize the player orientation actions for this experiment. This is done by defining 6 actions for player orientation, namely look horizontal with offset values $\{-8, -4, -2, 2, 4, 8\}$. We also conducted a test with only 2 actions, namely $\{-4, 4\}$ to reduce the actions space, but the decreased flexibility in player orientation seemed to result in worse results.

DDPG

For the next baseline, we decided to test how well the DDPG would perform on the domain. However, the DDPG possesses multiple challenges. First of all the output of the DDPG is continuous, which cannot be immediately used for our discrete actions. We solve this problem with the simple mapping

$$f(x) = \begin{cases} \text{if } x > 0 & \text{Action} \\ \text{else} & \text{NoOp,} \end{cases}$$

This is possible since all our discrete actions can be combined, for instance if both $Shoot$ and $Walk$ are selected we map that to the discrete action $WalkAndShoot$, and in the case of no actions being taken, it will be the equivalent of a NoOp.

The next problem is the fact that the DDPG will only output a single value, i.e. can only be used for a single action. This is solved by using multiple independent networks for each action. However, as mentioned in section 5 this is expected to result in unstable, or even no real learning at all.

MADDPG

Next, we wanted to add a MADDPG baseline. MADDPG still posses the problem of not being able to output discrete actions, so we will be using the same mapping as in the DDPG. As mentioned MADDPG is a Multi-Agent network, and is therefore able to output multiple values. We use this to output values to all the actions needed, and since we no longer use independent networks the training is expected to be more stable.

NNC

The NNC baseline is used to determine the success of using two independent networks. It uses a DDQN for discrete actions selection and a DDPG for player orientation. As the networks are not correlated at all, we would as with the DDPG expect the training to be unstable.

Results

The results of all the algorithms over a period of 2500 episodes in the *Deadly Corridor* scenario can be seen in fig. 3. As expected both DDPG and NNC performs suboptimal. In fact the results changing rapidly in a somewhat correlated way seems to indicate that the score is primarily affected by the changing noise and not so much network changes.

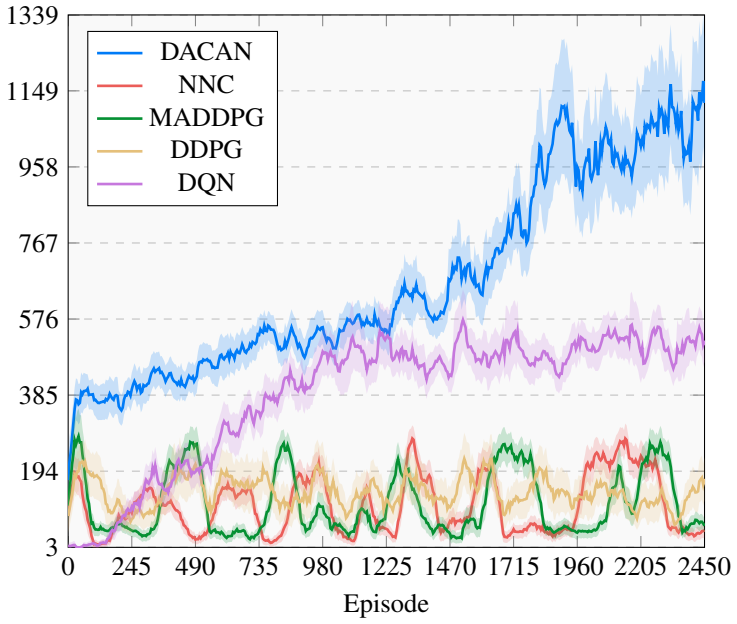

Figure 3: Evaluation of algorithms on Deadly Corridor

Somewhat more surprising is that the MADDPG performs similarly to DDPG and NNC. This could indicate that our implementation of MADDPG may contain errors, or that the hyper parameters chosen are inadequate for it to perform well. Alternatively it could also indicate that translating continuous actions to discrete actions using the mapping previously mentioned is not feasible in practice.

Next up is the DDQN which as expected demonstrates stable learning, until it plateaus at around episode 1000.

