Intelligent Systems for Pattern Recognition 2020/21 Survey Project Report

An Overview of Deep Reinforcement Learning

Pier Paolo Tarasco (619622) – p.tarasco@studenti.unipi.it

Master Degree in Computer Science

Date: 27/01/2022

Abstract

In this short survey we try to analyze 3 different approaches to train smart agents. The first one is based on learning the value function [2], the second one on learning directly a policy [1] and the third one integrates the two previous techniques with an advanced MCTS [4]. In the final chapter we analyze a way to introduce in our agents the desire to explore novel parts of the environment [3].

Contents

1 How to play Atari games via state-action values

In this section we'll see how to train an agent that can achieve the same performance as a professional human across 49 games of the Atari 2600 console. The agent learns to play directly by mapping raw observations combined with the current game score to the value of each action. In practice, we would like this agent to approximate the optimal action-value function:

$$Q^{\star}(s,a) = \max_{\pi} E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi]$$

which is the maximum cumulative discounted reward achieved by an agent following the policy π. We'll use a Deep CNN to estimate the action-value function which being a nonlinear function is known to create instability during training. The combination of using a Deep CNN to represent an action-value function along with strategies to reduce the instability of training is called a Deep Q-network or DQN in short, introduced in the paper [2].

1.1 Training an action-value function

The fundamental idea behind many RL algorithms is to estimate the action-value function by using the Bellman equation, an important property that holds for the optimal $Q^{\star}$ function. Formally it states the following:

$$Q^{\star}(s,a) = E_{s'}[r + \gamma \max_{a'} Q^{\star}(s',a')|s,a]$$

Intuitively it states that if you have access to the optimal value for each state and action, then the cumulative reward you can expect when being in state s and taking action a is the immediate reward r plus the discounted value of the best action taken in the next state s'.

The idea is then to define an iterative process where you update the action-value function via the Bellman equation as follows: $Q_{i+1}(s,a) = E_{s'}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$. This process is called Value Iteration and it converges to $Q^{\star}$ as $i \to \infty$.

Our Q function is parametrized by weights θ adjusted to minimize the MSE in the Bellman equation. The target will then be $y = r + \gamma \max_{a'} Q(s',a',\theta)$, note that in contrast with Supervised Learning here we have that the target itself depends on the current Q function and also it will change as we update the weights θ. At each stage of the optimization we keep an old copy of the parameters called θ_i^- to use as targets when optimizing the parameters θ_i. The weights are updated using SGD to minimize the loss function $L(\theta_i)$, defined as follows:

$$L(\theta_i) = E_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s',a',\theta_i^-) - Q(s,a;\theta_i))^2]$$

Deriving with respect to θ_i we obtain:

$$\nabla_{\theta_i} L(\theta_i) = E_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a, \theta_i)]$$

We remark that this algorithm is model-free since it does not learn the transition or reward dynamics and also is off-policy given that while learning the parameters of the current policy θ_i it follows the behaviour of an older policy.

1.2 Stabilizing the DQN training algorithm

By naively applying the previously introduced algorithm, the DQN is likely to diverge from a local minima and also it will not learn useful policies. This instability is due to:

1. The correlation in the sequence of observations
2. That the data distribution used for SL is biased towards the initial estimates of the values which might be wrong. This could create problems in the exploration since it can leads to re-visiting always the same states without considering other alternatives.
3. The correlation between the target value y and the action-values Q

To solve this problems they introduce a mechanism called Experience Replay which stores samples of experience (s, a, r, s'). During training a random minibatch of past experiences are sampled which helps mitigating the correlation in the sequence of observations. Also, this helps the agent to not overfit to the current observations which also mitigates the second problem of the exploration being biased towards the current Q values. The exploration problem and also the correlation between the target and current action-value are eased by computing the target values with an old copy of the DQN parametrized by weights θ_i^- which is only updated once every C epochs. Lastly, the error term is also clipped to be between -1 and 1.

1.3 Architecture

The observations in the Atari games are composed by the current image displayed by the emulator along with the current score. Since for each game the same hyperparameters are used they clipped all positive rewards to a maximum value of 1 and all negative rewards to a minimum of -1. This has the effect of limiting the scale of the error derivative making it easier to use the same learning rate. The original resolution of the image is 210x160 with a 128 colour palette. Before feeding it to the DQN the images are down-scaled to 84x84 and also the previous 4 images are stacked together to provide the network the ability of detecting changes. This is a necessity for example in the game of Pong where by only looking at 1 frame you cannot distinguish if the ball is moving towards or away from your

side. The final input is then 84x84x4 and is fed into a CNN with 3 convolutional layers followed by two fully-connected layers. Note that in theory the Q function should receive in input both the current state and action to provide the value. In practice however only the image is used as input and we'll have one output for each of the action to ease the computational time. Another technical detail to improve the performance is that the agent only perceives the observations once every 4 frames, and the action selected is then repeated for the next 4 observations before recomputing another action. For each game the agent experienced 50 million frames which is roughly equivalent to 38 days of game experience in total. The ϵ parameter that controls the degree of exploration is linearly reduced from 1 to 0.1 in the first 1000000 frames.

