DQN: Does it scales?

Harsh Satija

***Abstract*—Deep Reinforcement learning has recently emerged as a successful methodology for efficiently learning complex tasks and even succeeded in matching (and in some cases surpassing) human level performance in a variety of games [7] [17]. For our project we studied Deep Q-Networks (DQN) [7] which estimate the Q-values (or expected discounted returns for an action) for the current game state, where the game state is defined only by the last four frames the agent has encountered. However, one major downside of DQN is its huge training time and computational cost, which is primarily due to its dependency on how it maintains the game state (it preserves past frames as well as the size of the frame). In this project we attempt to study the various methods that are employed to address these issues. In particular we looked at adding the ability to maintain the game state using individual frames, and focusing the agent's attention on the most salient features of the input.**

I. Introduction

The computational cost of DQN is extraordinarily high, which is majorly due to the fact it's required to model the game as a Markov Decision Process (MDP) and maintain the past frames as part of the state description. As such, the computational complexity of DQN scales 'linearly' with:

- Size of the game image (size of frame)
- Number of past frames required to maintain the game state.

Our project is aimed at looking at ways to tackle the above two problems. Our main inspiration is drawn from Hausknecht and Stone [9], who provide an architecture that works under both the Markov assumption and for partially observable Markov Decision Processes (POMDPs), and that integrates information across frames. They call their architecture Deep Recurrent Q-Value Network (or DRQN), as it employs recurrent networks, particularly Long-Short Term Memory cells (or LSTMs) [5]) in order to learn temporal information. Additionally, work has been done on attention models for DQN [13] but the focus of that work was not to see the effect on computation time but rather in highlighting the regions of the game screen the agent focuses on when making decisions. Furthermore, by using a soft attention model similar to [13] we attempted to identify the most salient regions of the game screen and see how that affects the computation. For this project our focus was in implementing DQN and DRQN, to compare their performance, and see how they stack up against the attention model. In the following sections we'll describe the core components of of our agent, which are: the Convolutional Neural Network (CNNs), Deep Q-Networks (DQNs), Deep Recurrent Q-Networks (DRQN), and the attention module.

A. CNN

Convolutional Neural Networks (CNNs) [3], [4] constitute a special class of artificial neural network with an architecture that is designed to exploit the properties of overlapping regions in the visual field. Inspired by animal vision, CNNs are intended for tasks whose raw input data has a grid-like topology, like images (2-D topology). The convolution process, for which the architecture is named, involves generating an output tensor from the linear combination of the input data, f, and a parameter matrix, or *kernel*, g. Unlike in traditional neural networks, whose parameter matrices describe the interactions between every input and its corresponding output, the parameter matrix for a CNN is orders of magnitude smaller than the input data. This unique architecture affords us three distinct advantages over basic neural nets: (1) sparse interactions; (2) parameter sharing; and (3) equivariant representations. These features conspire to increase statistical efficiency and reduce memory and computation time. For the purposes of this project, we use convolutions to generate feature maps from local connections over the input layer.

The composition of a typical CNN, consisting of five main components (or layers), is as follows:

- INPUT holds the raw pixel values of the image, in this case an image of 48 x 48
- CONVOLUTION will compute the output of neurons that are connected to local regions in the input, each computing a dot product between the weights of the input and the region to which they are connected in the input volume. The convolution operator for a 2-D topology is defined as:

$$\begin{aligned} o[m,n] &= f[m,n] * g[m,n] \\ &= \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v] g[m-u, n-v] \end{aligned}$$

- ACTIVATION applies a non-linear activation function, such as hyperbolic tangent or sigmoid. Introducing non-linearity, so the output (or feature map) isn't merely a linear combination of the input data, is done for the purpose of constructing a universal function approximator.

$$h^k_{ij} = \tanh((W^k * x)_{ij} + b_k)$$

- The POOL layer will perform a down-sampling operation along the spatial dimensions (width, height), returning summary statistics of *collections* of outputs from the activation layer (rather than each individual output), thus resulting in a reduction of volume.
- And last, the FULLY-CONNECTED layer will compute the class scores via softmax. This is similar to a regular layer in a fully-connected neural network, where each neuron is connected to all the numbers in the previous volume. The softmax function is as follows:

$$s(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

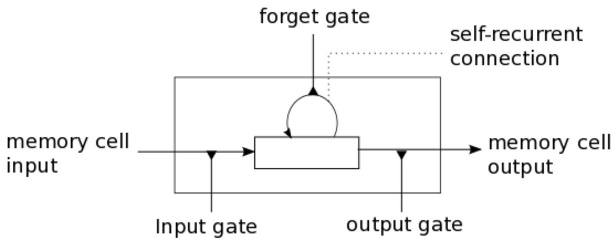

Fig. 1. Illustration of LSTM memory cell [6]

