Architectural Design In Deep Q-Learning

Moncef Msekni, Robert Lundberg and Movitz Lenninger

Abstract

Deep Learning models have previously been successfully applied to the domain of Reinforcement Learning (RL). To use RL in real-world applications, the agent has to be able to handle high dimensional sensory data to find a suitable representation of the environment. Furthermore, the agent has to be able to generalize well to situations it has previously not encountered. In 2012 DeepMind presented their DQN, a Q-learning network that were able to learn several Atari 2600 games. In this paper, we will further investigate different architectural variations of DeepMind's DQN and their impacts on learning Breakout. We show that adding to much non-linearity combined with a lack of dimensionality reduction in the early layers leads to oscillating networks. Furthermore, simply adding one inception-module does not seem to improve the performance of DQN:s. Finally, expanding the network and compensating by spatially factorizing the first layer performs close to our implementation of DeepMind's DQN structure.

1 Intoduction

In the last few year AI:s have briskly moved to a more holistic approach, with Deep Learning as the current state of the art in many domains. One of the challenges in AI is to learn and generalize from experience and one such learning approach is Reinforcement Learning (RL). Work done by Google's DeepMind achieved above human performance in multiple video games in 2015. Their work spurred new interest in RL and has lead to multiple papers with proposed improvements of their algorithm. The main cause of the success of DeepMind's AI is their use of Deep Neural Networks (DNN). The architecture and hyperparameters of DNN's can be altered to change the modeling capabilities of the network. Not a lot of published work has been done in the context of changing the network architecture from DeepMind's original. DeepMind's network has been optimized to converge and perform well on multiple games. Changing the design often leads to oscillating networks but it can still be interesting to asses how different networks affect the convergence rate and overall performance. In this paper different architectural designs will be tested, the goal is to get a better understanding of how different networks architectures affect the convergence and performance of the AI.

2 Background

The fundamental building blocks of neural networks were developed decades ago. The Perceptron was proposed by Frank Rosenblatt in 1957. The learning algorithm Backpropagation was derived in the 60s[10] but first proposed as an efficient method in neural networks in the 80s [7]. Mainstream adoption of neural networks came decades later when Deep neural networks (DNN) revolutionized the field of computer vision. Alex Krizhevsky showed, in 2012, that image classification could be done more accurately with deep neural network than with conventional methods that required a substantial amount of feature engineering[15]. The network architecture, convolutional neural network (CNN), was first proposed by Yann Lecun[3] in 1998 where it was applied on handwritten character recognition. The availability of powerful GPU:s and new techniques such as dropout [14] and ReLu[4] lead to the success of Krizhevsky's network in 2012. Deep learning has since then been used successfully in many domains, most notably in image recognition, speech and speaker technology, Natural Language Processing and also in RL.

2.1 Convolutional Networks

CNN:s are feed-forward neural networks that are loosely based on how the visual cortex processes visual information[3]. The weights in each layer is kept in filters (kernels). These filters are often of lower dimension than the input and the filters are therefore swept across the input. This reduces memory and allows for weight sharing. The weight sharing and sweeping of multiple filters results in a network with units that are activated by increasingly complex patterns further down the network. This principle is in accordance with David H. Hubels research on cats[9], where the neurons closest to the input fire to different rotations and positions of bars. Neurons close to the output fire to complex patterns.

Each layer in a CNN has three dimensions: height, width and depth. The depth is a result of the number of filters (kernels) used on the previous layer. The width and height is dependent on the filters' size and padding, i.e. adding zeroed valued channels around the actual values. Figure 1 illustrates the weight sharing.

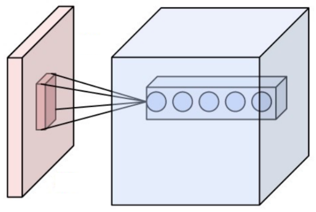


Figure 1: CNN

A full network is a set of layers with activation functions in between, which gives non-linearity and on-off[4] properties. Pooling operations can be added to give invariant features[13]. Regularization of CNN is of high importance due to the high complexity of the functions modeled by neural networks. Dropout has been the most common regularizer for CNN:s and works by randomly dropping units from the network during training[14]. Dropout can be seen as an ensemble of multiple networks and hence improves generalization. The evolution of CNN:s can be seen in Figure 2, 3 and 4. The networks in Figure 2 and 3 do not seem to differ very much and indeed the methods used are very much alike. The main differences between LeCun-5 and AlexNet are that AlexNet is trained on two GPU:s, uses ReLu instead of tanh as activation function and also utilities dropout. Google's inception-v4 is the currently the best classification CNN and reaches a score of 3.08% on

ImageNet.