1.3.1 Evaluation of the DQN algorithm

To access the advantages of using a non-linear approximator for the Q function along with the Experience Replay and the separate older Q function for the target values computation we compare the performance of the agent on 5 different games showing how the average score achieved per episode changes as we control for the Experience Replay (with/without replay), separate DQN for target (with/without target) and the non-linearity of Q (linear/non-linear). As we can see by table 1 the techniques introduced in the paper achieved the best results when combined forming the DQN agent. We can also see by comparing the Linear scores with the DQN without replay and without target that the non-linear Q is by nature unstable and so, if not managed correctly, it can hampers the performance even when against a simple linear policy.

Game	Linear	Without replay Without target	Without replay with target	With replay without target	DQN (With replay & with target)
Breakout	3	3.2	10.2	240.7	316.8
Enduro	62	29.1	141.9	831.4	1006.3
River Raid	2346.9	1453	2867.7	4102.8	7446.6
Seaquest	656.9	275.8	1003	822.6	2894.4
Space Invaders	301.3	302	373.2	826.3	1088.9

Table 1: Comparison of the average score per episode with different methods

1.4 Limitations of DQN

A general problem of value methods and in particular model-free ones is that they're not sample efficient and requires a lot of time to train. Also, this gets worse as the reward gets sparser since the agent relies initially on a random exploration which might not converge at anything useful at all if the environment is particularly complex. It also has

troubles to learn long-term series of actions, an example would be the inability to solve the Montezuma game.

2 Policy: Learning in the real world

In the domain of robots we would like to train agents capable of interacting with real world scenarios solving many tasks. Without even considering the complexity of the task, the first problem one encounters is how to represent real world data in an informative way. Many approaches in learning a policy that controls a robot tried to manually engineer a set of features to be used as input to the policy. While this is certainly appealing given the interpretability of the representation along with the reduction in the dimensionality of the observation, we can't be sure that this features are helping the policy to solve the task at hand and also we're limiting the agent to improve the representation adapting it to the problem. The paper [1] tries to answer the question if it's possible to learn better policies by representing together the vision and control modules allowing the agent to jointly learn them.

Our policy will then be represented as a Deep CNN that maps raw observations/images directly to actions. Using a NN comes with a number of problems:

- Training requires lot of data which we don't have, especially given the cost of running a real robot;
- Secondly, the task is partially observed since from a single image we cannot figure out the full state, such as the positions of the objects which must be extracted from the image.

In the paper they solved this problems by introducing a novel training technique called Guided Policy Search which converts policy search into a Supervised Learning problem and the training data is constructed by an efficient trajectory optimization procedure. This can be formalized as an instance of BADMM, an optimization algorithm which converges to a local optima. The state of the system is only fully observable at training time, while at test time the policy will only act based on images of the task. The idea is then to split the training of the full policy into two components: in the first one we'll train a simpler policy $p(u_t|x_t)$ which emits a probability distribution over the actions u_t conditioned on the full state x_t, while in the second part we'll optimize the policy $\pi_\theta(u_t|o_t)$ represented by the Deep CNN by making it consistent/equal to $p(u_t|x_t)$.

The goal of the task is to minimize the expected loss of an episode when following the trajectory emitted by π_θ. This can written as $E_{\pi_\theta(\tau)}[\sum_{t=1}^{T} l(x_t, u_t)]$. The probability of a trajectory $\pi_\theta(\tau)$ is computed as follows:

$$\pi_\theta(\tau) = p\left(\mathbf{x}_1\right) \prod_{t=1}^{T} \pi_\theta\left(\mathbf{u}_t \mid \mathbf{x}_t\right) p\left(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{u}_t\right)$$

Notice that π_θ here is conditioned on the state x_t, to retrieve this distribution we just have to marginalize over the observations as follows:

$$\pi_\theta\left(\mathbf{u}_t \mid \mathbf{x}_t\right) = \int \pi_\theta\left(\mathbf{u}_t \mid \mathbf{o}_t\right) p\left(\mathbf{o}_t \mid \mathbf{x}_t\right) d\mathbf{o}_t$$

2.1 Guided Policy Search with BADMM

Our goal is to minimize the expected loss when following the guiding distribution $p(u_t|x_t)$ subject to the constraint that $p(u_t|x_t) = \pi_\theta(u_t|x_t)$. This constraint is on a continuous space making the optimization untractable, we'll transform it into making the two distributions equals in expectation. First we multiply both side by $p(x_t)$ so that we'll be able to simplify the Lagrangian later on by merging the expectation of the loss with the expectation of the Lagrangian multiplier. The final constraint is then $E_{p(u_t|x_t)p(x_t)}[u_t] = E_{\pi_\theta(u_t|x_t)p(x_t)}[u_t]$, giving rise to the problem:

$$\min_{p,\pi_\theta} E_\pi[l(\tau)]$$

subject to

$$E_{p(u_t|x_t)p(x_t)}[u_t] = E_{\pi_\theta(u_t|x_t)p(x_t)}[u_t]$$

We solve this problem using the BADMM algorithm which is a two step optimization procedures where first we minimize the function with an initial guess for the Lagrangian multiplier λ and then we modify λ by its sub-gradient. The optimization procedure can be written as:

$$\begin{aligned}
\theta &\leftarrow \arg\min_\theta \sum_{t=1}^{T} E_{p(\mathbf{x}_t)\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)}\left[\mathbf{u}_t^{\mathrm{T}} \lambda_{\mu t}\right] + \nu_t \phi_t^\theta(\theta, p) \\
p &\leftarrow \arg\min_p \sum_{t=1}^{T} E_{p(\mathbf{x}_t,\mathbf{u}_t)}\left[\ell\left(\mathbf{x}_t, \mathbf{u}_t\right) - \mathbf{u}_t^{\mathrm{T}} \lambda_{\mu t}\right] + \nu_t \phi_t^p(p, \theta) \\
\lambda_{\mu t} &\leftarrow \lambda_{\mu t} + \alpha \nu_t \left(E_{\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)p(\mathbf{x}_t)}\left[\mathbf{u}_t\right] - E_{p(\mathbf{u}_t|\mathbf{x}_t)p(\mathbf{x}_t)}\left[\mathbf{u}_t\right]\right)
\end{aligned}$$

The two Lagrangians will be denoted as $\mathcal{L}_\theta(\theta, p)$ and $\mathcal{L}_p(\theta, p)$. This algorithm is particularly useful in our case since we do not know how to solve the first original optimization problem where we had a coupling between π_θ and p but the (B)ADMM formulation is a lot easier to handle. Indeed, our optimization problem is now transformed into 2 subproblems, the first one that minimizes $\mathcal{L}_p(\theta, p)$ is a trajectory optimization under

unknown dynamics problem while the second one is solved by matching the policy π_θ to p.

The $\phi(\cdot)$ functions in the two Lagrangians are an augmentation of the ADMM method called BADMM that limits the difference between the distributions, formally:

$$\phi_t^p(p,\theta) = E_{p(\mathbf{x}_t)}\left[D_{\mathrm{KL}}\left(p\left(\mathbf{u}_t \mid \mathbf{x}_t\right) \| \pi_\theta\left(\mathbf{u}_t \mid \mathbf{x}_t\right)\right)\right]$$
$$\phi_t^\theta(\theta,p) = E_{p(\mathbf{x}_t)}\left[D_{\mathrm{KL}}\left(\pi_\theta\left(\mathbf{u}_t \mid \mathbf{x}_t\right)\right) \| p\left(\mathbf{u}_t \mid \mathbf{x}_t\right)\right]$$

2.1.1 Trajectory Optimization under Unknown dynamics

The guiding policy p is a mixture of Gaussians one for each initial state x_1. Here we show the trajectory optimization for a single Gaussian $p_i(\tau)$ which will be applied to all the Gaussians in the mixture.

We have that $p(u_t|x_t)$ and $p(x_{t+1}|x_t,u_t)$ are time-varying linear Gaussians, given by:

$$p(u_t|x_t) = \mathcal{N}(K_t x_t + k_t, C_t)$$

$$p(x_{t+1}|x_t,u_t) = \mathcal{N}(f_{xt}x_t + f_{ut}u_t + f_{ct}, F_t)$$

This type of linear controller is easily learned and does not require a lot of samples. In contrast, this is only a good approximation when the dynamics themself can be represented as locally linear.

In order to optimize p we first have to fit the dynamics distribution $p(x_{t+1}|x_t,u_t)$ to the trajectories sampled from the previous guiding distribution denoted as $\hat{p}$. Our dataset will be a list of $\{x_t,u_t,x_{t+1}\}$ and so the parameters of our dynamics can be fitted with a simple linear regression procedure. Once we have the dynamics we can optimize the Lagrangian $\mathcal{L}_p(\theta,p)$ using the LQR algorithm to learn the new controller $p(u_t|x_t)$. Note that an additional constrain is added to limit the KL-divergence between the new and the old controller, this is needed since our dynamics model is only valid in a neighbourhood of the old controller and if we move too far away from those trajectories the policy learning would not be accurate.

Another problem is that in $\mathcal{L}_p(\theta,p)$ we have the KL-divergence between the policy $\pi_\theta(u_t|x_t)$ and $p(u_t|x_t)$, however π_θ is in theory only defined conditioned on the observation o_t. In order to compute this we use the same approach taken to linearize the dynamics. Practically when we collect data to construct $p(x_{t+1}|x_t,u_t)$, in addition to storing $\{x_t,u_t,x_{t+1}\}$ we'll also remember $E_{\pi_\theta(u_t|o_t)}[u_t]$. This will allow to learn an approximate $\pi_\theta(u_t|x_t)$.

2.1.2 Supervised Policy Optimization

To solve the other subproblem that match the policy π_θ to p we have to minimize the KL divergence between the two distributions as well as the expected value of $\lambda_{\mu t}^T u_t$. Since

our policy π_θ is Gaussian we can construct an objective function that we can minimize over θ using simple SGD.

2.2 Architecture of the Deep CNN for policy

The policy π_θ is a Deep CNN that predicts the mean and variance of a normal distribution from which we will sample our actions. The model receives in input raw observation and feed them into 3 convolutional layers. The last of them contains 32 kernels with dimension 109x109 that are passed through a special operator called Spatial softmax. Basically we first take the output of the ReLU that follows the conv layer, so we'll have in each channel c a value a_{cij} for each pixel i, j. Each of the channels are fed into a softmax computing $s_{cij} = \exp(a_{cij}) / \sum_{i',j'} \exp(a_{ci'j'})$. The idea is that now each of the channel contains a probability distribution over the location of a feature in the image, we now want to convert this representation into an exact coordinate (f_{cx}, f_{cy}). To compute this coordinates we take a weighted average of the probability distribution with some learned weights. Formally, we have $f_{cx} = \sum_{ij} s_{cij} x_{ij}$ and $f_{cy} = \sum_{ij} s_{cij} y_{ij}$. This two coordinates are concatenated with the robot current angles of the joints and the pose of the end-effector as well as their velocities and fed into two dense layers followed by a linear layer that has 7 units for the joint torques of the robot.

