Transfer between Atari Games

Gabriëlle Ras
Department of Artificial Intelligence
Radboud University Nijmegen
g.ras@student.ru.nl

Abstract

This paper looks at transfer learning between Deep Q-Networks in the context of playing three Atari 2600 games: Carnival, Demon Attack and Space Invaders. Experiments are done with different source networks to find out if this will increase performance compared to networks trained from scratch. We also experiment with the time the source networks train and look at the effects on the performance. And finally the similarity between learned features in the source networks is measured to see if there is a relationship between similarity in learned features and performance. We discovered that initialization with a pre-trained network does not always seem to work in a convincing manner, training time of the source networks does not matter much and the results do not indicate a relationship between similar learned features and performance.

1 Introduction

Deep learning is a fast-growing field, with new architectures and algorithms appearing every few weeks. Most of them are based on original parent architectures, while others are combinations of methods that already exist. An example of such a network is the Atari playing Deep Q-Network (DQN) by Mnih et al. [2013], which combines a Deep Convolutional Neural Network with Q-learning, a type of reinforcement learning. Unlike previous reinforcement learning models, DQNs can learn directly from high-dimensional sensory inputs, in this case raw pixel input. While the results are impressive, the network has to learn from scratch to play each game separately. There may be benefits to reusing already learned features that can be transferred between games. It was shown by Yosinski et al. [2014] that networks trained on natural images learn the same features in the initial layers. These are considered general features that are transferable to other networks with a similar task and data set. The questions is if the same is applicable to Atari 2600 input, since these visuals are not natural. In this paper we use pre-traied networks to investigate how knowledge is transferred between Atari Games, how different transfer weight initializations affect the performance of a DQN and the influence of the amount of time a network has trained before it was given another task to train on. We also investigate if networks that have learned similar features are better candidates for transfer. The idea is that given two similar games, different networks will learn similar features, thereby being a better candidate for transfer. Specifically we try to answer the following questions in the context of playing Atari with a DQN:

1. Does initialization with a pre-trained network increase performance?
2. Does the amount of training time of the pre-trained network affect the performance?
3. Is there a relationship between learned feature similarity and performance?

For the purposes of the experiments, we select three similar games. This means that the action-space is the same, the task is the same and visually the games are similar. Unlike Rusu et al. [2016] we were able to find a set of Atari games that are greatly similar. We chose the Atari 2600 games: Carnival (CA), Demon Attack (DA) and Space Invaders (SI). See Figure 1. All the games have the

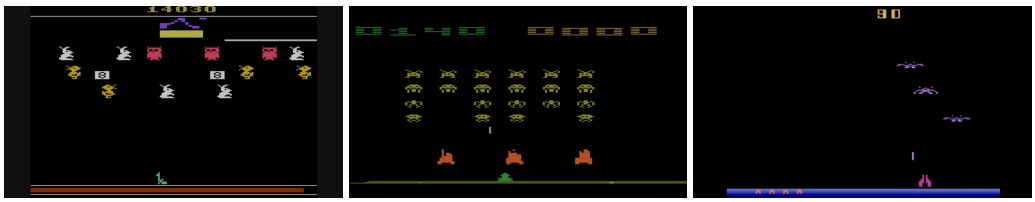


Figure 1: Screenshot of the three games used respectively: Carnival, Space Invaders and Demon Attack.

following gaming premise: The player is on the ground and has to shoot targets in the sky while avoiding occasional attacks from these targets that diminish the lives of the player. Each time the player successfully shoots the target, a counter with the score is increased. Given the similarity of the games transfer should be possible and our results indicate that pre-training does help in most cases. However, we have found that this does not happen.

