Recurrent Deep Q-Learning for PAC-MAN

Kushal Ranjan
kranjan@stanford.edu

Amelia Christensen
amyjc@stanford.edu

Bernardo Ramos
bramos@stanford.edu

Abstract

Classic Artificial Intelligence agents are limited by the creation of handcrafted features which include specific domain knowledge about the rules of the game which they are playing. In this project, we take the data-driven approach, and attempt to train deep neural networks to play the game PAC-MAN, using no specific game knowledge, only features extracted from raw-pixel images using a convolutional neural network paired with recurrent neural network layers. We experiment with several architectures and learning strategies, including supervised learning, transfer learning with an inception network, a small convolutional network, and a convolutional network combined with an LSTM unit.

1. Introduction

The recent decade has seen a a revolution in the predictive power of neural networks, especially for ”sensory discrimination” types of problems (e.g. visual object recognition, or natural language processing). Although the adoption of artificial neural networks to reinforcement learning paradigms has been slower, promising results have been attained. For example, Google DeepMind used Convolutional Neural Network (CNN) architectures as part of an algorithm termed ”Deep Q-Learning” (DQL) to play Atari games [15] remarkably well, sometimes even surpassing human performance. The team used a data-driven paradigm, in which they provided the agent only the game pixels and reward values, diverging from the more traditional approach of handcrafting features. The DeepMind team also had recent success training a DQL based agent to play Go at a level that surpassed that of a human professional player [20], a feat that pundits had previously proclaimed was more than a decade away.

These works illustrate the large potential power CNNs and deep reinforcement learning possess, which will facilitate flexible, intelligent game playing agents. However, there still exist situations in which DQL has not yet been successful, for example, in arenas in which rewards are sparse, or temporally removed from the actions which were causal to the agent receiving the reward. One example of such a game, where existing deep learning algorithms have failed to produce impressive results is PAC-MAN.

Throughout this work, we will study an application of Deep Q-Learning (more specifically, a CNN paired with a Recurrent Neural Network) in learning to play PAC-MAN using raw pixel images only. We hope that the combination of a convolutional network which should extract salient spatially invariant features from the game pixels, and the recurrent net which will allow the agent to integrate information about past action sequences that it has taken, will optimize the performance of the DQL network. We will also discuss the difficulties encountered when performing transfer versus bottom-up learning and provide an insight of future improvements.

2. Related Work

The DQL algorithm was first described by Minh et. al. [15] in 2013, where the DeepMind team performed standard approximate Q-Learning, replacing the Q function with a deep convolutional network. They then applied the same framework to all of the Atari games, in many cases achieving performance surpassing that of human experts. This demonstration spurred a number of groups to implement variations of DQL. [4, 10, 12, 16, 17, 23, 25]

Perhaps the most similar of these implementations to our work is the Deep Recurrent Q-Learning framework published by Matthew Hausknecht and Peter Stone in 2015 [9]. Hausknecht and Stone replace the last fully connected layer from the convolutional networks used by Minh et. al. with an LSTM, and then train the network on the Atari games. While they do not find that the addition of the LSTM either positively or negatively impacts performance in normal playing conditions, they do notice that in cases when there are flickers on the game screen, or some other inconsistency in the test data, their network is able to perform better than when just a convolutional network is used. They conclude that the addition of the LSTM provides robustness against test time variations in the statistics of the environment.

Another relevant application of Deep Reinforcement Learning, this time to the game Go, was published by DeepMind in Nature this year [20]. Although the approach was quite different from the previous DQL work, both for the combination of roll outs and MonteCarlo Tree search and the use of two networks, one to estimate the value of the board at any given time, and another to select an action, the authors used supervised learning to initialize the values of the action policy network and the value network, a method which we adopted.

Although there are examples of approximate Q learning or DQL on top of handcrafted features (e.g. [3]) that perform quite well, to the best of our knowledge, there is currently no DQL implementation of PAC-MAN that uses convolutional layers to extract features from the game board that comes even close to rivaling scores achieved by human players.

