Learning Control Policies from High-Dimensional Visual Inputs

CS231n Final Project

Irwan Bello
Stanford University
ibello@stanford.edu

Yegor Tkachenko
Stanford University
yegor@stanford.edu

Abstract

We investigate how reinforcement learning can be combined with convolutional networks to learn to control an agent from high-dimensional visual inputs in the context of simple video games. We apply deep Q-learning to train a network as a Q-value function approximator on the space of state-action combinations of a game.

First, we replicate the noted approach [5] and train a network to play the simple Atari game Breakout. Second, we apply common visualization techniques to a trained network and provide analysis of how it works and extracts features, concluding that the network is overfitting to the particular task of playing Breakout. Third, we discuss the potential directions for future research.

1. Introduction

For a long time reinforcement learning algorithms have been mostly successful in solving problems with state spaces of low dimensionality.

When applied to tasks that involve high-dimensional inputs (e.g. pixel data), the prevalent approach has been to rely on handcrafted features that condense the relevant information into a low-dimensional feature representation. The quality of such a system depends on the quality of the feature representation (whether the features allow for a good reconstruction of the input) and its utility for the original learning problem (whether the features are actually useful for learning a policy). [8]

Success has been reported with using neural networks as a dimensionality reduction technique and then applying RL in the reduced feature space [4]. Especially good results were obtained when the neural network encoder was fine-tuned via a loss function corresponding to a particular task at hand. This allowed the system to learn which low level features are actually important for the task and then carry out dimensionality reduction while preserving that information. As an example, dimensionality reduction guided by reinforcement learning would be able to detect that the value of a particular pixel in the visual input is of great importance when making a choice on what action to take and would preserve its value through the dimensional collapse, embedding it in the high-level features, whereas dimensionality reduction without fine-tuning would likely convolute and lose this information.

However the most radical breakthrough of the recent years came from using a neural network as a direct function approximator, linking an arbitrarily complex state space to a Q-value func-

tion [5, 6]. According to the reported results, the system was able to learn how to play some simple Atari games using the described approach - i.e. by taking as inputs the pixel data, information about the action taken and achieved reward and ultimately estimating the expected value of each action for a given point of the state space.

2. Background

2.1. Reinforcement Learning

Reinforcement learning explores a problem, where an agent tries to maximize a cumulative reward in an a priori unknown environment by relying on interactions with the environment. The environment can be modelled as a Markov Decision Process (MDP), i.e. a set of states S, a set of actions A, a transition function T: $S \times A \times S \rightarrow [0, 1]$, which defines a probability distribution over possible next states given the current state and an action, and a reward function $S \times A \times S \rightarrow \mathbb{R}$. The goal of the agent is to find the policy $\pi : S \rightarrow A$ that maximizes the discounted cumulative reward $R_T = \sum_{t=T}^{\infty} \gamma^t r_t$ where γ is a discounted factor between 0 and 1, r_t denotes the reward obtained at time-step t and T is the current time-step.

Model-free RL algorithms, which learn an optimal policy without constructing an estimate of the MDP, rely on estimating the action-value function $Q^*(s, a)$, which is the maximim expected cumulative reward achievable when being in state s and performing action a. Given the action-value function we then get the optimal policy with:

$$\pi(s) = argmax_a Q^*(s, a)$$

The optimal action value function Q^* satisfies the *Bellman optimality equation*, which is based on the intuition that the optimal strategy when selecting an action is to maximise the expected value of $r + \gamma Q^*(s', a')$:

$$Q^*(s, a) = E_{s' \sim T(s,a,.)}[r + \gamma \max_{a'} Q^*(s', a') | s', a']$$

$$= \int_{s'} T(s, a, s')(R(s, a, s') + \gamma \max_{a'} Q^*(s', a')))$$

The process of iteratively updating Q using the Bellman equation is called value-iteration and yields a sequence of Q_i which converges towards Q^* as $i \rightarrow \infty$ [3]. In our case, value iteration doesn't scale because the iterations need to be performed on each state-action pair, without any generalization.