2 Background

2.1 Transfer Learning

Aggarwal [2014] loosely defines transfer learning as *"the ability of a system to recognize and apply knowledge and skills learned in previous tasks to novel tasks or new domains, which share some commonality"*. The main idea behind transfer learning is to extract knowledge from a related domain, called the *source*, to enhance the performance of a learning algorithm in the domain of interest, called the *target*. In our case the source is a game G_s which is an object in the set of games $G = \{CA, SI, DA\}$ and the target is a game G_t from the set of games $G - G_s$. There are three main questions in transfer learning: 1) what to transfer, 2) how to transfer and 3) when to transfer. In our case we will 1) transfer learned features and weights by 2) instantiating a network with a trained network at 3) the beginning of the training phase.

2.2 Reinforcement Learning

In reinforcement learning, the agent interacts with an environment by selecting actions in a way that maximises future rewards. In our case the environment is the Atari 2600 emulator. At each time-step the agent selects an action a_t from a set of legal game actions $A = \{1, ..., K\}$. The action modifies the environment and also the game score. The agent only observes the screen, which is given as a vector of raw pixel values. This is known as an observation x_t. The agent does not have access to the internal state of the emulator. Additionally, the agent is given knowledge of the game score as it receives a reward r_t representing the change in game score.

The goal of the agent is to learn a policy that maximises future rewards. A policy is a function that yields the next action based on a sequence of past observations and actions: $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$. If we consider a future time-step in the game, we can predict how much our accumulated reward at that time-step would be. We call this the future discounted *return* at time t: $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r^{t'}$ where γ is the factor which future rewards are discounted by and T is the time-step at which the game terminates. We want to find the optimal policy π that maximizes R based on the sequence s of past actions and rewards, by giving a value to each action in A and then choose the action with the highest value. We denote this as the optimal action-value and it is given by the function $Q^*(s, a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$.

In our case we use the DQN to estimate $Q(s, a; \theta) \approx Q^*(s, a)$ where θ are the weights of the network. The network is trained by minimising a sequence of loss functions $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\Big[(y_i - Q(s, a; \theta_i))^2\Big]$ where $y_i = \mathbb{E}_{s'\sim E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over s and a.

2.3 Atari Deep Q-Networks

A DQN is a Deep Convolutional Neural Network trained with a modified version of the Q-learning algorithm. The goal behind the network is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates. This is inspired by a a backgammon-playing program which learnt entirely by reinforcement learning and self-play, see Tesauro [1995]. In this work the architecture updates the parameters of a network that estimate the value function, directly from on-policy samples of experience, $s_t, a_t, r_t, s_{t+1}, a_{t+1}$, drawn from the algorithm's interactions with the environment.

In contrast to this method, Mnih et al.'s DQN uses a technique called experience replay, see Lin [1993], where the experience of the agent is stored at each time-step as $e_t = (s_t, a_t, r_t, s_{t+1})$ in $D = (e_1, ...e_N)$, pooled over many episodes or games into a replay memory. The agent selects actions according to an ϵ-greedy policy. The action is performed and the reward and observation are stored in D. Then a sample of experiences e is drawn at random from D and a gradient descent step is applied on $y - (Q(s_t, a_t; \phi)^2)$ where s_t and a_t are from the sampled $e_t = (s_t, a_t, r_t, s_{t+1})$. The weights are then updated. It must be noted that when we use a pre-trained network the replay memory of the pre-trained network is also copied to the new network.

This approach has several advantages over standard online Q-learning: 1) Each step of experience is potentially used in many weight updates, which allows for greater data efficiency. 2) Randomizing the samples breaks the correlations that are present between samples when learning directly from consecutive samples and therefore reduces the variance of the updates. 3) By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

2.3.1 Pre-processing and Model Architecture

The raw Atari frames are 210×160 pixel images with a 128 color palette. This is computationally demanding so the input dimensionality is reduced by converting the RGB image to grey-scale and down-sampling it to a 110×84 image. Then an 84×84 region of the playing area is cropped and this is the final input.

The input to the neural network consists of an $84 \times 84 \times 4$ array, representing a stack of four most recent image frames. The first hidden layer convolves 16 8×8 filters with stride 4 with the input array and applies rectifier nonlinearity. The second hidden layer convolves 32 4×4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. There are six valid actions in total.