3. Methods and Technical Approach

Our goal is to develop a reinforcement learning agent for PAC-MAN that uses a combination of CNN's and recurrent neural network with LSTM units to model the Q-value function $Q(s, a)$. In this section we discuss the technical details surrounding Reinforcement Learning (section 3.1) and Deep Q-Learning (section 3.2), show our motivation and hypothesis for introducing RNN's (section 3.3), and describe the concrete architectures we will use to compare our results (section 3.4). Finally, we describe a supervised learning task that will give an insight of the learning capacity under the RL paradigm (section 3.5).

3.1. Reinforcement Learning Overview

Let $\mathcal{S}$ be the set of states, and $\mathcal{A}$ the set of actions of a Markov Decision Process (MDP). In our setting, the set $\mathcal{S}$ consists of all the possible configurations of PAC-MAN, the ghosts and the food in a given level, and $\mathcal{A}$ is the set of actions PAC-MAN can take: left, right, up, down, or stay.

In Reinforcement Learning, we are concerned with finding the policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the value function $V_\pi(s)$, defined as the sum of the expected (discounted) rewards when following policy π starting in state s. To this end, we define the Q-value of a policy as

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')(\text{Reward}(s, a, s') + \gamma V_\pi(s')),$$

where $T(s, a, s')$ denotes the probability of the underlying Markov Decision Process transitioning from state s to s' after taking action a, and $0 \leq \gamma \leq 1$ is the discount parameter.

A necessary condition for a policy π_{opt} to be optimal is to satisfy

$$\pi_{\text{opt}}(s) = \arg\max_{a \in \mathcal{A}} Q_{\pi_{\text{opt}}}(s, a) \qquad \forall s \in \mathcal{S}, \tag{1}$$

hence a strategy for finding the optimal policy is to directly estimate Q_{opt} (the Q-function of an optimal policy) and use (1) to find $\hat{\pi}_{\text{opt}}$. One way to do this is to fit a neural network, which we parametrize by θ, with an objective loss given by

$$L(\theta) = \sum_{(s,a,r,s')} \left(\hat{Q}_{\text{opt}}(s, a; \theta) - (r + \gamma \hat{V}_{\text{opt}}(s'))\right)^2,$$

where (s, a, r, s') are the values of the states, actions, rewards and successive states in the observed paths and $\hat{V}_{\text{opt}}(s) = \hat{Q}_{\text{opt}}(s, \hat{\pi}_{\text{opt}}(s))$. The network can then be trained using stochastic gradient descent updates.

In a real-world setting, an optimal-policy agent is both concerned with collecting rewards following the estimated optimal strategy, and finding a good estimator of the Q function across the state-action space. The latter requires a thorough exploration of $\mathcal{S} \times \mathcal{A}$, which can be achieved by deviating from the optimal policy with a certain probability ϵ. Such a strategy is called ϵ-greedy, and is formally given by

$$\pi_\epsilon(s) = \begin{cases} \hat{\pi}_{\text{opt}}(s) & \text{w. prob. } 1 - \epsilon \\ a \in \mathcal{A} \text{ selected randomly} & \text{w. prob. } \epsilon, \end{cases}$$

For the purposes of this work we will use ϵ-greedy exploration to compare the performance of different Q-Learning approaches.

3.2. Deep Q-Learning and Convolutional Neural Networks

In section 3.1 we discussed the general Q-Learning procedure without specifying the form of $\hat{Q}(s, a; \theta)$. Regular Q-Learning with function approximation takes $\theta \in \mathbb{R}^m$ of the form

$$\hat{Q}(s, a; \theta) = \phi(s, a) \cdot \theta,$$

where $\phi(s, a) \in \mathbb{R}^m$ is the vector of features describing the state s and the action a. The term Deep Q-Learning applies when the output of $\hat{Q}$ is generated from a deep neural network with inputs $\phi(s, a)$, which in our case includes the set of pixels of the image generated from state s. Motivated by the success of CNN's in vision applications (cf. section 1), we will use convolutional layers in the test architectures of section 3.4.

Our architectures will be structured so that activations $\hat{q}_t$ of the last layer at time t represent an expected Q value for the action that corresponds to that neuron, i.e. $\hat{q}_t = (\hat{Q}(\text{up}), \hat{Q}(\text{down}), \hat{Q}(\text{right}), \hat{Q}(\text{left}), \hat{Q}(\text{stay})) \in \mathbb{R}^5$. The action PAC-MAN will take is then

$$a_t = \arg\max_a \hat{q}_t(a),$$

as output neurons will be trained to contain the ‘scores’ for performing each action.