2.2. Deep Q-Learning and Experience Replay

Instead, deep Q-learning uses a neural network as a Q-function approximator (called deep Q-network or DQN), which is trained to minimize the $L2$ norm between the state-action value predicted by the network and the observed state-action value experienced by the agent.

Note that reinforcement learning was shown to be unstable or even to diverge when approximating the state-action value function with non-linear functions (such as a neural networks) [7].

This instability has several causes which are rooted in the difference in assumptions made by supervised learning and reinforcement learning. Supervised learning assumes that data samples are independently drawn from a fixed underlying distribution, while RL deals with sequences of correlated states sampled from a distribution that changes as the agent adopts new policies over time. The problem of retaining previously learned representation while acquiring new ones is known as catastrophic forgetting and has been shown to be very acute for neural networks [2].

To alleviate such issues, deep Q-learning uses a mechanism called experience replay, similar to how learning occurs in the hippocamp region of the brain [6]: the DQN is trained on episodes sampled uniformly from a replay memory, thus smoothing the data distribution and removing correlations in the sequence of observations.

Use of a separate network for generating the targets in the Q-learning update coupled with only periodical updates of the action-value targets is another trick that can be helful in alleviating the

issue, as most recently proposed in [6]. More precisely, every x number of updates the network Q is cloned to obtain a target network Q', which is then used for generating $\max_{a'} Q'_i(s', a')$ targets for the updates to Q.

More formally, the DQN is trained with the following loss:

$$L(\theta_i) = E[(r+\gamma \max_{a'} Q^*(s', a', \theta_{-i}) - Q(s, a, \theta_i))^2]$$

where the expectation is taken over (s, a, s', r) - experiences sampled uniformly from the replay memory. θ_i in the above equation are the parameters of the network at iteration i and θ_{-i} are the parameters used to compute the target at iteration i.

The training is performed with epsilon-greedy exploration (the agent chooses the best action with probability epsilon and chooses a random action otherwise) and no regularization is used. In particular, dropout is usually advised against in such regression settings.

3. Implementation

3.1. Technical setup

The game we play is Breakout. Our main code builds on the freely distributed implementation from Nathan Sprague `https://github.com/spragunr/deep_q_rl` based on Theano + PyLearn2. The code utilizes the RL-Glue framework to interact with the Atari emulator, and concurrently trains a conv-net using experience replaty tuples.

In training the net we use the Arcade Learning Environment game simulator for Atari, from which we generate experience tuples (starting image, action, reward, resulting image) [1]. At each time step an action generated by the conv net is passed to the emulator, which itself yields a new reward and resulting image. The idea of periodic updates from [6] is not implemented in this code.

While we utilize the code engine provided by Nathan, we needed to make a significant number of adjustments to various elements of the package to make it run on our machine - from editing the code that allows interaction between Theano, RL-Glue and Atari Emulator to dealing with various numerical problems within Theano code, forcing us to adjust portions of it.

Table 1: Nathan Sprague's network architecture

Layer	Size of activation volume
Input	84 * 84 * 4
Conv8-16	20 * 20 * 16
Conv4-32	9 * 9 * 32
FC-256	256
FC-4	4

We also had trouble unpickling the parameters of the network trained on Theano and thus took advandtage of the very recently released code from Google DeepMind `https://sites.google.com/a/deepmind.com/dqn/`. We train a separate network and apply visualization techniques.

3.2. Preprocessing and model architecture

Note that a single frame cannot capture the entire state of the game: in general 2 and 3 frames are required to respectively capture speed and acceleration. In our case, the input to the network consists 4 frames preprocessed in luminance scale (see [6]). Action space consisted of 4 actions: right, left and 2 nil actions.

The network construction involved no padding and square filters. The output screen of the game is cropped to 84x84 matrix (bottom part of the screen).

E-greedy exploration is turned on during training and turned off during testing.

The difference between the 2 code implementations have different structures. These are outlined in Tables 1 and 2. Table 3 shows (some) of the hyperparameters used in both implementations (see [6] for a complete list of hyperparameters).