B. RNN and LSTMs

1) RNN: While Convolutional Networks are specialized for processing grid-like topological data, Recurrent Neural Networks (RNNs) are uniquely equipped to handle sequential data of the form $x^1, ..., x^T$. RNNs are particularly useful for retaining information about past entries in a data sequence. The feature that confers this advantage is called *parameter sharing*, which is similar to but distinct from the means of parameter sharing in CNNs. For RNNs, sharing doesn't emerge from the application of a static kernel matrix across the input data, but rather by making each output a function of previous outputs. RNNs enable us to exploit a signal's temporal structure, which is omitted by traditional feed-forward neural networks (which have unique parameters for each element of the input data). The RNN, hidden state update is given by:

$$h_t = tanh(W_{hidden-to-hidden} h_{t-1} + W_{input-to-hidden} x_t)$$

2) LSTM: A typical RNN, however, suffers from a vanishing gradient problem where the gradient signal gets so small that learning either becomes very slow or stops working altogether. This also affects the ability of the network to form long-term learning dependencies. Therefore, we use a long short-term memory (LSTM) network [5] to address these issues. The fundamental unit of an LSTM is called a *memory cell* [Figure 1] which is composed of four main elements: an input gate, a neuron with a self-recurrent connection (i.e. a connection to itself), a "forget" gate, and an output gate. The gates serve to modulate the interactions between the memory cell itself and its environment. [6]

To update the memory cells, we first compute the values for i_t, the input gate, and $\widetilde{C_t}$ the candidate value for the states of the memory cells at time t :

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\widetilde{C_t} = tanh(W_c x_t + U_c h_{t-1} + b_c)$$

Second, we compute the value for f_t, the activation of the memory cells' forget gates at time t :

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

Given the value of the input gate activation i_t, the forget gate activation f_t and the candidate state value $\widetilde{C_t}$, we can compute C_t the memory cells' new state at time t. And with the new state of the memory cells, we can compute the value of their output gates and, subsequently, their outputs. [6]

$$C_t = i_t * \widetilde{C_t} + f_t * C_{t-1}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$
$$h_t = o_t * tanh(C_t)$$

II. DQN

Deep Q-Networks, originally developed by David Silver and Google's Deep Mind project [], operate on the principles of the psychological school of *behaviorism*. The philosophy of behaviorism, championed by B.F. Skinner and Ivan Pavlov, first gained traction in the 1940s, and postulates that human behavior is ultimately driven by cognitive conditioning and reinforcement learning. Although this theory fails to adequately capture the nuances of human cognitive function, it provides an interesting conceptual framework from which to develop and train less complex minds: those of computers. Specifically, DQNs blend the notion of reinforcement learning with convolutional neural networks (described above). By interacting with its given environment, a DQN agent generates a series of mutually dependent observations, actions, and rewards. Much like the human caricature conjured by B.F. Skinner, the agent tries to develop a *policy*, $\pi = P(a|s)$, from which it selects actions, *a*, so as to maximize its future cumulative reward, *r*.

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

To address the tendency of reinforcement learning agents to diverge when paired with a convolutional neural network, Silver and his colleagues employed three tricks. The first is referred to as *experience replay*, which randomizes over the data to purge correlations in the observation sequence and smooth changes in the data distribution. An agent's *experiences* are stored as $e_t = (s_t, a_T, r_t, s_{t+1})$ at each time step in the dataset $D_t = (e_1, ...e_t)$. Secondly, the DQN iteratively updates the action values (Q) so as to shift them in the direction of the target values (which, by contrast, are only periodically updated), using a separate target network $\hat{Q}$. This serves to diminish correlations between the action values and those of the target. The target network essentially decouples the feedback resulting from the network, and therefore generates its own targets. Lastly, our DQN uses an adaptive learning rate method in RMSProp [8] (although other methods are available).