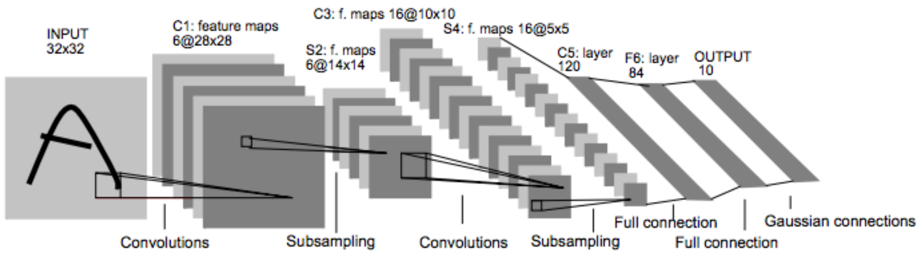


Figure 2: LeNet-5, used by Yan LeCun in 1998 to classify handwritten digits

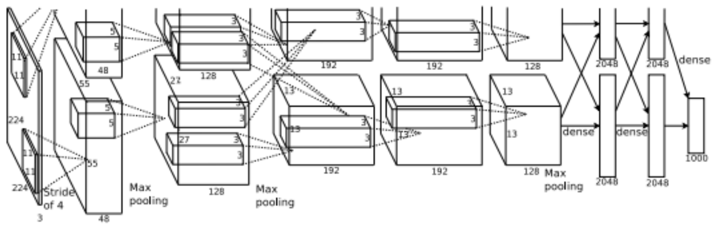
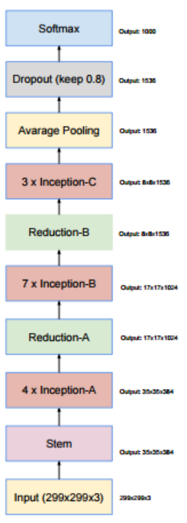
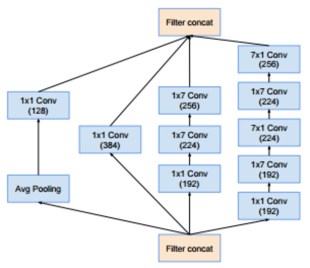


Figure 3: AlexNet which was used in 2012 to win the ImageNet competition



(a) The current state-of-the-art network in image classifiction, A 156-layer network with multiple modules.



(b) Example of a module used in inceptionNet-v4

Figure 4

2.2 Reinforcement Learning

Reinforcement learning lies somewhere in between supervised and unsupervised learning. Rather than providing the ground truth of each sample by labels, as in supervised learning, RL uses a reward signal to rate the performance of the network. The goal of the agent becomes to interact with the environment, by selecting actions, in such a way that maximizes the total reward. This means that the network has to, by itself, figure out which actions give high rewards. Therefore, when training the network there is no need of having labeled data at hand, but rather there has to be a way to evaluate and decide the reward of an performed action. This self-reliance and independence from supervised labels is the strength of reinforcement learning[11].

2.2.1 Markov process and the Bellman equation

The process of reinforcement learning can be represented as a Markov Decision Process, a generalization of Markov Chains [19]. An agent interacts with an environment E through selecting actions from a set of legal actions $A = \{a_1, ..., a_n\}$. At each time step, the environment, which is in a specific state, is observed and the agent has to select an action from A, which sometimes results in a reward r. The action taken transforms the environment into a new state and the the process repeats. This continues until the game (episode) ends. An important assumption is that the next state only depends on the current state and the current action, but not on any previous states or actions - i.e the process should satisfy the *Markov Property.*

The goal of the agent is to accumulate as high total reward as possible during an episode. Assuming that the task is a Markov Decision Process, then the total future reward for any possible set of actions can be calculated, theoretically, by a reward function. Thus, it is possible to plan for future rewards when selecting the next action. However, since most environments are not entirely deterministic, rewards in the near future should be prioritized before distant and more uncertain rewards. Therefore, future rewards are scaled such that distant rewards are counted less. This is called the discounted future reward and can be defined iteratively as:

$$R_t = r_t + \gamma R_{t+1}$$

where γ represents the scaling factor and is a value between 0 and 1. Setting γ to 1 implies that the environment is fully deterministic and predictable, contrastingly setting γ to 0 implies that the environment is hard to predict and that only the immediate reward should be taken into account.