3.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) allow for the network to have ”memory” over past actions. LSTM (Long

Algorithm 1: batch Deep-Q-Learning

```
Initialize $cache \leftarrow \{\}$
Initialize Q-network parameters randomly
for $t$ = $1$ to $M$ do
    with probability $\epsilon$, $a_t$ is random
    otherwise, select $a_t$ = $\max_a(\hat{Q}(s, a; \theta))$
    execute action and observe new image
    store $(s_t, a_t, r_t, s_{t+1})$ in cache
    set $s_t$ = $s_{t+1}$
    if episode modulo batch size == 0 then
        perform gradient descent update on cached sequence
        clear cache
    end
end
```

Short Term Memory) units – introduced by Hochreiter and Schmidhuber [11] – are specialized neural network units that can remember input patterns over an arbitrary number of passes. They have been successfully used for natural language processing applications such as speech recognition and grammar modeling.

PAC-MAN is a game whose strategies require planning over potentially long sequences of actions. Our hypothesis is that a PAC-MAN RL agent with added memory will have an increased capacity of representing states to determine the best action sequences. We will test this hypothesis in section 5 by testing the performance of ConvNet architectures with and without LSTM layers.

There is, however, an implementation caveat for using recurrent neural networks. While LSTM layers will hopefully add a better state-action sequence representation of the game, they require the entire sequence of actions of a game to perform a single backpropagation step. This increases the computational time our agent requires for learning. To cope with this issue, we propose a learning procedure where a backpropagation step is performed after a batch of actions is performed. This updating procedure is described in Algorithm 1.

Using batch Q-learning introduces a trade-off between a more rapid learning procedure with the overall long-term representation of past actions. While batch Q-Learning does keep the hidden state of the LSTM layer as if learning was performed using the entire game sequence, the gradient of actions past a certain batch will be ignored, which potentially has an impact in long-memory sequences. Throughout this work we used batch sequences of length 6, a choice driven by the trade-off between the run-time of a single game and the amount necessary for tuning the network over a reasonable number of hours.

3.4. Architectures

In this section we present the proposed DQL architectures, designed to test the effects of performing transfer learning (Inception-LSTM network, section 3.4.1), adding an LSTM to a CNN trained from scratch (Conv-LSTM network, section 3.4.2), and training a vanilla ConvNet without RNN features (ConvNet Experience Replay, section 3.4.3).

3.4.1 Inception v3 + LSTM (Transfer Learning)

Our first architecture is designed to test the effect of transfer learning (the practice of using part of a trained network as feature extractors in another architecture), in the network's learning capacity. Works such as Taylor and Stone [22] emphasize the benefits of using TL in reinforcement learning applications, which stresses our motivation for combining a pretrained network with the proposed RNN structure.

Figure 1 depicts the flow of our Inception-LSTM architecture. It uses Google's Inception v3 [21] as a feature extractor and adds layers that will be trained using the procedure described in section 3.3. The architecture is structured as follows:

1. The raw image is processed to create features using Google Inception v3's fourth max-pool.
2. The features are fed into a fully connected layer (FC1).[1]
3. The result is the input of an LSTM cell, whose hidden state contains memory of past actions.[2]
4. An FC-5 layer outputs the estimated Q-values for the five actions (right, left, up, down and stop).

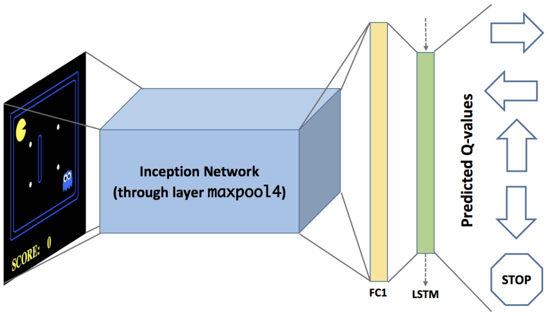

Figure 1. Inception-LSTM Architecture.

[1] The initial size is taken to be 200, though in section 5 we also consider a size of 50.