The loss function of the iterative update is as follows:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\ U(D)}[(r + (\gamma) max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i)^2]$$

Here, γ is the *discount factor*, θ_i are the parameters of the Q-network at iteration i, and θ_i^- are the network parameters for computing the target iteration.

A significant limitation of DQNs, however, is that they are constrained by the information provided by the last four states only. This is sufficient for a suite of narrowly defined tasks; however, many of the world's most compelling problems

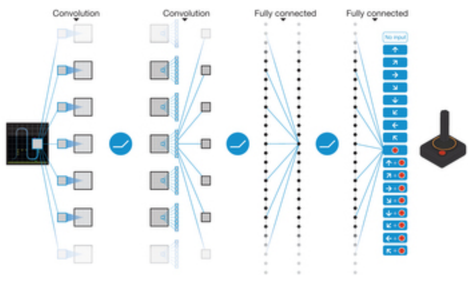

Fig. 2. DQN architecture [7]

require more robust memory. In the context of DQNs, problems that demand retention of information from many previous states become partially observable Markov decision processes (POMDPs) for which DQNs are insufficient.

III. DRQN

For problems in which the full state of the system is obscured or unobtainable, the Markov property, on which DQNs are predicated, no longer applies. To better capture the properties of these non-idealized environments, partially observable Markov decision processes (POMDPs) are used instead. The 4-tuple design of the DQN is replaced with a 6-tuple system (S, A, P, R, Ω, O) in which the agent receives an observation $o \in \Omega$ from the partially observed underlying state of the system, $o\ O(s)$. Nothing about the architecture of the traditional DQN admits of understanding the latent state of the system, which hinders the agent's ability to generate good Q-values. This is because the observed state of the system is not always reflective of the true state. To correct for this, Matthew Hausknecht and Peter Stone [9] equipped the DQN agent with recurrency, resulting in the DQRN architecture. DRQNs share the same basic foundation of DQNs, but substitute the first fully connected layer with an LSTM.

Following the convolutions ordinarily involved in DQN (labeled Conv1-Conv4 in Fig.3) the outputs are pushed through the LSTM layer, after which a final fully connected layer computes a Q-value for the system.

To fully exploit the recurrence of the DRQN, the backward passes through the system must have long sequences of game screens and target values. In order to achieve this, we first zero the initial LSTM hidden state, and then proceed with a *boostrapped random update*. This means that episodes are randomly selected from the replay memory described above, and use only timesteps of unfolded iterations. Unfolding (or unrolling) a recursive computation, like the one used in the LSTM layer, results in the aforementioned sharing of parameters across the deep network structure.

The process of unfolding merely consists in applying the definition of the recursive function $T-1$ times, where T is the number of time-steps involved. This dissolves the recurrence of the system, which can now be represented as an acyclic network. In dynamical systems analogous to our DRQN, the state of the system (in our case, the hidden state, $h^{(t)}$ is defined by the past state(s), $h^{(t-1)}$, an external signal, $x^{(t)}$, and the system parameters, θ, such that $\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}, \theta)$. Predictions are then computed using $\mathbf{h}^{(t)}$ to map an arbitrarily

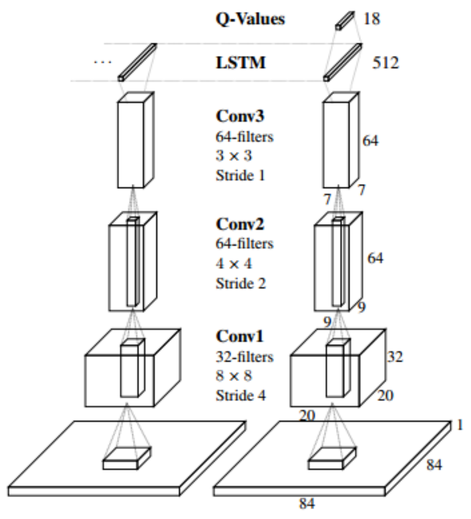

Fig. 3. DRQN architecture [9]

long sequences of the external input to a fixed-length vector in $\mathbf{h}^{(t)}$. Consequently, $\mathbf{h}^{(t)}$ is wielded by the agent as a necessarily lossy summary of the history of inputs.