In Q-learning, the *Optimal Value Function* $Q(s, a)$ is defined as the maximum discounted future reward:

$$Q(s_t, a_t) = maxR_{t+1}$$

where s and a is the state and the action taken. $Q(s, a)$ serves as a action-selection policy for an RL agent. In the case of games, Q(s,a) would represent the best possible score at the end of the game given you take action a in the state s. Basically, it represents the **Q**uality of an action in a specific state. The Q-function can be iteratively defined using the Bellman equation:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

where s' and a' is the next state and the next action[12].

2.2.2 Deep-Q-Network (DQN)

The task of Q-learning agent is to learn the Optimal Value Function, Q(s,a). The problem is that it in practice impossible to cover every possible combination of states and actions. This is where the deep learning comes in. DNN:s are good at extracting features from highly structured data, making it capable of representing the Q-function for important states and generalize to unseen states. The core idea is to use a deep neural network to approximate the Q-function, $\hat{Q}(s, a, \theta) \approx Q(s, a)$ where θ represents the weights of the network. Thus, the approximated Q-function, $\hat{Q}$, becomes a function of the state of the environment, the action and the state of the network.
Instead of providing both state and action as input to the network, a more efficient way is to only feed the state to the network. Then the network can predict the Q-values for all possible actions in that state. Hence, only one forward pass is needed to decide the best action.
Since the Q-values can assume any real values, approximating the Q-function becomes a regression task and can therefore be optimized with following squared error loss function[19]:

$$L = (r + \gamma \max_{a'} \hat{Q}(s', a', \theta) - \hat{Q}(s, a, \theta))^2$$

Here the term $r+\gamma \max_{a'} \hat{Q}(s', a', \theta)$ is called the *target* and the term $\hat{Q}(s, a, \theta)$ is called the *prediction*.
The algorithm to update the network (weights) then becomes:

1. Do a forward pass of the current state s and get all Q-values $Q(s, a)$.
2. Take the action $arg \max_a Q(s, a)$, get the reward r and do a forward pass of the new state s'. Calculate the max output from the network $\max_{a'} Q(s', a')$.
3. Apply a L2 regression loss over the target and the prediction.
4. Backpropagate the gradient to update the weights.

Training the network is difficult and there are a lot of tricks you have to use to get it to converge. The most important one is called experience replay[19] and is a temporal reinforcement learning version of randomizing the samples. Experience replay is used by storing transitions <s,a,r,s'> as experiences in a memory called the replay memory. The network is trained using random batches sampled from the memory instead of training on the experiences drawn continuously from the game. This way, the training data are, at least, partly decorrelated.

A general problem with reinforcement learning is how to decide when to be satisfied with the learned strategy and when to search for a potentially better one. This is called the exploration-exploitation dilemma. To find a balance between the two, Deep Mind's DQN uses a ϵ-greedy exploration[19] - with probability ϵ the agent chooses a random action and otherwise the action with the highest Q-value. ϵ is initially set to 1 but is decreased as training progresses. Another problem when using neural networks is that the algorithm uses a forward pass of the same network to compute both the target and the prediction, which can lead to divergence and oscillations [20]. A solution to this problem is to have separate target and training networks. To accomplish this, two networks with identical architectures are created, and the target network is copied every n iteration from the training network. Note that the training network is updated every iteration.