[2] We take the first LSTM size to be 150, and in section 4.1 we also consider a size of 15. This also applies to the Conv-LSTM architecture presented in 3.4.2.

While we expect transfer learning to provide an accelerated learning process, we should keep in mind that the nature of the application in which Inception v3 was trained (the ILSVRC challenge for object classification) is fundamentally different than ours. This means features created out of PAC-MAN images may not prove as meaningful as in object classification tasks.

3.4.2 ConvNet + LSTM

Our next architecture follows a CNN structure with an added LSTM layer. Unlike the Inception-LSTM architecture, all layers of this network will be trained from scratch. Figure 3.4.2 shows the Conv-LSTM structure, whose flow is described as follows:

1. The image goes through three stacks of 3x3 Convolution–ReLU–2x2 max-pool layers with 8, 16 and 32 filters, respectively.

2. The result is fed into two fully connected layers (FC1, FC2). [3]

3. FC2 connects to an LSTM cell.

4. An FC of size 5 outputs $\hat{Q}(s, a)$.

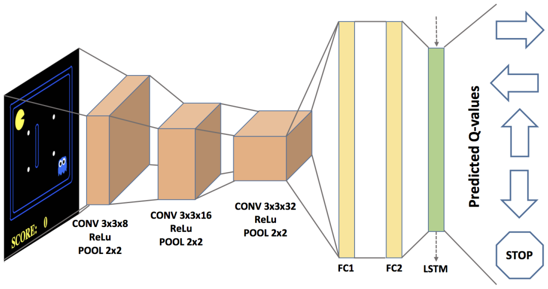

Figure 2. Conv-LSTM Architecture.

3.4.3 ConvNet Experience Replay (Non-RNN network)

Our two previous networks have included LSTM cells before outputting action scores. This network is intended to provide a benchmark similar to the model used by Mnih et al. [15], where the PAC-MAN agent learns to play without accounting for memory cells. The flow is described as follows:

1. The first layers are the same as ConvNet-LSTM (see Figure 3.4.2), except the FC2 layer's size is turned to 5.

2. The FC2 layer outputs the estimated Q-values. The LSTM cell is omitted to provide a non-memory benchmark for the proposed architectures.

To follow the steps of Mnih et al. [15], we will use an experience replay learning process, differing from the one used for LSTM structures. This process stores the past history of (s, a, r, s') tuples, and then updates the network based on a random past training examples, to remove correlation structure in the training examples.

3.5. Supervised Learning for Game Replaying

Consider the supervised learning task that uses same architectures as in section 3.4, but replacing the step loss function with cross-entropy

$$\ell_t(\theta) = -\sum_{a \in \mathcal{A}} 1_{\{a_t = a\}} \log \left(\frac{e^{\text{score}(a;\theta)}}{\sum_{a'} e^{\text{score}(a';\theta)}} \right),$$

where $(a_t)_t$ is a sequence of actions taken at time t, score$(a; \theta)$ is the output of the last layer corresponding to action a given the parameters θ, and the total loss is

$$L_S(\theta) = \sum_t \ell_t(\theta).$$

When optimizing with respect to the loss function L_S, the network learns to distinguish the actions PAC-MAN should take, based on the games provided (we train our networks using games played by a human). We believe using parameters tuned for minimizing L_S will prove useful as a weight initialization for minimizing the reinforcement learning loss, as learned weights incorporate knowledge about the PAC-MAN dynamics. We will explore the results of using the supervised learning procedure for weight initialization in section 4.1.

Moreover, having near-zero classification losses across various games suggests that the network can learn sequential strategies. Indeed, we can test the performance of the learned agent by letting it play in a RL setting and looking at the obtained scores. As we will see in section 4.1, our architectures *can* learn to play after being tuned in a supervised fashion.

3.6. PAC-MAN Implementation

Our code was built on top of a pre-existing Python implementation of PAC-MAN. The implementation is that used in UC Berkeley's Introduction to Artificial Intelligence course and was built by John Denero, Dan Klein, and Pieter Abbeel [1].

[3] We consider two different sizes: first, FC1 and FC2 will both have 200 neurons, and in section 4.1 we will reduce them to 50 and 25, respectively.