2.3 Pretraining techniques

To reduce the training time consequently reducing the number of interactions performed by the real robot we perform 2 different pretraining techniques. To speed up the training of the Deep CNN we initialize the convolutional layers by training another CNN with the same structure on a different task. This task is a pose regression problem where the robot moves around the target object recording both images and positions and then adjust it's weights to predict the position given the image. However since only a small amount of data can be captured also the first conv layer is initialized from a model trained on the ImageNet Classification task.

While to pretrain the guiding trajectory distribution p we start with 15 iterations of guided policy search without optimizing the Deep CNN. The idea is to not waste time trying to match π and p when at the start the trajectories provided by p are still more or less random.

2.4 Experiments

To evaluate the performance of the GPS algorithm introduced in this paper we'll analyze two of the experiments of the paper:

1. A 2D/3D control task is solved with first only the guiding distribution p comparing it to other policy methods when the state is visible. Then with policy methods that trains neural networks on the observations;

2. An evaluation of the special operator Spatial softmax.

2.4.1 Peg insertion experiment

In this experiment we'll compare the performance of the introduced method on a 2D and 3D insertion task where the agent has to learn a policy to move an arm such that the end peg inserts into a slot. In Figure 1 we can see the performance of different policy methods when using the full state x_t, in this first experiment only the guiding distribution p is used. Since the peg is 0.5 units long, distances above this threshold corresponds to policy that cannot solve the task. We can already see how even when having knowledge of the full state of the system most of the policy methods cannot solve this task due to complexity of the dynamics involved.

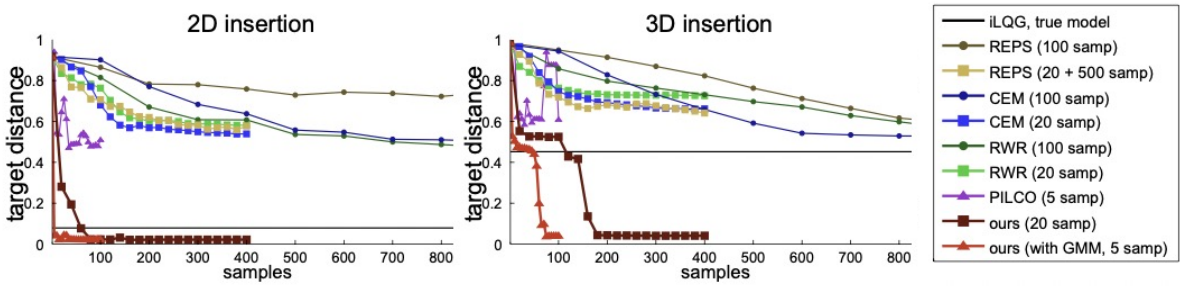


Figure 1: Distance from the target comparing different policy methods with the full state

In Figure 2 the same experiment is performed this time by training neural network policies. This time the policy is able to generalize and solve the task by only being fed the image.

2.4.2 Spatial softmax performance

To test whether the Spatial softmax improve the performance of our policy when compared to classical CNN architectures we compare the test error results on the Pose estimation regression task we described before. As we can see in Figure 3 the Spatial softmax achieves the lowest test error proving that learning feature point coordinates is useful when the task consists of recognizing the exact position of an object in a scene.

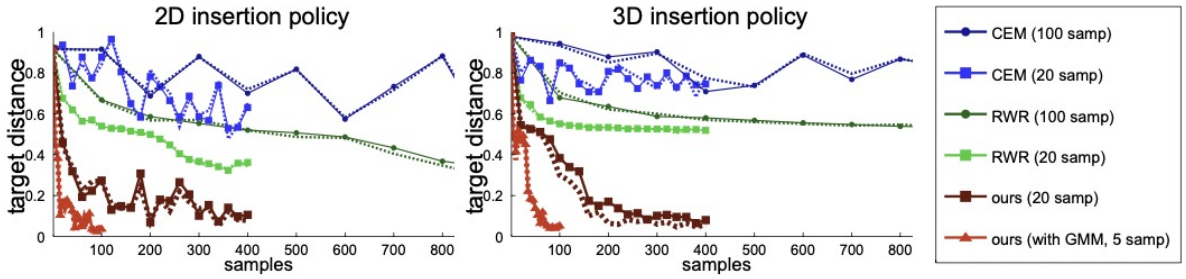


Figure 2: Distance from the target comparing different policy methods to train neural networks policies

network architecture	test error (cm)
softmax + feature points (**ours**)	**1.30 ± 0.73**
softmax + fully connected layer	2.59 ± 1.19
fully connected layer	4.75 ± 2.29
max-pooling + fully connected	3.71 ± 1.73

Figure 3: Test error and sd for the Pose regression task

2.5 Limitations

One of the limitations of this approach shared with all the other methods in this report is the need to manually write a reward function custom to the task at end. Also, while the simple time-linear Gaussian controller used as the guiding distribution is able to train policies surely able to complete complex tasks it's not clear how it would scale up to larger and more complex environments.