Table 2: Google DeepMind's network architecture

Layer	Size of activation volume
Input	84 * 84 * 4
Conv8-32	20 * 20 * 32
Conv4-64	9 * 9 * 64
Conv3-64	7 * 7 * 64
FC-512	512
FC-4	4

Table 3: Training parameters by implementation

	Nathan S.	DeepMind
Learning rate	0.0002	0.00025
Replay memory	1 mln.	1 mln.
Discount factor	0.95	0.99
History lengths	4	4
Min epsilon	0.1	0.1
Start epsilon	1	1
Epsilon decay	10 mln.	10 mln.
Minibatch size	32	32

4. Experiments

4.1. Training results

Building on Nathan Sprague's implementation, we ran the code for Breakout game for 3 days on NVIDIA GeForce GT 650M GPU and obtained data for 50 training epochs (2.5 million iterations). The DQN figured out how to play at a child-level, understanding it is controlling the paddle and that it shouldn't let the ball reach the bottom of the screen. See figures 1,2 and 3 and submitted video.

We note that this implementation fails to learn the optimal strategy of tunneling, as covered in [6], supposedly because of fewer iterations and a difference in architecture. As stated earlier, we also do not implement the trick from DeepMind's paper to facilitate convergence in Nathan's code. (We do observe tunneling in several instances, but it is more of a random occurrence than a result of careful planning). Still, the results prove that the learning procedure works.

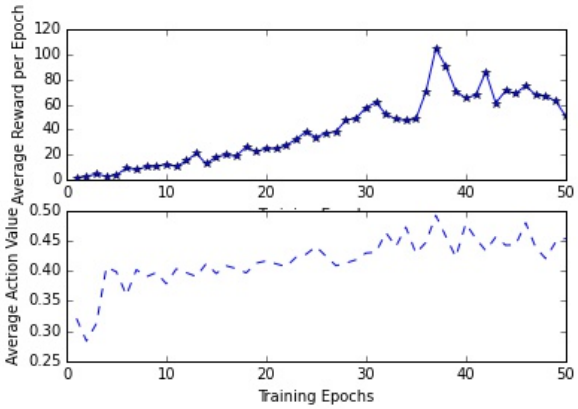

Figure 1: Training history - Nathan Sprague's implementation

However, we had difficulties extracting parameters from the files produced by Theano code. Concurrently, we asked several members of the Deep Q-Learning community to share some of their trained networks - so that we could have a larger sample to work on when examining how the network works. While we received a lot of feedback, we kept running into the same problems when unpickling the network.

Thus, we decided to repeat the training with Google's code, the output of which we could actually process.

We ran Google's Torch code for 3 days on AWS EC2 GPU-enabled instance for 15 million iterations (the code is unsurprisingly much better optimized and thus faster than Nathan's version). The trained DQN achieves comparable performance to that of Nathan's implementation.

4.2. Visualization

We now unpack our trained network (DeepMind implementation) and study its properties. A common visualization technique for convolutional networks [9] is to plot the weight of each filter in the first convolutional layers expecting to recognize the low level features that each filter is trained to extract.