The implementation provides 5 moves: left, right, up, down, and stay. The score starts at 0 and changes at each timestep as follows:

- The score drops by 1.
- Eating food increases the score by 10.
- Eating a ghost increases the score by 200.
- Eating the last piece of food (winning the game) increases the score by 500.
- Being eaten by a ghost (losing the game) decreases the score by 500.

This implementation includes an interface for general Q-learning, but not for Deep Q-learning, which was implemented from scratch.

3.7. Computational Setup

We used TensorFlow [2] in Python as our deep learning library. The machines we used all ran OS X without Tensorflow-compatible GPUs. Thus, all code was run on 2.5 GHz Intel Core i7 CPUs.

4. Dataset and Features

Since the bulk of our training is reinforcement learning, our dataset was primarily generated as the agent progressed through the PAC-MAN game. It consisted of raw RGB images of the PAC-MAN game board. We used two game grids for our experiments: one simple, 5x6 grid with 4 pieces of food and a ghost; and one larger grid with many pieces of food and multiple ghosts. Example images are shown in Figures 3 and 4.

These images were downsampled using opencv [5] before being fed to our Q-nets from their original size of 540x540 pixels to 224x224 pixels. Screenshots were taken once per move using the ImageGrab functionality available in the Python Imaging Library. The project was developed in Python and we used Tensorflow for designing our deep architectures.

Q-learning feature extraction is handled by a convolutional network on top of the raw images. Since these convolutional layers are trained in conjunction with the rest of the Q-net, they learn features that are salient to Q-learning decisions.

4.1. Supervised Learning Data

Training deep convolutional networks is difficult, especially when considering the low rate at which we acquire data (once per move). Since obtaining thousands of training points may take on the order of days, we created a dataset of 6,500 labeled image-move training examples by playing PAC-MAN manually. This dataset was used when using our own, randomly initialized convolutional layers for feature extraction, as they allowed us to train those layers more effectively prior to any reinforcement learning. These images are standardized to zero-mean-unit-variance pixels.

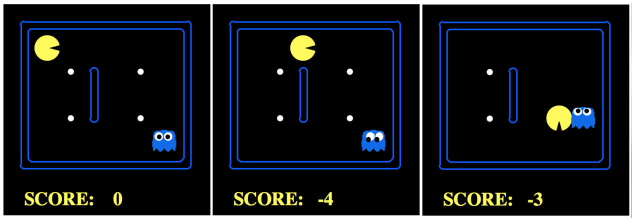

Figure 3. Example images from the grid `mediumGrid` used by the deep Q-nets. The original 540x540 images were rescaled to 224x224 before being passed to the networks. This grid was used for our initial reinforcement learning

Finally, we used a different grid than our previous attempts – a larger grid with more food and multiple ghosts, shown in figure 4 – for this phase of our experiments. We did so because, since the supervised training set was generated manually, it was possible for us to generate a greater diversity of training examples by using a more complex game grid.

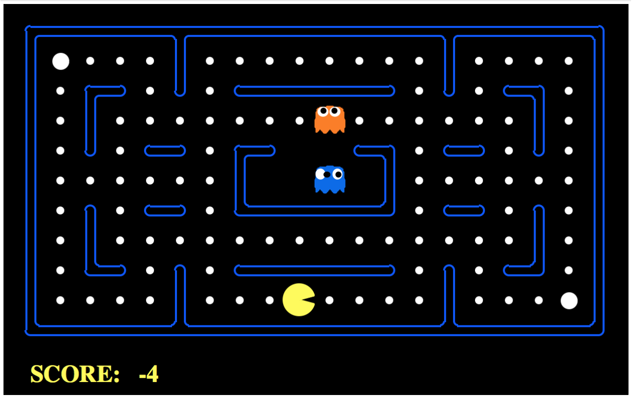

Figure 4. Example image from the grid `mediumClassic`. This grid was used once we started using supervised learning to pretrain the convolutional layers in our models.

5. Results

5.1. Initial Results

We first ran several thousand reinforcement learning iterations with our ConvNet-LSTM and Inception-LSTM architectures. The ConvNet architecture was that described in section 3.4.2 with activation depths of 8, 16, and 32 in the convolutional layers. All models were trained by Adam optimization of the DQL loss described in section 3.1. The resulting average step loss over 288 games for both models on `mediumGrid` is plotted in figure 5. We see a heavy drop in average loss within the first 100 games, after which it seems to level off.