3 Related Work

Most work on transfer learning focuses on transfer where a model is pre-trained on a source domain and the output layers of the model are converted to suit the target domain. Enhanced performance with this method was achieved across many domains where deep learning is applicable. Yosinski et al. [2014] showed that copying up to the first three layers of a deep network and then re-training the network increases performance significantly because networks trained on natural images learn the same features in the initial layers.

Experiments by Erhan et al. [2009] point out that pre-training adds robustness to a network with a deep architecture, gives consistently better generalization and that it is more effective for lower layers than for higher layers. However, pre-training does not always help: If the layers are small, pre-trained deep networks perform systematically worse than randomly initialized deep networks.

A big open question in transfer learning is finding out which tasks are relevant to transfer from, as it is known that transfer from inappropriate tasks can decrease performance on the new task. Ammar et al. [2014] used a data-driven automated similarity measure for MDPs to indicate when transfer will be useful and to automatically select tasks to use for transfer. Their results show that this measure is correlated with the performance of transfer and therefore can be used to identify similar source tasks

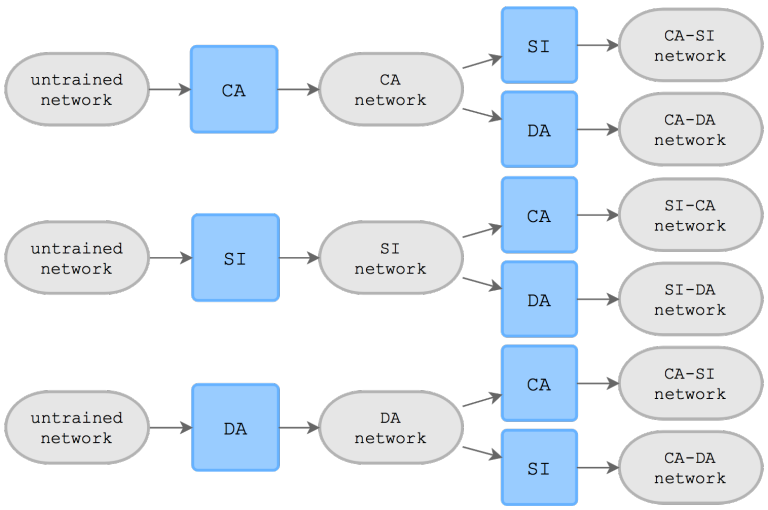


Figure 2: A visualization of the experiment setup where we have three games (CA, SI and DA). Each untrained DQN is trained for 100 epochs on the three games respectively to obtain N_s, where $s = G_s$ (the game you get knowledge from). We make a copy of each network and train each copy on a target game, obtaining $N_{s,t}$, where $t = G_t$ (the game transfer knowledge to). For example, if $G_t = CA$, we compare the performance of N_{CA} to the performance of $N_{SI,CA}$ and $N_{DA,CA}$. In other words, we want to know if the knowledge obtained when learning SI or DA will transfer to CA.

for transfer learning. This implies that the more similar the source and target task states, the better suited the source is for transfer.

Li et al. [2015] showed that some features are learned reliably in multiple networks, while other features are not consistently learned. This has implications for transfer learning, in the sense that if a transfer method seems to be unsuccessful, it could be that the source network failed to learn transferable features at the time of the transfer. Or its reverse: if a transfer method worked well it might be that a source network learned rare transferable features and that this might not generalize.