At last we can see DACAN showing stable and fast learning all the way through the experiment. The results imply that DACAN is useful for combining discrete and continuous actions in an advanced domain with high dimensional input. It also indicates that combining discrete and continuous actions may be favorable over simply using discrete actions and potentially discretizing input that should otherwise be continuous.

Comparing to DDQN in big action spaces

Following the above results it is clear that DACAN was the best performing method, also significantly better than DDQN. However, the scenario was pretty good for DDQN with only a few actions available. Next up we want to test the DDQN against DACAN in another scenario, namely *Defend The Center*.

> *The purpose of this scenario is to teach the agent that killing the monsters is GOOD and when monsters kill you is BAD. In addition, wasting amunition is not very good either. Agent is rewarded only for killing monsters*

so he has to figure out the rest for himself.

Map is a large circle. Player is spawned in the exact center. 5 melee-only, monsters are spawned along the wall. Monsters are killed after a single shot. After dying each monster is respawned after some time. Episode ends when the player dies (it's inevitable because of limitted ammo).[10]

The goal here is to prove that traditional DDQN do not scale to big actions spaces, and that DACAN will outperform DDQN significantly (with regards to tasks related to continuous actions), when forcing DDQN into large action spaces. This time we give the DDQN 200 different actions for changing the players orientation, namely $\{-10, -9.9, -9.8, \ldots, 9.8, 9.9, 10\}$. Although this may seem like a lot, it is nowhere near the in practice infinite[11] amount of values that DACAN can pick. In defend the center we will only have a single discrete action, that is $Shoot$, and of cause the $NoOp$ action. Therefore, in order to perform well in this scenario, it is vital that the agent becomes good at controlling the player orientation.

The results can be seen in chart fig. 4.

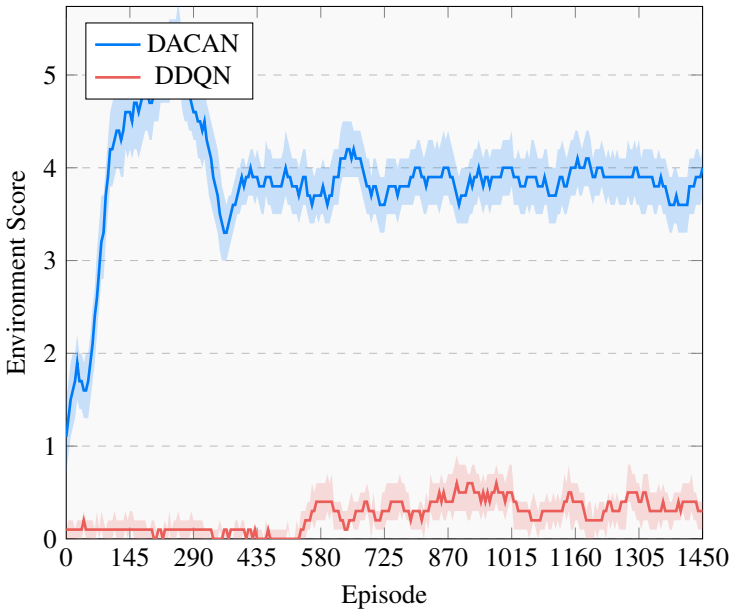

Figure 4: Evaluation of algorithms on Defend The Center

As can be seen in the chart, the big action space makes exploration really hard for DDQN, and it struggles to learn anything. DACAN on the other hand learns quickly, and although it may not quite reach the results we were hoping for (only killing an average of 5 enemies), it is apparent that having player orientation controlled with a continuous action, as opposed to a big amount of discrete actions, is very beneficial for learning.

[10] `https://github.com/mwydmuch/ViZDoom/tree/master/scenarios`

[11] A standard 64 bit floating point variable can represent approximately 14 trillion values between -1 and 1.

7 Conclusion

This paper presents two novel approaches for combining discrete and continuous actions in DRL networks. This hybrid of continuous and discrete actions are useful for environments requiring a high level of precision for continuous inputs, like in the case of controlling player orientation in the DOOM environment.

