Concolidated Deep Actor Critic Networks

Tamis Achilles van der Laan

Technische Universiteit Delft

TU Delft

Concolidated Deep Actor Critic Networks

by

Tamis Achilles van der Laan

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday September 30, 2015 at 14:00.

Student number:	4057694		
Project duration:	November 1, 2014 – September 30, 2015		
Thesis committee:	Prof. dr. ir. M. Reinders,	PRB,	Chair
	Prof. dr. ir. M. Loog,	PRB,	Supervisor
	Prof. dr. ir. M. Spaan,	PRB,	External
	Prof. dr. ir. J. Kober,	DCSC,	Supervisor

I would also like to thank *Laurens van der Maaten* and *Robert Babuska* for their excellent guidance and advice regarding my thesis.

This thesis is confidential and cannot be made public until September 30, 2015.

An electronic version of this thesis is available at `<url>`.

TU Delft Delft University of Technology

Consolidated Deep Actor Critic Networks

Tamis Achilles van der Laan
Delft University of Technology, 2628 CB Delft, The Netherlands.
tamis.vanderlaan@bisc.nl

***Abstract*—The works [20], [21] have demonstrated the power of combining deep neural networks with Watkins Q learning. They introduce deep Q networks (DQN) that learn to associate High dimensional inputs with Q values in order to produce discrete actions, allowing the system to learn complex strategies and play Atari games such as Breakout and Space invaders. Although powerful the system is limited to discrete actions. If we wish to control more complex systems like robots we need the ability to output multidimensional continuous actions. In this paper we investigate how to combine deep neural networks with actor critic models which have the ability to output multidimensional continuous actions. We name this class of systems deep actor critic networks (DACN) following the DQN naming convention. We derive and experiment with four methods to update the actor. We then consolidate the actor and critic networks into one unified network which we name consolidated deep actor critic networks (C-DACN). We hypothesize that consolidating the actor and critic networks might lead to faster convergence. We test the system in two environments named Acrobot (under actuated double pendulum) and Bounce (continuous action Atari Breakout look alike).**

***Index Terms*—reinforcement learning, actor critic models, artificial neural networks, convolution networks, deep learning, experience replay**

I. Introduction

A wide range of problems can be modeled as Markov decision processes (MDPs). These systems can be viewed as a modification of the markov process where the transition probability from the current state to the next is influenced by a decision or action that can be selected. A reward is associated with each state transition and the goal is to learn the best decision/action for each state called the policy as to maximize the accumulated rewards. algorithms that learn such policies are called reinforcement learning (RL) algorithms. For a detailed introduction to reinforcement learning see [32]. In order to apply RL to problems with a high dimensional input space, function approximators (FAs) are used. The most successful of these are based on simple linear FAs and hand crafted features, the construction of which heavily relies on experience and domain expertise. Advances in the field of artificial neural networks (ANN) and deep learning has made it possible to train large and deep networks which can be used as FA for RL. For a detailed introduction to deep learning see [3]. The benefit of these FAs is that they learn features autonomously eliminating the need for feature engineering and limit the need for domain knowledge. The deep Q network (DQN) [20], [21] is the first successful system that combines RL and ANN and applies this system in order to learn to play Atari 2600 games with human level performance. Although impressive the DQN system is limited to discrete actions only. If we wish to control more complex environments such in the case of robotics we require the ability to output multidimensional continuous actions. A class of RL algorithms called actor critic models (ACM) have been devised that can deal with multidimensional continuous action spaces. They do this by parameterizing the policy separately from the value function. The policy is named the actor and the value function is called the critic. In this paper we adapt the fundamental ideas comprising the DQN system to actor critic models. We derive a new system we call Deep Actor Critic Networks in line with the DQN naming convention. We derive four methods for updating the parameters of the actor. We also experiment with consolidating the NN of the actor and the critic allowing them to share common features. We hypothesize that this might lead to faster convergence. Both variants and all four update methods are then evaluated based on two environments with continuous action spaces called Bounce and Acrobot.

II. Markov decision process

A discrete time Markov process (DT-MP) [12] consists of a pair (S, P) where S defines the set of all possible states the agent can be in at time t ($\mathbf{s}_t \in S$). This set can be finite or infinite in which case we talk about a discrete time finite state Markov process or a discrete time infinite state Markov process respectively. The set P is a probabilistic mapping $\mathrm{P} : \mathrm{S} \times \mathrm{S} \rightarrow [0, 1]$ which represent the probability of transitioning from the current state $\mathbf{s}_t$ to $\mathbf{s}_{t+1}$. For the DT-MP, time is defined to start at zero and runs in whole integer steps ($t \in \mathbb{N}^+$). The DT-MP obeys the Markov Property which states that the the next state $\mathbf{s}_{t+1}$ only depends on the previous state $\mathbf{s}_t$ at any time t. An example would be a DT-MP model of a particle in space being pushed around by other particles. The agent represents the particle in space that transitions from it's current state/position to the next state/position based on a multinomial Gaussian around it's current position.

A discrete time Markov decision process (DT-MDP) extends the DT-MP with decision theory in order to model sequential decision problems. It does this by extending the DT-MP with actions and rewards. The DT-MDP is defined by the tuple $(\mathrm{S}, \mathrm{A}, \mathrm{R}, \mathrm{P})$. S is the set of all possible states as we have seen before. A is the set of all actions that the agent can perform in time step t ($a_t \in \mathrm{A}$). $\mathrm{R}(\mathbf{s}_t, a_t, \mathbf{s}_{t+1})$ is a mapping $\mathrm{R} : \mathrm{S} \times \mathrm{A} \times \mathrm{S} \rightarrow \mathbb{R}$ which represents the reward the agent gains associated with a transition from state $\mathbf{s}_t$ to $\mathbf{s}_{t+1}$ and action a_t. $\mathrm{P}(\mathbf{s}_t, a_t, \mathbf{s}_{t+1})$ is a probabilistic mapping $\mathrm{P} : \mathrm{S} \times \mathrm{A} \times \mathrm{S} \rightarrow [0, 1]$ which represent the probability of transitioning from state $\mathbf{s}_t$ to $\mathbf{s}_{t+1}$ based on the action

at selected by the agent. Note that the sets in the tuple $(\mathrm{S}, \mathrm{A}, \mathrm{R}, \mathrm{P})$ defining the DT-MDP is fixed and independent of time. Because of this fact we say that the DT-MDP is stationary. An example of a MDP is Grid World. Grid world consists of an agent that lives in a finite state space in the form of a finite grid. The agent can move from it's current state/cell to one of the adjacent states/cells in each time step. The agent gains positive or negative reward by moving to specific squares on the grid. In decision theory we are interested in making one or more decisions as to maximize expected utility, reward or negative Loss. In the case of DT-MDP we wish to maximize the expected discounted cumulative reward under the deterministic policy π which maps states to actions ($\pi : \mathrm{S} \rightarrow \mathrm{A}$). The difference between DT-MDP and standard decision theory lies in the fact that we use a policy which uses state information and is capable of recovering from bad decisions. Using standard decision theory we can only make decisions ahead of time which is called planning. The decisions will be independent of the current state and hence cannot recover from bad decisions. The expected cumulative discounted reward (ECDR) is defined as:

$$\mathrm{V}^{\pi}(\mathbf{s}) = \mathbb{E}_{\mathrm{P}}[\sum_{k=0}^{T} \gamma^k r_{t+k+1} | \mathbf{s} = \mathbf{s}_t, \pi]$$

Here r_t is the reward received at time t. The discount factor γ is bounded $(0 \leq \gamma < 1)$ and bounds the ECDR to the set of real numbers in the infinite horizon case $T \rightarrow \infty$ given that the reward signal is bounded:

$$\frac{\min(\mathrm{R})}{1-\gamma} \leq \mathrm{V}^{\pi}(\mathbf{s}_t) \leq \frac{\max(\mathrm{R})}{1-\gamma}$$

The discount factor can also be thought of as a parameter that specifies the importance between long and short term rewards.

If we cannot completely observe the state but instead can only observe a part of the state the MDP model no longer suffices. In this case the environment should be modeled as a partially observable Markov decision process (POMDP). If adversarial agents are present a MDP is also no longer sufficient as the policy of the adversary is not necessarily fixed. In such a case a stochastic game (SG) [27] can be used to model the environment. In this work we limit our selfs to a MDP environment. Note that learning a deterministic policy is sufficient to solve a MDP. A deterministic policy is however insufficient to solve SG's and can improve performance on POMDP's [30].