3 Actor-Critic: How to play Go

In all perfect information games there is an optimal value function $v^\star(s)$ which determines the outcome of the game assuming perfect play by both players. The idea behind solving this type of games such as Chess or Go is to recursively compute the value function in all the possible states in order to select the best action. The problem is that this search tree contains $O(b^d)$ possible states, where b is the number of legal moves in each state and d is the game length. In Go we have $b \approx 250$ legal moves and $d \approx 150$ moves per match making exhaustive search unfeasible. The goal of the paper which mastered the game of Go [4] is to reduce the search tree intelligently by only considering plausible/good moves predicted by a policy when expanding the search tree and also cutting-off the tree without playing the game until the end by estimating the outcome with a value function. The policy and value function will then be combined into a MCTS in order to select the

best move at each board state. The model can be divided into 3 stages:

1. Learning policies that mimic human play style
2. Improving the previous policies by self-play
3. Learn to predict the outcomes by learning the value function

We'll analyze each of this stages separately and then see how we can combine them to search for good moves in Section 3.4.

3.1 Supervised Learning of policy networks

To provide a good starting point for the next RL stage we begin by training a policy $p_\sigma(a|s)$ to predict moves made by experts. The policy is represented as a Deep CNN followed by a softmax layer which outputs a probability distributions over all legal moves a. The input s is an augmented feature representation of the original 19x19 board resulting in a 19x19x48 input.

To train $p_\sigma(a|s)$ we randomly sample state-action pairs (s, a) and use Stochastic Gradient Ascent to maximize the likelihood of predicting the human move a in state s, formally:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial\sigma}$$

The policy is trained on 30 million moves and achieves a 57% accuracy on a holdout test. Since larger networks require more time at inference time a second policy $p_\pi(a|s)$ is trained with the same objective but consisting only of a linear softmax of small pattern features. This achieved a test accuracy of 24% with the advantage of requiring only around 0.07% of the time taken by the larger policy $p_\sigma(a|s)$ to perform prediction.

3.2 Improving by self-play

We now want to improve the performance of $p_\sigma(a|s)$ by policy gradient RL. As a first step we initialize a new policy p_ρ identical to $p_\sigma(a|s)$. We then play games between the current p_ρ and a random previous p_ρ, this requires saving the network every k iterations but stabilizes the training by reducing overfitting to the current opponent. The reward function $r(s)$ is zero for all non-terminal states and the outcome $z_t = \pm 1$ depends on the perspective: 1 for the winner and -1 for the loser. The weights ρ are then updated by Stochastic Gradient Ascent by increasing or decreasing the probability of the actions taken during a game based on the final outcome, formally:

$$\Delta\rho \approx \frac{\partial \log p_\rho(a_t|s_t)}{\partial\rho} z_t$$

After training the RL policy was tested against the policy that mimics expert humans winning more than 80% of the matches.

3.3 Predict the outcome

In theory we would like to train a value function which converges to the optimal one $v^\star(s)$, however this is not possible since it would require already having access to the optimal one. So, we train our value function $v_\theta(s)$ to predict the outcome of the game where both players uses our strongest policy p_ρ. The model architecture is similar to the one used by the policies however it only predicts a single value. The weights θ are optimized via SGD on the MSE between the predicted value $v_\theta(s)$ and the outcome z. The training data is generated by storing the outcome and only one random state s from a game played by the policy p_ρ against itself. Even if this procedure requires discarding a lot of moves the authors found it necessary to prevent overfitting. This is due to the fact that successive positions are strongly correlated since they differ from only one stone.

3.4 MCTS for move selection

In AlphaGo the move is selected by constructing a Search Tree and at each time step selecting the most visited action during the MCTS.

The Search Tree stores board states as nodes and creates an edge for each possible legal action. For a node containing state s we'll define the edge (s, a) as the connection between s and s' where s' is the result of executing action a in state s. For each edge we store an action-value $Q(s, a)$ along with a value $N(s, a)$ that counts the number of times the search passed through that edge along with the prior probability $P(s, a)$.

Starting from the root representing the current state of the game, we visit the tree by simulation. A simulation consists of descending the tree by playing a game where at each time step t the action a_t is selected as follows:

$$a_t = \underset{a}{\text{argmax}}[Q(s_t, a) + u(s_t, a)]$$

where $u(s, a)$ is a bonus which balance the exploration vs exploitation during the search, formally we have $u(s, a) \propto \frac{P(s,a)}{1+N(s,a)}$. The idea of the bonus is that at the start of the search the prior probability of the action weights more meaning that we tend to exploit actions that we think are good. But, as $N(s, a)$ gets higher at some point we'll switch into exploring other actions having less prior probability of being played by the policy.

During the simulation, at a time L we'll reach a leaf node which might be expanded and if so we'll ask the policy p_σ for the prior probabilities of each action. Then, the value of the leaf node s_L is evaluated in 2 different ways: once by the value network $v_\theta(s_L)$

and another by the outcome z_t of a game played by the fast rollout policy p_π. The 2 estimations are then combined by a weighted interpolation: $V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$.