4 Methods

4.1 Experiment Setup

The basic idea of the experiment is to test whether or not using a previously trained network on one game (source) is beneficial for learning to play a similar game (target). We create a DQN N_s and train it to play a source game G_s and we create another DQN N_t and train it to play a target game G_t. Then we take N_s and train it to play G_t which yields $N_{s,t}$. Also N_t is trained to play G_s which yields $N_{t,s}$. In Figure 2 this setup is visualized when we have three games. In the experiments we vary the time at which the layers and weights of source network N_s are copied. For experiment 1 we copy the values at epoch 50 and in experiment 2 we copy the values at epoch 100. This is represented as N_{S50} and N_{S100}, meaning source network at epoch 50 and 100 respectively. The layers contain the learned features information and the weights represent the strength between the connections in the layers. At epoch 50 we could say that the network is less experienced at playing G_s than at epoch 100: the learned features are less developed and the weights are less biased. We compare the two different initializations to see if experience has an effect on the performance of $N_{s,t}$. The hypothesis is that source networks copied at epoch 50 should perform better than source networks copied at epoch 100 because they are less biased towards the source game. Each experiment is run three times and the average is taken. In addition we also compare the similarity between learned features of the source game G_s and the target game G_t to find out if there is a relationship between learned feature similarity in source networks and the performance of the network after re-training. An example: if it results that network 1 trained on game 1 and network 2 trained on game 2 learn similar features, then network 1 is a good candidate for transfer to learn game 2 and the reverse, because since network 1 learned similar features as network 2, network 1 should provide an increase in performance when used as a

pre-trained network to learn game 2. The network consists of five layers, but we compare features between only the first four layers, so the two convolutional layers and the two fully-connected layers. We do not look at the last layer because it is the output layer, giving the probabilities of the allowed actions. So for example, the distance is measured between the features in the second convolutional layer of network N_{DA} and the features in the second convolutional layer of network N_{CA}.

4.2 Technical Specifications

The code used to experiment with was the Theano-based implementation by Sprague [2015]. The technical specifications of the implementation can be found at `http://redsphinx.github.io/DQNblog/`. We have left all parameters at their default value. Each model is trained for 100 epochs. Each epoch consists of a number of games, which are called episodes. In each training epoch, the agent has 50000 steps to play as many episodes as possible without dying. At each step, the selected action gets repeated a number of times. We keep track of how many steps each episode takes. At the end of an episode, which occurs when the agent dies or when all 50000 steps have been used, those steps get subtracted from the remaining number of steps and a new game is started.

5 Results

To compare the performance between N_s and $N_{s,t}$ we look at the average action value $\overline{Q}$ and the average reward per episode $\overline{R}$. These are chosen as performance measures because they were used as performance measures in the original paper by Mnih et al. [2013]. Each of these measures is per epoch. $\overline{Q}$ provides an estimate of how much discounted reward the agent can obtain by following its policy from any given state. $\overline{Q}$ will eventually converge to a constant value, indicating that the network reached an optimal policy. $\overline{R}$ also converges to a maximum value because there is a maximum score that can be achieved in 1 episode. To show that $N_{s,t}$ performs better than N_s we want both $\overline{Q}_{s,t}$ and $\overline{R}_{s,t}$ to be larger than $\overline{Q}_t$ and $\overline{R}_t$ respectively.

5.1 Average action value $\overline{Q}$

Looking at Figure 3 we compare $\overline{Q}$ values. When playing CA, we see that $\overline{Q}_{DA,CA}$ and $\overline{Q}_{SI,CA}$ are both smaller than $\overline{Q}_{CA}$, for initialization with N_{S50} as well as N_{S100} for both source games DA and SI. However there is a noticeable upwards trend in the values of $\overline{Q}_{DA,CA}$ and $\overline{Q}_{SI,CA}$. More epochs are needed to be certain, but it seems that $\overline{Q}_{DA,CA}$ and $\overline{Q}_{SI,CA}$ will eventually reach and surpass $\overline{Q}_{CA}$. When playing DA, we see that $\overline{Q}_{CA,DA}$ and $\overline{Q}_{SI,DA}$ are both smaller than $\overline{Q}_{DA}$, for initialization with N_{S50} as well as N_{S100} for both source games CA and SI. When playing SI, we see that $\overline{Q}_{CA,SI}$ is greater than $\overline{Q}_{SI}$ from about epoch 45, for initialization with N_{CA50} as well as N_{CA100}. $\overline{Q}_{DA,SI}$ reaches $\overline{Q}_{SI}$ at about epoch 50, for initialization with N_{DA50} as well as N_{DA100}. Overall there is no large difference to be seen between initialization with N_{S50} and initialization with N_{S100}. Both initializations yield similar values.