III. Reinforcement learning

A. Bellman equation

The Bellman equation allows us to solve the MDP by means of dynamic programming (DP). It defines the value function V in the current state as the ECDR under the current deterministic policy π:

$$\begin{aligned}
\mathrm{V}^{\pi}(\mathbf{s}) &= \mathbb{E}_{\mathrm{P}}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | \mathbf{s} = \mathbf{s}_t, \pi] \\
&= \mathbb{E}_{\mathrm{P}}[r_{t+1} + \gamma \cdot \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | \pi] \\
&= \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}')[\mathrm{R}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}') + \\
&\quad \gamma \cdot \mathbb{E}_{\mathrm{P}}[\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+2} | \pi]] \\
&= \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}')[\mathrm{R}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}') + \gamma \cdot \mathrm{V}^{\pi}(\mathbf{s}')]
\end{aligned}$$

The Bellman equation shows the value function under the current policy can be decomposed in terms of itself. This makes solving for V under the current policy π amenable to DP. In order to find the optimal policy π^*, we look for the policy that maximizes the value function:

$$\begin{aligned}
\mathrm{V}^{*}(\mathbf{s}) &= \max_{\pi} \mathrm{V}^{\pi}(\mathbf{s}) \\
&= \max_{\pi(\mathbf{s})} \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}')[\mathrm{R}(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}') + \gamma \mathrm{V}^{*}(\mathbf{s}')]
\end{aligned}$$

This last equation is called the Bellman optimality equation. Given that we computed V^* the optimal policy is now equal to selecting the action with the highest value:

$$\pi^{*}(\mathbf{s}) = \arg\max_{a} \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \mathrm{V}^{*}(\mathbf{s}')]$$

These equations were as the name implies first derived by Richard Bellman in 1957 [2].

B. Value iteration

Its clear we require the optimal value function in order to find the optimal policy. Although the value function can be solved for exactly using DP, it can also be approximated using value iteration (VI). VI works by using a look up table $\hat{\mathrm{V}}$ with a value for each state often initialized to zero. We iterate over each state and compute the next value of $\hat{\mathrm{V}}$ based on the Bellman optimality equation. The new values of $\hat{\mathrm{V}}$ now lay closer to the optimal value V^*:

$$\hat{\mathrm{V}}(\mathbf{s}) \leftarrow \max_{a} \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \hat{\mathrm{V}}(\mathbf{s}')]$$

The policy is selected based on our estimate $\hat{V}$:

$$\hat{\pi}(\mathbf{s}) = \arg\max_{a} \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \hat{\mathrm{V}}(\mathbf{s}')]$$

C. Q iteration

Note that in order to execute the current policy π we are required to compute the action that leads to the highest expected value by looking on step forward using the DT-MDP. We can eliminate this computational overhead, trading it for a higher memory requirement, by defining a value function dependent both on the current state and action. This value function is known as the Q value ($\mathrm{Q} : \mathrm{S} \times \mathrm{A} \rightarrow \mathbb{R}$). First we remove the maximum action argument using the Q function:

$$\mathrm{Q}^*(\mathbf{s}, a) = \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \mathrm{V}^*(\mathbf{s}')]$$

Notice that $\mathrm{V}^*(\mathbf{s}') = \max_{a'} \mathrm{Q}^*(\mathbf{s}', a')$ hence we can replace the optimal value function and get an expression solely in terms of Q:

$$\mathrm{Q}^*(\mathbf{s}, a) = \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \max_{a'} \mathrm{Q}^*(\mathbf{s}', a')]$$

The equivalent to value iteration for Q learning is called Q iteration (QI):

$$\hat{\mathrm{Q}}(\mathbf{s}, a) \leftarrow \sum_{\mathbf{s}' \in S} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \max_{a'} \hat{\mathrm{Q}}(\mathbf{s}', a')]$$

Finding the optimal policy in the current state s now comes down to a lookup which does not require explicit knowledge of the model:

$$\hat{\pi}(\mathbf{s}) = \arg\max_{a} \hat{\mathrm{Q}}(\mathbf{s}, a)$$

Q iteration was introduced by Watkins in 1989 [37] together with Q learning which we introduce in the next subsection. A clear limitation of QI is the action selection by the policy. The policy operates by finding the action with the maximum Q value. For finite size action sets of small size finding the action with the maximum Q value is not a problem. However for large to infinite sized action sets finding the action with a maximum Q value becomes intractable.

D. Temporal difference learning

QI requires knowledge of all the possible states S, the transition function P and reward function R in order to find a good estimate of Q^*. The standard QI algorithm is thus model dependent. temporal difference learning (TDL) allows us to perform QI without explicit knowledge of S, P, R by means of sampling experience tuples $(\mathbf{s}, a, r, \mathbf{s}')$. By exploring the unknown DT-MDP by means of following the current policy or some behavioral distribution β the agent receives for each time step a experience tuple. This tuple consists of $\mathbf{s} \in \mathrm{S}$, $a \in \mathrm{A}$, $r \in \mathrm{R}$, $\mathbf{s}' \in \mathrm{S}$ and provides enough information to perform a temporal difference update. The temporal difference update is based on the temporal difference error δ. The error is based on the difference between the old value and the newly computed value. temporal difference learning can be used to learn the Q function as followed:

$$\begin{aligned} \delta &= r + \gamma \max_{a'} \hat{\mathrm{Q}}(\mathbf{s}', a') - \hat{\mathrm{Q}}(\mathbf{s}, a) \\ \hat{\mathrm{Q}}(\mathbf{s}, a) &\leftarrow \hat{\mathrm{Q}}(\mathbf{s}, a) + \alpha\delta \end{aligned}$$

Where α is called the learning rate ($0 < \alpha < 1$) and is used to keep a running average in order to take into account the transition probability P. In order to converge to the correct Q value, the learning rate α should decay as $t \rightarrow \infty$. Note that the sum over states falls away because experience tuples are automatically sampled according to the transition probability distribution P. The policy remains the same as before:

$$\hat{\pi}(\mathbf{s}) = \arg\max_{a} \hat{\mathrm{Q}}(\mathbf{s}, a)$$

This method is called Q learning (QL). Q learning is said to be model free, on-line and off-policy. It is model free as it does not requires a explicit model of the DT-MDP and on-line because it only utilizes the latest required experience tuple to update it's Q values, after which the experience tuple is discarded. QL is off-policy because it can act according to a behavioral distribution β and is not constrained to act based on it's policy π. Choosing between acting according to some behavioral distribution or the current policy is called exploration and exploitation respectively. The choice of acting according to the behavioral distribution or current policy determines the quality of the samples leading to a better policy estimate and the total accumulated reward gathered in one episode of play. The choice when to select between exploration and exploitation is called the exploration exploitation trade-off. Different methods of choosing between exploration and exploitation have been developed with different properties.

E. Temporal difference learning using function approximation

TDL makes use of lookup tables to capture the Q values. This method of computing Q values becomes unpractical as the number of possible states grows very large. In fact it becomes impossible to use when the state space is continuous and hence the set of possible states S is infinite. In these cases we can represent the Q values using a function approximator (FA). A FA makes use of a parameter vector $\boldsymbol{w}$ to shape the approximator. The goal is to shape the FA in order to match the true underlying Q values as good as possible. The error between the estimate of the FA and the experience tuple is called the temporal difference error:

$$\delta = r + \gamma \cdot \max_{a'} \hat{\mathrm{Q}}_{\boldsymbol{w}}(\mathbf{s}', a') - \hat{\mathrm{Q}}_{\boldsymbol{w}}(\mathbf{s}, a)$$

Gradient descent can be used to train the FA. This is done by performing gradient descent on some loss function based on the temporal difference error δ, for example the mean squared error loss function:

$$\mathfrak{L} = \frac{1}{2}\delta^2$$

Applying gradient descent we get the following update rule for the FA parameter vector $\boldsymbol{w}$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \cdot \delta \cdot \frac{\partial \mathfrak{L}}{\partial \boldsymbol{w}} \hat{\mathrm{Q}}_{\boldsymbol{w}}(\mathbf{s}, a)$$

FAs applied to TDL suffer from the following problems:

1) *Catastrophic Forgetting:* Many FAs trained on one function, then trained on a second function tend to forget the first function. The same holds when trained on parts of one function in sequential order. This becomes problematic in the context of on-line TDL using a FA, because the FA is trained in the same order as experience tuples are observed. Due to the ordering the FA tends to forget it's previously learned Q values. Catastrophic Forgetting is predominantly problematic in ANN as modifying the parameters in one layer may effect the layer above [28].
2) *Oscillations:* Updating Q values may corrupt other Q values. Correcting these Q values may corrupt the Q values we updated in the first place. This interplay between updating and corrupting can cause oscillations in the Q values stopping the learning process.
3) *Divergence:* The parameters of FAs trained on the temporal difference error using gradient descent are susceptible to divergence. The problem is best illustrated by the star network introduced in the work [1]. Figure 1 shows the star network, which consists of 6 states with a reward function that is always zero. The FA is linear in parameters w_0 to w_6 and should learn to set all it's parameters to 0. Each state is approximated by a linear combination of two parameters. During training each transition is observed equally often. All the values of the parameters are initialized positive. Parameter w_6 initial value is much larger than the other parameter initial values. When the system is trained w_0 will be updated and increased 5 times while w_6 is updated only once, causing all the parameters to diverge to positive infinity while parameter w_6 will diverge to negative infinity.

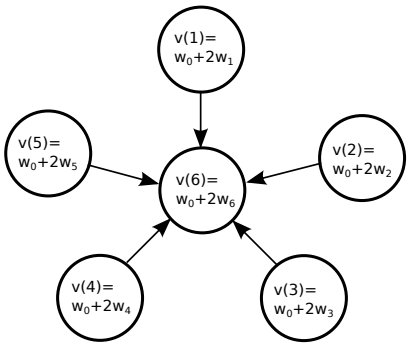


Figure 1: The Star problem

The problem of catastrophic Forgetting in RL was addressed by Lin in 93 [17] who introduced the concept of experience replay which can be seen as a version of Sweep Rehearsal [22] applied to RL. The idea behind experience replay is to build a database of experience tuples called replay memory. Batches of experience tuples are sampled randomly from replay memory in order to brake correlations between experience tuples. These batches are used to train the FA, the act of which is called experience replay. This way old experiences are no longer forgotten by the FA as they are replayed.

The second and third problem of oscillation and divergence was solved by Gordon in 95,99 [10], [11]. His technique called fitted value iteration (FVI), which separates value iteration from the act of function approximation. Instead of interleaving value iteration and function approximation one after another, he proposed to first do a step of value iteration to compute the target values and then do multiple steps of function approximation to fit the FA to the target values. The stopping criteria can then be a error threshold or a maximum number of iterations. The same method can also be used for Q learning, called fitted Q iteration (FQI) [7].

IV. Deep Learning

Many standard FAs make use of a combination of a linear mapping with parameter matrix W, input vector $\boldsymbol{x}$ and a element wise non-linear function ϕ in order to produce approximated values ($\hat{\mathbf{y}} : \mathbb{R}^N \rightarrow \mathbb{R}^M$) which maps an N dimensional input space to a M dimensional output space:

$$\hat{\mathbf{y}}(\boldsymbol{x}) = \phi(W^T \boldsymbol{x})$$

Note that if we want to use a bias terms we add another row of parameters to W and append the constant 1 to the the end of the input vector $\mathbf{x}$. In case we are interested in approximation of multiple inputs simultaneously the input $\mathbf{x}$ and the output $\hat{\boldsymbol{y}}$ become matrices:

$$\hat{Y} = \phi(W^T X)$$

The parameters of this general form of FAs are often adjusted using Gradient Descent based on a chosen Loss function $\mathfrak{L}$ in order to better approximate a given data set $\mathfrak{D}$ of input output pairs:

$$\mathfrak{D} = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), ..., (\mathbf{x}^{(n)}, \mathbf{y}^{(n)})\}$$

Gradient Descent is performed by computing the gradient of the parameters W with respect too the Loss function $\mathfrak{L}$. After which the parameters are updated by adding the negative gradient weighted by a constant learning rate α too the current parameters:

$$W_i \leftarrow W_i - \alpha \frac{\partial \mathfrak{L}}{\partial W_i}$$

Examples of such systems are linear regression, were the activation function is the identity function and the loss function equal to the mean squared error. Another example is logistic regression where the activation is equal to the logistic function and the loss function equal to the log likelihood. The more popular FA called the support vector machine/regression is also an example of this general form.

A. Artificial Neural Networks

One way of interpreting artificial neural networks (ANN) is to look at ANN as an extension of the general form by stacking these individual FAs on top of each other were the output of the previous layer forms the input for the next layer. The equation for the l'th layer is defined as:

$$\begin{aligned}\hat{\mathbf{y}}_l(\hat{\mathbf{y}}_{l-1}) &= \phi(W_l^T \hat{\mathbf{y}}_{l-1}) \\ \hat{\mathbf{y}}_0 &= \mathbf{x}\end{aligned}$$

Such a layer is also referred to as a fully connected layer for reasons that will clear in a later section. The ANN is thus defined as a stack of arbitrary many general FAs with an arbitrary number of outputs per layer. The output off the complete ANN is computed by passing the input through the layers to the top layer, the process of which is called forward propagation. Again we can chose an arbitrary loss function and use gradient descent for optimization. In order to utilize gradient descent we must compute the gradient with respect to the parameters of each layer. This is less trivial then before as the parameters of one layer are now dependent on the layers above. The algorithm for computing the gradient of the parameters with respect to the loss is called the backpropagation algorithm the invention of which often credited to Werbos in 74 [39] and comes down to applying the chain rule for each layer.

In order to compute the derivative of the parameters with respect tot the loss we start by realizing that the loss of a arbitrary layer l is dependent on the actual loss function and the layers above $\mathfrak{L}(\hat{\mathbf{y}}_l)$. If we take this loss function for the layers above as given we can compute the gradient of the parameters with respect to this loss function as followed:

$$\begin{aligned}\frac{\partial \mathfrak{L}}{\partial W_{l,i,j}} &= \frac{\partial \mathfrak{L}}{\partial \hat{y}_{l,i}} \cdot \frac{\partial \hat{y}_{l,i}}{\partial W_{l,i,j}} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{y}_{l,i}} \cdot \phi'(w_{l,i}^T \hat{\mathbf{y}}_{l-1}) \cdot \frac{\partial w_{l,i}^T \hat{\mathbf{y}}_{l-1}}{\partial W_{l,i,j}} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{y}_{l,i}} \cdot \phi'(w_{l,i}^T \hat{\mathbf{y}}_{l-1}) \cdot \hat{y}_{l-1,j}\end{aligned}$$

Note that $w_{l,i}^T$ denotes the i'th transposed column of the matrix W_l. We see that in order to compute the gradient of the parameters we require the derivative of loss function with respect to the layers above ($\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l}$). If we compute the derivative of the loss function from the point of view of the layer above $l+1$, we see that we can express the derivative of the loss function ($\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l}$) in terms of the derivative of derivative of the loss function of the layer above ($\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}}$):

$$\begin{aligned}\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l} &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot \frac{\partial \hat{\mathbf{y}}_{l+1}}{\partial \hat{\mathbf{y}}_l} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot Diag[\phi'(W_{l+1}^T \hat{\mathbf{y}}_l)] \cdot \frac{\partial W_{l+1}^T \hat{\mathbf{y}}_l}{\partial \hat{\mathbf{y}}_l} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot Diag[\phi'(W_{l+1}^T \hat{\mathbf{y}}_l)] \cdot W_{l+1}^T\end{aligned}$$

This forms the core of the backpropagation algorithm as we can start at the top layer, compute the derivative of the loss function and then propagate this derivative down to compute the local loss for each layer based on the known derivative of the loss in the layer above. The local loss is then used to compute gradients with respect to the parameters. The schematic in Figure 2 exemplifies the process for a 3 layer ANN. ANNs form a complete end to

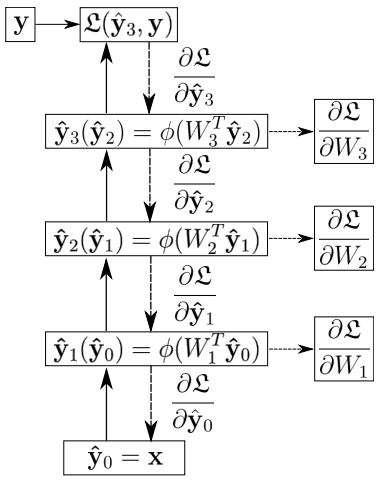


Figure 2: Example of forward and backpropagation in a 3 layer neural network used for computing the output $\hat{\mathbf{y}}_3$ and the derivative of the parameters with respect to loss.

end differentiable learning system. Theoretically they can solve any approximation problem as shown by the universal approximation theorem [5]. In practice however ANNs performed suboptimal compared to other methods such as support vector machine/regression (SVM/SVR) by Vapnik [4]. Using hand engineered features and or using kernels, the SVM/SVR allowed for superior performance over ANNs on many tasks. Around 2009 ANNs made a comeback with superior performance on several benchmark datasets.