Once we have terminated n simulations we update the edge information as:

$$N(s,a) = \sum_{i=1}^{n} \mathbb{1}(s,a,i)$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbb{1}(s,a,i) V(S_L^i)$$

Basically each edge stores the mean evaluation of all the simulations passing though that edge and also the number of times it got visited.

Once the simulations are finished we pick the next move based on the most visited edge from the root. The Search Tree is reused only focusing on the subtree connected to the chosen action while the other subtrees can be discarded to free memory. A remark is that when storing the prior probabilities for the moves, the policy p_σ performed better than the stronger RL policy p_ρ. This is probably because humans are more diverse when it comes to selecting good moves while the RL policy only optimizes for the strongest one. Nonetheless, for the value network the one derived from the RL policy performed better in the search.

On a more technical note the prior probabilities stored in $P(s,a)$ are initially set to the results from another faster policy p_τ and are later replaced as soon as the GPU is able to pool the results from p_σ. Also, when computing p_σ we use a temperature parameter β that controls the uniformity of the action distribution. To allow for the GPU to catch up with the amount of states it has to evaluate with p_σ another parameter n_{thr} is dynamically adjusted such that the number of evaluation we ask to the GPU doesn't surpass the number of evaluations it can provide.

3.5 Results

To evaluate the performance of AlphaGo the authors constructed a tournament where different Go programs played against each other. The final elo ratings can be seen in the last column of Figure 4 showing that AlphaGo is the best one. Another experiment was conducted to analyze the impact of the various components that make AlphaGo, in Figure 5 we can see how different choices for the policy network or whether to only evaluate the state with the rollout policy or with the value network impact the final elo rating.

3.6 Limitations of AlphaGo

A certain limitation of this work is the requirement of having a complete and perfect model of the environment limiting the scenarios where this algorithm can be applied.

Short name	Computer Player	Version	Time settings	CPUs	GPUs	KGS Rank	Elo
α^d_{rvp}	Distributed AlphaGo	See Methods	5 seconds	1202	176	–	3140
α_{rvp}	AlphaGo	See Methods	5 seconds	48	8	–	2890
CS	CrazyStone	2015	5 seconds	32	–	6d	1929
ZN	Zen	5	5 seconds	8	–	6d	1888
PC	Pachi	10.99	400,000 sims	16	–	2d	1298
FG	Fuego	svn1989	100,000 sims	16	–	–	1148
GG	GnuGo	3.8	level 10	1	–	5k	431
CS_4	CrazyStone	4 handicap stones	5 seconds	32	–	–	2526
ZN_4	Zen	4 handicap stones	5 seconds	8	–	–	2413
PC_4	Pachi	4 handicap stones	400,000 sims	16	–	–	1756

Figure 4: Elo rating for different programs

Short name	Policy network	Value network	Rollouts	Mixing constant	Policy GPUs	Value GPUs	Elo rating
α_{rvp}	p_σ	v_θ	p_π	$\lambda = 0.5$	2	6	2890
α_{vp}	p_σ	v_θ	–	$\lambda = 0$	2	6	2177
α_{rp}	p_σ	–	p_π	$\lambda = 1$	8	0	2416
α_{rv}	$[p_\tau]$	v_θ	p_π	$\lambda = 0.5$	0	8	2077
α_v	$[p_\tau]$	v_θ	–	$\lambda = 0$	0	8	1655
α_r	$[p_\tau]$	–	p_π	$\lambda = 1$	0	0	1457
α_p	p_σ	–	–	–	0	0	1517

Figure 5: Elo rating for different versions of AlphaGo

Also, it require a lot of data on the problem and the construction of features to enhance the board state before feeding it to the various networks. This last two problems were tackled by a later paper by the same authors that introduced AlphaGo Zero [5], an agent that does not require any data and also only uses the white and black stones positions as input features.

4 Exploration with Intrinsic Motivation

In scenarios where rewards are sparse our agent is likely to perform very poorly if it does not explore the environment in a smart way.

To allow the agent to learn useful policies in this environment one solution is to modify the reward function by adding a term which favours exploration. This term will be called the Intrinsic reward/motivation.

The Intrinsic Reward is proportional to how hard it is for the agent to predict the result of its action in the environment. A naive way to do this would be to train a Deep NN capable of prediction the next state/image $\hat{x}_{t+1}$ given x_t and a_t and then our intrinsic reward could be $\|\hat{x}_{t+1} - x_{t+1}\|_2$. While this would serve as intuition for

the method presented in the paper [3], it's easy to see how this falls short when the environment changes in a stochastic way without our influence allowing our model to receive intrinsic reward when not necessary. A practical example of this would be an agent exploring an house and receiving rewards for staring at the TV since the different frames would change x_t.

The goal is then to avoid taking into account hidden dynamics independent of our actions and only focus on the changes that were caused by us or that somehow influences us. To accomplish this we transform the images into a Feature Space that only captures relevant information. The Feature Space is constructed by training an Inverse Dynamics Model that predicts the action a_t given the state x_t that caused the action and the resulting state x_{t+1}. The intuition is that since the model only has to learn to predict the action it will discard irrelevant parts of the image and only focus on those parts that were changed by the action.