5.2 Average reward per episode $\overline{R}$

Looking at Figure 4 we compare $\overline{R}$ values. Notice that the y-axis differs between games, this is because each game is different and has their own scoring system. For example, in CA you can get a higher score per episode than in DA and SI. When playing CA, $\overline{R}_{DA,CA}$ and $\overline{R}_{SI,CA}$ reach $\overline{R}_{CA}$ at about epoch 70, for initialization with N_{DA50} and N_{SI50} respectively. For initialization with N_{DA100} and N_{SI100} this happens near epoch 90. Note that at epoch 100 both $\overline{R}_{DA,CA}$ and $\overline{R}_{SI,CA}$ are greater than $\overline{R}_{CA}$. For initialization with N_{S100} there seems to be an upward trend after epoch 50. More epochs are needed, but it could imply that N_{S100} initialization for CA leads to higher $\overline{R}$ over time. When playing DA, we see that $\overline{R}_{CA,DA}$ and $\overline{R}_{SI,DA}$ are both smaller than $\overline{R}_{DA}$, for initialization with N_{S50} as well as N_{S100} for both source games CA and SI. When playing SI, we see that $\overline{R}_{CA,DA}$ and $\overline{R}_{SI,DA}$ both ultimately have a greater value than $\overline{R}_{DA}$ at epoch 100, for initialization with N_{S50} as well as N_{S100} for DA. For CA with initialization at epoch 50 only. For comparing $\overline{R}$ values we can see that there is some difference between initialization with N_{S50} and initialization with N_{S100}.

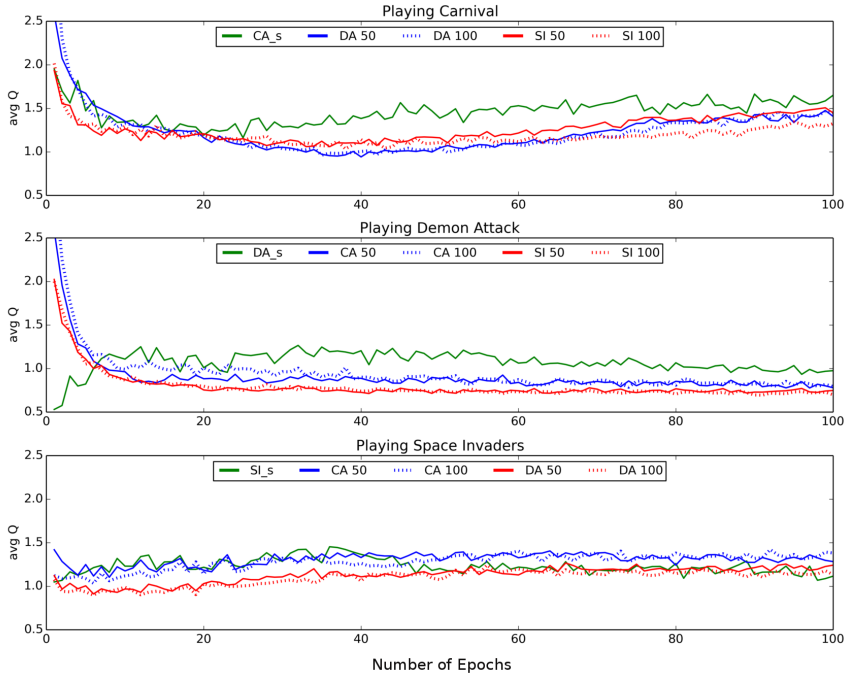


Figure 3: Comparison of $\overline{Q}$ between the source network and the re-trained network. On the y-axis we have the average Q and on the x axis we have the number of epochs. If we compare CA_s with DA50 it means that we compare the source network for CA with a network that has been pre-trained on DA for 50 epochs and then re-trained on CA for 100 epochs.