This resurgence is mainly due to three important aspects that kept ANN performance down initially:

1) *Computational Power:* In order to gain good performance on interesting datasets large ANN with multiple layers are required. In the early days of ANN development the computational power to construct and learn such ANN was not available. Due to the exponential increase in computing power and the introduction of GPU computing it became possible to train the large ANN required for good performance.
2) *Dataset Size:* It turns out that the size of the dataset used to tune the ANN is important. In the classic case were we combine feature engineering and the SVM/SVR humans utilize their knowledge of the world to extract suitable features for the SVM which leads to good performance. The ANN has no such knowledge of the world and therefore requires larger amounts of training examples to learn the same features.
3) *Gradient Saturation:* It was discovered that the activation function used at each layer of the ANN can hamper

learning performance. This was most predominantly noticeable with the two most widely used activation functions, the sigmoid and TANH activation functions. In the backpropagation process we compute the new derivative of the loss function for the layer below the current layer by multiplying with the derivative of the activation function $\phi'(W_{l+1}^T \hat{\mathbf{y}}_l)$. If this derivative is very small in each layer the derivative of the loss function for each layer will also become small and cause extremely slow learning rates or cause the gradient to die out completely. Small gradients can also cause numerical instability. Several methods have been developed to counteract this problem. Examples of such methods are ReLU [23] which due to there linearity do not suffer from gradient saturation, proper parameter initialization [31] which avoids regions where the gradient saturates and Batch Normalization [14] which adds another learnable parameter which is used to escape from saturated gradients.

Deep learning is a branch of machine learning which came into being together with the resurgence of ANNs. Deep learning refers to the concept of learning hierarchies of structure or computation. The aspect of ANNs referred to as deep learning are the layers of general FAs. The idea is that features in a layer will be based on features in lower layers leading to a system that is combinatorial in nature.

B. Convolution neural networks

Convolution neural networks were inspired by the work of Hubel and Wiesel [13] on the cats visual cortex. They showed neurons in the visual cortex that fired based on simple patterns like edges in subregions of the visual field. Neurons that detect the same pattern tiled over the visual field were found. Different neural network models which replicated this behavior were invented the first often accredited too Fukushima in 1980 [9] and the more well known version [16]. With convolution artificial neural network (CANN) we indicate a ANN which makes use of one or more convolution layers (CL). The idea behind a CL is based on the observation that if we train a ANN for example on image data, we find by means of inspection that each layer specializes in the detection of certain patterns or structures the kinds of which depends on the topology of the ANN. So each element in the co-domain of each layer outputs a value that indicates if a certain feature, structure or pattern is present in the layer below. The features learned display two kinds of properties, the first of which is called locality. Locality refers to the idea that these features are triggered by patterns in the input that are near each other. For example, if the input is a image and we inspect the co-domain of the first layer of the ANN a value in the co-domain might be triggered that detects a diagonal edge feature in the upper right corner. The feature ignores all other information in the image and only looks at the upper right corner of the image setting all other parameters associated with the rest of the image to zero (Figure 3).

The second aspect is parameter duplication, which refers to the same features being learned in different locations of the input and hence have the same parameters. For example if we go back to the example which detects a diagonal edge in the upper right corner we might find another feature detector which also detects a diagonal edge but in the lower right corner of the input image (Figure 3).

Figure 3: The figure shows a hypothetical case where the parameters of two element from the first layer learned to detect approximately the same diagonal edge in two different locations of the input image.

Convolution artificial neural networks exploits these redundancies by learning local position invariant features by means of convolution. It works by specifying the size of the feature window which is convolved over the input with a specified stride. Each window has it's own parameters and outputs the result of the convolution with the input for the next layer to process. One can liken the process too applying the general FA for a smaller input many times over the input in different locations to produce the new output. Because the input to the CL can have different structures for example a 2d input structure in the case of image data or a 3d structure in the case of voxel data the convolution layer is defined in terms of tensors. However for simplicity and clarity we derive the equations over a one dimensional input using one filter which reduces to discrete 1D convolution:

$$\begin{aligned} \hat{\mathbf{y}}_l(\hat{\mathbf{y}}_{l-1}) &= \phi(\mathbf{w}_l * \hat{\mathbf{y}}_{l-1}) \\ \hat{\mathbf{y}}_0 &= \mathbf{x} \end{aligned}$$

Because a CL uses convolution, the number of window parameters $\mathbf{w}_l$ can be set very small leading to a radically reduced number of parameters compared to the original fully connected layer. This leads to a significant increase in learning speed and allows us to construct much larger networks. Training a convolution layer like a fully connected layer requires computation of the gradient of the loss function with respect to the parameters, which is done in the following manner:

$$\begin{aligned}\frac{\partial \mathfrak{L}}{\partial w_{l,i}} &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l} \cdot \frac{\partial \hat{\mathbf{y}}_l}{\partial w_{l,i}} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l} \cdot Diag[\phi'(\mathbf{w}_l * \hat{\mathbf{y}}_{l-1})] \cdot \frac{\partial \mathbf{w}_l * \hat{\mathbf{y}}_{l-1}}{\partial w_{l,i}} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{y}_{l,i}} \cdot Diag[\phi'(\mathbf{w}_l * \hat{\mathbf{y}}_{l-1})] \cdot \begin{bmatrix} \hat{y}_{l-1,N-i+1} \\ \vdots \\ \hat{y}_{l-1,M-i+1} \end{bmatrix}\end{aligned}$$

In order to compute the gradient of the parameters with respect to the loss we again need to compute the derivative of the loss with respect to the layers above the current layer $\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l}$:

$$\begin{aligned}\frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_l} &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot \frac{\partial \hat{\mathbf{y}}_{l+1}}{\partial \hat{\mathbf{y}}_l} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot Diag[\phi'(\mathbf{w}_{l+1} * \hat{\mathbf{y}}_l)] \cdot \frac{\partial w_{l+1} * \hat{\mathbf{y}}_l}{\partial \hat{\mathbf{y}}_l} \\ &= \frac{\partial \mathfrak{L}}{\partial \hat{\mathbf{y}}_{l+1}} \cdot Diag[\phi'(\mathbf{w}_{l+1} * \hat{\mathbf{y}}_l)] \cdot \begin{bmatrix} flip[\mathbf{w}_{l+1}^T] \\ \vdots \\ flip[\mathbf{w}_{l+1}^T] \end{bmatrix}\end{aligned}$$

Apart from multi-dimensional convolution and multiple output filters we can also use a stride larger then 1 for convolution. A stride larger then 1 allows for a faster computation time at the cost of accuracy. In our example we have use convolution without padding, which leads to a accuracy loss around the edges.

V. Deep Q networks

Many have worked on combining ANN and RL algorithms. A very successful system was created by Tesauro in 95 called TD-GAMMON [34] which combined VI, TD learning and ANNs to learn to play backgammon and achieved top human level play. In 93 Lin combined WQL with robotics and introduced the concept of experience replay [17]. Although impressive the lack of computational resources meant that the system could only deal with low complexity environments. FQI was combined with ANNs in a system called neural fitted Q iteration (NFQI) [25] which allowed for more accurate approximations of Q values. All these systems were successful combining ANN and RL on a small scale. A truly impressive result on a large scale was achieved by DeepMind in 2014[20], [21] who utilized GPU's and new developments in ANN such as ReLU to create a system that learns to play Atari 2600 games often with human level performance by capturing the last four frames of the screen as the current state/input. They combined Q learning, experience replay, a modified version of FQI and CANN to build a powerful system called deep Q networks (DQN). DQNs use a modified version of FQI where instead of using a separate preconstructed target value set two neural networks are used. The first neural network $\hat{\mathrm{Q}}$ with parameters $\boldsymbol{\theta}^-$ produces the target values while the second neural network $\hat{\mathrm{Q}}$ with parameters $\boldsymbol{\theta}$ is trained on these target values. After a fixed number of iterations C the parameters $\boldsymbol{\theta}^-$ are replaced with the updated parameters $\boldsymbol{\theta}$. Ignoring specifics such as pre-processing and initialization and using gradient descent for illustrative purposes we get algorithm 1.