2.2.3 Reinforcement Learning in games

AI is often thought of as an agent taking actions in an environment. For basic and repetitive tasks robots have been used extensively both in academia and in the industry. A problem with robots is that they are physical objects that have to be built and can therefore be both expensive and cumbersome to work with. In reinforcement learning, where the AI learns from experience, the training would take a lot of time since you can not speed up the learning and any real environments fitted for the task would have to be created. Instead of actually creating the environment and the physical robots, one can use simulators that can be sped up and in which the AI can be tested in multiple different environments. Games are perfect for the task, there are thousands of games with different complexity to choose from. From simple board games to complex computer games that already simulate environments where AI:s can be tested. AI using reinforcement learning with CNN has successfully learned to play both boardgames[17] and video games[20]. The Atari Learning Environment (ALE) is often used to benchmark the general intelligence of AI:s, i.e. the AI is tested on multiple environments to assess if it can learn to navigate in different environments. Julian Togelius has written extensively about the benefits of using games to develop AI on his personal website[16].

2.3 Architecture Design

2.3.1 Inception Networks

In the article "Going deeper with convolutions" [18], a new architectural design was proposed. In short, their idea was to make parallel network structures, organized into modules, within the network. The same research team had used this design to create GoogLeNet, which set the new state of the art in classification and detection of images in the ImageNet competition in 2014. The authors hoped that the new architecture could be a way of increasing the network size efficiently. The naive way, they say, would be to simply add more layers (increasing depth) and adding more units to each layer (increasing the width). However, increasing the network size results in a larger number of parameter. Consequently, the network is more likely to overfit and requires a larger training set. Adding more layers also increases the risk of having weights close to or equal to zero. Since the computational resources are limited, this results in a less and less efficient representation. The core idea with Inception modules is to find a more efficient representation by trying to approximate a local sparse structure using already existing components (i.e convolution filters). This structure (one module) can then be repeated spatially to increase the size of the network.

The authors suggested that such a Inception module should consist of four parallel paths, each using 1x1, 3x3, 5x5 convolutions or max pooling, respectively. However, to be able to concatenate the four paths after the convolutions, 1x1 convolutions needs to be added to the 3x3, 5x5 and pooling paths as dimension reduction or expansion.

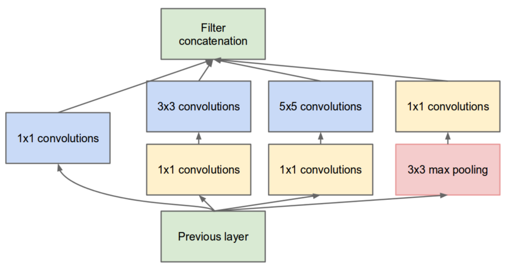


Figure 5: Inception module presented by Szegedy et al [18]

2.3.2 Design considerations for Inception networks/Factorizing convolutions

There are several general design principles to consider for convolution networks. Some of which are presented in the article ”Rethinikng the Inception Architecture for Computer Vision” [5]. One of the strength of the Inception network, the authors claim, is the extensive use of dimension reductions. To further increase this benefit of lowering the number of parameters, the authors suggest factorizing convolutions with large filter sizes. For example, a 5x5 convolution can be replaced by two 3x3 convolutions resulting in the same sized output with the same receptive field. The resulting weight sharing thus decreases the number of parameters. In the example where a 5x5 convolution is replaced as described above, the number of parameters will decrease by a factor of $2 * 3^2/5^2 = 0.64$. Hence, factorizing the convolution kernels can be a way to further reduce the number of parameters needed.

3 Methods

The project code is based on Devsisters corp.'s Tensorflow implementation [1] of Deep Mind's DQN [20]. Their implementation became our starting point when creating our own network structures. The code was provided on GitHub. The implementation is limited to use the Breakout game, a game where you are trying to break as many blocks as possible by bouncing a ball from a controllable platform onto the blocks, see Figure 6. The environment is provided by OpenAI's Gym toolkit. We trained our networks on a GTX

680 2GB and a GTX 970 4GB. Due to other hardware restrictions we were capt at 2GB of GPU-memory. Each network was trained for roughly 3-4 days.