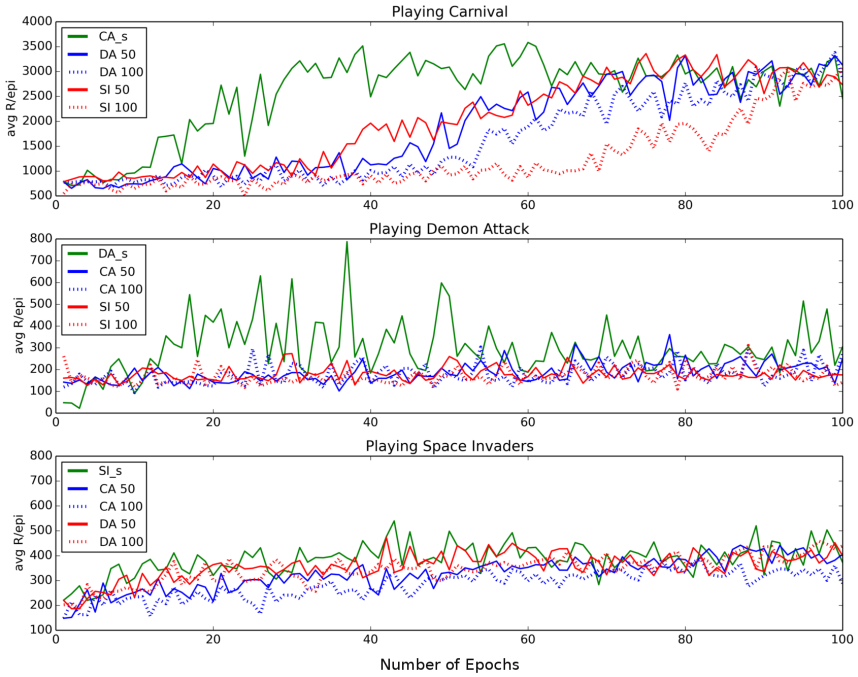
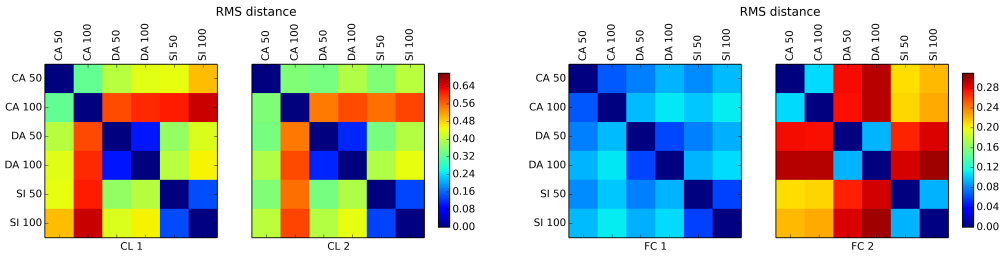


Figure 4: Comparison of $\overline{\mathrm{R}}$ between the source network and the re-trained network. On the y-axis we have the average reward per episode and on the x axis we have the number of epochs. If we compare CA_s with DA50 it means that we compare the source network for CA with a network that has been pre-trained on DA for 50 epochs and then re-trained on CA for 100 epochs.



(a) Convolutional layers

(b) Fully-connected layers

Figure 5: RMS distance matrix between learned kernels per layer per source network. In 5a we look at the average RMS distance between the features in convolutional layers 1 and 2. If we take a (row, column) pair such as (CA100, DA100) at CL1 we can see the average distance between the features in these networks in convolutional layer 1, the networks being source CA at epoch 100 and source DA at epoch 100 respectively. The colors indicate how similar the features in the layers are on average: the more blue te color, the smaller the average distance, hence the more similar the features are on average. And the more red the color, the greater the distance, hence the less similar the features are on average. In 5b the same is done for the fully-connected layers.