Algorithm 1 DQN

For $t = 1$ to ∞ **do**:

1) With probability ϵ select a random action a_t. Otherwise select $a_t = \arg\max_a \hat{\mathrm{Q}}(\mathbf{s}_t, a|\boldsymbol{\theta})$
2) Execute action a_t and observe reward r_t and state $\mathbf{s}_{t+1}$.
3) Store experience tuple $(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1})$ in $\mathfrak{D}$.
4) Sample a random experience tuple from replay memory $(\mathbf{s}, a, r, \mathbf{s}') \sim \mathfrak{D}$.
5) Compute $\delta = r + \gamma \cdot \max_a \hat{\mathrm{Q}}(\mathbf{s}', a|\boldsymbol{\theta}^-) - \hat{\mathrm{Q}}(\mathbf{s}, a|\boldsymbol{\theta})$.
6) Update parameters:
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{\partial \frac{1}{2}||\delta||_2^2}{\partial \boldsymbol{\theta}}$$
7) Every C steps transfer parameters $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$.

DQNs makes use of a deep CANN the architecture of which van be seen in Figure 4. Because the replay memory is not fixed but instead grows dynamically standard gradient descent wont work properly as the required learning rate depends on the size and variability of the target values. The NFQI algorithm dealt with this by utilizing the RPROP algorithm by Riedmiller [26] which is far more robust to dynamically changing training sets. The DQN algorithm uses a batch variant of this algorithm called the RMSPROP algorithm invented by Tieleman but never officially published [35]. An interesting aspect of the DQN system is that we can

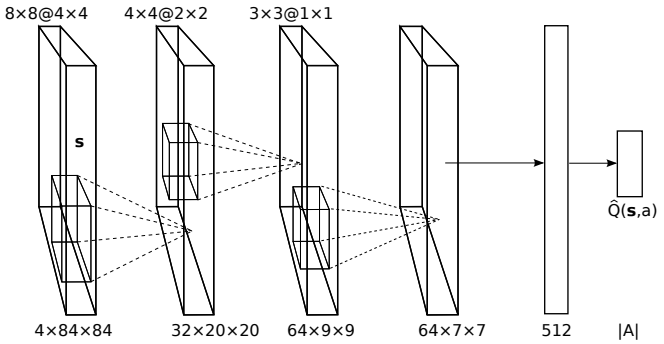


Figure 4: The DQN architecture is composed of three convolution layers and two fully connected layer. All layers use the ReLU activation function except for the last layer which uses the identity activation function. The input consists of 4 temporal sequential 84×84 black and white images of the game and the output layer generates the Q values associated with each possible action. The bottom shows the output dimensions of the layer and the top shows the window and stride used in the convolution.

utilize ANN architectures developed for different domains such as Object Detection and speech recognition and with a few modifications incorporate them directly into the DQN system. This then allows one to design agent that operate in different domains dealing with with different types of inputs received from the environment.

VI. Actor critic models

The DQN system showed immense success at learning to play Atari 2600 games. However the system is based on WQL and thus can only deal with discrete actions. If we wish to control continuous action systems like robots we cannot use the DQN system. The bottleneck of WQL in combination with continuous actions is the need to select the maximum action for our policy. In a limited sized discrete action set this operation is fast but with continuous actions finding the maximum action is intractable.

Actor critic models (ACM) [8] are a class of RL models that separate the policy from the value approximation process by parameterizing the policy separately. The parameterization of the value function is called the critic and the parameterization of the policy is called the actor. The actor is updated based on the critic which can be done in different ways, while the critic is update based on the current policy provided by the actor. This creates coupling between the two systems. The standard ACM uses a parameterized stochastic policy $\pi : S \times A \rightarrow [0, 1]$ for the actor which is learned by using the temporal difference error provided by the critic. The temporal difference error is used to make the executed action for the current state more or less probable in the positive and negative case respectively. An example of this would be using the standard value function and a stochastic policy based on the Gibbs softmax method. The temporal difference error is defined as:

$$\delta = r_t + \gamma \mathrm{V}(\mathbf{s}_{t+1}) - \mathrm{V}(\mathbf{s}_t)$$

The policy uses the Gibbs softmax parametrization defined as:

$$\pi(\mathbf{s}, a) = \Pr[a_t = a | \mathbf{s} = \mathbf{s}_t] = \frac{e^{\kappa(\mathbf{s},a)}}{\sum_{a'} e^{\kappa(\mathbf{s},a')}}$$

Updating the stochastic policy is based on the temporal difference error:

$$\kappa(\mathbf{s}_t, a_t) \leftarrow \kappa(\mathbf{s}_t, a_t) + \alpha \cdot \delta$$

The next action is drawn according to the stochastic policy:

$$\pi(\mathbf{s}_t) \sim \pi(\mathbf{s}_t, \cdot)$$

Although successful in combination with different FAs it is impractical to combine the standard ACM with ANNs. The reason for this is the stochastic policy. The Bellman equation using a stochastic policy and a continuous action space is equal to:

$$\mathrm{V}^{\pi}(\mathrm{s}) = \int_{a \in \mathrm{A}} \pi(\mathbf{s}, a) \sum_{s' \in \mathrm{S}} \mathrm{P}(\mathbf{s}, a, \mathbf{s}')[\mathrm{R}(\mathbf{s}, a, \mathbf{s}') + \gamma \cdot \mathrm{V}^{\pi}(\mathbf{s}')]$$

We see that we need not only sum over all possible states but we also need to integrate over all possible actions. This means that in order to use TD learning the experience tuple $(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1})$ must be drawn according to the current policy. This in turn means that the method has to be on-line and on-policy in order to work. This prohibits us from using experience replay in it's current form.

The work by Degris & Sutton [6] introduce a off-policy ACM which uses a fixed behavioral distribution β for exploration in combination with importance sampling[1] (IS). When exploration is performed actions are drawn from the behavioral distribution and IS is used to transform these samples into samples drawn from the current policy distribution. We can use this system to implement experience replay by saving the parameters of the actor ($\boldsymbol{\kappa}$) within the experience tuple $(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1}, \boldsymbol{\kappa})$ it is then possible to use IS to transform the sample from the old policy into a sample drawn from the current policy. Although possible there are significant problems associated with this approach:

1) We need to save the parameters of the actor $\boldsymbol{\kappa}$ inside the experience tuple. If we make use of ANNs with a large number of parameters this becomes impractical.
2) IS can introduce bias resulting in poor estimates of the current policy derailing the learning process.
3) It makes learning from a physical teacher impossible as the teachers behavioral distribution β cannot be known.

Nonetheless systems have been constructed based on this principle, utilizing small ANNs [38], [19].

VII. Deep actor critic Network

Due to the complexities associated with stochastic policies we make use of a deterministic policy. This means we give up the possibility to produce a mixed optimal strategy when dealing with adversarial agents as described in section 2. This does however opens up the possibility to use the off-policy temporal difference error as described before together with experience replay and FQI to update the Q values. A added benefit of this method is the fact we can use an external teacher to train the system. Apart from fixing the Q values we now also fix the policy produced by the actor:

$$\delta_C = r + \gamma \cdot \hat{\mathrm{Q}}(\mathbf{s}', \boldsymbol{\pi}(\mathbf{s}|\boldsymbol{\kappa}^-)|\boldsymbol{\theta}^-) - \hat{\mathrm{Q}}(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta}) \qquad (1)$$

Both the critic and the actor are parameterized using a neural network. We can use a modified version of the DQN architecture (Figure 4) for both the actor and critic networks. This will however introduce a redundancy as there will be a large overlap between the features learned in the convolution layers of both the actor and the critic networks. In order to save computation power and potentially improve learning speed we consolidate the convolution layers of the actor and critic networks leading to the architecture in Figure 5. This allows the critic and actor to share learned features from the convolution layers. Although consolidation of the actor and critic networks might seem straight forward it introduces another form of coupling between the actor and and critic which might interfere with update process. Apart from updating the critic Q values we also need to update the actor policy. We identify one method to update the actor based on the critics gradient and two new methods based on sampling

[1] Importance sampling is a technique used to generate samples from a distribution A using samples drawn from a distribution B.