Our setting is different given that the depth of

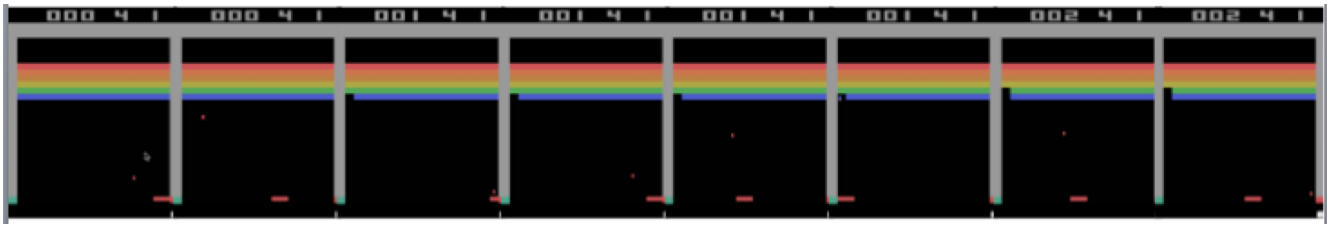

Figure 2: Epoch 1 (50,000 iterations) - Nathan Sprague's implementation

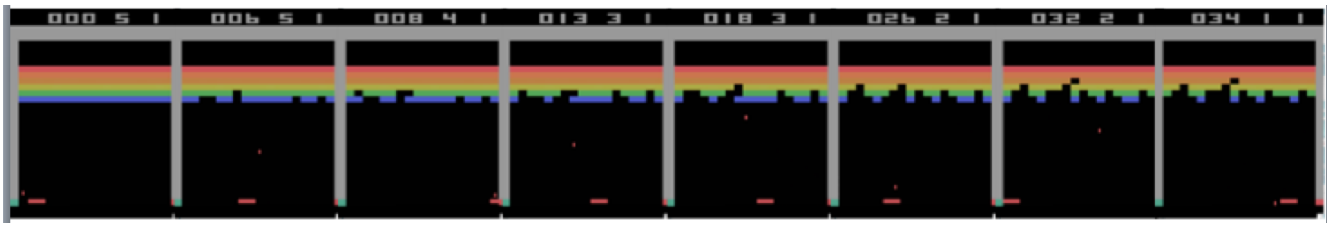

Figure 3: Epoch 37 (1,850,000 iterations) - Nathan Sprague's implementation

our input shape corresponds to the time dimension rather than RGB channels. We thus display each depth slice of our filters on a separate plot, the intuition being that the sequence of 4 images represents a template for the sequence of images (i.e. the current state).

No apparent patterns were found when observing the weights of any of the first convolutional layer's filters (see Figure 5 for a sample filter). This suggests that the DQN is strongly overfitting the task of playing Breakout (the training does not involve any regularization), thus not necessarily focusing on features usually met in computer vision (such as edges and texture for example).

Figure 6 shows the activation maps of the three convolutional layers for the sample state displayed in Figure 4. Note that some activations are blank, possibly summarizing that all bricks are present in our sample state. Simply looking at a few sample states and their corresponding activation maps did not provide us with a clear understanding of what each filter is picking up in the network.

We also looked at the states yielding the maximum action values (those were the states corresponding to new screens - the network has learned that a new screen leads to higher action values),

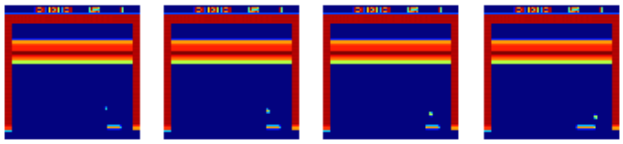

Figure 4: 4 frames of a sample state

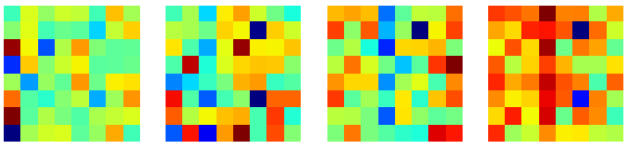

Figure 5: Weights of a sample filter from the first layer (4 images to the 4 frames in the sequence)

the minimum action values and the extreme differences in values between different values. No notable results were found in the 2 latter experiments.

5. Future Research

Although deep Q-learning has proven to be an incredibly powerful method for learning optimal action policies over arbitrarily complex state spaces, its potential remained far from being explored.