5.3 Learned Features Similarity

In Figure 5 we plot the distance between features in each layer of the source network in 5a and 5b. RMS is used to calculate the average distance per source network per layer: $\sqrt{\frac{\sum_{j=1}^{n}(w1_j - w2_j)^2}{n}}$ where w is the weights of the network and $n = f \times k \times c$, f is the number of filters, k is the kernel dimension and c is the number of channels. We can see that in 5a the distance of CL1 is bigger than in CL2. In 5b the opposite happens: The distance of FC1 is smaller than in FC2. We can interpret this distance change as: 1) Because the games are similar, eventually the networks will learn similar kernels. This is in line with the observation that kernels in CL2 are more similar than kernels in CL1. 2) Features in FC layers get specialized no matter the similarity of the game. Unfortunately there is no discernible pattern between Figure 5 and Figures 3 and 4. We think that a more adequate measuring method is needed.

6 Conclusion

The results have been condensed in Table 1 for a snapshot moment at epoch 100. Now we can compare the snapshot of the state of the networks with the graphs in Figure 3 and Figure 4 to compare the gradual progress over 100 epochs vs. the performance at the end. Looking at Figure 3 and Figure 4, we can see that larger values of $\overline{Q}$ coincide with larger values of $\overline{R}$ per episode from epoch 0 to 100, even though in Table 1 there is no clear indication of a relationship between $\overline{Q}$ and $\overline{R}$. Seven out of twelve times, pre-training resulted in an increase in $\overline{R}$ at epoch 100. Sometimes it seems that pre-training increases performance near the end, but not always. However looking at the complete progress over time from epoch 0 to epoch 100 the source network does much better. This distinction in two ways of interpreting the results yields a bit of a mixed bag: On the one hand we can say that it worked when strictly looking at the 100^{th} epoch while on the other hand we can argue that there is not much improvement and that the source network performs better when looking at the total $\overline{R}$ per 100 epochs. This is largely because the networks seem to stabilize around the 100^{th} epoch. We conclude that more epochs are needed to verify that the stability and value of $\overline{R}$ is consistent over time and that in general transfer does not work. The amount of training time of the source models yields inconsistent increase or decrease in performance. For some games initialization with a source network at 50 epochs increases performance, but for others initialization with a source network at 100 epoch increases performance. These results are inconclusive. Since it is difficult to relate four layers individually to an increase in performance, we look at the average RMS distance per network $RMS_{avg} = \frac{RMS_{CL1}+RMS_{CL2}+RMS_{FC1}+RMS_{FC2}}{4}$. Comparing RMS_{avg} we find the data inconclusive to say anything about how feature similarity in source networks affects performance.

	Q at 100	R at 100	RMS avg	RMS CL1	RMS CL2	RMS FC1	RMS FC2
CA source	1.65	2450.95	0	0	0	0	0
SI50CA	1.45	2728.88	0.37	0.62	0.55	0.1	0.21
SI100CA	1.33	3072.27	0.4	0.66	0.59	0.11	0.23
DA50CA	1.41	3126.04	0.38	0.58	0.55	0.1	0.28
DA100CA	1.45	2859.92	0.4	0.61	0.58	0.11	0.29
DA source	0.98	306.25	0	0	0	0	0
CA50DA	0.78	251.93	0.3	0.44	0.39	0.09	0.28
CA100DA	0.79	272.83	0.4	0.61	0.58	0.11	0.29
SI50DA	0.75	175.31	0.3	0.4	0.4	0.09	0.29
SI100DA	0.7	141.96	0.31	0.4	0.44	0.11	0.3
SI source	1.12	371.88	0	0	0	0	0
CA50SI	1.28	406.38	0.31	0.5	0.4	0.1	0.22
CA100SI	1.38	288.03	0.4	0.66	0.59	0.11	0.23
DA50SI	1.24	398.73	0.3	0.43	0.4	0.09	0.29
DA100SI	1.13	439.3	0.33	0.46	0.44	0.11	0.3