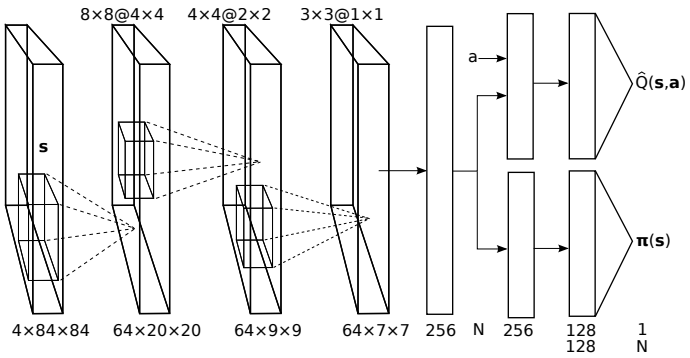


Figure 5: The DACN architecture is composed similarly to the DQN architecture (Figure 4). The main difference lays in the last layers were we split into two separate networks representing the actor and critic respectively. We also use leaky ReLU [18] instead of the standard ReLU in order to combat dead units. The parameters of the convolution layers are thus shared between the actor and the critic. Note that N indicates the action dimensionality.

and the temporal difference. We also invent a new hybrid method which combines the sample and gradient based update methods. These actor update methods are described in the next sub sections. Combining everything together we derive algorithm 2. Note that the algorithm uses a single experience tuple to update the network and gradient descent for illustrative purposes. In our implementation we use batch updates as well as ADAM [15] as our optimization technique.

Algorithm 2 (C)-DACN-(S,TD,G,H)

For $t = 0$ to ∞ **do**:

1) With probability ϵ select random action $\mathbf{a}_t$. Otherwise select $\mathbf{a}_t = \boldsymbol{\pi}(\mathbf{s}_t|\boldsymbol{\kappa})$.
2) Execute action $\mathbf{a}_t$ and observe reward r_t and state $\mathbf{s}_{t+1}$. Store experience tuple $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ in $\mathfrak{D}$.
3) Sample a random experience tuple $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}') \sim \mathfrak{D}$.
4) Compute δ_C according to eq. (1).
5) Compute $\boldsymbol{\delta}_A$ according to eq. (2, 3, 4 or 5).
6) Update the actor:
$$\boldsymbol{\kappa} \leftarrow \boldsymbol{\kappa} + \beta \cdot \frac{\partial \frac{1}{2}||\boldsymbol{\delta}_A||_2^2}{\partial \boldsymbol{\kappa}}$$
7) Update the critic:
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{\partial \frac{1}{2}||\delta_C||_2^2}{\partial \boldsymbol{\theta}}$$
8) Every C steps transfer actor critic parameters ($\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\kappa}^- \leftarrow \boldsymbol{\kappa}$).

A. Actor Updating, Sample - S

The simplest method uses samples from the experience replay directly by comparing the value of the sampled action $\hat{\mathrm{Q}}(\mathbf{s}, \mathbf{a})$ and the value produced by the policy $\hat{\mathrm{Q}}(\mathbf{s}, \boldsymbol{\pi}(\mathbf{s}))$. If the Q value associated to the sampled action is strictly larger $\hat{\mathrm{Q}}(\mathbf{s}, \mathbf{a}) > \hat{\mathrm{Q}}(\mathbf{s}, \boldsymbol{\pi}(\mathbf{s}))$ we update the parameters towards this action. We update according to the slope of the secant between the sample action and policy action in terms of the $\hat{\mathrm{Q}}$ values:

$$\boldsymbol{\delta}_A = \underset{\hat{\mathrm{Q}}(\mathbf{s},\mathbf{a})>\hat{\mathrm{Q}}(\mathbf{s},\boldsymbol{\pi}(\mathbf{s}))}{\mathbf{1}} \cdot \frac{\hat{\mathrm{Q}}(\mathbf{s}, \mathbf{a}) - \hat{\mathrm{Q}}(\mathbf{s}, \boldsymbol{\pi}(\mathbf{s}))}{\mathbf{a} - \boldsymbol{\pi}(\mathbf{s})} \tag{2}$$

Note that the devision is element wise with respect to the denominator. Using the slope of the secant rather then the difference between actions allows us to break the symmetry in cases where the current policy is caught between two peaks of equal breadth but different height. We expect the rate at which the policy is learned to diminish over time as the policy approaches the optimal policy due to the number of samples drawn from replay memory with a strictly larger $\hat{\mathrm{Q}}$ value becomes less probable. The effectiveness of this update method is highly dependent on the exploration strategy used.

B. Actor updating, temporal difference - TD

Going one step further we can use the temporal difference to update the actor by moving towards or away from actions associated with positive or negative reward respectively according to the magnitude of the temporal difference:

$$\boldsymbol{\delta}_A = \delta_C \cdot \frac{\mathbf{a} - \boldsymbol{\pi}(\mathbf{s})}{||\mathbf{a} - \boldsymbol{\pi}(\mathbf{s})||_2}$$

Work by [36] however suggests that better performance can be obtained only updating when $\delta_C > 0$. We found this could be relevant even though they use a stochastic policy.

$$\boldsymbol{\delta}_A = \underset{\delta_C>0}{\mathbf{1}} \cdot \delta_C \cdot \frac{\mathbf{a} - \boldsymbol{\pi}(\mathbf{s})}{||\mathbf{a} - \boldsymbol{\pi}(\mathbf{s})||_2} \tag{3}$$

They also suggest using the sign of the temporal difference to update the actor, i.e. neglecting the magnitude of the temporal difference. This was done in order to make the learning parameter of the actor invariant to the variation of the reward function adding stability to the learning process. This in our case is not needed as the use of the ADAM optimization algorithm [15] already makes the learning parameter of the actor invariant to variation of the reward function.

C. Actor updating, gradient - G

The last way of updating the actor is by directly applying gradient ascent on the $\hat{\mathrm{Q}}$ function. The gradient of $\hat{\mathrm{Q}}$ can be calculated by using the backpropagation mechanism. This is done by backpropagating the constant 1 through the network down to the input actions. The actor is updated as followed:

$$\boldsymbol{\delta}_A = \left.\frac{\partial \hat{\mathrm{Q}}(\mathbf{s}, \mathbf{a})}{\partial \mathbf{a}}\right|_{\mathbf{a}=\boldsymbol{\pi}(\mathbf{s})} \tag{4}$$

This method is called action dependent heuristic dynamic programming (ADHDP) and was developed by Prokhorov & Wunsch in 1997 [24]. As one can imagine it could be possible that this method becomes stuck in a local optima, converging to a suboptimal policy. Although possible one can also imagine cases where due to the strong coupling between actor and critic, in the case where the actor gets stuck in a local optima,

a "in between" policy is learned which causes the critic values to change and the actor to escape the local optima. There might also exist multiple optimal policies as well as many near optimal policies which would make this method practical.