We then use this Feature Space to train another model that represent the Forward Dynamics in the Feature Space. This model will learn to predict the next feature state representation $\phi(x_{t+1})$ given the current one $\phi(x_t)$ and the action taken a_t, resulting in $\phi(x_{t+1}) = F(\phi(x_t), a_t)$.

The intrinsic reward r_t^i will then be equal to the error between the feature state representation of the next state and the one predicted by the Forward model, formally: $r_t^i = \phi(x_{t+1}) - F(\phi(x_t), a_t)$.

In case our environment provides an extrinsic reward r_t^e we can combine it with the intrinsic one by summing them. The total reward will be $r_t = r_t^i + r_t^e$.

4.1 Architecture

Our agent is composed of a policy $\pi(s_t; \theta_P)$ which is represented by a neural network with parameters θ_P optimized to maximize the expected reward:

$$\max_{\theta_P}[E_{\pi(s_t;\theta_P)}[\sum_t r_t]]$$

The policy can be learned with various method but in the paper the A3C policy learning was used.

The presented architecture consists of two neural networks. The first one has 2 submodules: one that encodes the observation s_t into the Feature Space emitting $\phi(s_t)$, while the second one takes $\phi(s_t)$ and $\phi(s_{t+1})$ to predict the action a_t that caused it.

Training this NN amounts to learning the function g called the Inverse Dynamics:

$$\hat{a} = g(s_t, s_{t+1}; \theta_I)$$

We optimize the parameters θ_I as follows:

$$\min_{\theta_I}[L_I(\hat{a}_t, a_t)]$$

In the case that the actions are discrete the output of g is a soft-max distribution over the possible actions. We can then interpret minimizing the function L_I as Maximum Likelihood estimation of θ_I under a multinomial distribution.

The second neural network learns a function f called the Forward Dynamics, indicated by:

$$\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t; \theta_F)$$

The parameters θ_F are optimized with the objective:

$$\min_{\theta_F}[\frac{1}{2}\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|_2^2]$$

The Intrinsic Reward is then:

$$r_t^i = \frac{\mu}{2}\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|_2^2$$

The neural networks θ_F and θ_I combined are called the Intrinsic Curiosity Module (ICM).

We can combine the losses for θ_P, θ_F and θ_I in a single equation:

$$\min_{\theta_P,\theta_F,\theta_I}[-\lambda E_{\phi(s_t;\theta_p)}[\sum_t r_t] + (1-\beta)L_I + \beta L_F]$$

where the hyperparameters $\lambda > 0$ and $0 \leq \beta \leq 1$ serves as tradeoffs between the importance of the various losses.

We remark again that the function g is only trained to predict the action by learning the Inverse Dynamics, as such it will ignore irrelevant parts of the image. This will result in the agent not receiving any intrinsic reward for reaching states that are unpredictable. While it will be given an high reward when we're not able to predict well the feature space embedding which corresponds to non-explored parts of the environment.

4.1.1 ICM Architecture

The Intrinsic Curiosity Module (ICM) consists of 2 neural networks for the forward and inverse dynamics. The inverse model builds the Feature Space representation of the input image $\phi(s_t)$ using four convolutional layers, the final dimensionality is 288. Then the inverse model will predict the action by concatenating $\phi(s_t)$ and $\phi(s_{t+1})$ and passing them into two dense layers where the last layer has the same number of units as the number of actions.

The forward model concatenates $\phi(s_t)$ with a_t and has a first hidden layers of 256 units and then a final one with the same dimension as $\phi(s_{t+1})$ (288).

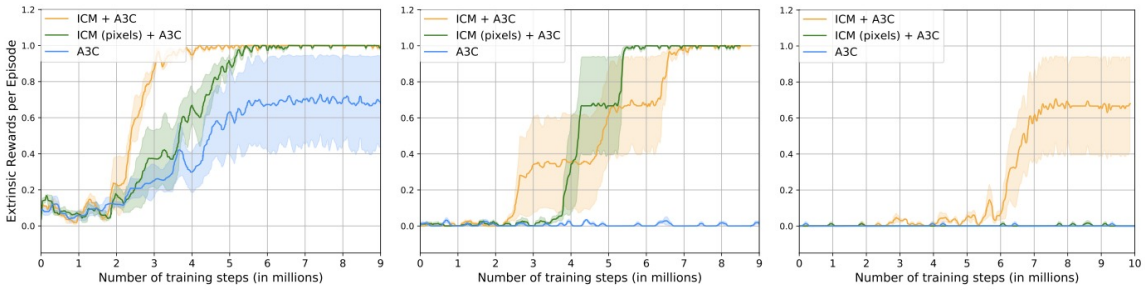
The policy is represented as a Deep CNN taking as input a state s_t represented by concatenating the previous three frames along with the current one. Each frame is a grey-scale 42x42 image and as for DQN there is action repeat.

4.2 Experiments and Results

To evaluate the performance of the introduced architecture 'ICM + A3C', the authors compared the performance against vanilla A3C with ϵ-greedy exploration and a variant of ICM called 'ICM-pixels + A3C'. This last model does not compute the inverse dynamics needed to learn the Feature Space $\phi(\cdot)$ and has the intrinsic reward equal to the error in predicting the next observation in pixel space. In order to predict the full observation deconvulational layers were added to the forward model in order to reconstruct the original image. Since this variant is forced to operate on the images it should achieve worse results when there are irrelevant and non-predictable components of the image.