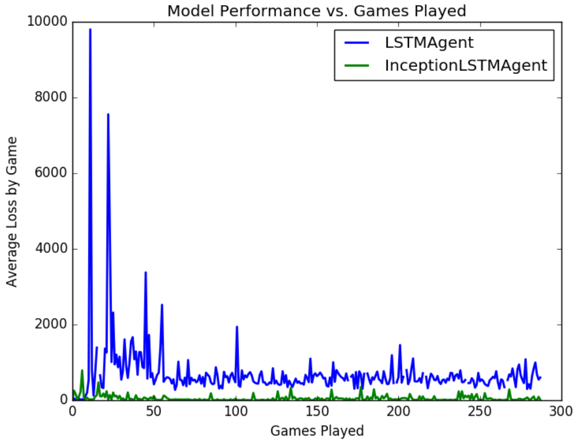

Figure 5. Average step loss per game over 288 games for both the ConvNet-LSTM agent and the Inception-LSTM agent. These values were calculated by averaging the DQL loss for batches of size 6 over the moves of each game.

Despite the drop in loss, we did not see a significant improvement in PAC-MAN performance by our agents. The resulting action patterns were largely cycles of left-right or up-down movements, with no inclination to avoid the ghost or to pursue food. This suggests one of two things:

- that PAC-MAN is too difficult to model with networks of the size that we used, and so even a seemingly converged network would have subpar performance, or
- that the few thousand DQL training iterations that we used were insufficient to properly train the network.

In the following sections we will show evidence against the first hypothesis. In particular, we will see how architectures with a significantly reduced number of parameters are able to learn game strategies when the problem is turned into a supervised learning task rather than a reinforcement learning one. As a consequence, the possibility that our architectures were misspecified (i.e. the addition of an RNN layer not generalizing enough, or layer sizes not being enough for a good state representation) appears to be weak.

For the case of randomly initialized convolutional layers – as opposed to using the Inception network – it's certainly likely that only a few thousand training iterations is insufficient to train those layers so to properly extract salient features from the PAC-MAN game board. This is supported by the observation that average loss using the pre-trained Inception network is significantly lower than that using randomly initialized convolutional layers, despite Inception being trained for ILSVRC rather than for PAC-MAN feature extraction.

Should hypothesis 2 be correct, then it would require 1000 hours to fully train a single CNN for a reasonable number of iterations – say, 325,000. We would then need 1.5 months to evaluate our RL agents at an acceptable capacity. The main restriction is that our PAC-MAN base code requires a physical screen grabbing, which ruled out any possibility of using remote GPUs or services such as AWS.

The following sections describe the procedure followed to cope with the restriction of a reduced training time. Moreover, we provide support for the hypothesis that learning the PAC-MAN game *is* feasible – at least in a supervised learning task. Finally, we use the tuned supervised learning parameters as the RL weight initialization as an attempt to improve the learning speed and explore the results.

5.2. Supervised Learning for Weight Initialization

To circumvent the difficulty of training deep convolutional networks with reinforcement learning described above, we attempted a supervised learning approach to pretrain the convolutional layers in our models. In particular, we used the dataset detailed in 4.1 to initially train the weights of our convolutional layers before applying reinforcement learning. We used an Adam optimizer with a learning rate of 10^{-3}, dropping that value as training loss leveled out. The agent used here is similar to the standard DQL agent detailed in [15], except that the supervised learning takes the place of experience replay; in fact, training batches were sampled randomly from the dataset, similar to experience replay's sampling from a move history cache.

The advantage of supervised learning over reinforcement learning is primarily speed. Supervised learning enables us to use batch training to significantly speed up the training process, rather than the one-by-one nature of reinforcement learning. First, we reduce the number of parameters used in each layer.[4] Table 1 shows the number of parameters used for each architecture before and after the change of the supervised learning tasks.

Architecture	First approach	Final sizes
Conv-LSTM	220,000	46,000
Inception-LSTM	18,000,000	4,700,000
Experience Replay	175,000	45,000

Table 1. Approximate number of parameters tested for each architecture.