This works presents the NNC method, which naively combine two independent networks, namely a continuous and a discrete network. Utilizing this method each agent is essentially training in an environment which is made random by the other networks training, leading to unstable or no learning at all. The results, shown in fig. 3, shows that, as expected, the performance of the NNC method is inadequate for the scenarios in which it has been tested. Thus, we deem that the NNC agent does not warrant any further work.

The primary contribution, the DACAN method, performs well in the DOOM environment on the scenarios on which it has been tested. These are precisely the type of domains that DACAN is designed for, since it contains the discrete actions for moving the player around in the world, and the continuous actions for determining the player orientation, thus allowing the agent to look around in the environment with a much higher precision than if discretized player orientation actions were used.

The primary novelty is a successful integration between multiple discrete and continuous actor networks, with a central Q-network serving as the critic which manages the learning between actors. We introduce n continuous actor networks, one for each continuous action, meaning that we have a central critic network, a discrete actor network, n continuous actor networks, and target networks for each of these. This approach leads to excellent results, far exceeding the baseline networks also developed for the experiments, as can be observed in fig. 3. In addition, as can be seen in fig. 4, DACAN also significantly outperforms the traditional DDQN when the action space of DDQN is raised to be somewhat comparable to the precision achieved when using continuous values. This is simply because DDQN cannot scale to handle such big action spaces.

The DACAN approach provides very promising results and further work in other environments should be pursued

8 Future work

Given the time frame of the project and the limited server resources, we were unable to train all algorithms on multiple environments for an extended period of time. Future work should test especially DACANs performance against multiple environments, to further verify the results.

For some of the same reasons as stated above, DACAN has only been tested on a Proof of Concept level with a single continuous output and limited discrete actions. DACAN should in future work be tested with a more complex action space, including potentially multiple continuous outputs (for instance including the vertical axis for player orientation).

DACAN should also be tested against state of the art discrete and continuous hybrid agents like Hybrid SAC in order

to determine how well it compares against its main competition.

Additional further work could also investigate possibilities for improving the discrete actor in DACAN, by for instance introducing some of the improvements that have been known to benefit ordinary discrete Actor-Critic networks.

References

Delalleau, O.; Peter, M.; Alonso, E.; and Logut, A. 2019. Discrete and continuous action representation for practical rl in video games.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2019. Continuous control with deep reinforcement learning.

Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, P.; and Mordatch, I. 2020. Multi-agent actor-critic for mixed cooperative-competitive environments.

Matheron, G.; Perrin, N.; and Sigaud, O. 2019. The problem with ddpg: understanding failures in deterministic environments with sparse rewards.

Miranda, L. J. 2017. Understanding softmax and the negative log-likelihood". `https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/`.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2016. A deep hierarchical approach to lifelong learning in minecraft.

Uhlenbeck, G. E., and Ornstein, L. S. 1930. On the theory of the brownian motion. *Phys. Rev.* 36:823–841.

Wang, Z.; Bapst, V.; Heess, N.; Mnih, V.; Munos, R.; Kavukcuoglu, K.; and de Freitas, N. 2017. Sample efficient actor-critic with experience replay.

9 Appendix

Discrete actions

Binary buttons have only 2 states "not pressed" if value 0 and "pressed" if value other then 0.

- ATTACK
- USE
- JUMP
- CROUCH
- TURN180
- ALTATTACK
- RELOAD
- ZOOM
- SPEED
- STRAFE
- MOVE_RIGHT
- MOVE_LEFT
- MOVE_BACKWARD
- MOVE_FORWARD
- TURN_RIGHT
- TURN_LEFT
- LOOK_UP
- LOOK_DOWN
- MOVE_UP
- MOVE_DOWN
- LAND
- SELECT_WEAPON1
- SELECT_WEAPON2
- SELECT_WEAPON3
- SELECT_WEAPON4
- SELECT_WEAPON5
- SELECT_WEAPON6
- SELECT_WEAPON7
- SELECT_WEAPON8
- SELECT_WEAPON9
- SELECT_WEAPON0
- SELECT_NEXT_WEAPON
- SELECT_PREV_WEAPON
- DROP_SELECTED_WEAPON
- ACTIVATE_SELECTED_ITEM
- SELECT_NEXT_ITEM
- SELECT_PREV_ITEM
- DROP_SELECTED_ITEM

Continuous actions

Buttons whose value defines the speed of movement. A positive value indicates movement in the first specified direction and a negative value in the second direction. For example: value 10 for MOVE_LEFT_RIGHT_DELTA means slow movement to the right and -100 means fast movement to the left.