Depending on the nature of the problem, and the training methodology, information about the past sequence can be preferentially preserved, as opposed to indiscriminately storing the entire corpus of sequence information. Thus, unfolding yields two primary benefits: (1) the learned model is defined in terms of the length of history states, meaning it has the same input size regardless of sequences length; and (2) the same transition function, f, can be coupled with the same parameters at every time step.

This culminates in the DRQNs improved ability to generalize to sequences that didn't appear in the training set, thus allowing it to succeed in situations that require POMDPs.

IV. Soft Attention Model

Attention based models take inspiration from how humans observe a scene, and how they dynamically filter only the salient features of a noisy image. The images registered in the eye have a non-uniform resolution throughout — with the maximum resolution at the center, and gradually decreasing fields of resolution near the periphery. Thus, when humans observe something, the features of the event that register most strongly are the ones their gaze is explicitly focused on. Rapidly scanning a scene in this way is referred to as a saccade. Attention models try to incorporate this behavior into the network's learning architecture. Inspired by recent advances in attention mechanisms for caption generation [10], machine translation [12] and object recognition [11], we try to model an attention mechanism similar to the Deep Attention Q-Network (DARQN) proposed by Sorokin et al (Fig. 4, [13]). Three primary networks are used in this model — a CNN, an attention network, and an RNN. The CNN takes the current frame and produces $m \times m$ feature vectors, each with dimension D. The output from the CNN is fed into the attention network, g which transforms these into a set of vectors $x_t = x_t^1, ..., x_t^L, x_t^i \in \mathbb{R}^D, L = m * m$ and combines it

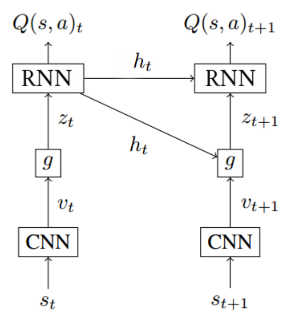

Fig. 4. DARQN's Blocks

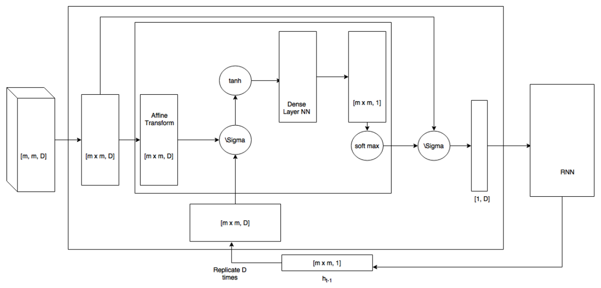

Fig. 5. Soft Attention Network Architecture

with the previous hidden state h_{t-1} of the RNN. The resulting output is a weight of relative importance for each of the $m \times m$ feature vectors. We then take the linear combination of feature vectors multiplied by their weights to produce a *context vector*, z_t,

$$z_t = \sum_i g(x_t^i, h_{t-1}) * x_t^i$$

Where, we define our attention module $g(x_t, h_{t-1})$ as:

$$g(x_t^i, h_{t-1}) = softmax(W_{\text{dense}}(\tanh(A_t)) + b_{\text{dense}})$$
$$A_t = \text{Affine}(x_t^i) + W.h_{t-1}$$

Here, W and W_{dense} are the weights of the network. $Affine(x) = Bx + d$ is an affine transformation with some weight matrix B and bias d. The module is shown in Fig. 5. Note that the entire module is differentiable and can be be trained through regular back-propagation through time.

This module can be inserted into the recurrent layer of the existing DRQN architecture, and can be trained in the same way as the regular DRQN.

V. Experiments

In Theano [1], we implemented each of the models described above, and used Lasagne [2] for the convolutional neural networks. We the subjected each model to the Arcade Learning Environment as the Atari emulator [14]. For each experiment we used the game Pong.

For DQN we chose the same training parameters as used by Deepmind in [7]. For DRQN, the convolutional part of the network was the same as DQN, except for the fact that instead of an input of 84×84 ×4 we used only one frame and therefore had 84×84×1). Additionally, instead of the dense part we used LSTM with 512 hidden units. For training the DRQN, episodes were selected randomly from the replay memory and updates began at random points in the episode and proceeded only for 4 unrolled timesteps. For, the attention model, we couldn't get the code running with the LSTM (due to faulty implementation) so we used RNN instead. In this case, too, the convolutional module was unchanged, but for the RNN we used 256 hidden units for every connection in the recurrent module. We have attached our implementation for all models in the supplemental files. We trained our models on NVIDIA Titan X with the CUDA 7.5 version.