D. Actor updating, sample/gradient hybrid - H

In a effort to improve upon the above primary update methods we add one extra update method that combines both the sample based and the gradient based update methods into one hybrid method. In this hybrid method we use the sample based method in case the Q value of the sample is larger then the Q value of the policy and otherwise we use the gradient to update method. The rational behind this approach is that the sample based method might be less susceptible to get stuck in a local optimum, but has a hard time converging to the exact optimum. Hence we use sample based updating to move towards a global optimum and gradient based updating to converge locally. Of course there is still no guaranty we reach the optimum policy. Combining both method results in the following hybrid update method:

$$\boldsymbol{\delta}_A = \begin{cases} \frac{\hat{\mathrm{Q}}(\mathbf{s},\mathbf{a}) - \hat{\mathrm{Q}}(\mathbf{s},\boldsymbol{\pi}(\mathbf{s}))}{\mathbf{a}-\boldsymbol{\pi}(\mathbf{s})} & \hat{\mathrm{Q}}(\mathbf{s},\mathbf{a}) > \hat{\mathrm{Q}}(\mathbf{s},\boldsymbol{\pi}(\mathbf{s})) \\ \frac{\partial \hat{\mathrm{Q}}(\mathbf{s},\mathbf{a})}{\partial \mathbf{a}}\Big|_{\mathbf{a}=\boldsymbol{\pi}(\mathbf{s})} & \hat{\mathrm{Q}}(\mathbf{s},\mathbf{a}) \leq \hat{\mathrm{Q}}(\mathbf{s},\boldsymbol{\pi}(\mathbf{s})) \end{cases} \tag{5}$$

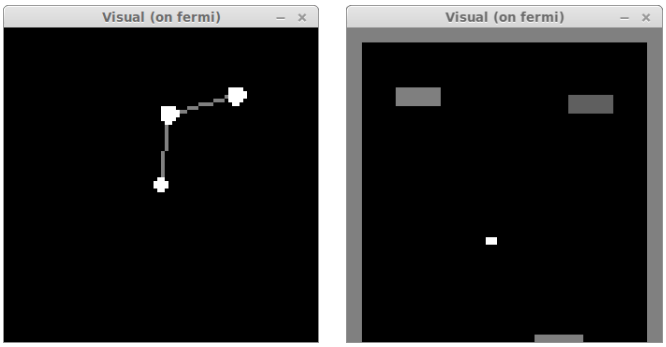
What makes this method interesting is that the sample update is a crude approximation of the derivative and hence of the same order of magnitude as the derivative which leads to a smooth integration of both methods.

E. Deterministic Policy Gradient Theorem

David Silver in the paper [29] derives a deterministic policy version of the gradient theorem for actor critic models. This theorem shows that the deterministic policy gradient is a limiting case of the stochastic policy gradient as the variance of the stochastic gradient goes to zero. If we fix the parameters of all layers except for the top layer and use the gradient based actor update eq. 4 the deterministic policy gradient theorem holds for our system and the system will convergence. Although we don't fix the parameters of the lower layers, this is a interesting fact to note.

VIII. Experiments

In order to compare the three update methods we ran experiments using two environments (Figure 6). The first environment is called Bounce and is inspired by the classic Atari game Breakout. The goal is too Bounce the ball using the paddle in such a way that it stays inside the environment while hitting the correct blocks and avoiding other blocks. The Bounce environment models a continuous action environment with delayed rewards. The second environment is called Acrobot which is a classic problem in control. The environment models a double pendulum with a actuator attached between the two arms (middle vertex). The goal is to find the policy which let's the double pendulum balance upright with both arms. The Acrobot environment models a continuous action environment



(a) Acrobot (b) Bounce

Figure 6: Acrobot and Bounce testing environments.

with direct reward feedback. We test a consolidated and non-consolidated version of the network for each of the four algorithms described in the previous section surmounting to 8 versions in total. We use a discount factor of $\gamma = 0.9$ for the Acrobot environment and a discount factor of $\gamma = 0.96$ for the Bounce environment. The learning rates were set to $\alpha, \beta = 0.000025$ for both the actor and the critic. We use a batch size of $b = 32$ and swap the parameters of the network every $C = 512$ cycles. We linearly decay the exploration rate ϵ from 1.0 to 0.0 over 100000 iterations. In between every 2000 iterations we run 2000 evaluation steps without exploration and learning in order to evaluate the current policy. The policy is evaluated by recording the average rewards and Q values. For the ADAM optimization method we set both decay rates to 0.95 and the correction term to 10^{-11}.

A. Acrobot

The Acrobot environment [33] represents an acrobat which can swing by applying torque between its torso and legs (Figure 6a). The goal of the Acrobot is to balance upright. The state of the Acrobot is specified by the angle between the base and the first arm (θ_1), the angle between the first and the second arm (θ_2) and there angular velocities $\dot{\theta}_1$ $\dot{\theta}_2$. A torque τ can be applied by a controller on the joint between the two arms. The reward function is computed as:

$$r = \frac{1}{10} \cdot \frac{1}{2} \cdot (\frac{y_1}{l_1} + \frac{y_2 - y_1}{l_2})$$

Where l_1 and l_2 denote the length of arm 1 and arm 2 respectively. Wee see that the reward is defined by the dot product between the direction of each arm with the up vector. This surmounts to a reward of 0.1 when the double pendulum is still and upright and -0.1 when the double pendulum is hanging down. The action space lays between $a \in [-1.0, 1.0]$ and corresponds to a torque between $\tau \in [-5.0, 5.0]$. The angular velocity is limited between $\dot{\theta}_1, \dot{\theta}_2 \in [-5.0, 5.0]$. The experimental results for the Acrobot environment can be seen in Figures 7, 8.

B. Bounce

The goal of the Bounce environment is to Bounce the ball in such away it does not leave the environment and only hits the

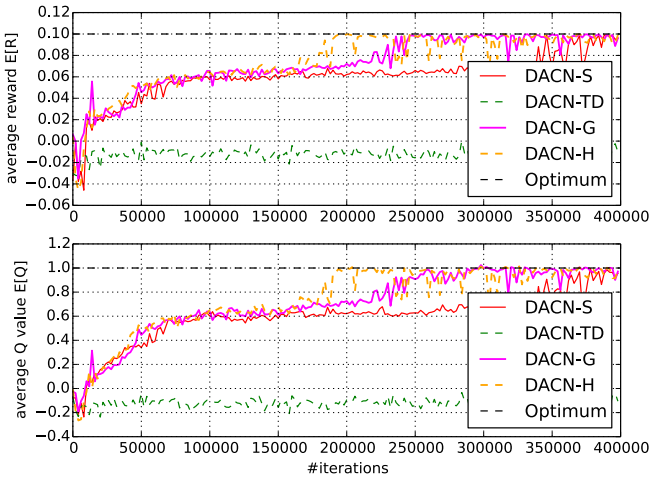


Figure 7: Experimental results for the Acrobot test environment using two separate networks for the actor and critic respectively.

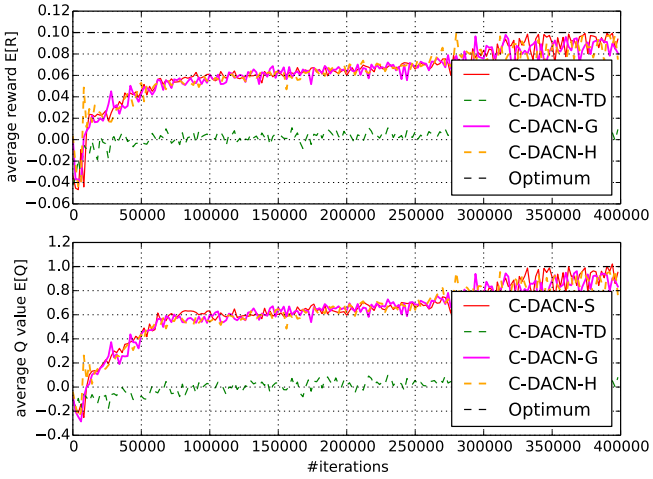


Figure 8: Experimental results for the Acrobot test environment using a single consolidated network for the actor and critic.

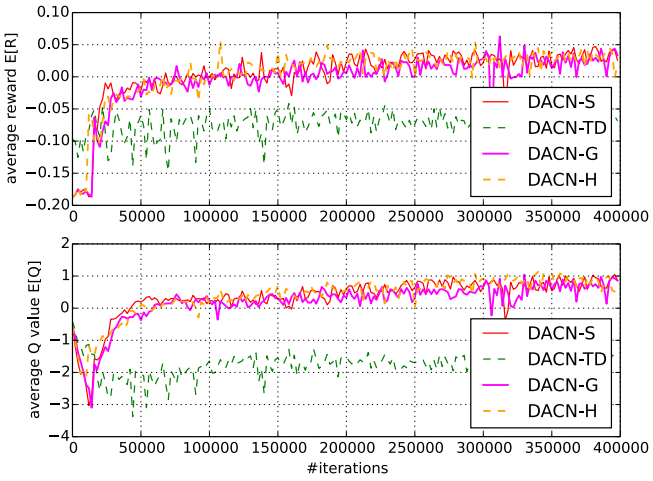


Figure 9: Experimental results for the Bounce test environment using two separate networks for the actor and critic respectively.