In case of TURN_LEFT_RIGHT_DELTA and LOOK_UP_DOWN_DELTA values correspond to degrees. In case of MOVE_FORWARD_BACKWARD_DELTA, MOVE_LEFT_RIGHT_DELTA,
MOVE_UP_DOWN_DELTA values correspond to Doom Map unit.

- LOOK_UP_DOWN_DELTA
- TURN_LEFT_RIGHT_DELTA
- MOVE_FORWARD_BACKWARD_DELTA
- MOVE_LEFT_RIGHT_DELTA
- MOVE_UP_DOWN_DELTA

10 Hyper parameters

Default hyper parameters

Hyperparameter	Value	Description
Frame width	$80px$	Width of frame after down scaling with σ
Frame height	$80px$	Height of frame after down scaling with σ
Frame stacking	4	Number of frames to include in input to network
Frame skipping	4	Number of frames to skip / repeat action
Base Layer 1 type	Convolutional	Type of layer
Base Layer 1 kernel size	8×8	Size of filter to convolve across frame
Base Layer 1 stride size	4×4	Step size of filter
Base Layer 1 output size	32	Number of outputs from layer
Base Layer 1 activation	ReLU	Activation function applied on output from layer
Base Layer 2 type	Convolutional	Type of layer
Base Layer 2 kernel size	4×4	Size of filter to convolve across frame
Base Layer 2 stride size	2×2	Step size of filter
Base Layer 2 output size	64	Number of outputs from layer
Base Layer 2 activation	ReLU	Activation function applied on output from layer
Base Layer 3 type	Convolutional	Type of layer
Base Layer 3 kernel size	3×3	Size of filter to convolve across frame
Base Layer 3 stride size	1×1	Step size of filter
Base Layer 3 output size	32	Number of outputs from layer
Base Layer 3 activation	ReLU	Activation function applied on output from layer
Base Layer 4 type	Dense	Type of layer
Base Layer 4 output size	1568	Number of outputs from layer
Base Layer 4 activation	ReLU	Activation function applied on output from layer
Optimizer	ADAM	Gradient updater / optimizer used when updating gradients
Weight Initializer	XAVIER	Method for initializing weights and biases
Batch size	32	Samples to use from replay buffer when training
Gamma	0.99	Discount factor used
Experience Replay Max Size	100000	The maximum size of the Experience Replay buffer in elements

DDQN

Hyperparameter	Value	Description
Target network update frequency	1000	How often the target networks are updated. Recall that these are frozen for a length of time.
Epsilon start	1	The start value of the epsilon-greedy value, i.e. how much we explore in the beginning
Minimum Epsilon	0.1	The smallest epsilon value.
Learning Rate	0.00025	Factor to be applied to gradients before updating network.

DDPG

Hyperparameter	Value	Description
Learning Rate	0.00025	Factor to be applied to gradients before updating network.
Noise function	Simplex Noise	The noise function used in the implementation.
Tau	0.001	The size of the soft updates to the target network

MADDPG

Hyperparameter	Value	Description
Learning Rate	0.00025	Factor to be applied to gradients before updating network.
Noise function	Simplex Noise	The noise function used in the implementation.
Tau	0.001	The size of the soft updates to the target network

NNC

Hyperparameter	Value	Description
Learning Rate	0.00025	Factor to be applied to gradients before updating network.
Noise function	Simplex Noise	The noise function used in the implementation.
Tau	0.001	The size of the soft updates to the target network

DACAN

Hyperparameter	Value	Description
Learning Rate	10^{-7}	Factor to be applied to gradients before updating network.
Noise function	Simplex Noise	The noise function used in the implementation.
Tau	0.001	The size of the soft updates to the target network
Actor to critic layer size	1568	
Actor to critic layer activation	ReLU	