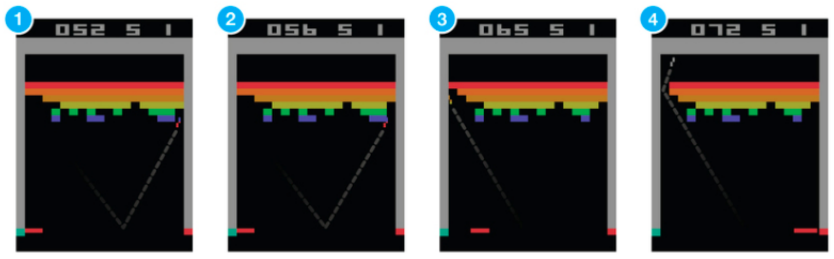


Figure 6: Illustraion of the game breakout

3.1 Architecture

First, we trained using the same architectural structure as in Mnih et al, see Table 1. This structure was then expanded using additional layer(s). The first attempt was to replace the 8x8 convolution kernel with three smaller kernels. The architecture is described in Table 2. A second structure was attempted, where yet again the 8x8 convolution kernel was replaced, but this time using a spatial factorizing technique, see 2.3.2. The 8x8 kernel was replaced by two spatially smaller kernels in such a way that they together gave the same dimension reduction as the vanilla DQN, see Table 3. In the last attempt, an Inception module was added to the original structure. The module was added between the second and the third layer in the original structure, see Table 4. We made one adjustment to the module compared to [18], namely that the pooling path was removed. The argument for removing the pooling unit is that pooling is thought to increase translational invariance. However, translational invariance might be beneficial for classification purposes but not for playing an arcade game (where the spatial location of the ball and paddle matters a great deal). As for Mnih et al., the inputs to all networks are four consecutive 84x84 pixel images of the screen.

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Table 1: Architecture of Deep Mind's DQN

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	84x84x4	4x4	2	32	ReLU	41x41x32
conv2	41x41x32	3x3	1	32	ReLU	39x39x32
conv3	39x39x32	3x3	1	32	ReLU	37x37x32
conv4	37x37x32	4x4	2	64	ReLU	17x17x64
conv5	18x18x64	3x3	1	64	ReLU	15x15x64
fc6	16x16x64			512	ReLU	512
fc7	512			18	Linear	18

Table 2: Architecture of DeeperNet

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	84x84x4	4x4	2	32	ReLU	41x41x32
conv2	84x84x4	3x3	2	32	ReLU	20x20x32
conv3	20x20x32	4x4	2	64	ReLU	9x9x64
conv4	9x9x64	3x3	1	64	ReLU	7x7x64
fc5	7x7x64			512	ReLU	512
fc6	512			18	Linear	18

Table 3: Architecture of FactorizeNet

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
inception3	9x9x64				ReLU	9x9x64
conv4	9x9x64	3x3	1	64	ReLU	7x7x64
fc5	7x7x64			512	ReLU	512
fc6	512			18	Linear	18

Table 4: Architecture of InceptionNet

3.2 Training methodology

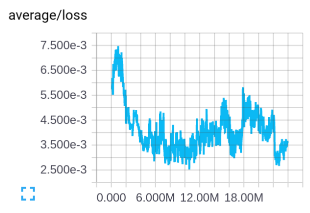
Training was performed with linearly decreasing exploration. The ϵ-greedy policy was decreased from 1 to 0.1 during 200 000 transitions, after which ϵ was kept fixed. Furthermore, a replay memory was utilized for training. The training was performed on mini-batches of saved transitions in the replay memory, as described in 2.2.2. Each batch consisted of 32 transitions and the memory consisted of the 200 000 latest transitions. The reward was fixed to +1 for all positive values and fixed to -1 for all negative rewards. Zero reward remained unchanged.
The loss function for iteration i is defined as the difference in the approximated Q-value at iteration i using the latest weights, θ_i, and the sum of the reward of iteration i and the approximated Q-value of iteration $i+1$ using older weights, θ_i^-.

$$L_i(\theta_i) = E_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i)\right)^2\right] \tag{1}$$

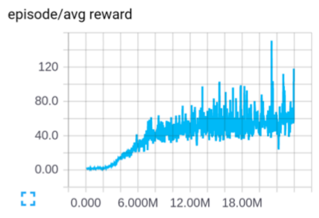
The error, $r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i)$, used in the loss function is clipped to be in range [-1,1]. This helps stabilizing the performance of the algorithm [20]. The older weights θ_i^- in the target network are updated once every 10'000 iterations to the current weights from the training network, $\theta_i^- = \theta_i$.