[4] Concretely, we reduced all fully connected layer sizes from 200 to be 50 and 25, respectively, and changed the LSTM unit size from 150 to be 10.

By using batches of size 100, we were able to perform 11,000 updates (corresponding to 1,100,000 examples sampled from the dataset of 6,500 images) in just 8 hours. In contrast, performing 1,100,000 updates of reinforcement learning on our Conv-LSTM model would take roughly 5 months.

Indeed, we see that supervised learning provides for impressive game performance. Figure 6 shows the game scores obtained by agents trained with the supervised learning method. At each of the 6 benchmark locations – 100, 4800, 6700, 8800, 9500, and 10,900 – we ran the agent on 5 games to gauge its performance. We can see a clear upward trend as training proceeds. Figure 7 shows the step cross-entropy loss over 20,000 steps of the same training; the downward trend also demonstrates steady convergence. These two plots provide evidence that the ConvNet agent was being trained effectively, and would potentially provide well-trained convolutional layers for our Conv-LSTM DQL agent.

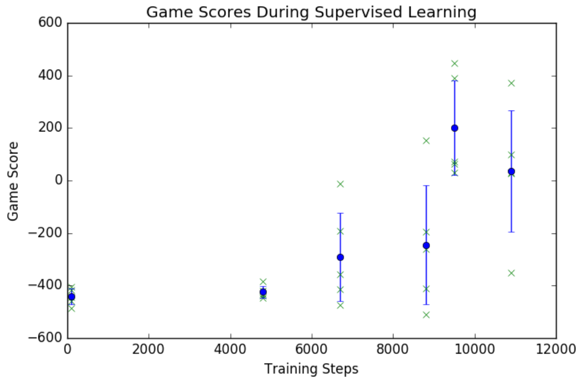

Figure 6. Average game scores at 6 points during training of a supervised learning agent. We see the general upward trend as training proceeds. This implies the ability to learn a reasonable PAC-MAN agent despite our relatively small models.

The implication of the score increase is that, despite what we had seen with the reinforcement learning agents, it is possible for a network with the number of parameters that we are using to be trained into a reasonably successful PAC-MAN agent. In particular, the convolutional layers generated by this method should be usable in a DQL agent, bypassing the need for training those layers using DQL over far more steps than feasible.

5.3. DQL After ConvLayer Pre-training

As in Figure 6, we saw steady game performance improvement over the course of our supervised learning. We did this with the intention of using the pre-trained convolutional layers from these models to expedite training on our DQL models.

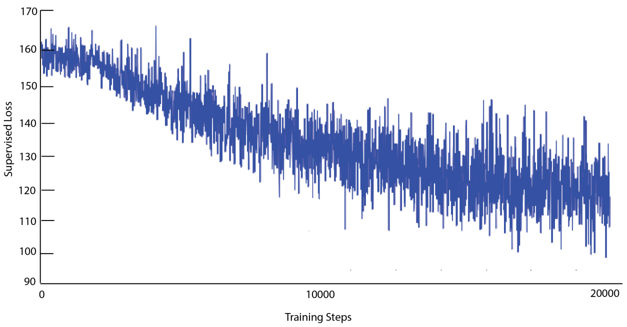

Figure 7. Cross-entropy loss over 20,000 supervised training steps. We see a gradual decrease in loss; this corresponds to Figure 6 to suggest that the ConvNet agent trained by supervised learning was indeed being trained effectively.

However, our results did not significantly improve beyond the supervised learning. In the 100 games played via DQL following the pretraining, our agent's performance did not increase; it obtained an average final game score of -42 over 31 games after training, lower than the averages seen in the later stages of the supervised learning, likely due to the addition of a relatively high $\epsilon = .2$, to encourage exploration.

The performance stagnation seen at this stage would suggest that our issues largely stem from the difficulty of DQL over many iterations. The same agent running only on supervised learning performs reasonably well, suggesting that our convolutional layers have learned features salient to Q-learning. Thus, performance due to DQL may only be achieved by more extensive training, which is currently infeasible due to our hardware limitations and the requirement that the PAC-MAN screen be visible at all times in order for the Deep Q-learning code to use it.