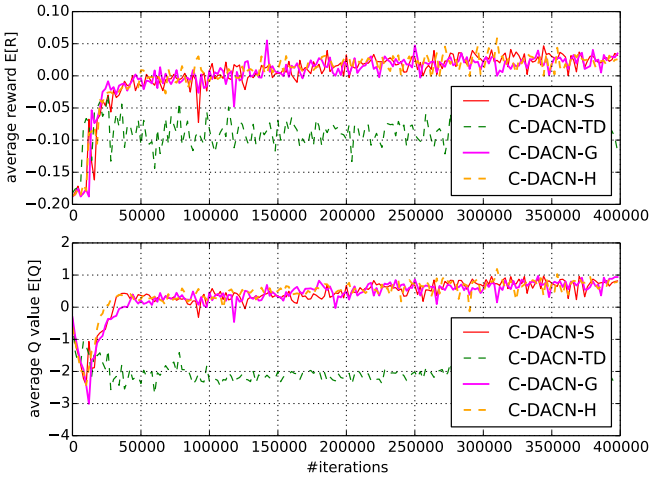


Figure 10: Experimental results for the Bounce test environment using a consolidated network for the actor and critic.

right block (Figure 6b). The left block gives -1.125 reward and the right blocks give $+1.125$ reward. When the ball leaves the environment we get -2.25 reward. And the position of the ball is reset to the center of the environment. The ball is then given a random initial velocity downwards and a smaller initial random velocity towards the sides. Three walls are present on the left right and top to keep the ball inside the environment. Thus the ball can only escape via the bottom. The paddle can be moved left or right with a continuous amount represented by the action values in the range $a \in [-1, 1]$. When the ball and the paddle collide the ball inherits a bit of horizontal velocity from the paddle which represents friction between the ball and paddle. This allows the agent a level of directional control over the ball. The experimental results for the Bounce environment can be seen in Figures 9, 10.

IX. Discussion

In Figure 7 we see the Acrobot environment experiments using the separated network architecture. We see that the sample, gradient and hybrid update methods clearly differ in terms of finding the optimal policy. We expect the hybrid method to perform best as it combines the sample and gradient based update methods allowing the method to potentially escape local minima's while at the same time converge locally. We then expect the gradient based update method to converge as it always has a local gradient available to update the actor parameters. Last we expect the sample based method to converge as it can only update itself based on samples with a $\hat{Q}$ value strictly larger then it's current $\hat{Q}$ value and hence will update more slowly as it reaches the optimum. The results in Figure 7 confirm these expectations.

When we compare the results of the separated network architecture in Figure 7 with the results of the consolidated architecture in Figure 8 we see that the performance differences between the three methods disappear. All methods seem to perform equally well. We speculate that the reason for this may be due to the fact that features are shared between the actor and critic in the convolution layers (Figure 5). This feature sharing might allow the features required by the actor to be formed by the critic equalizing the relative performance of the three actor update methods.

We hypothesized that consolidating the actor and critic networks would lead to faster convergence. However, from Figures 7, 8 we see that initially the separated and consolidated architectures perform equally well after approximately 175000 iterations the separated architecture outperforms the consolidated architecture. The cause for this could be that the actor and critic might interfere with each other in terms of opposite directional parameter updates in the shared layers. This could lead to a kind of rope pulling effect where the net resulting parameter updates become very small. One definite benefit of the consolidated architecture opposed to the separated architecture is the lower time complexity which may decrease learning time.

Figures 7, 8, 9, 10 show that the temporal difference update method for the actor does not perform well. This result is unexpected and the reason is not clear but could be explained by the fact that the critic temporal difference error never reaches zero and oscillates around the optimal Q value. This would cause the actor to always update no matter how small the temporal difference error, in effect randomizing the actor policy.

In Figure 9 we see the result for the Bounce environment using a separated architecture. We see that there is no difference between the hybrid, sample and gradient update methods. This could be due to the nature of the Bounce environment or due to inaccuracies associated with the size of the stride used in the convolution layers limiting the accuracy of the network causing suboptimal performance. It also does not reach the optimal policy, visual inspection shows that the agent hits the right block consistently but looses track of the ball once in a while.

X. Future research

The results obtained through experiments show there are differences between the update methods, but the experiments are to limited in size in order to thoroughly determine which method works best in what circumstance. In the future we wish to do more experiments and theory to get clearer picture as to which update methods work best. An example of such a experiment would be a environment where the goal is to select a white circle on the screen for a positive reward and a negative reward for a selection outside the white circle. Such an environment have a very sparse gradient and hence would be a good test case for comparing the sample and gradient based update methods. Due to the sparse gradients we would expect the sample based update methods to outperform the gradient based method.

Another interesting topic which requires more in depth research is the reason why the consolidated architecture converges later compared to the separated architecture. A deeper understanding may lead to more effective update methods improving performance which may make the consolidated architecture preferable over the separated architecture.

Lastly understanding the reason why the temporal difference update method fails may lead to a modified more successful version, which may potentially improve the convergence rate.

XI. Conclusion

In this paper we have presented a way combining deep learning with actor critic models in order to deal with continuous action spaces. We have investigated four methods to update the actor as well as consolidating the actor critic network architectures.

We hypothesized that consolidating the actor and critic networks would allow for fewer learning iterations as features between the two networks are shared. Limited experiments however imply that it takes more iterations as supposed to less. However the time complexity of the consolidated architecture is lower and thus may proof more practical.

For the update methods we hypothesized that the hybrid method would work best as it combined both global and local convergence. After which the gradient method would perform second best due to the availability of the gradient and lastly the sample based method. Our experiments seem to indicate this is also the case in practice when using a non consolidated network for the actor and critic. For the consolidated version however the differences between the methods disappear.

Experiments showed that the temporal difference update method is not applicable in this context and failed to produce a policy.

REFERENCES

[1] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the twelfth international conference on machine learning*, pages 30–37, 1995.

[2] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[4] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[5] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24, 2001.

[6] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[7] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. In *Journal of Machine Learning Research*, pages 503–556, 2005.

[8] Eugene A Feinberg and Adam Shwartz. *Handbook of Markov decision processes: methods and applications*, volume 40, pages 453–453. Springer Science & Business Media, 2012.

[9] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[10] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Proceedings of the twelfth international conference on machine learning*, pages 261–268, 1995.

[11] Geoffrey J Gordon. Approximate solutions to markov decision processes. *Robotics Institute*, page 228, 1999.

[12] Ronald A Howard. Dynamic programming and markov processes.. 1960.

[13] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[17] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[18] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[19] Francisco S Melo and Manuel Lopes. Fitted natural actor-critic: A new algorithm for continuous state-action mdps. In *Machine Learning and Knowledge Discovery in Databases*, pages 66–81. Springer, 2008.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] Ole-Marius Moe-Helgesen and Havard Stranden. Catastophic forgetting in neural networks. *Dept. Comput. & Information Sci., Norwegian Univ. Science & Technology (NTNU), Trondheim, Norway, Tech. Rep*, 2005.

[23] 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.

[24] Danil V Prokhorov, Donald C Wunsch, et al. Adaptive critic designs. *Neural Networks, IEEE Transactions on*, 8(5):997–1007, 1997.

[25] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer, 2005.

[26] Martin Riedmiller and Heinrich Braun. Rprop-a fast adaptive learning algorithm. In *Proc. of ISCIS VII, Universitat*. Citeseer, 1992.

[27] Lloyd S Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095, 1953.

[28] N Sharkey and A Sharkey. Catastrophic forgetting in connectionist networks: Causes, consequences and solutions. *Anal Catastrophic Interference*, 7(3-4):301–329, 1995.

[29] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[30] Satinder P Singh, Tommi Jaakkola, and Michael I Jordan. Learning without state-estimation in partially observable markovian decision processes. In *ICML*, pages 284–292. Citeseer, 1994.

[31] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1139–1147, 2013.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*, volume 1, chapter 11.3. MIT press Cambridge, 1998.

[34] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[35] Tieleman and Hinton. Rmsprop, coursera: Neural networks for machine learning. 2012.

[36] Hado Van Hasselt, Marco Wiering, et al. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 272–279. IEEE, 2007.

[37] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge England, 1989.

[38] Paweł Wawrzyński. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks*, 22(10):1484–1497, 2009.

[39] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.