VI. Results

Our DQN model was able to fully master the game of Pong (in around 90 epochs) where the average time per epoch was 2759 seconds. The total time it took to train for Pong was roughly 3 days (70 hours). In our experiments we found the DRQN model, which we trained for 50 epochs (roughly 80 hours), to be 2.3× slower than the DQN model, where the average time per epoch was 6132 seconds. The training curve for both DQN and DRQN is given in Figure 6.

Unfortunately, our attention model is unable to learn and improve the training reward even after training for 25 epochs (avg time per epoch 2680 seconds; total training time approximately 20 hours). We suspect the reason for this is due to the fact that we are using a vanilla RNN instead of an LSTM.

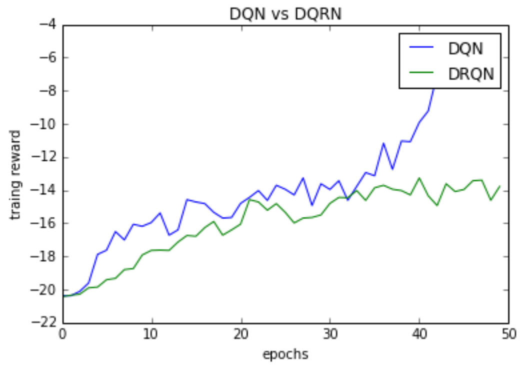

Fig. 6. DQN vs DRQN Learning Trend first 50 epochs

VII. Future Work

Our foremost priority is to get the attention model working, for which we need to add our attention model in the LSTM cell. We are also going to test our models on smaller games for which we don't require the CNNs.

Another interesting prospect would be to substitute our attention model with the Spatial Transformer Module, which is also a fully differentiable network module and has shown the ability to capture attention.

One very interesting direction of possible research is to use glimpses as used by [11] [15] which uses a glimpse network that drives the network's attention across important regions of the frame. A glimpse is a multi-resolution image which consist of a small, high resolution region of an image, with concentric regions of decreasing resolutions (typically we have 3 kinds of resolution patches: high, medium and low). The input to the RNN would now be a product of ‘glimpse’ instead of the entire frame. The agent would then have to learn on which location to focus next, and the optimal action-to-state decision, given the current frame. Training with glimpses instead of full frames greatly reduces the number of parameters that need to be trained for.

Here we have confined our experiments to ALE. During the course of this project, we were able to get an alternate PyGame Learning Environment(PLE)[19]. PyGame is an open source platform that facilitates game development. As such, many of the games are developed by amateurs and are hence prone to have many bugs. We could not test our current network on this playform, but in the future, we would like to test our agent’s ability to learn on such buggy games.

REFERENCES

[1] http://deeplearning.net/software/theano/

[2] http://lasagne.readthedocs.org/en/latest/

[3] http://deeplearning.net/tutorial/lenet.html

[4] http://cs231n.github.io/convolutional-networks/

[5] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[6] http://deeplearning.net/tutorial/lstm.html

[7] 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.

[8] Tieleman, T., and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012

[9] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable MDPs. CoRR, abs/1507.06527, 2015.

[10] Kelvin Xu, Jimmy Ba, Ryan Kiros, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. arXiv preprint arXiv:1502.03044, 2015.

[11] Volodymyr Mnih, Nicolas Heess, Alex Graves, and koray kavukcuoglu. Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, Advances in Neural Information Processing Systems 27, pages 2204–2212. Curran Associates, Inc., 2014.

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

[13] Ivan Sorokin, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, Anastasiia Ignateva. Deep Attention Recurrent Q-Network . arXiv preprint:1512.01693, 2015

[14] M. G. Bellemare , Y. Naddaf, J. Veness and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents , 2013

[15] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. In Proceedings of the International Conference on Learning Representations (ICLR), 2015.

[16] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3-4):229–256, 1992.

[17] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel and Demis Hassabis http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[18] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu. Spatial transformer networks. CoRR, arXiv: abs/1506.02025, 2015

[19] Tasfi, Norman. PyGame Learning Environment. https://github.com/ntasfi/PyGame-Learning-Environment