We trained our network using the RMSprop algorithm [6], a mini-batch version of stochastic gradient descent, a learning rate of 0.00025 was used. Decay was set to the default value, 0.9, in TensorFlow[2] (as value not stated in the paper by Mnih et al) and the momentum was set to 0.95. The initial weights were randomly sampled from a Gaussian distribution with mean 0 and standard deviation 0.02.

4 Result

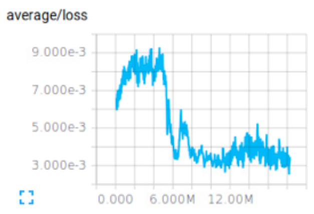


(a)

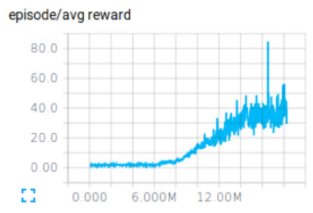


(b)

Figure 7: Results from vanilla DQN

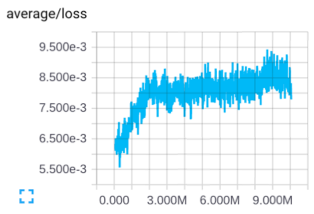


(a)

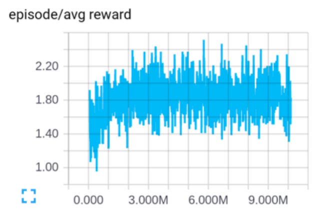


(b)

Figure 8: Results from InceptionNet

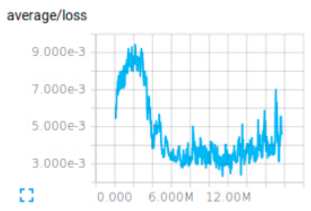


(a)

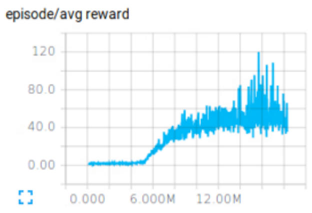


(b)

Figure 9: Results from the DeeperNet



(a)



(b)

Figure 10: Results from FactorizeNet

	Human[19]	DeepMind's DQN 2012 [19]	DeepMind's DQN 2015 [20]	Our DQN	InceptionNet	DeeperNet	FactorizeNet
BreakOut	31	168	401.2	116	54	1.8	103

Table 5: Comparison of avarge rewards from different network performances

5 Discussion and conclusion

Our agents did not preform nearly as well as the DQN agent presented by DeepMind, not even when we used the same network architecture as DeepMind. There could be several possible explanations for this. First of all, they specified the rough training time but not number of iterations for their agent to learn each game. Even though our training times were within the same range, the difference in hardware likely resulted in much fewer training iterations for our agents. Furthermore, due to time limitations and the scope of the project, the main bulk of the code was imported. The authors claimed to have converted the DeepMind's code into TensorFlow code. However, we found several errors in the code making the networks not exactly identical. Even though we found some, there might still exist other mistakes that we overlooked.

Due to memory limitations we had to decrease the size of the replay memory. The DQN keeps 10^6 samples of transitions in their memory, we only save $2 * 10^5$ (20 %) for every net other than the vanilla DQN. The exploration phase is reduced by the same amount. These reductions naturally affects the performance of the networks. The smaller memory size puts more emphasis on recent transitions when training. A faster decay of ϵ leads to less exploration of the state-space. These two combined could result in a convergence to undesired local minima and hence suboptimal behavior.

The average loss (over an epoch) is not strictly decreasing in RL, as opposed to most image classification tasks. The reason for this is twofold. The agent can start to observe states which it has not encountered before (via exploration). This sometimes leads to the discovery of new better strategies. Secondly, the loss function uses the approximated values for training and target Q-value. Furthermore, as we trained the network on mini batches, the loss function becomes less smooth since it sees different parts of the training data for each batch. Another problem we discovered is that the networks usually requires several million iterations before any effect of the training can be seen. This could be an indication of bad initialization of the weights. It could also be an effect of shorter exploration phase.