4.2.1 Robustness to sparsity

As a first experiment we evaluate the ICM in a 3D navigation task "VizDoom" where the agent has to find a target goal placed around 350 steps away from the start position when following the optimal policy. The agent receives a reward of 1 if it reaches the goal and 0 otherwise. The episode also ends if the agent does not reach the goal in a maximum of 2100 time steps. To verify whether the ICM is beneficial we construct 3 experiments where the rewards gets more and more sparse by spawning the agent farther away from its goal. In Figure 6 we can see that the performance of the baseline "A3C" decreases as the reward gets sparser and also that "curious" A3C agents are superior in all the settings. A possible hypothesis as to why the ICM-pixels behaves worse than ICM is that the agent is spawn at each episode in a different one of the 17 rooms with different textures. And as the number of textures increases it gets harder to predict in pixel space thus degrading the quality of the intrinsic reward.



(a) "dense reward" setting (b) "sparse reward" setting (c) "very sparse reward" setting

Figure 6: Average reward in the navigation task in 3 different sparsity reward

4.2.2 Robustness to stocastich dynamics

To test whether the ICM is affected by uncontrollable dynamics the agent observation is augmented with a fixed region of random noise which spans for 40% of the original

image. The same experiment as before, in the sparse configuration (b), is executed and the results are shown in Figure 7. In the case of ICM the agent is still able to learn the task on the contrary of ICM+pixels proving the ICM robustness to noise.

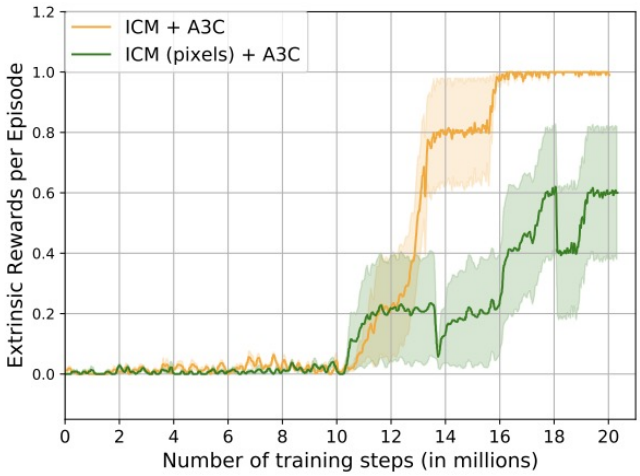


Figure 7: Average reward in the navigation task where random noise was added to the input image

4.2.3 Comparison with other Exploration techniques

As for the experiments before we compare the performance of ICM against the baseline A3C and also another policy method: TRPO. We also augment the TRPO method with the VIME exploration technique to empirically verify whether the ICM is better. On average the TRPO receives a 0.26 reward which increases to 0.46 when used with VIME. A3C is still not able to learn the task at all with an average score of around 0, while the ICM + A3C agent is able to fully solve the environment with an average reward of 1.

4.2.4 Exploration only training

To see the effect of the ICM intrinsic reward the authors re-run the previous experiment this time disabling the extrinsic reward. The agent is now driven only with the intrinsic reward and so we should see its exploration behaviour. We can study in Figure 8 a map of an environment where the agent starts at the position and orientation represented as a red arrow while the visited rooms are colored in yellow. While, in Figure 9, we can compare the same behaviour this time when the agent conducts random exploration. We can conclude that the exploration clearly is smarter when using ICM.

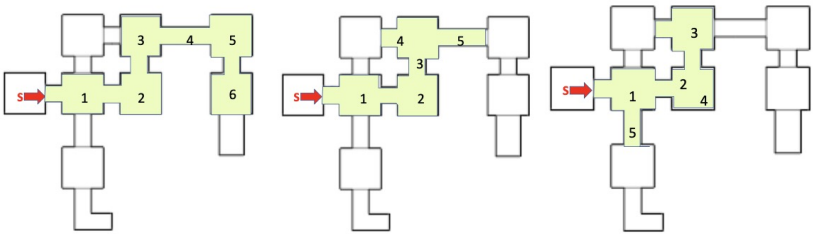


Figure 8: Exploration behaviour of the ICM, visited rooms colored in yellow

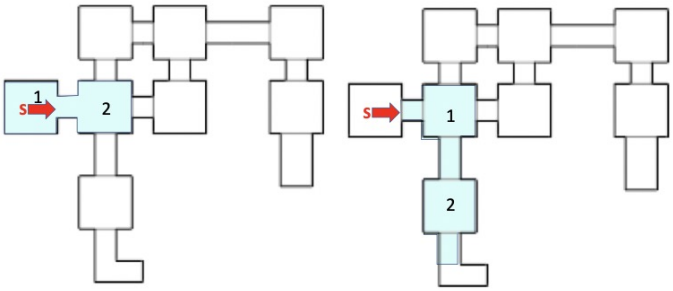


Figure 9: Random exploration behaviour, visited rooms colored in blue

References

[1] Sergey Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 1334–1373. ISSN: 1532-4435.

[2] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[3] Deepak Pathak et al. "Curiosity-Driven Exploration by Self-Supervised Prediction". In: ICML'17 (2017), pp. 2778–2787.

[4] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[5] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.