Table 1: A summary of the experiment results at epoch 100. In the top row: Q at 100 means $\overline{Q}$ at epoch 100, R at 100 means $\overline{R}$ at epoch 100, RMS CL means root mean squared distance between the convolutional layers and RMS FC means root mean squared distance between the fully-connected layers. RMS avg is the mean of the other four RMS, RMS avg can be seen as the average distance between features per entire network, and not just per one layer. In the leftmost column: the source is the network trained on the source game. "SI50CA" should be read as *source SI network copied at epoch 50 re-trained on CA*. Each game in the column is compared to it's source game. For instance "SI50CA" is compared to "CA source", while "CA50DA" is compared to "DA source". Note that all the source games have an RMS of zero, because the distance between the features in the layers of the source game and itself is zero.

7 Discussion and Future Work

It was surprising to discover that transfer does not work in DQN when playing Atari because it has been successful in a plethora of other domains. A reason that transfer is unsuccessful could be that the network learns features that are irrelevant to the task, but that are relevant to the specific game. These provide better initialization than random, as can be seen from the larger starting values in Figure 3 and Figure 4, but lead to a slower increase in performance. As proposed in Rusu et al. [2016], it could be the case that the transferred representation is good enough for a functional, but suboptimal policy. However, in Rusu et al. [2016] the network architecture was very different from a DQN, but seeing as both architectures implement a type of memory, the same can hypothesis can be applicable to both architectures. Concerning the comparison between similarity in learned features, a better method of measuring similarity is necessary as the average RMS "blurs away" too much feature detail by taking an average. One alternative is to measure the correlation between the matched layers. Using t-SNE is another possibility as the data is relatively high dimensional, see Maaten and Hinton [2008]. An investigation of the MDP similarity measure should also be conducted to evaluate whether it can be used with a DQN and Atari games, see Ammar et al. [2014]. It may not work because MDPs for Atari games are large and are constructed as the network is trained. All results indicate that the networks need to train for more epochs to see if the results will stabilize after a while. More research is needed to investigate what happens at later epochs as 100 epochs is not enough. Also, experiments need to be done with games that are completely different from any of the games chosen in this paper, to compare the transfer between dissimilar games and similar games. Lastly, we could consider copying only certain layers and research the effects of leaving out the replay memory when copying. To determine which layers would be best for transfer we could apply Average Perturbation Sensitivity to evaluate the degree to which source layers contribute to the target task.

References

Charu C Aggarwal. *Data Classification: Algorithms and Applications*. CRC Press, 2014.

Haitham Bou Ammar, Eric Eaton, Matthew E Taylor, Decebal Constantin Mocanu, Kurt Driessens, Gerhard Weiss, and Karl Tuyls. An Automated Measure of MDP Similarity for Transfer in Reinforcement Learning. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. In *AISTATS*, volume 5, pages 153–160, 2009.

Yixuan Li, Jason Yosinski, Jeff Clune, Hod Lipson, and John Hopcroft. Convergent Learning: Do Different Neural Networks Learn the Same Representations? *arXiv preprint arXiv:1511.07543*, 2015.

Long-Ji Lin. Reinforcement Learning for Robots Using Neural Networks. Technical report, DTIC Document, 1993.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive Neural Networks. *arXiv preprint arXiv:1606.04671*, 2016.

Nathan Sprague. Theano-based Implementation of Deep Q-learning. `https://github.com/spragunr/deep_q_rl`, 2015.

Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38 (3):58–68, 1995.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable are Features in Deep Neural Networks? In *Advances in Neural Information Processing Systems*, pages 3320–3328, 2014.