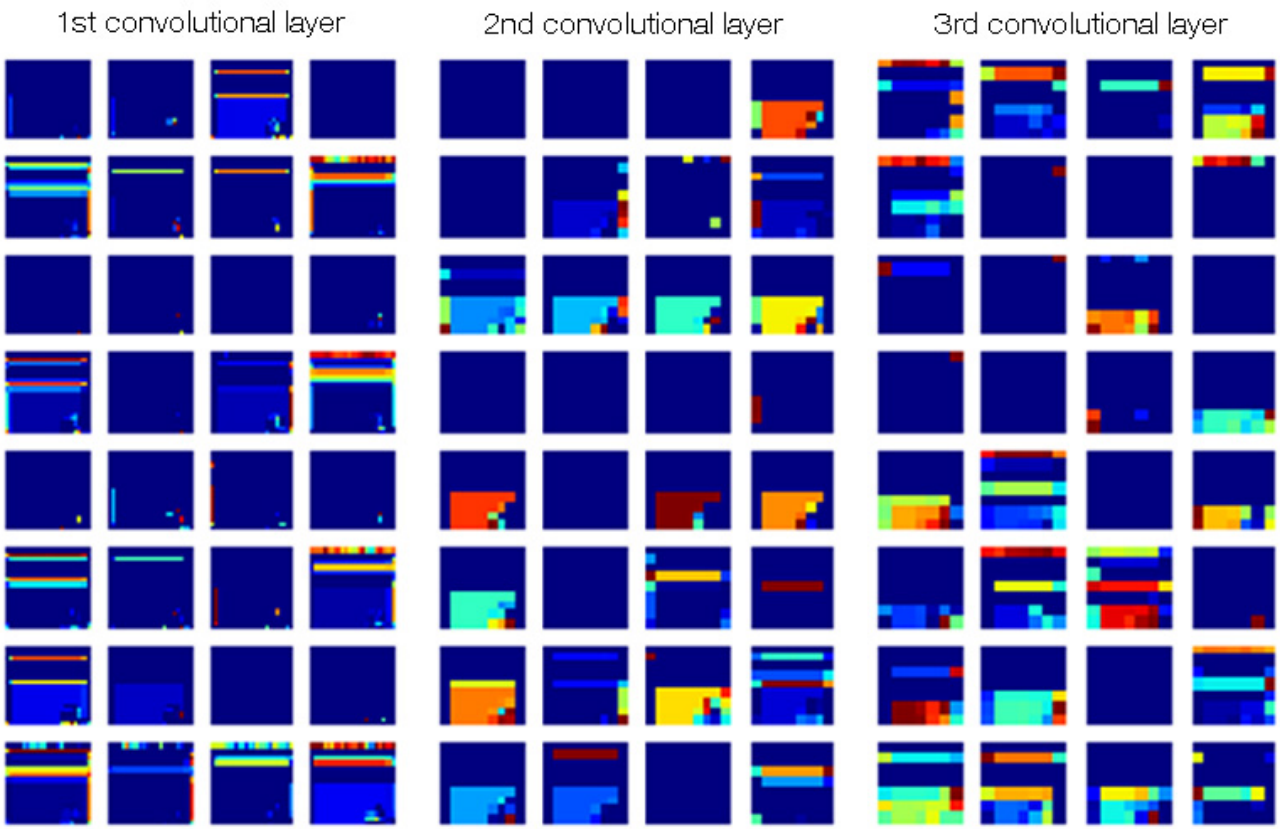

Figure 6: Activations maps of our sample state

Additional insights and possibly a validation of the overfitting hypothethis may be given by applying deconvnet [9] to get the top patches of each filter (note that it would require to adapt the method to the time dimension).

Fighting overfitting and coming with regularization techniques adapted to sequential problems while getting reliable convergence seems to be a challenging and promising future area of research (e.g: penalizing useless actions, applying batch normalization, training a network to play different games).

On a more general level, great opportunity lies in advancing our understanding of how neural nets could handle large action spaces. While existing implementations can handle limited discrete action spaces with relative ease, continuous action spaces of multiple variables or very large discrete action spaces are still not well supported.

Finally, transfer learning in deep Q-learning seems to hold great potential and may be one path to a truly general autonomous game playing machine.

6. Conclusion

We are able to replicate the recently reported success of deep Q-Learning for game control from high-dimensional visual inputs. The method allows us to achieve human-level performance on Atari game "Breakout", building on 2 different code implementations - by Nathan Sprague and Google DeepMind. We also provide analysis on the features a trained DQN pays attention to, arguing that the DQN is largely overfitting its current task. Finally we briefly discuss some potential directions for future research in the field.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[2] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgeting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[3] M. Kochenderfer. *Decision making under uncertainty: Theory and applications*. MIT press, 2015.

[4] S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. *Proceedings of the International Joint Conference on Neural Networks*, 2010.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv: ...*, pages 1–9, 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[7] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions on*, 42(5):674–690, 1997.

[8] M. Wiering and M. van Otterlo. *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. *arXiv preprint arXiv:1311.2901*, 2013.