Our results with the vanilla DQN outperforms all the other architectures, see Figures 7, 8, 9 and 10. However, the comparison is not entirely fair since the vanilla DQN was trained with a larger replay memory and longer exploration. Using the full memory proved to be unfeasible in our other architectures due to hardware limitations. As already discussed a smaller memory and shorter exploration could be the reason why the vanilla DQN finds a better strategy. However, the difference in performance could also depend on the differences in representation. The network architecture in vanilla DQN could simply be better. DeeperNet does not converge at all. The loss function increases and the average rewards oscillates around 1.5. The poor performance could be due to the high degree of non-linearity, caused by the two extra ReLU layers, and lack of dimensionality reduction in the early layers. The InceptionNet does in fact converge to learning a strategy. The training is less efficient than for DQN, the average reward started to grow first after six million iterations. The final strategy obtained was not as good either with an average reward per game at roughly 54. InceptionNet has less non-linearity than the DeeperNet and has a faster dimensionality reduction, which could be why InceptionNet converges but not DeeperNet. The FactorizeNet both converged and performed better than InceptionNet. It is difficult to compare it to the vanilla DQN considering the difference in memory size and exploration time. However, given the same memory size and exploration time, FactorizeNet could potentially be on par with vanilla DQN, considering they got very similar results despite the training differences. Further investigation would be required to get conclusive results.

Interestingly, FactorizeNet is performing better than InceptionNet even though it contains less parameters. Larger networks usually give better results on high dimensional problems. A larger network that still gives worse results might indicate over-fitting or not enough training. Comparing InceptionNet's and FactorizeNet's average reward, one can observe that the InceptionNet requires longer time before it starts to improve. The learning curve is also less steep for InceptionNet. InceptionNet could also be more harmed by the limitations in memory and exploration because of its more complex structure. Deep Mind's DQN does not use any form of regularization through dropout, which could be necessary when larger networks are used.

In conclusion, we have shown that there are other network architectures that also are capable of learning a representation by RL. Although, DQN still outperforms the other networks. The InceptionNet showed performance on roughly the same level as humans and the FactorizeNet got results very close to the DQN, see Table 5. The key aspects of the success of the DQN could be due to its generous dimensionality reduction and relatively low complexity using quite few non-linearity ReLU layers. DQN is probably a result of rigorous testing and cross validation.

References

[1] Devsisters corp. github. `https://github.com/devsisters/DQN-tensorflow`. Accessed: June 2016.

[2] Tensorflow documentation. `https://www.tensorflow.org/versions/r0.8/api_docs/python/train.html`. Accessed: June 2016.

[3] Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haner. Gradientbased learning applied to document recognition. *IEEE*, 11 1998.

[4] Xavier Glorot, Antoine Bordes and Yoshua Bengio. Deep sparse rectifier neural networks. *JMLR Workshop and Conference Proceedings*, 2011. p. 315-323.

[5] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[6] Geoffrey Hinton, Nitish Srivastava and Kevin Swersky. Lecture: Overview of mini batch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: June 2016.

[7] David E. Rumelhart, Geoffrey Hinton and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, (323):533–536, 10 1986.

[8] Sepp Hochreiter and Jurgen Schmidhubers. Long short-term memory. *Neural Computation*, (9):1735–1790, 1997.

[9] D. H. Hubel. Cortical unit responses to visual stimuli in nonanesthetized cats. *American Journal of Ophthalmology*, (46):11O–122, 1958.

[10] Henry J. Kelly. Gradient theory of optimal flight paths. *Ars Journal*, 1960.

[11] Tambet Matiisen. Nervana blog post: Demystifying deep reinforcement learning. `https://www.nervanasys.com/demystifying-deep-reinforcement-learning/`. Accessed: June 2016.

[12] Francisco S. Melo. Convergence of q-learning: a simple proof. `http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf`. Accessed: June 2016.

[13] Dominik Scherer, Andreas Muller and Sven Behnke. 2010. *ICANN*, September.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[15] Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[16] Julian Togelius. Julian togelius's blog. `http://julian.togelius.com/`. Accessed: June 2016.

[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. Mastering the game of go with deep neural networks and tree search. *Nature*, (529):484–489, January 2016.

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.