6. Conclusion

In this project, we explored different architectures and deep reinforcement learning training paradigms for the game PAC-MAN. We began with a standard DQL implementation, where the Q-network is a convolutional neural network, and experimented with adding an LSTM recurrent network after the fully connected layers, in an attempt to add sequence memory to the network.

We also experimented with using a pretrained Inception network to extract features, and with an LSTM connected after the convolutional layers. Finally, we collected a database of PAC-MAN images and moves by playing the game ourselves, and then used supervised learning with a cross-entropy loss to pretrain both the purely convolutional Q-agent, and a convolutional Q-agent with an LSTM.

We found that because of the vastly larger number of iterations, we were able to achieve convergence in the supervised learning scenario, whereas we did not see the same behavior in the pure reinforcement learning scenario. While the agents never reached human level performance, they did clearly pick up some aspects of human-like play, especially in the opening sequences of the game. For example, both the convolutional-only and convolutional + LSTM networks learned to immediately eat the power pellets, a strategy we employed often when playing the game to create the database.

7. Future work

With more computational power, or more training time, we would like to be able to train an agent purely with reinforcement learning, not resorting to supervised learning. Demis Hassabis, founder of DeepMind, agrees in [6] that their AlphaGo Go playing agent could have been trained without initial supervised learning given more time. We think that with further optimization of network architecture and other hyper-parameters, and increased training time, it should be possible to reinforcement train a reasonably good PAC-MAN agent.

References

[1] A. Wong, P. A. The pac-man projects. Documentation for the PAC-MAN implementation. 4

[2] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 5

[3] Bom, L., Henken, R., and Wiering, M. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on* (April 2013), pp. 156–163. 2

[4] Borsa, D., Graepel, T., and Shawe-Taylor, J. Learning Shared Representations in Multi-task Reinforcement Learning. *ArXiv e-prints* (Mar. 2016). 1

[5] Bradski, G. *Dr. Dobb's Journal of Software Tools*. 5

[6] Byford, S. Deepmind founder demis hassabis on how ai will shape the future. An interview with Demis Hassabis, founder of DeepMind, on their AlphaGo Go player. 8

[7] DeLooze, L. L., and Viner, W. R. Fuzzy q-learning in a nondeterministic environment: developing an intelligent ms. pac-man agent. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* (2009), IEEE, pp. 162–169.

[8] François-Lavet, V., Fonteneau, R., and Ernst, D. How to discount deep reinforcement learning: Towards new dynamic strategies. *CoRR abs/1512.02011* (2015).

[9] Hausknecht, M. J., and Stone, P. Deep recurrent q-learning for partially observable mdps. *CoRR abs/1507.06527* (2015). 1

[10] Heinrich, J., and Silver, D. Deep Reinforcement Learning from Self-Play in Imperfect-Information Games. *ArXiv e-prints* (Mar. 2016). 1

[11] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780. 3

[12] Jiang, N., and Li, L. Doubly robust off-policy evaluation for reinforcement learning. *CoRR abs/1511.03722* (2015). 1

[13] Mertikopoulos, P., and Sandholm, W. H. Learning in games via reinforcement and regularization. *ArXiv e-prints* (July 2014).

[14] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. *CoRR abs/1602.01783* (2016).

[15] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013). 1, 4, 6

[16] Ong, H. Y., Chavez, K., and Hong, A. Distributed deep q-learning. *CoRR abs/1508.04186* (2015). 1

[17] Parisotto, E., Ba, L. J., and Salakhutdinov, R. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR abs/1511.06342* (2015). 1

[18] PythonWare. Python Imaging Library (PIL).

[19] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229* (2013).

[20] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489. 1

[21] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567* (2015). 3

[22] Taylor, M. E., and Stone, P. Transfer learning for reinforcement learning domains: A survey. *The Journal of Machine Learning Research 10* (2009), 1633–1685. 3

[23] van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. *CoRR abs/1509.06461* (2015). 1

[24] Wang, Z., de Freitas, N., and Lanctot, M. Dueling network architectures for deep reinforcement learning. *CoRR abs/1511.06581* (2015).

[25] Zaremba, W., and Sutskever, I. Reinforcement learning neural turing machines. *CoRR abs/1505.00521* (2015). 1